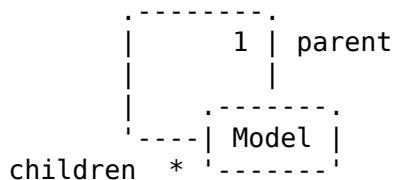


Simulation models and treeness



Miguel González Cuadrado <mgcuadrado@gmail.com>

12th May 2020

Contents

Intro	ii
Treeness	iv
I Model	I

2	Model life and ownership	I
2.1	Kinds of pointer	2
2.2	Implementations for the pointers	4
3	Model creation	5
3.1	Child model creation	6
3.2	Root model creation	8
4	Model destruction	9
5	Model access	10
6	Services	10
7	Feature installation	10

List of Figures

List of Tables

Intro

I'm sure someone must have written something about this, but i haven't found any reference (i haven't looked too hard

either): big object-oriented programming frameworks often feature a core, central concept implemented as a oversized base class. The user of the framework is expected to derive from this base class extensively. As the framework grows bigger, the base class keeps accumulating more and more features that the framework developer wants all derived classes to have: constructs like debugging facilities, serialisation, error reporting, and many field-specific functionality get tacked onto the poor base class.

I'm not ranting about this state of affairs. I think it's inevitable, and in a certain sense desirable: although this makes the learning curve steeper for the programmer using the framework, the rewards soon outweigh the effort. But knowing the issue, the framework designer must always strive to keep the overgrowth under control, and make each feature as clear and lean as possible.

In real-time simulation as i've been practising it, the central concept revolves around state evolution and tree-like organisation paralleling the reality that's being modeled. I've called the core base class "Model". The "Model" class' concerns are:

- organisation in the form of a tree;
- usage of the tree structure as a means of communication and propagation;
- the (observable) state (reification, management, evolu-

tion);

- naming and debugging facilities.

This is the first article¹ in a series about the design of Model. Each installment will delve into a single topic that constitutes a separate feature or design consideration for Model. This first one looks at treeness. The others will be about (probably)

- communication and propagation,
- state,
- wrapping up and odds and ends (model constructor specification macros, config object manipulation...).

Treeness

The tree structure is the natural organisation for simulation models, since it matches the way we look at reality and the way we organise our analysis at several levels:

- an aeroplane (represented by a tree node) contains aerodynamic surfaces (child nodes);
- several aeroplanes (nodes) with a common goal can be organised into a fleet of aeroplanes (parent node); the aeroplanes in a fleet are sibling nodes;

¹[ˈɑːrtɪkəl]kəl]

- gravity and atmosphere (nodes) reign over an aeroplane (child node);
- simulation services (numerical integration, servers, et cetera) are located near the top of the tree, ancestor to all the nodes they serve;
- the root of the tree is the prime access to all the tree nodes.

Additionally, once treeness is in place, it can be used very effectively for such things as

- human-readable identification of nodes;
- action propagation down the tree (most importantly, state management and evolution);
- communication across the tree (information and requests can be passed up and down the tree);
- text message addressing and handling.

As we'll see, although implementing a general tree structure is an easy task, the specifics of "Model" call for specific features that i've solved in a specific way that works for me.

Let's rock.

1 Model

The `Model` class is the basic type of nodes in the model tree. It contains a lot of functionality, but no real simulation-related behaviour. So, most of the time, model instances have a type derived from `Model`, that specifies various bits of behaviour, concerning state, services, communication, and the like. Nevertheless, pure `Model` instances are also useful in structuring the tree (e.g., such an instance may serve as the root of a subtree that contains models that record the evolution of other models), or they can be fitted with services (§6) or features (§7).

2 Model life and ownership

That is, what governs model creation, and more important, destruction; how is a model kept alive.

There are several strategies for model life and ownership, and several non-strategies like holding models in the stack or controlling life explicitly. Everything i've tried came tumblin' down when i started implementing Passamonte, where models are required to spawn and disappear dynamically, and to keep promiscuous relationships among them. Everything except this.

2.1 Kinds of pointer

At the risk of overstructuring this section, but for increased clarity, i've subsectioned the exposition of the different kinds of pointer and their life-maintaining requirements.

