



» THE INNER PRODUCT

WRITING REUSABLE CODE

Observations From The Trenches

AS PROGRAMMERS, WE'RE CONSTANTLY reusing code. Sometimes it's in the form of low-level OS function calls or game middleware, and sometimes it's code that our teammates wrote. Unless you're writing top-level game script code, chances are the code you're writing will be used by other humans at some point in its lifetime.

I'm purposefully not labeling reusable code "libraries" or "middleware." Those are just two of the many forms of code we end up reusing. Copying some source files onto your project, calling an API function, and instantiating a class written by someone else on your team are all different forms of code reuse.

IMPLEMENT FIRST, ABSTRACT LATER

Without a doubt, the most difficult part of writing reusable code is making sure it solves a problem correctly and meets everybody's needs. That's not an easy task when we're writing code for ourselves, so it's even more difficult when we're doing it for other people. Too many libraries get it only half right, and they solve some problems while introducing a bunch of new ones. Or they force the user to jump through all sorts of hoops to get the desired result.

We need a clear understanding of the exact problem we're trying to solve with our code.

Libraries often fall into the trap of presenting an implementation-centric interface. That is, their interface is

based on the implementation details of the library, rather than how developers are going to use it in their programs.

The best way I've found to address both those shortcomings is to start by implementing code that solves the problem for one person—just one! Forget about multiple users and code reusability for now. If you don't have immediate and constant access to that one person, you need to play that role and create a game or application that is as close as possible to what one of your users is going to be developing.

By using this approach, I find that the interface of the code developed is much more natural because it's based on the experience of having solved the problem at least once. Otherwise, you run the risk of creating an interface that is not a good fit and forcing everything to conform to it in unnatural ways.

Once you have implemented a solution, take a moment to think before you dive into doing any more work. Many times you'll find there's no reason to abstract it any further since your code is only going to be used in one place. If that's the case, step away from the keyboard and go do something more productive. You can always come back later when there's a real need to reuse it later in the project or in a future game.

There are exceptions to this approach of implementing a solution first and abstracting it later. Some problems are very simple or very well understood, so we might be able to jump in and implement the reusable solution directly (for example, an optimized search algorithm, or a compression function). It's also possible you've implemented a similar system several times before, meaning you know exactly at what kind of level to expose the interface and how things should look. In that case, it's perfectly valid to draw on your past experience. Just try to avoid the second-

system effect: the tendency to follow up a successful and simple first system with an overly complex one with all the ideas that didn't make it into the first.

SETTING GOALS

Most successful reusable code is created with specific goals about how it is meant to be used. Sometimes those goals are explicitly stated, though most of the time they're implied in the code design.

Some of the most common goals are flexibility, protection, simplicity, robustness, and performance. Obviously, a game programmer cannot meet all those goals at once. Even if you could, you probably shouldn't try. It would be a tremendous waste of time and resources. Stop thinking in the abstract, and start thinking about your one user. What does she need? What is the most important goal for her?

Many APIs and middleware packages are designed with protection as one of the primary goals. They don't want the user to accidentally do anything wrong. In itself, it's not a bad goal, and it can often be implemented by having clean and unambiguous interfaces, clearly named types and functions, and strongly typed data types.

Unfortunately, a design with protection as a main goal can often result in encumbered interfaces, slow performance, and inflexible and verbose code. There is nothing more frustrating than wanting to do something that is explicitly being protected against and having to work around the interface.

If your target users are professional game developers, give them the benefit of the doubt and don't try to overprotect all your code. Save that for the scripting API exposed to junior designers and released to the customers with the game. If you're concerned about programmers using your code correctly and not making mistakes, provide good sample code,

tests, and documentation. If that's not enough and you feel that everyone would benefit from some level of protection, try to keep it to a minimum and maybe even provide lower-level functions that bypass it for power users.

Whatever the primary goal, the libraries

and APIs I prefer to work with help me get whatever I need done, while staying out of the way as much as possible.

ARCHITECTURE

There are many different ways to architect code that is intended for reuse. The best approach will depend on the particular code: How complex is it? How much of it is there? What are its goals?

