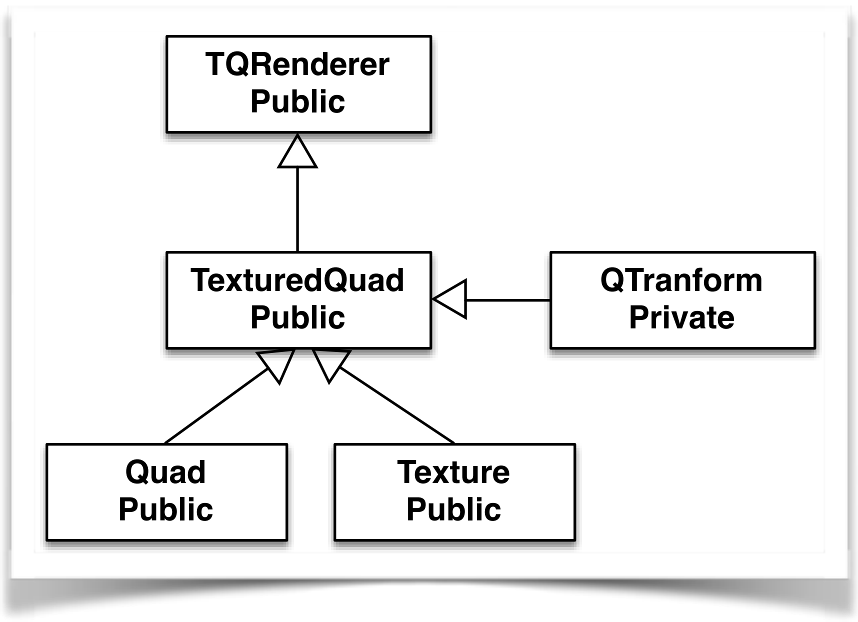
In a layered approach to an application design such as MVC design paradigm, you may wish to have reusable portions of your code factored into reusable methods and classes. You'll then group those methods and classes into a framework, forming your model portion of your code, or your intellectual property. You may then use this model or framework in your view and controller code, and across all your applications. You're therefore creating a separation between your model code used often, and you're view and controller code used once and customized per application. And by having this separation, you can have your model code abstract platform dependent features, and present a uniform interface across all your applications.



In this sample code our model portion of the code is comprised of `Quad`, `Texture`, `TexturedQuad`, and `TQRenderer` (textured quad renderer) classes. Each of these classes present a standard interface for usage across one or more applications. There is also a group of `simd` based methods for forming *linear transformation matrices of perspective geometry* that we want to export as a public interface and be part of our model code.

Additionally, there is the QTransform (quad transform) functor object in the client code. If we're building a framework encapsulating our model code, only the setters of this object will be exposed through `TexturedQuad`, and subsequently, through `TQRenderer` class. Therefore, there is no need to expose the interfaces of this class as public. And as such the interface for this class in a framework environment will be private.

Furthermore, in this sample we're demonstrating a design patterns for opaque data model programming paradigm. In order to demonstrate this design pattern we'll look at the implementation of the `Quad` class.

Firstly, let us look at the public interface the client will see for the Quad class:

```
// Opaque class encapsulating the data for quad
class QuadData;

class Quad
{
public:
    // Constructor
    Quad(id<MTLDevice> device);

    // Destructor
    virtual ~Quad();

    // Copy Constructor
    Quad(const Quad& rQuad);

    // Assignment operator
    Quad& operator=(const Quad& rQuad);
        :
        :
    // Encode a quad using a render encoder
    void encode(id<MTLRenderCommandEncoder> renderEncoder);

private:
    QuadData *mpQuad;
};
```

Note the forward declaration of the QuadData, referenced by a pointer as a private instance variable mpQuad. The implementation file includes the type definition of the QuadData opaque data

```
class QuadData
{
public:
    // textured Quad
    id <MTLBuffer>  m_VertexBuffer;
        :
        :
    // Table for indices
    NSUInteger m_Index[kQuadIndexMax];
};
```

In our layered approach to design, we will never expose the actual instance variables that are part of the QuadData class, we will only only expose functionality operating on this opaque data reference through public methods of the Quad class. And to have this distinct separation we will implement some static methods operating on the QuadData object.

Some static methods for the QuadData object are:

```
static QuadData *AAPLMetalQuadCreate(id<MTLDevice> device);

static QuadData *AAPLMetalQuadCreateCopy(QuadData * const pQuadSrc);

static void AAPLMetalQuadDelete(QuadData *pQuad);
```

Note that we have above a static constructor, deep-copy constructor, and a delete methods. If, for example, we had opted to implement our own reference counted `QuadData` object using atomic counters, then instead of a static delete method, we would have retain and release methods here instead (reference <http://ubm.io/1kIjTJo>).

Next, we wish to expose this functionality to the client. To do so, we will expose the functionality through constructor, copy constructor, assignment operator and the virtual destructor of the class:

```
    // Constructor
    Quad::Quad(id<MTLDevice> device)
    {
        mpQuad = AAPLMetalQuadCreate(device);
    }

    // Destructor
    Quad::~Quad()
    {
        AAPLMetalQuadDelete(mpQuad);
    }

    // Copy Constructor
    Quad::Quad(const Quad& rQuad)
    {
        QuadData *pQuad = AAPLMetalQuadCreateCopy(rQuad.mpQuad);

        if(pQuad != nullptr)
        {
            AAPLMetalQuadDelete(mpQuad);

            mpQuad = pQuad;
        }
    }
```

```
// Assignment operator
Quad& Quad::operator=(const Quad& rQuad)
{
    if(this != &rQuad)
    {
        QuadData *pQuad = AAPLMetalQuadCreateCopy(rQuad.mpQuad);

        if(pQuad != nullptr)
        {
            AAPLMetalQuadDelete(mpQuad);

            mpQuad = pQuad;
        }
    }

    return *this;
}
```

This will bring us to another interesting topic. In a collaborative approach to layered application architecture, typically the person or the team responsible for the core application functionality will approach this problem in a slightly different fashion.

In our example, the methods operating on the QuadData made here static will be made public. That is to say, the interface file will now expose these methods,

```
// Opaque data object
class QuadData;

// Constructor for creating a opaque data object.
QuadData *AAPLMetalQuadCreate(id<MTLDevice> device)

// Create a deep-copy of the opaque data object.
QuadData *AAPLMetalQuadCreateCopy(QuadData * const pQuadSrc);

// Delete the opaque data object
void AAPLMetalQuadDelete(QuadData *pQuad);
```

In this approach, we will still use forward declaration for the `QuadData` class, details of which, as before, will be hidden from the client and only exposed in the implementation file. We may adopt this design if we are targeting one or more platforms. And if we are targeting more than one platform, the platform specific data types and objects can be guarded with conditional compilation. And since these are merely a group interfaces, we can adopt language bindings to target C++, Objective-C, or other programming and scripting languages.