2.1.1 Life-preserving pointers: child pointer, root pointer

There must be a single life-maintaining resource for each model, so that death can be controlled precisely. And the life maintaining must be dynamic. We all knew this led to shared pointers or some similar implementation, but the important point here is that each model has exactly one sharedish pointer pointing to it. And the sharedish pointer is kept by the model's parent, because when a node goes away, all its descendence must go away with it. Therefore, we have two life-preserving kinds of pointers: the *child pointer*, which is a shared pointer to a model kept by its parent, and the *root pointer*, which is a shared pointer to a parentless model.

2.1.2 Non-life-preserving parent pointer

A node therefore knows all its children, through the child pointers. But a node must also know its parent. This knowledge must not be life preserving, and it needn't even be

any smart, since as long as a node exists, we're sure it parent does too (thanks to what's been explained above about life-maintaining child pointers). We'll call this kind of pointer *parent pointer*, and i can already tell you it's as dumb a pointer as you can get, but i've wrapped it in a uniformising coat to make it look similar to its smarter counterpart child pointer. Making it any smarter made some constructor-time operations impossible, like a model creating during its own construction a child that needs to access its parent, since any smart pointer must wait until the construction of the whole object to be valid.

2.1.3 Non-life-preserving model pointer

We also need an efficient way for a model to keep a pointer to another model. Here, the kind we want is C++'s weak pointer, which can be converted on demand to the shared (child) pointer it's associated to, but doesn't contribute to the life expectancy of the model pointed to. The dynamicity of the model scenario is supported by the fact that the weak pointer becomes effectively null when its associated shared pointer goes away; so, even though a model may be pointing to another pointer whose life it doesn't control, it will automatically know whether the other model is still alive or not when it needs to access it.

2.2 Implementations for the pointers

Child pointer, which i’ve named “`cptr<M>`” in the `Model` class (where “`M`” is an optional template argument, which defaults to “`Model`”) is simply a renaming of `shared_ptr<M>`. We don’t need more than that, since it’s gonna be private to `Model`, and `Model` knows precisely what to do with it, when, and how.

Model pointer, which i’ve named “`mptr<M>`” in the `Model` class, is what i’ve termed an automatic weak pointer. It works as, and is implemented in terms of, `weak_ptr<M>`. In simulation, you’ll have three main uses for `mptr<M>`, namely copying it, checking it for nonnullness, and dereferencing it. Copying is ok, but the other two require you to acknowledge in syntax that it has a specific implementation, if you use the naked `weak_ptr<M>`, specifically “not `p.expired()`”, and “`p.lock()->`”; and this happens all the time. Therefore, the automatic weak pointer replaces those operations with the more natural “`operator bool()`” and “`operator->()`”. This is the syntax you’d use with a plain `shared` pointer, so it’s important to remember automatic weak pointers don’t keep models alive, despite their syntax. Other bells and whistles: resetting, dynamically casting, converting to pointer-to-const, equality comparisons, and “`owner_less()`” specialisation for the sake of maps with automatic weak pointer keys.

Parent pointer, which i’ve named “`pptr<M>`” in the `Model`

class, is a plain C++ pointer wrapped to resemble `m_ptr<M>` as much as practical.

Root pointer, which i’ve named “`root_model_t<M>`” in the `Model` class, is `shared_ptr<M>` wrapped to check correct ownership (“`use_count()==1`”) and to implement correct model dismantlement on destruction (see §4).

3 Model creation

Now that the tree concept takes in charge its nodes’ life, we can go a step further and encapsulate node creation. By doing so, the policy is enforced, and the syntax becomes rather pleasing.

Each model gets, upon construction, its name and a pointer to its parent. The name must be specified explicitly most of the time, since it’s specific to each model instance, and can’t be inferred. The parent is specified by the parent itself. These two pieces of information are put together into a single object, that i call “`config`”, whose type can be extended to accommodate more pieces of information as need arises.

The `Model` class, and any class that derives from it, has a constructor whose first argument is the `config` object. A derived class can inspect this object, but it mainly forwards it to the `Model` constructor, which uses it to set up the object’s

parent and name².

