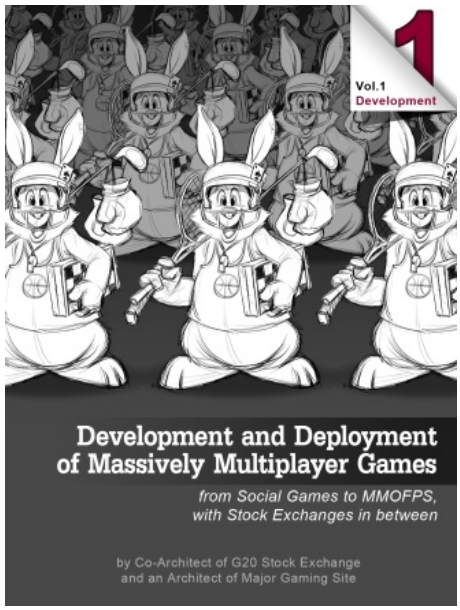
Chapter IV. DIY vs Re-Use: In Search of Balance from upcoming book "Design&Development of MMOG"

posted November 16, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko

[[*This is Chapter IV from the upcoming book "Development&Deployment of Massively Multiplayer Online Games", which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the "release" book to those who help with improving; for further details see "Book Beta Testing". All the content published during Beta Testing, is subject to change before the book is published.*

To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]



DIY
Initialism of *do it yourself*
— Wiktionary —

In any sizable development project there is always a question: "What should we do ourselves, and what we should reuse?" Way too often this question is answered as "Let's re-use whatever we can get our hands on", without understanding all the implications of re-use (and especially about the implications of improper re-use, see, for example, [NoBugs2011]). On the other hand, an opposite approach of "DIY Everything" can easily lead to the projects which cannot possibly be completed on one person's life time, which is usually "way too long" for games. In this chapter we will try to discuss this question in detail.

In the game realm the answers to “DIY vs Re-Use” question reside on a pretty wide spectrum, from “DIY pretty much nothing” to “DIY pretty much everything”. On the one end of the spectrum, there are games which are nothing more but “skins” of somebody-else’s game (in such cases, you’re usually able to re-texture and re-brand their game, but without any changes to gameplay; changes to meshes and/or sounds may be allowed or disallowed). In this case, you’re essentially counting on having better marketing than your competition (as everything else is the same for you and your competition). This approach may even bring some money, but if you’re into it, you’re probably reading the wrong book (though if you’re running your own servers, some tricks from Part [[TODO]] might still be useful and may provide some additional competitive advantage, but don’t expect miracles in this regard).

“On the one end of the spectrum, there are games which are nothing more but “skins” of somebody-else’s game. In this case, you’re essentially counting on having better marketing than your competition

On the other end of the spectrum, there are game development teams out there which try to develop pretty much everything, from their own 3D engine, their own TCP replacement, and their own channel security (using algorithms which are “much better” than TLS), to their own graphics and sounds (fortunately, cases when the developers are trying to develop their own console and their own OS are very few and far between). This approach, while may be fun to work on, may

have problems with providing results within reasonable time, so your project may easily run out of money (and as the investors understand it too, running out of money will happen sooner rather than later).

Therefore, it is necessary to find a good balance between the parts which you need to re-use, and the parts you need to implement yourself.

Business Perspective: DIY Your Added Value

First of all, let's take a look at "DIY vs Re-Use" question from the business point of view. While business perspective is not exactly the point of this book, in this case is way too intertwined with the rest of our discussion to set it aside.

From the business point of view, you should always understand what "added value" your project provides for your customers. In other words – what is that thing which differentiates you from your competition? What is the unique expertise you provide to your players?

When speaking about "DIY vs 3rd party reuse" question, it is safe to say that

At least, you should develop your Added Value yourself

The motivation behind the rule above is simple: if you're re-using everything (including gameplay, world map, and meshes), with only cosmetic differences (such as textures) then your game won't be really different from the other games which are doing the same thing. To succeed commercially, you need a distinguishing factor (sometimes 'pure luck' qualifies as such, but luck is not something you can count on).

The rule of Added Value is normally taken care of at a business level. However, even after this rule is taken into consideration, you still need to make "DIY vs reuse" decisions for those things which don't constitute the added-value-for-end-users (or at least are not *perceived* to constitute the added value at the first glance). In this regard, usually it more or less boils down to one of three approaches described below.

