Service Locator

Game Programming Patterns / Decoupling Patterns

Intent

Provide a global point of access to a service without coupling users to the concrete class that implements it.

Motivation

Some objects or systems in a game tend to get around, visiting almost every corner of the codebase. It's hard to find a part of the game that *won't* need a memory allocator, logging, or random numbers at some point. Systems like those can be thought of as *services* that need to be available to the entire game.

For our example, we'll consider audio. It doesn't have quite the reach of something lower-level like a memory allocator, but it still touches a bunch of game systems. A falling rock hits the ground with a crash (physics). A sniper NPC fires his rifle and a shot rings out (AI). The user selects a menu item with a beep of confirmation (user interface).

Each of these places will need to be able to call into the audio system with something like one of these:

```
// Use a static class?
AudioSystem::playSound(VERY_LOUD_BANG);

// Or maybe a singleton?
AudioSystem::instance()->playSound(VERY_LOUD_BANG);
```

Either gets us where we're trying to go, but we stumbled into some sticky coupling along the way. Every place in the game calling into our audio system directly references the concrete `AudioSystem` class and the mechanism for accessing it—either as a static class or a singleton GoF.

These call sites, of course, have to be coupled to *something* in order to make a sound play, but letting them poke at the concrete audio implementation directly is like giving a hundred strangers directions to your house just so they can drop a letter on your doorstep. Not only is it a little bit *too* personal, it's a real pain when you move and you have to tell each person the new directions.

There's a better solution: a phone book. People that need to get in touch with us can look us up by name and get our current address. When we move, we tell the phone company. They update the book, and everyone gets the new address. In fact, we don't even need to give out our real address at all. We can list a P.O. box or some other "representation" of ourselves instead. By having callers go through the book to find us, we have *a convenient single place where we control how we're found.*

This is the Service Locator pattern in a nutshell—it decouples code that needs a service from both *who* it is (the concrete implementation type) and *where* it is (how we get to the instance of it).

The Pattern

A **service** class defines an abstract interface to a set of operations. A concrete **service provider** implements this interface. A separate **service locator** provides access to the service by finding an appropriate provider while hiding both the provider's concrete type and the process used to locate it.

When to Use It

Anytime you make something accessible to every part of your program, you're asking for trouble. That's the main problem with the Singleton GoF pattern, and this pattern is no different. My simplest advice for when to use a service locator is: *sparingly*.

Instead of using a global mechanism to give some code access to an object it needs, first consider *passing the object to it instead*. That's dead simple, and it makes the coupling completely obvious. That will cover most of your needs.

But… there are some times when manually passing around an object is gratuitous or actively makes code harder to read. Some systems, like logging or memory management, shouldn't be part of a module's public API. The parameters to your rendering code should have to do with *rendering*, not stuff like logging.

Likewise, other systems represent facilities that are fundamentally singular in nature. Your game probably only has one audio device or display system that it can talk to. It is an ambient property of the environment, so plumbing it through ten layers of methods just so one deeply nested call can get to it is adding needless complexity to your code.

In those kinds of cases, this pattern can help. As we'll see, it functions as a more flexible, more configurable cousin of the Singleton pattern. When used well, it can make your codebase more flexible with little runtime cost.

Conversely, when used poorly, it carries with it all of the baggage of the Singleton pattern with worse runtime performance.

Keep in Mind

The core difficulty with a service locator is that it takes a dependency—a bit of coupling between two pieces of code—and defers wiring it up until runtime. This gives you flexibility, but the price you pay is that it's harder to understand what your dependencies are by reading the code.

The service actually has to be located

With a singleton or a static class, there's no chance for the instance we need to *not* be available. Calling code can take for granted that it's there. But since this pattern has to *locate* the service, we may need to handle cases where that fails. Fortunately, we'll cover a strategy later to address this and guarantee that we'll always get *some* service when you need it.

The service doesn't know who is locating it

Since the locator is globally accessible, any code in the game could be requesting a service and then poking at it. This means that the service must be able to work correctly in any circumstance. For example, a class that expects to be used only during the simulation portion of the game loop and not during rendering may not work as a service—it wouldn't be able to ensure that it's being used at the right time. So, if a class expects to be used only in a certain context, it's safest to avoid exposing it to the entire world with this pattern.

