Module Systems (2) – Draft

Fin

January 23, 2022

Abstract

Previously, we had a choice about whether or not a module should be seen as a component of a codebase, or as a library/language extension. We already established a rule to follow so that choice can be made following an objective approach. That rule was the following:

1. When the module is used in a finite amount of other modules, it is a component in the codebase.
2. When the module is used in a non determined amount of other modules, ie: the amount of possible modules is ”infinite”, it is an extension on the core language.

In the following text we will have a deeper look into how we should make this distinction. Do note that we’ll both use the term libraries and language extensions for the same thing.

1 Distinction between libraries and components

First of all it is easy to see that the only modules that qualify for being seen as libraries, are those that do not import other modules of the codebase. By definition, modules that depend on other modules in the codebase, are components in that codebase. We can also easily see that a module needs to be (be it in the present or at some point in the uture that the programmer anticipates for) imported in at least two other modules to have a chance of being a library. When these criteria are already met, it is still possible that the module is still a component in the codebase instead of a library. To situate the modules that meet the criteria discussed above, there are three options:

1. Put the imported module in the cluster of the importing modules if the imported modules are in fact part of the same cluster.
2. Rework the graph (and clusters) of the codebase if the importing modules are in different clusters of the codebase. This could mean that you have to sort of *merge* the different clusters in which the importing modules find themselves.
3. Make the module a ”local library”, we haven’t introduced this concept yet in the previous article, so that’s what we’ll do in the next section.

2 Local Libraries

Pondering on the module system in the state we previously had raised some questions for me. The idea of only having one global namespace for libraries didn’t sit well with me. The reason for this being, is that every sub-component of a program can be seen as a little program itself. And thus it seems like those little programs should have their own (local) library namespace as well. Perhaps it is easiest to illustrate with an example: Imagine you have a codebase for a game engine. Now that codebase is at the highest level separated in two ”sub-programs”, those being the physics engine and the rendering engine. Now you want to import libraries specifically for rendering tasks. It’s easy to see thatit doesn’t make sense that the physics engine can use these libraries, it’s completely unnecessary. And although for libraries this isn’t too big of a problem, it kind of bloats the library namespace too. So we have now established that each cluster, will at least need a namespace for their own local libraries.

Now the question becomes, are the libraries of a cluster also visible to all of that cluster's subclusters? I think the answer heavily depends. On the one hand you could say yes, because if it should only be visible in the current cluster, then the module should just be seen as a component in the program. But there's the possibility that some subclusters might need it, while other's won't. I that case I think there's nothing wrong with having the library of the cluster visible to *all* of its subclusters, instead of trying to set a constraint on which subclusters can or cannot import the library. The reason for this being that if it is used in different levels of the hierarchy and the use of the library is already scattered that much, it's obviously not a case where your codebase has been separated in completely different parts that have no logic in common whatsoever. So it is a sign that other subclusters, perhaps that are added in the future, will also benefit from the library. And in those cases where the library is clearly only needed by a definite finite amount of clusters, you can slo always copy the code and have two parts that are totally unrelated in your codebase but they do the same thing.

3 Code Reuse vs Modularity

Finally, I wanted to talk about this topic because often times these two concepts seem to be mutually exclusive. That is, if you have a very modular codebase where no module has any clue of the other surrounding modules, then you'll probably have to do the tedious tasks of reworking every module separatly, and repeating yourself multiple times in whatever fixes you make to the code. On the other hand, the pinnacle of having code reuse would be an incomprehensible web of spahetticode where as much modules are entangled as possible. The downside of this is obviously that for a programmer it becomes hard to follow what the codebase actually does, plus, the power of reworking many parts of the codebase all at once actually backfires when you want something to change at one place but not at the other.