Engine-Centric Approach: an Absolute Dependency a.k.a. Vendor Lock-In

Probably the most common approach to game development is to pick a game engine, and to try building your game around that engine. Such game engines usually don't implement all the gameplay (instead, they *provide you with a way* to implement your own gameplay on top of the engine), so you're fine from the Added

Value point of view. For the sake of brevity, let's refer to this "3rd-party engine will do everything for us" approach as a much shorter "Engine-Centric" Approach.

"The biggest problem with building your game around 3rd-party game engine is that in this case, the game engine becomes your Absolute Dependency

The biggest problem with building your game around 3rd-party game engine is that in this case, the game engine becomes your Absolute Dependency; in other words, it means that "if the engine is discontinued, we won't be able to add new features, which will lead us to close sooner rather than later". Another way to see the very same thing, is in terms of Vendor Lock-In: as soon as you have an Absolute Dependency, you're locked in to a specific 3d engine vendor, and vice versa. Therefore, we will use terms Absolute Dependency and Vendor Lock-In interchangeably.

While by itself Absolute Dependency a.k.a. Vendor Lock-In is not a show-stopper for building around 3rd-party game engine (and indeed, there are many cases when you should do just that), you need to understand implications of this Absolute Dependency.

First of all, (as we will discuss in more detail in Chapter [[TODO]]), for "Games with Undefined Life Span" (as defined in Chapter I), the risks of having 3rd-party Absolute Dependency are much higher than for "Games with Limited Life Span". Having your game engine as a Vendor Lock-In for a limited-time project is often fine even if your choice is imperfect; having the very same Absolute Dependency "forever and ever till death do us part" is a much bigger deal, which can easily lead you to a disaster if your choice turns out to be a wrong one.

Moreover, usually, for "Games with Undefined Life Span", you shouldn't count on assumptions such as "Oh, it is a Big Company so they won't go down" (while the company might not go down, they still may drop this engine, or drop support for those-features or those-platforms you cannot survive without). While for a limited time, such risks can be estimated and are therefore manageable (in many cases, we can say with enough confidence "they will support such-and-such feature in 3 years from now"), relying on a 3rd party doing something "forever and ever" is usually too strong of an assumption.

Engine-Centric Approach: Pretty Much Inevitable for MMORPG/MMOFPS

In spite of the risks above, it should be noted that there are several MMO genres where developing a game engine yourself is rarely feasible. In particular, it applies to MMORPGs and MMOFPS. The engines for these games tend to be extremely complicated, and it will normally take much-more-time-than-you-have to develop them. Fortunately, in this field there are quite a few very decent engines with pretty good APIs separating the engine itself and your game logic. In future chapters we will keep in mind three specific game engines, and will discuss their pros and cons with relation to the issues we are raising. These engines are Unity 5, Unreal Engine 4, and CryEngine (previously known as CryEngine 3). Apologies to fans of other game engines, but I simply cannot cover all of the engines in existence; still, principles behind are usually rather similar, so you should be able to make your own judgements based on general principles outlined in this book.

"In future chapters we will keep in mind three specific game engines, and will discuss their pros and cons with relation to the issues we are raising. These engines are Unity 5, Unreal Engine 4, and CryEngine

For MMORTS the situation is much less obvious; depending on specifics of your game, there are much more options. For example, (a) you may want to use 3rd-party 3D engine like one of the above (though this will work only for relatively low number of units, you need to study very carefully engine's capabilities in this regard), (b) you may use 2D graphics (or pre-rendered 3D, see Chapter [[TODO]] for details), with your own engine, (c) you may want to develop your own 3D engine (optimized for large crowds but without features which are not necessary for you), or (d) you may even make a game which runs as 2D on some devices, and as 3D on some other devices (see Chapter [[TODO]] for further discussion of dual 2D/3D interfaces).

For all the other genres, whether to use 3rd-party engine, is a completely open question, and you will need to decide what is better for your game; often, for non-MMORPG/non-MMOFPS games, and if your game is intended to have an Undefined Life Span, it is better to develop game engine yourself than to re-use a 3rd-party game engine (even when you have your own game engine, you may use 3rd-party 3D rendering engine, or even several such 3D engines – see Chapter [[TODO]] for further details).