Sample Code

Getting back to our audio system problem, let's address it by exposing the system to the rest of the codebase through a service locator.

The service

We'll start off with the audio API. This is the interface that our service will be exposing:

```
class Audio
{
public:
  virtual ~Audio() {}
  virtual void playSound(int soundID) = 0;
  virtual void stopSound(int soundID) = 0;
  virtual void stopAllSounds() = 0;
};
```

A real audio engine would be much more complex than this, of course, but this shows the basic idea. What's important is that it's an abstract interface class with no implementation bound to it.

The service provider

By itself, our audio interface isn't very useful. We need a concrete implementation. This book isn't about how to write audio code for a game console, so you'll have to imagine there's some actual code in the bodies of these functions, but you get the idea:

```
class ConsoleAudio : public Audio
{
public:
  virtual void playSound(int soundID)
  {
    // Play sound using console audio api...
  }

  virtual void stopSound(int soundID)
  {
    // Stop sound using console audio api...
  }

  virtual void stopAllSounds()
  {
    // Stop all sounds using console audio api...
  }
};
```

Now we have an interface and an implementation. The remaining piece is the service locator—the class that ties the two together.

A simple locator

The implementation here is about the simplest kind of service locator you can define:

```
class Locator
{
public:
  static Audio* getAudio() { return service_; }

  static void provide(Audio* service)
  {
    service_ = service;
  }

private:
  static Audio* service_;
};
```

The technique this uses is called dependency injection, an awkward bit of jargon for a very simple idea. Say you have one class that depends on another. In our case, our `Locator` class needs an instance of the `Audio` service. Normally, the locator would be responsible for constructing that instance itself. Dependency injection instead says that outside code is responsible for injecting that dependency into the object that needs it.

The static `getAudio()` function does the locating. We can call it from anywhere in the codebase, and it will give us back an instance of our `Audio` service to use:

```
Audio *audio = Locator::getAudio();
audio->playSound(VERY_LOUD_BANG);
```

The way it "locates" is very simple—it relies on some outside code to register a service provider before anything tries to use the service. When the game is starting up, it calls some code like this:

```
ConsoleAudio *audio = new ConsoleAudio();
Locator::provide(audio);
```

The key part to notice here is that the code that calls `playSound()` isn't aware of the concrete `ConsoleAudio` class; it only knows the abstract `Audio` interface. Equally important, not even the *locator* class is coupled to the concrete service provider. The *only* place in code that knows about the actual concrete class is the initialization code that provides the service.

There's one more level of decoupling here: the `Audio` interface isn't aware of the fact that it's being accessed in most places through a service locator. As far as it knows, it's just a regular abstract base class. This is useful because it means we can apply this pattern to *existing* classes that weren't necessarily designed around it. This is in contrast with Singleton GoF, which affects the design of the "service" class itself.

A null service

Our implementation so far is certainly simple, and it's pretty flexible too. But it has one big shortcoming: if we try to use the service before a provider has been registered, it returns `NULL`. If the calling code doesn't check that, we're going to crash the game.

I sometimes hear this called "temporal coupling"—two separate pieces of code that must be called in the right order for the program to work correctly. All stateful software has some degree of this, but as with other kinds of coupling, reducing temporal coupling makes the codebase easier to manage.

Fortunately, there's another design pattern called "Null Object" that we can use to address this. The basic idea is that in places where we would return `NULL` when we fail to find or create an object, we instead return a special object that implements the same interface as the desired object. Its implementation basically does nothing, but it allows code that receives the object to safely continue on as if it had received a “real” one.

To use this, we’ll define another “null” service provider:

```
class NullAudio: public Audio
{
public:
  virtual void playSound(int soundID) { /* Do nothing. */ }
  virtual void stopSound(int soundID) { /* Do nothing. */ }
  virtual void stopAllSounds()        { /* Do nothing. */ }
};
```

As you can see, it implements the service interface, but doesn’t actually do anything. Now, we change our locator to this:

```
class Locator
{
public:
  static void initialize() { service_ = &nullService_; }

  static Audio& getAudio() { return *service_; }

  static void provide(Audio* service)
  {
    if (service == NULL)
    {
      // Revert to null service.
      service_ = &nullService_;
    }
    else
    {
      service_ = service;
    }
  }

private:
  static Audio* service_;
  static NullAudio nullService_;
};
```

You may notice we're returning the service by reference instead of by pointer now. Since references in C++ are (in theory!) never `NULL`, returning a reference is a hint to users of the code that they can expect to always get a valid object back.

