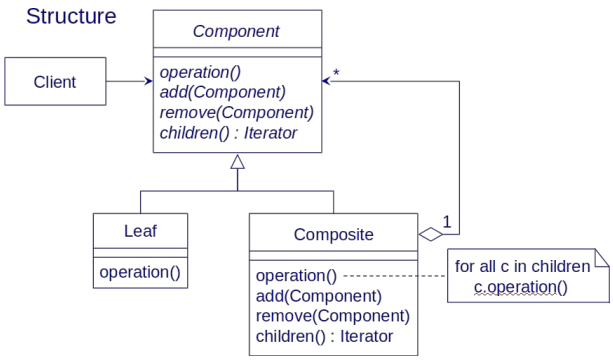
Clayton Turner
CSIS 603 - Design Patterns

Composite Pattern: Android's ComposeShader



A composite pattern is one whose intent is to compose objects into tree structures to represent part-whole hierarchies. This evolves into deeper and deeper trees because objects that are composed together (i.e. a composite) can then be composed with another object of the composite/original type. This is able to be done through subclassing - a composite needs to be related to the original type and this is commonly seen through the utilization of inheritance. This allows clients to ignore the differences between composite objects and atomic (non-composited - represented as a leaf node) objects, treating them uniformly (as per Dr Moore's slides). The original class being inherited from is an abstract class, which is why the aforementioned atomic objects are referred to as a leaf node as they are a concrete implementation of the original class while the composite is a special composed version of the leaf. Both the leaf and composite inherit from the abstract component class. The following is information from Dr Moore's slides to be referenced later:

The component is required to:

1) Declares an interface for objects in the composition
2) Implements default behavior common to all classes as needed
3) Declares an interface for accessing/managing children

The leaf:

1) Represents leaf objects in the composition (no children)
2) Defines behavior for primitive objects in composition

The composite:

1) Defines behavior for components having children
2) Stores child objects
3) Implements child-related operations defined in Component by delegating to all child objects

The client:

1) Manipulates objects in the composition through the Component interface.

The class I will be detailing the composite pattern for is the ComposeShader class within the Android operating system. ComposeShader is a subclass of the Shader class. The Shader class is not directly used, but, rather, a subclass of it is used within the Paint class through a setShader

method call (as per the Shader Android reference), which satisfies the un-instantiability of the component class (i.e. abstract). The Shader class is used for returning horizontal spans of colors during drawing. The ComposeShader class is used for returning the composition of two other shaders. The ComposeShader class inherits 3 methods from the Shader class: void finalize(); boolean getLocalMatrix(Matrix localM); void setLocalMatrix(Matrix localM);. The Shader class is then thereby providing an interface for the objects in the composition. Each of these methods are overriden so there is no default behavior common to all classes needed to be passed along. The get method described earlier for the Shader class allows accessing of the children objects and the set method allows the managing of the same objects. The subclassed Shader that is passed back from the Paint setShader method, if not a composition, would be considered the leaf in this scenario. Because we know it is not a composite as the ComposeShader has not been called, then it must not have children, satisfying the first requirement. All this subclass needs to know about is itself for the horizontal color shading, so no methods are explicitly required as the Paint class can override its own shading if it needs to re-do shading or simply clear it; so all the behavior for these leaves are defined in that none is really needed. The ComposeShader, in its usage will take two of these subclassed shaders and compose them together. There are two modes for combining these shades (Xfermode and PorterDuff.Mode), but that is not important to describe the composition, but rather it can further affect the display of the shader itself (through alterations to saturation, brightness, etc.) What is important about the mode passed to the ComposeShader is that the mode itself stores the composed version of the shaders and the original shaders themselves are still stored in the ComposeShader, satisfying the requirements for defining behavior for components having children and the storage of child objects. Lastly, the Paint class functions as the client here as it is the one, as aforementioned, gathering these shaders and actually doing the tinkering to display to the screen in a specific manner.