Unless your goals are to make quick throwaway applications, I strongly recommend against using a framework type of architecture [see Figure 1]. A framework is a system that allows you to create your program by adding a few bits of custom functionality. Framework architectures seem like they will provide a clean and easy way to use complex code, but they are inevitably very restrictive, making it difficult, if not impossible, to do things with them beyond what they were intended to.

A more flexible approach is to use a layered architecture [see Figure 2]. Each layer is relatively simple and provides a well-defined set of functionality. Higher-level layers build on top of lower-level ones to create more complex or more specific functionality.

Keep in mind that it's not necessary or even desirable to have higher-level layers completely abstract out and hide the lower-level ones. By letting layers be fully transparent, they allow you to mix and match the level at which you want to access the code. This can be very important, especially toward the end of a game's development, when the team is fixing some bugs or trying to squeeze some more performance out of the engine.

Here's an example of how layered architecture can work. One layer can expose functionality to create and manipulate pathfinding networks and nodes. Another layer can implement searches and other queries on those networks. A third, higher-level layer, can expose functions to reason on the state of the network.

A toolkit architecture [see Figure 3] is the most flexible of all. It provides small, well-defined modules or functions that have very few dependencies on

other modules. This allows users to pull whatever modules they need into their games to meet their needs and nothing else. Users can also start by reusing some modules, with the intention of replacing them down the line when they want to go beyond the existing functionality. Needless to say, toolkit architectures are particularly well suited to game development.

OBJECT ORIENTED

I tend to avoid complex class hierarchies in most of my code, but it's especially important to steer clear of them when writing reusable code. The main drawback of class hierarchies is that they impose a very rigid structure on the code.

If you want to remain object-oriented, a better approach is to emphasize the composition of objects instead of inheritance. It allows users to more easily pull in the functionality they need and create their own objects based on their own constraints.

You can even go a step further and provide purely procedural interfaces: plain static functions that operate on data types. Interfaces based on static functions are often much easier to grasp than interfaces that involve classes and inheritance. They are also much more convenient for users to wrap and use in many different ways.

Do you remember what you learned about object-oriented design and having private data? Forget it and keep everything accessible! You may think you're doing the user a favor by making some variables private and reducing the complexity of the interface—and that's partly true—but eventually, your users will want to have access to some of those variables that you took pains to hide. And if they really want to, they will get to them, even if it means direct addressing into an object or vtable.

It's true that large code bases can be intimidating, and exposing all the internal details along with the regular interface would make it unwieldy for new users. An effective solution to this problem is to separate the public interface from what is intended for internal use only, while still making it available through some other

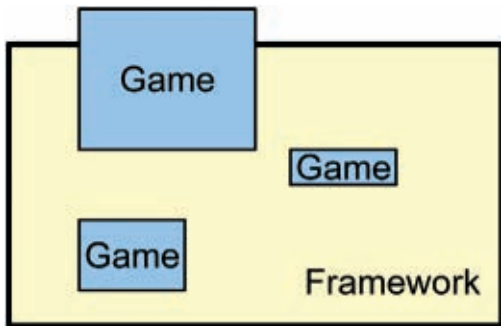


FIGURE 1 Framework Architecture

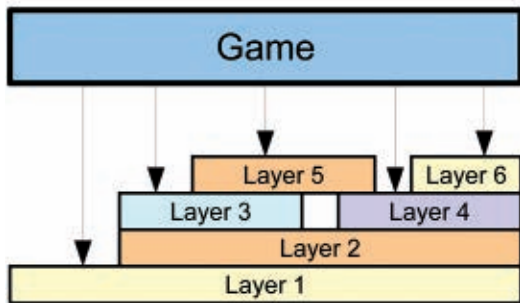


FIGURE 2 Layered Architecture

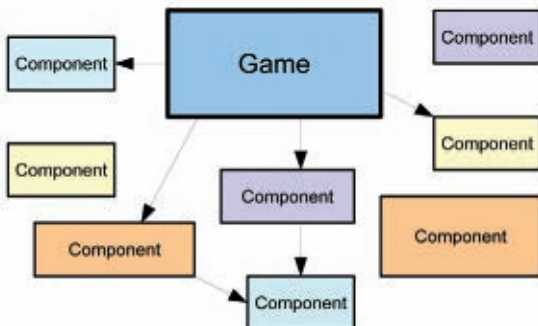


FIGURE 3 Toolkit Architecture

means. For example, private data and functions could be wrapped in a different namespace or simply in a different set of headers. Any method works as long as the interface is clearly separated, but experienced developers can still get to the implementation details to get their hands dirty with them.