The other thing to notice is that we're checking for `NULL` in the `provide()` function instead of checking for the accessor. That requires us to call `initialize()` early on to make sure that the locator initially correctly defaults to the null provider. In return, it moves the branch out of `getAudio()`, which will save us a couple of cycles every time the service is accessed.

Calling code will never know that a "real" service wasn't found, nor does it have to worry about handling `NULL`. It's guaranteed to always get back a valid object.

This is also useful for *intentionally* failing to find services. If we want to disable a system temporarily, we now have an easy way to do so: simply don't register a provider for the service, and the locator will default to a null provider.

Turning off audio is handy during development. It frees up some memory and CPU cycles. More importantly, when you break into a debugger just as a loud sound starts playing, it saves you from having your eardrums shredded. There's nothing like twenty milliseconds of a scream sound effect looping at full volume to get your blood flowing in the morning.

Logging decorator

Now that our system is pretty robust, let's discuss another refinement this pattern lets us do—decorated services. I'll explain with an example.

During development, a little logging when interesting events occur can help you figure out what's going on under the hood of your game engine. If you're working on AI, you'd like to know when an entity changes AI states. If you're the sound programmer, you may want a record of every sound as it plays so you can check that they trigger in the right order.

The typical solution is to litter the code with calls to some `log()` function. Unfortunately, that replaces one problem with another—now we have *too much* logging. The AI coder doesn't care when sounds are playing, and the sound person doesn't care about AI state transitions, but now they both have to wade through each other's messages.

Ideally, we would be able to selectively enable logging for just the stuff we care about, and in the final game build, there'd be no logging at all. If the different systems we want to conditionally log are exposed as services, then we can solve this using the Decorator GoF pattern. Let's define another audio service provider implementation like this:

```
class LoggedAudio : public Audio
{
public:
  LoggedAudio(Audio &wrapped)
  : wrapped_(wrapped)
  {}

  virtual void playSound(int soundID)
  {
    log("play sound");
    wrapped_.playSound(soundID);
  }

  virtual void stopSound(int soundID)
  {
    log("stop sound");
    wrapped_.stopSound(soundID);
  }

  virtual void stopAllSounds()
  {
    log("stop all sounds");
    wrapped_.stopAllSounds();
  }

private:
  void log(const char* message)
  {
    // Code to log message...
  }
  Audio &wrapped_;
};
```

As you can see, it wraps another audio provider and exposes the same interface. It forwards the actual audio behavior to the inner provider, but it also logs each sound call. If a programmer wants to enable audio logging, they call this:

```
void enableAudioLogging()
{
  // Decorate the existing service.
  Audio *service = new LoggedAudio(Locator::getAudio());

  // Swap it in.
  Locator::provide(service);
}
```

Now, any calls to the audio service will be logged before continuing as before. And, of course, this plays nicely with our null service, so you can both *disable* audio and yet still log the sounds that it *would* play if sound were enabled.

Design Decisions

We've covered a typical implementation, but there are a couple of ways that it can vary based on differing answers to a few core questions:

How is the service located?

- **Outside code registers it:**

  This is the mechanism our sample code uses to locate the service, and it's the most common design I see in games:

  - *It's fast and simple.* The `getAudio()` function simply returns a pointer. It will often get inlined by the compiler, so we get a nice abstraction layer at almost no performance cost.

- *We control how the provider is constructed.* Consider a service for accessing the game's controllers. We have two concrete providers: one for regular games and one for playing online. The online provider passes controller input over the network so that, to the rest of the game, remote players appear to be using local controllers.

  To make this work, the online concrete provider needs to know the IP address of the other remote player. If the locator itself was constructing the object, how would it know what to pass in? The `Locator` class doesn't know anything about online at all, much less some other user's IP address.

  Externally registered providers dodge the problem. Instead of the locator constructing the class, the game's networking code instantiates the online-specific service provider, passing in the IP address it needs. Then it gives that to the locator, who knows only about the service's abstract interface.