And if you're going to re-use a 3rd-party engine (for whatever reason), make sure to read and follow "You Still Need to Understand How It Works" section below.

Engine-Centric Approach: You Still Need to Understand How It Works

When introducing 3rd-party game engine as an Absolute Dependency, you still need to understand how the engine works under the hood. Moreover, you need to know a lot about engine-you're-about-to-choose *before* you make a decision to allow the engine Vendor to Lock you In. Otherwise, 6 months down the road you can easily end up in situation "oh, this engine *apparently* cannot implement this feature, and we absolutely need it, so we need to scrap everything and start from scratch using different game engine".

Of course, there will be tons of implementation details which you're not able to know right now. On the other hand, you should at least go through this book and see how what-you-will-need maps into what-your-engine-can-provide, aiming to:

- understand what exactly are the features you need
- make sure that your engine provides those features you need
  - if some of the features you need, are not provided by your game engine (which is almost for sure for an MMOG), at least that you should know that you can implement those "missing" features yourself on top of your game engine

While this may look time consuming, it will certainly save a lot of time down the road. While introducing Absolute Dependency may be a right thing to do for you, this is a Very Big decision, and as such, MUST NOT be taken lightly.

Engine-Centric Approach: on "Temporary" dependencies

Nothing is so permanent as a temporary government program
— Milton Friedman —

"Eliminating dependency on your game engine is pretty much hopeless without rewriting the whole thing.

If you want to use 3rd-party game engine to speed up development, and count on the approach of "we'll use this game engine for now, and when we're big and rich, we will rewrite it ourselves", you need to realize that removing such a big and fat dependency as game engine, is not realistic. Eliminating dependency on 2D engine, sound engine, or any other such engine may be possible (though requires extreme vigilance during development, see "Modular Approach: on "Temporary" dependencies" section below). On the other hand, eliminating dependency on your game engine is pretty much hopeless without rewriting the whole thing.

The latter observation is related to number of "interface points"[1] which arise when you integrate with your game engine; for a typical game engine you have lots and lots of such points. Moreover, these interface points tend to be of very different

nature (ranging from mesh file formats to API callbacks with pretty much everything else you can think of, in between). To make things worse, the better is the game engine you're using, the more perfectly legitimate uses you have for those interface points, and the more locked-in you become as a result (while having all the good reasons for doing it). Due to these factors, IMNSHO, the task of making your program game-engine-agnostic is orders of magnitude more complicated than making your program cross-platform (which is also quite an effort to start with), so think more than twice before attempting it.

---

[1] let's define an "interface point" as a point, where your program (and more generally, your whole game development process) interacts with the game engine

"Re-Use Everything in Sight" Approach: An Integration Nightmare

If you've decided not to make a 3rd-party engine your Absolute Dependency, then the second approach often comes into play. Roughly it can be described as "we need such-and-such feature, so what is the 3rd-party component/library/... we want to ~~borrow~~ re-use to implement this feature?"

Unfortunately, way too many developers out there think that this is exactly the way it should be done. (mis-)Perception along the lines of "hey, re-use is good, so there can be nothing wrong with re-use" is quite popular with developers; for managers it is "it saves on the development time" pro-reuse argument which usually hits home.

"When your code does *nothing* beyond dealing with peculiarities and outright bugs of 3rd-party libraries, *it cannot possibly*

However, in practice it is not that simple. Such "reuse everything in sight" projects way too often become an integration nightmare. As one of developers of such a project (who was responsible for writing an installer) has put it: "Our product is load of s**t, and my job is to carry it in my hands to the end-user PC, without spilling it around". As you can see, he wasn't too fond of the product (and the product didn't work too reliably either, so the product line was closed within a few years). Even worse, such "reuse everything in sight" projects were observed to become spaghetti code very quickly; moreover, from my experience, when your code does *nothing* beyond dealing with peculiarities and outright bugs of 3rd-party libraries, *it cannot possibly be anything but spaghetti*. Oh, and keep in mind that indiscriminate re-use has been observed as a source of some of the worst software bugs in the development history [NoBugs2011].

be anything but spaghetti

The problem with reusing everything you can get your hands on, can be explained as follows. With such an indiscriminate re-use, some of modules/components you are using, will be inevitably using less-than-ideal for the job; moreover, even if the component is good enough now, it may become much-less-than-ideal when[2] – the Business Requirements change. And then, given that the number of your not-so-ideal components is large enough, you find yourself in an endless loop of "hey, trying to do this with Component A has broken something-else with Component B, and fixing it in Component B has had such-and-such undesired implication in Component C, and so on…".