EXTENSIBILITY

As soon as you make your code available to a wide range of developers, you'll find that people want to use it in progressively more complex and bizarre situations. Your code might have completely solved the case for your first couple of users, and since it's layered and modular, it can meet a lot of different requirements, too. But eventually, some people will start taking it to extremes that you hadn't imagined.

You could add more options and more modules and more callbacks to your code, allowing programmers to hook up into almost any part of the code and replace it with their own. The problem, though, is that you've taken something that was relatively simple and made it into a large, ugly, fully customizable behemoth that tries to keep everybody happy. Doesn't that sound like a lot of APIs we know and hate? Most successful products try to meet the needs of some people completely rather than meet everybody's needs part way. That means that, for a small percentage of your users, your code is not going to meet their needs. Instead of altering your code, you should give them the means to adapt it themselves. How? Give them the source code.

Without source code, developers feel caged and constrained. They know they can't look behind interfaces, let alone modify anything in case something goes wrong (and we all know something will go wrong). The more code there is, and the more a project relies on it, the more important it is to have access to the full source code.

Many teams will refuse to use some libraries or middleware unless the full source code is available. As soon as you make source code available to your users, you immediately put them

more at ease because they feel they're in control, and you allow those with special requirements to make whatever modifications they need.

Even those developers without a need to modify the code will be able to browse it and see how certain functions are implemented. Giving developers access to the full source code encourages them to not only use the code, but also fix bugs and suggest performance improvements, so it's a win-win situation for everybody.

For extra bonus points, the source code should be accompanied by a set of tests, and the more comprehensive, the better (unit tests, functional tests, and so forth). Hopefully, you created all those tests while you were developing the code, so distributing it along with the source code shouldn't require any extra effort. But it will make a huge difference to your users. It will give them much more confidence to modify the code to suit their needs if they can see that all the tests are still passing.

UPGRADES

As soon as you release some code and you have your first user, the question of how to deal with new versions arises. There are many ways you can go about it, depending on how often you'll release new versions, and how important it is to maintain backward compatibility.

One extreme is to change the code and the interface to fit new features, changes in architecture, or for any other reason. Whenever you release a new version, users will have to choose to remain with their current version or upgrade to the latest and make whatever changes are necessary. This is a common approach in open source projects and internal company code.

The opposite extreme is to set the interface in stone, and keep it the same in every version no matter what. This can be good for users because they can get new versions without any extra work on their part, but it can be very constraining as well. It can become impossible to add new features, and it can prevent performance optimizations. This approach is more common on code that will become the foundation of many programs, like OS

libraries and low-level APIs.

A good compromise is to keep interfaces the same during minor version releases and only change them whenever a major version is released. That way, other developers only have to put in the time to upgrade to a major release if they really need the new features at that time.

Another approach is to not change existing functions or classes, but introduce new ones and slowly deprecate the old ones over time. Code continues to work, but users can take advantage of the new functions. After a few versions, you can completely drop deprecated functionality, at which point most people will have upgraded already. If you do this, make sure to label deprecated functions with a `#pragma warning` or something similar, informing developers that the feature will be phased out and that they can start thinking about upgrading to the new interface.

The easier you make the transition to the new version, the better for your users, and the more likely they will be to continue using your code. For example, you can provide scripts that parse their source code and upgrade it to match the new interfaces. That can be a bit risky, but it can be really worthwhile if you have a lot of required changes that are relatively mechanical, such as renamed functions or changed parameter order.

Is there any point to all this talk of interfaces if you made your source code available? Yes, very much so.

Most developers will look at the source code to know how things work under the hood, but they probably won't modify it. Even if they do, they know that's something they do at their own risk, so they'll be more than willing to make a few changes whenever a new version is released. Everybody else will still definitely benefit from a relatively stable interface.

REDUCE, REUSE, RECYCLE

Writing reusable code starts with solving a problem and solving it well. The rest should all fall from there, and you can pick whichever method is more appropriate for your particular code and how you want to share it. ❖