

Re: Properties Shared Amongst Objects

Source: <http://coding.derkeiler.com/Archive/General/comp.object/2008-02/msg00178.html>

- *From:* "H. S. Lahman" <hsl@xxxxxxxxxxxxxxxxxxxxx>
 - *Date:* Mon, 18 Feb 2008 18:19:08 GMT
-

Responding to Sanford...

The applications that I'm writing are dll's (dynamic linked libraries) that are run inside of a host. The host can query my application for certain information. For example, it can ask for the number of parameters my application uses. It can also ask for a parameter's value, name, label, and a human readable string representing its current value. Also, it can set a parameter value. Parameter values from the host's perspective are always in the range of [0, 1].

So the view of my application from the host's perspective is very flat, a list of parameters and their values (there are other ways that the host and my application interact, but for the purposes of this post, I'll skip those details).

A DLL is basically just a reusable subsystem. In that context one usually strives to provide a subsystem interface that reflects the invariants of the DLL subject matter. That provides a generic interface to clients while allowing DLL implementations to be swapped transparently to the clients. Typically "flat" interfaces work well in that context, so all seems as it should be so far.

<parenthetical aside>

Unfortunately sometimes the clients want to use some other interface than the pristine one provided by the DLL. In that case the Facade pattern comes into play and one can link a custom interface into the client that provides its view while its implementation knows how to convert that view to the interface the DLL provides. (If you haven't checked it out, the category on Application Partitioning in my blog has a post about a general subsystem interface model.) I mention this because there may be some additional leeway in where the "flattening" occurs.

</parenthetical aside>

I might have something like this in my code:

```
enum ParameterId
{
    AmplitudeLevelId,
```

Re: Properties Shared Amongst Objects

```
FrequencyId
};

//
// Called by the host:
//

float GetParameterValue(int parameterId)
{
float value;

switch(parameterId...

return value;
}

void SetParameterValue(int parameterId, float value)
{
// ...
}

void GetParameterName(int parameterId, char *name)
{
// ...
}
```

Whereas this flat view is useful for the host, it's not so useful for my application. For example, a frequency parameter value may need to be mapped to a more useful range than the [0, 1] range the host uses, say, [20, 20000]. Also, an amplitude parameter value may need to map parameter values to an exponential curve. In addition, it's useful for my application to create an object oriented architecture in order to manage behavior. After some consideration, I might group some of the parameters into classes. Using the above example, I could have this:

This sort of change is really a mapping between the client's internal representation of a value and the DLL's internal representation. (The same is true for mapping <.5 to OFF in your other example below.)

I think this is really a job for the interface. Neither client nor DLL are confused about the semantics of the value, but they are speaking different languages about its implementation (syntax). For any complex semantics one can usually provide multiple different syntaxes for accessing it. In OO-land interfaces represent syntactic views of the implementation semantics. So logically I see this sort of conversion to be the interface's job.

This maps quite conveniently into the notion of a Facade pattern. The Facade is an object with an implementation. That implementation lives to resolve syntactic mismatches. So I would look to do these sorts of conversions in the interface rather than explicitly within the DLL subject matter.

<caveat>

The operative phrase is 'these sorts of conversions'. Your examples are relatively simple mappings of /data/. A

Re: Properties Shared Amongst Objects

some point the conversion may become so complex that one wants to deal with it explicitly within the DLL subject matter. More likely, one may need different /behaviors/ that are dynamically determined based on incoming parametric data. In that case, one needs a different mechanism within the DLL subject matter.

As mentioned, the Strategy pattern is often useful. One reason is that the mechanics map very nicely into a flat, data-driven interface. That allows the interface to interpret context by examining parameters and then dynamically instantiate the [Context] to [Strategy] association. Then the DLL solution goes about business as usual by navigating to whoever is there at the end of the association.