To make sure that managers (who're usually very fond of re-use, because of that "it saves the development time" argument), also understand the perils of indiscriminate re-use: you (as a manager) need to keep in mind that indiscriminate re-use very frequently leads to "oh, we cannot implement this incoming Business Requirement because our 3rd-party component doesn't support such-and-such feature" (which, if happens more than a few times over the life span of the project, tends to have rather bad impact on the bottom line of the company). Or describing it from a different perspective: if your developers are doing their own component, it is them who're responsible that this "we cannot implement Business Requirement" thing never happens; at the moment when you force (or allow) them to "use such and such library", you give them this excuse on a plate 😉 .

BTW, to make it perfectly clear: I'm *not* arguing that *any* re-use is evil; it is only *indiscriminate* re-use which should be avoided. What I am arguing for, is "Responsible Re-use" (a.k.a. "Modular") approach described a little bit below.

---

[2] it is indeed 'when', not 'if'! – see Chapter I

"DIY Everything": The Risk of Never-ending Story

Another approach (the one which I myself am admittedly prone to), is to write everything yourself. Ok, very few developers will write OS themselves, but for most of the other things you can usually find somebody who will be arguing that "this is the most important thing in the universe, and you simply MUST do it this way, and there is nothing which does it this way, so we MUST do it ourselves".

There are people out there arguing for rewriting TCP over UDP[3], there are people out there arguing that TLS is not good enough, so you need to use your own security protocol, there are people out there arguing for writing crypto-quality RNG based their own algorithm[4] , there are quite a few people out there writing their own in-memory databases for your game,

and there are even more people out there arguing for writing your own 3D engine.

> "There are people out there arguing for writing crypto-quality RNG using their own algorithm

Moreover, depending on your circumstances, some of these things may even make sense; however, writing *all of these things together* will lead to a product which will never be released, almost inevitably.

As a result, with all my dislike to the 3rd-party dependencies, I shall admit that we do need to re-use *something.* Now the next question is: "What exactly we should re-use, and what should we write ourselves?"

---

[3] I shall admit that I was guilty of such suggestion myself for one of the projects, though it has happened at a later stage of game development, which I'm humbly asking to consider as a mitigating circumstance

[4] once it took me several months to convince external auditor that implementing RNG *his way* is not the only "right" RNG implementation, with the conflict eventually elevated to The Top Authority on Cryptography (specifically, to Bruce Schneier)

"Responsible Re-Use" a.k.a. "Modular" Approach: Looking for Balance

As it was discussed above (I hope that I was convincing enough), there are things which you should re-use, and there are things which you shouldn't. The key, of course, is all about the question "What to Re-use and What to DIY?". While the answer to this question goes into realm of art (or black magic, if you prefer), and largely follows from the experience, there are still a few hints which may help you in making such a decision:

- Most importantly, all decisions about re-use MUST NOT be taken lightly; it means that *no clandestine re-use should be allowed,* and that all re-use decisions MUST be approved by an architect (or by consensus of senior-enough developers). Discussion on "to re-use or not to re-use" MUST include both issues related to licensing, and issues related to reuse-being-a-good-thing-in-the-long-run (you can be sure that arguments about it being a good thing in the short run *are* brought forward).
- To decide whether a specific re-use will be a good-thing-in-the-long-run, the following hints may help:
  - "glue" code is almost universally DIY code; while it is unlikely that you will have any doubts about it, for the sake of completeness I'm still mentioning it here

- if writing your own code will provide some Added Value (which is visible in the player terms), it is a really good candidate for DIY. And even if it doesn't touch gameplay, it can still provide Added Value.
  One example: if your own communication library will provide properties which lead to better user-observable connectivity (than the one currently used by competition), it does provide Added Value (or a competitive advantage, if you prefer), and therefore may easily qualify for DIY (of course, development costs still need to be taken into account, but at least the idea shouldn't be thrown away outright). In another practical example, if you're considering re-using Windows dialogs (or MFC), and your own library provides a way to implement i18n without the need for translators to edit graphics (!) for each-and-every dialog in existence – it normally qualifies as an "Added Value" (at least compared to MFC, let's postpone further discussion about i18n until Chapter [[TODO]]).