3.1 Child model creation

Thanks to this convention of having the config object as the first constructor argument, child model creation is designed to be done like this:

```
1  auto radio= // radio is a model pointer to the newly  
    created radio  
2      helicopter // parent of the new model  
3      ->make_child<Radio>( // Radio is the type of  
    the new model  
4          {"fm_radio"}, // name of the new model  
5          fm_radio_spec, fm_radio_tag); // additional  
    constructor args
```

The `make_child<>()` call creates the new model, and registers it as a new child to the model on which it was invoked (`helicopter` in the example). The concrete type of the new model, which determines the constructor the arguments are passed to, is specified as a template argument (`Radio` in the example). If none is specified, `Model` is assumed.

²Well, actually, because of an implementation detail, namely virtual derivation from the `named` class, each derived class must invoke `named` constructor explicitly, but this detail is hidden by the `Model` constructor specification macros in most cases.

The first argument to `make_child<>()` is a configuration instance, without the parent specification, which is filled in by `make_child<>()` automatically. It contains at least the name, and possibly other data, depending on the type of the new model (by default, a derived type inherits the `config` type from its base class). You don't need to specify the `config` specific type, but must group all its components in curly braces. I feel this makes this first argument stand out nicely from the rest of the arguments. Usually, you'll need at least a name, "fm_radio" in the example.

The rest of the arguments to `make_child<>()` are passed verbatim to the specific model constructor. You can have anything there for derived model classes, including templated arguments, or constructor overloads.

You may have models that created other models that are children to it. No problem. But you may also need to do some setup tasks after a model (and all its children) has been completely constructed, and has gotten a shared pointer. Indeed, if the new model needs to tell anybody else about itself, for instance, it must pass a model pointer; but it won't be able to produce a model pointer during its constructor, since the shared pointer the model pointer references isn't ready yet. For this purpose, any model class can implement the virtual method "`setup_this()`", that is called by `make_child<>()` after the new model has been registered with its child pointer. As a bonus that may be useful sometimes, when a model "m" is

created that creates other child models, things are done so the whole subtree rooted at `m` is completely created before their setup methods are invoked, starting at `m`, and propagating depth first.

The `make_child<>()` method also performs some additional construction-related checks and chores, that are specific to features described in other articles.

The return value of `make_child<>()` is a model pointer to the newly created model. You can simply ignore it, but you can also keep it for later use, or directly use it to call a method on it, even a chained `make_child<>()` call.

3.2 Root model creation

The root model is not created with `make_child<>()` because it has no parent, and we need the return value to be life preserving. Instead, the very similar static `Model` method `make_root<>()` is used. It works in almost the same way, as `make_child<>()`, but instead of returning a model pointer to the newly created model, it returns a root pointer to it. You'll have to keep the root pointer to maintain the whole tree alive.

4 Model destruction

Model destruction is somewhat special, in that in order to comply with state management rules (see the article about state), it's not done explicitly. Instead, the model is instructed to go to a stage called "removing", which causes it to request its own removal from its parent and destruction during the next "update" phase.

An exception is the root node, which is destroyed by losing the root pointer to it (either destroying the pointer, or resetting it).

In all cases, the impending destruction of a model first causes the virtual method "dismantle_this()" to be called on it and propagated depth-first down its descendancy, before any destructor is called. A derived model class can override this method to perform specific tasks just prior to destruction.

At any rate, in case this is useful for some other kind of tree that's not concerned with state management rules, where it not for this rules, the following invocations would be accessible for child removal (where "c" is a model pointer to the model to be removed, and "p" is a model pointer to its parent):

- `p->remove_child("cname")` (where "cname" is the name of the model to be removed),
- `p->remove_child(c)`,

- `Model::remove(c)`.

5 Model access

If you have a model pointer to a model “p”, you can get

- a list of model pointers to its children, with `p->children()`;
- a model pointer to the child with a specific name “cname”, dynamically converted to a specific type “M” (which defaults to “Model”), with `p->child<M>("cname")`; it will throw if there’s no child with that name, and return a null pointer if the dynamic type conversion doesn’t succeed.

You should usually avoid accessing children by their name, and try to get pointers to models programmatically (from pointers returned at creation, or by scanning lists obtained with “children()”, or through communications).

6 Services

7 Feature installation