References

ComposeShader: http://developer.android.com/reference/android/graphics/ComposeShader.html
https://android.googlesource.com/platform/frameworks/base.git/+/android-m-preview-1/graphics/java/android/graphics/ComposeShader.java
Shader: http://developer.android.com/reference/android/graphics/Shader.html
https://android.googlesource.com/platform/frameworks/base.git/+/android-m-preview-1/graphics/java/android/graphics/Shader.java
Paint: http://developer.android.com/reference/android/graphics/Paint.html
https://android.googlesource.com/platform/frameworks/base.git/+/android-m-preview-1/graphics/java/android/graphics/Paint.java

Adding code for the ComposeShader from the android.googlesource link from above

ComposeShader.java

```
package android.graphics;
```

```
/** A subclass of shader that returns the composition of two other shaders, combined by
    an {@link android.graphics.Xfermode} subclass.
*/
public class ComposeShader extends Shader {
    private static final int TYPE_XFERMODE = 1;
    private static final int TYPE_PORTERDUFFMODE = 2;
    /**
     * Type of the ComposeShader: can be either TYPE_XFERMODE or
TYPE_PORTERDUFFMODE
     */
    private int mType;
    private Xfermode mXferMode;
    private PorterDuff.Mode mPorterDuffMode;
    /**
     * Hold onto the shaders to avoid GC.
     */
    @SuppressWarnings({"UnusedDeclaration"})
    private final Shader mShaderA;
    @SuppressWarnings({"UnusedDeclaration"})
    private final Shader mShaderB;
    /** Create a new compose shader, given shaders A, B, and a combining mode.
        When the mode is applied, it will be given the result from shader A as its
        "dst", and the result from shader B as its "src".
        @param shaderA  The colors from this shader are seen as the "dst" by the mode
        @param shaderB  The colors from this shader are seen as the "src" by the mode
        @param mode     The mode that combines the colors from the two shaders. If mode
                        is null, then SRC_OVER is assumed.
    */
    public ComposeShader(Shader shaderA, Shader shaderB, Xfermode mode) {
        mType = TYPE_XFERMODE;
        mShaderA = shaderA;
        mShaderB = shaderB;
        mXferMode = mode;
        init(nativeCreate1(shaderA.getNativeInstance(), shaderB.getNativeInstance(),
                (mode != null) ? mode.native_instance : 0));
    }
    /** Create a new compose shader, given shaders A, B, and a combining PorterDuff mode.
        When the mode is applied, it will be given the result from shader A as its
        "dst", and the result from shader B as its "src".
```

```
        @param shaderA  The colors from this shader are seen as the "dst" by the mode
        @param shaderB  The colors from this shader are seen as the "src" by the mode
        @param mode     The PorterDuff mode that combines the colors from the two shaders.
    */
    public ComposeShader(Shader shaderA, Shader shaderB, PorterDuff.Mode mode) {
        mType = TYPE_PORTERDUFFMODE;
        mShaderA = shaderA;
        mShaderB = shaderB;
        mPorterDuffMode = mode;
        init(nativeCreate2(shaderA.getNativeInstance(), shaderB.getNativeInstance(),
                mode.nativeInt));
    }
    /**
     * @hide
     */
    @Override
    protected Shader copy() {
        final ComposeShader copy;
        switch (mType) {
            case TYPE_XFERMODE:
                copy = new ComposeShader(mShaderA.copy(), mShaderB.copy(), mXferMode);
                break;
            case TYPE_PORTERDUFFMODE:
                copy = new ComposeShader(mShaderA.copy(), mShaderB.copy(),
mPorterDuffMode);
                break;
            default:
                throw new IllegalArgumentException(
                        "ComposeShader should be created with either Xfermode or PorterDuffMode");
        }
        copyLocalMatrix(copy);
        return copy;
    }
    private static native long nativeCreate1(long native_shaderA, long native_shaderB,
            long native_mode);
    private static native long nativeCreate2(long native_shaderA, long native_shaderB,
            int porterDuffMode);
}
```