"If writing your own code will provide some Added Value (which is visible in the player terms), it is a really good candidate for DIY

- If you're about to re-use something with a very well defined interface (API/messages/etc.), *and* where the interface does what you want *and* is not likely to change significantly in the future – it is a really good candidate for re-use. Examples include TLS, JPEG library, TCP, and so on.
- If you're about to re-use something which has much more non-trivial logic inside than it exposes APIs outside – it *might* be a good candidate for re-use. One such example is 3D engines (unless you're sure you can make them significantly better than the existing ones, see the item on Added Value above). It is usually a good idea, however, to have your own isolation layer around such things, to avoid them becoming an Absolute Dependency. Such an isolation layer should be usually written in a manner described in [[TODO]] section below (as described there, dependencies *are* sneaky, so you need to be vigilant to avoid them).
- If you're about to re-use something for the client side (or for non-controlled environment in general), *and* it uses a DLL-residing-in-system-folder (i.e. even if it is a part of your installer, it is installed in a place, which is well-known and can be overwritten by some other installer) – double-check that you cannot make this DLL/component private[5], otherwise seriously consider DIY. This also applies to re-use of components, including Windows-provided components.
  The reason for this rather unusual (but still *very* important in practice) recommendation is the following. It has been observed for real-world apps with install base in millions, that reliance on something-which-you-

don't-really-control introduces a pretty nasty dependency, with such dependencies failing for some (though usually small) percentage of your players. If you have 10 such dependencies each of which fails for mere 1% of your users – you're losing about 1-(0.99[10])~=9% of your player base (plus also people will complain about your game not working, increasing your actual losses many-fold).
Real-world horror stories in this regard include such things as:

"Don't think that such failures "are not your problem" - from the end-user perspective, it is *your* program which crashes, so it is *you* who they will blame for the crash

- program which used IE to render not really necessary animation, failing with one specific version of IE on player's computer
- some Win32 function (the one which isn't really necessary and is therefore rarely used) was used just to avoid parsing .BMP file, only to be found failing on a certain brand of laptops due to faulty video drivers[6]
- some [censored] developer of a 4th party app replaced stock mfc42.dll with their own "improved" version causing quite a few applications to fail (ok, this one has became more difficult starting from Vista or so, but it is still possible if they're persistent enough).

And don't think that such failures "are not your problem" – from end-user perspective, it is *your* program which crashes, so it is *you* who they will blame for the crash. In general, the less dependencies-on-specific-PC-configuration your client has – the better experience you will be able to provide to your players, and all the theoretical considerations of "oh, having a separate DLL of 1M in size will eat as much as 1M on HDD and about the same size of RAM *while our app is running*" are really insignificant compared to your players having better experience, especially for modern PCs with ~1T of HDD and 1G+ of RAM.

- Keep in mind that "reuse via DLLs" on the client side introduces well-defined points which are widely (ab)used by cheaters (such as bot writers); this is one more reason to avoid re-using DLLs and COM components (even if they're private). This also applies to using standard Windows controls (which are very easy to extract information from); see Chapter [[TODO]] for further discussion of these issues. Re-use via statically linked libraries is usually not affected by this problem.[7]

- If nothing of the above applies, and you're about to write yourself something which is central/critical to your game – it may be a good candidate for DIY. The more critical/central the part of your code is – the more likely related changes will be required, leading to more and more integration work, which can easily lead to the cost of integration exceeding the value provided by the borrowed code. About the same thing from a different angle: for the central/critical code you generally want to have as much control as you possibly can.
- If nothing of the above applies, and you're about to re-use something which is of limited value (or is barely connected) to your game – it may be a good candidate for re-use. The more peripheral the part of the code is – the less likely related changes will have a drastic effect on the rest of your code, so costs of the re-integration with the rest of your code in the case of changes will hopefully be relatively small.
- Personally, if in doubt, I usually prefer to DIY, and it works pretty well with the developers I usually have on my team. However, I realize that I usually work with the developers who qualify as "really really good ones" (I'm sure that most of them are within top-1%), so once again, your mileage may vary. On the other hand, if for some functionality all the considerations above are already taken into account and you're still in doubt (while being able to keep a straight face) on "DIY vs re-use" question, probably this specific decision on this specific functionality doesn't really matter too much.