- *We can change the service while the game is running.* We may not use this in the final game, but it's a neat trick during development. While testing, we can swap out, for example, the audio service with the null service we talked about earlier to temporarily disable sound while the game is still running.

- *The locator depends on outside code.* This is the downside. Any code accessing the service presumes that some code somewhere has already registered it. If that initialization doesn't happen, we'll either crash or have a service mysteriously not working.

- **Bind to it at compile time:**

  The idea here is that the "location" process actually occurs at compile time using preprocessor macros. Like so:

```
class Locator
{
```

```
public:
  static Audio& getAudio() { return service_; }

private:
  #if DEBUG
    static DebugAudio service_;
  #else
    static ReleaseAudio service_;
  #endif
};
```

Locating the service like this implies a few things:

- *It's fast*. Since all of the real work is done at compile time, there's nothing left to do at runtime. The compiler will likely inline the `getAudio()` call, giving us a solution that's as fast as we could hope for.

- *You can guarantee the service is available*. Since the locator owns the service now and selects it at compile time, we can be assured that if the game compiles, we won't have to worry about the service being unavailable.

- *You can't change the service easily*. This is the major downside. Since the binding happens at build time, anytime you want to change the service, you've got to recompile and restart the game.

- **Configure it at runtime:**

Over in the khaki-clad land of enterprise business software, if you say "service locator", this is what they'll have in mind. When the service is requested, the locator does some magic at runtime to hunt down the actual implementation requested.

Reflection is a capability of some programming languages to interact with the type system at runtime. For example, we could find a class with a given name, find its constructor, and then invoke it to create an instance.

Dynamically typed languages like Lisp, Smalltalk, and Python get this by their very nature, but newer static languages like C# and Java also support it.

Typically, this means loading a configuration file that identifies the provider and then using reflection to instantiate that class at runtime. This does a few things for us:

- *We can swap out the service without recompiling*. This is a little more flexible than a compile-time-bound service, but not quite as flexible as a registered one where you can actually change the service while the game is running.

- *Non-programmers can change the service*. This is nice for when the designers want to be able to turn certain game features on and off but aren't comfortable mucking through source code. (Or, more likely, the *coders* aren't comfortable with them mucking through it.)

- *The same codebase can support multiple configurations simultaneously*. Since the location process has been moved out of the codebase entirely, we can use the same code to support multiple service configurations simultaneously.

  This is one of the reasons this model is appealing over in enterprise web-land: you can deploy a single app that works on different server setups just by changing some configs. Historically, this was less useful in games since console hardware is pretty well-standardized, but as more games target a heaping hodgepodge of mobile devices, this is becoming more relevant.

- *It's complex*. Unlike the previous solutions, this one is pretty heavyweight. You have to create some configuration system, possibly write code to load and parse a file, and generally *do some stuff* to locate the service. Time spent writing this code is time not spent on other game features.

- *Locating the service takes time*. And now the smiles really turn to frowns. Going with runtime configuration means you're burning some CPU cycles locating the service. Caching can minimize this, but that still implies that the first time you use the service, the game's got to go off and spend some time hunting it down. Game developers *hate* burning CPU cycles on something that doesn't improve the player's game experience.

What happens if the service can't be located?

- **Let the user handle it:**

  The simplest solution is to pass the buck. If the locator can't find the service, it just returns `NULL`. This implies:

  - *It lets users determine how to handle failure.* Some users may consider failing to find a service a critical error that should halt the game. Others may be able to safely ignore it and continue. If the locator can't define a blanket policy that's correct for all cases, then passing the failure down the line lets each call site decide for itself what the right response is.

  - *Users of the service must handle the failure.* Of course, the corollary to this is that each call site *must* check for failure to find the service. If almost all of them handle failure the same way, that's a lot duplicate code spread throughout the codebase. If just one of the potentially hundreds of places that use the service fails to make that check, our game is going to crash.