</caveat>

As another parenthetic comment, there is no rule that requires the Facade to be a single class. I have once worked on a system where the programmatic interface was, itself, an entire subsystem with about 80 KLOC of code. [The client represented digital testing as an event stream while the tester itself was pattern-based. So the entire event stream had to be read and temporarily stored so that it could be analyzed as a whole to determine pattern boundaries.] Hopefully you are nowhere near that level of syntactic mismatch. I mention this just to underscore the lengths one can go to to preserve the isolation of the DLL subject matter implementation so that it is completely indifferent to who the client is.

```
class Amplifier
{
public:
float GetAmplitudeLevel() const;
void SetAmplitudeLevel(float amplitudeLevel);
void GetAmplitudeName(char *name) const;
void GetAmplitudeLabel(char *label) const;

// ...
};

class Oscillator
{
public:
float GetFrequency() const;
void SetFrequency(float frequency);
void GetFrequencyName(char *name) const;
void GetFrequencyLabel(char *label) const;

// ...
};
```

It then becomes a matter of mapping parameter changes to the component objects that represent those parameters.

What I noticed after factoring parameters into several classes is that I was duplicating code for converting parameter values to/from the host. For example, I noticed that some parameters behave like an on/off switch, mapping parameter values less than 0.5 as "Off" and greater than or equal to 0.5 to "On." Also, some parameters represented a selection, e.g. Red, Green, Yellow, etc. In all, there are three or four parameter types I discovered.

Re: Properties Shared Amongst Objects

So I thought it would be advantageous to create a set of parameter classes. The base parameter class would provide functionality common to all parameter classes. Derived parameter classes would specialize parameter values, e.g. a SwitchParameter to represent an on/off parameter.

My amplifier class could then look like this:

```
class Amplifier
{
private:
    ExponentialParameter amplitudeLevel;

public:
    Amplifier() :
    amplitudeLevel("Amplitude", "dB");
    {
    }

    float GetAmplitudeLevel() const
    {
    return amplitudeLevel.GetValue();
    }

    void SetAmplitudeLevel(float level) const
    {
    amplitudeLevel.SetValue(level);
    }

    // ...
};
```

As a practical matter, Facades commonly do a very similar thing. In the bowels of their implementation they invoke the right conversion from a table of functions using the parameter identity. Then a conversion method that applies to several parameters can be implemented once. (Obviously this only works if the mapping of parameter to conversion is fixed, which it should be for simple syntax mismatches.)

So far, so good. But as I mentioned in my original post, I might have several components that belong to a group. The components in the group all sharing the same parameter/property values. So why not create the parameter objects separately and pass them to each component in the group?

Assuming your conversions are all similar to your examples, one way to attack this in the interface is to index the conversion jump table by both parameter ID and group ID. To properly dispatch the incoming message the Facade always needs to understand how the client's identity maps into the DLL's implementation identity.

The client, though, may have no knowledge of groups within the DLL subject matter. In that case, it is fair for

Re: Properties Shared Amongst Objects

the Facade to look it up the same way a factory object in the DLL implementation would to decide how to instantiate a component. IOW, it is just another lookup table somewhere in the DLL that provides the context mapping to groups and the Facade can use that same lookup table to get a group ID to use in its own conversion table.

This might seem a bit arcane at first, but the combination of jump tables and lookup tables is actually pretty robust. You can capture pretty complex business rules and policies about context this way in simple data tables. Then all you have to do is modify the tables when things change. As a bonus the executable code is reduced from potentially many decision statements to a couple of simple lookups.

Note that your code is doing a very similar thing. But to keep track of what is actually going on one needs to "walk" it through several classes. If you can do the conversions in the Facade, then all of the conversion is in that one place (albeit not quite as elegantly because of the verbosity of the table initialization).

There is nothing wrong with me that could not be cured by a capful of Drano.

H. S. Lahman

hsl@xxxxxxxxxxxxxxxxxxxx

Pathfinder Solutions

<http://www.pathfindermda.com>

blog: <http://pathfinderpeople.blogs.com/hslahman>

"Model-Based Translation: The Next Step in Agile Development". Email

info@xxxxxxxxxxxxxxxxxxxx for your copy.

Pathfinder is hiring: http://www.pathfindermda.com/about_us/careers_pos3.php.

(888)OOA-PATH

.