- **Halt the game:**

  I said that we can't *prove* that the service will always be available at compile-time, but that doesn't mean we can't *declare* that availability is part of the runtime contract of the locator. The simplest way to do this is with an assertion:

```
class Locator
{
public:
  static Audio& getAudio()
  {
    Audio* service = NULL;
    // Code here to locate service...

    assert(service != NULL);
    return *service;
  }
};
```

If the service isn't located, the game stops before any subsequent code tries to use it. The `assert()` call there doesn't solve the problem of failing to locate the service, but it does make it clear whose problem it is. By asserting here, we say, "Failing to locate a service is a bug in the locator."

> The Singleton chapter explains the `assert()` function if you've never seen it before.

So what does this do for us?

- *Users don't need to handle a missing service.* Since a single service may be used in hundreds of places, this can be a significant code saving. By declaring it the locator's job to always provide a service, we spare the users of the service from having to pick up that slack.

- *The game is going to halt if the service can't be found.* On the off chance that a service really can't be found, the game is going to halt. This is good in that it forces us to address the bug that's preventing the service from being located (likely some initialization code isn't being called when it should), but it's a real drag for everyone else who's blocked until it's fixed. With a large dev team, you can incur some painful programmer downtime when something like this breaks.

- **Return a null service:**

  We showed this refinement in our sample implementation. Using this means:

- *Users don't need to handle a missing service.* Just like the previous option, we ensure that a valid service object will always be returned, simplifying code that uses the service.

- *The game will continue if the service isn't available.* This is both a boon and a curse. It's helpful in that it lets us keep running the game even when a service isn't there. This can be really helpful on a large team when a feature we're working on may be dependent on some other system that isn't in place yet.

  The downside is that it may be harder to debug an *unintentionally* missing service. Say the game uses a service to access some data and then make a decision based on it. If we've failed to register the real service and that code gets a null service instead, the game may not behave how we want. It will take some work to trace that issue back to the fact that a service wasn't there when we thought it would be.

  We can alleviate this by having the null service print some debug output whenever it's used.

Among these options, the one I see used most frequently is simply asserting that the service will be found. By the time a game gets out the door, it's been very heavily tested, and it will likely be run on a reliable piece of hardware. The chances of a service failing to be found by then are pretty slim.

On a larger team, I encourage you to throw a null service in. It doesn't take much effort to implement, and can spare you from some downtime during development when a service isn't available. It also gives you an easy way to turn off a service if it's buggy or is just distracting you from what you're working on.

What is the scope of the service?

Up to this point, we've assumed that the locator will provide access to the service to *anyone* who wants it. While this is the typical way the pattern is used, another option is to limit access to a single class and its descendants, like so:

```
class Base
{
  // Code to locate service and set service_...

protected:
  // Derived classes can use service
  static Audio& getAudio() { return *service_; }

private:
  static Audio* service_;
};
```

With this, access to the service is restricted to classes that inherit `Base`. There are advantages either way:

- **If access is global:**
    - *It encourages the entire codebase to all use the same service.* Most services are intended to be singular. By allowing the entire codebase to have access to the same service, we can avoid random places in code instantiating their own providers because they can't get to the "real" one.
    - *We lose control over where and when the service is used.* This is the obvious cost of making something global—anything can get to it. The Singleton GoF chapter has a full cast of characters for the horror show that global scope can spawn.
- **If access is restricted to a class:**
    - *We control coupling.* This is the main advantage. By limiting a service to a branch of the inheritance tree, we can make sure systems that should be decoupled stay decoupled.
    - *It can lead to duplicate effort.* The potential downside is that if a couple of unrelated classes *do* need access to the service, they'll each need to have their own

reference to it. Whatever process is used to locate or register the service will have to be duplicated between those classes.

(The other option is to change the class hierarchy around to give those classes a common base class, but that's probably more trouble than it's worth.)

My general guideline is that if the service is restricted to a single domain in the game, then limit its scope to a class. For example, a service for getting access to the network can probably be limited to online classes. Services that get used more widely like logging should be global.

See Also

- The Service Locator pattern is a sibling to Singleton GoF in many ways, so it's worth looking at both to see which is most appropriate for your needs.
- The Unity framework uses this pattern in concert with the Component → pattern in its `GetComponent()` method.
- Microsoft's XNA framework for game development has this pattern built into its core `Game` class. Each instance has a `GameServices` object that can be used to register and locate services of any type.

← Previous Chapter ≡ About The Book § Contents Next Chapter →