"The more critical/central the part of your code is – the more likely related changes will be required, leading to more and more integration work, which can easily lead to the cost of integration exceeding the value provided by the borrowed code.

Note that as with most of the other things in real life, all the advice above should be taken with a good pinch of salt. Your specific case and argumentation may be very different; what is most important is to *avoid making decisions without thinking,* and to *take at least considerations listed above into account.*

The approach presented above, can be seen as a "Responsible Re-Use"; on the other hand, we'll refer to it quite a lot in the subsequent chapters, so for the sake of brevity, we'll usually name it as "Modular Approach" (or "Modular Architecture").

---

[5] roughly equivalent to "moving it to your own folder"

[6] why such a function has had anything to do with drivers – is anybody's guess

[7] Strictly speaking, statically linked well-known libraries can also make life of cheater a bit easier, but this effect is usually negligible compared to the Big Hole you're punching in your own code when using DLLs

Modular Approach: Examples

Here are some examples of what-to-reuse and what-not-to-reuse (YMMV really significantly) under the "Responsible Re-Use" (a.k.a. "Modular") guidelines:

- OS/Console: usually don't really have choice about it. Re-use.
- Game Engine: depends on genre, but for MMORPG/MMOFPS is pretty much inevitable (see "Engine-Centric Approach: Pretty Much Inevitable for MMORPG/MMOFPS" section above)
- TCP/TLS/JPEG/PNG/etc.: usually a really good idea to re-use. One potential (though *quite rare*!) exception is TCP, but see detailed discussion on it in Chapter [[TODO]] first. On client-side it is much better to re-use them (and pretty much everything else) as static libraries rather than as DLLs, due to the reasons outlined above
- 3D Engine: an open question; see further discussion on it in Chapter [[TODO]].
- Ever-changing shared controls such as IE HTML Control: many of them are still error-prone, buggy, and are changed a lot depending on version of IE installed on client PC. Hence, it is better to avoid re-using them if you can (replacing them with much simpler 3rd-party libraries, which usually aren't that function-rich, but are much more predictable).
- On the other hand, much simpler basic controls such as text, don't have the problem of being changed too often; still, you need to consider effects related to bot fighting as mentioned above and described in Chapter [[TODO]], so using these for critical information might be not a good idea; on the third hand 🙂 , usually you will be able to replace them later without too much hassle, so it might be ok to use them to speed things up (aiming to replace them later, when bots become a problem)
- Core logic of your game. This is where your added value is. DIY
- Something which is very peripheral to your game. This is what is not likely to cause too much havoc to replace. Re-use (as long as you can be sure what exactly you're re-using on the client side, see above about DLLs etc.)

"Still, you need to consider effects related to bot fighting, so using these for critical information might be not a good idea

Modular Approach: on "Temporary"

dependencies

If you're planning to use some module/library only temporary (to speed up first release), and re-write it later, "when we're big and rich", it might work, but you need to be aware of several major caveats on the way. First of all, you need to realize that this won't work for replacing the whole game engine (see "Engine-Centric Approach: on "Temporary" dependencies" section above).

Second, you need to be extremely vigilant when writing your code. Otherwise, when the "we're big and rich" part comes, the 3rd-party module will become so much intertwined with the rest of your code, that separating it will amount to rewriting everything from scratch (which is rarely an option for an up-and-running MMOG).

So, if you're going to pursue this approach, you should at least:

- write in Big Bold Letters in your design documents, that your dependency on Module X is only temporary, and that you plan to get rid of it later
- make your own Module MyX with it's own API. The closer your own APIs to the needs of your game – the better; dumb wrappers around the 3rd-party modules should be avoided. Your Module MyX should do what-your-specific-game-needs-to-do (and not what-3rd-party-module-is-able-to-provide). The mapping between the two API sets ("your own" one and "3rd-party" one) is what your own module should do, however trivial it may seem at first (don't worry, if your APIs are centered around your game, and not around the 3rd-party comphonent, the "meat" of your own module will grow as you develop). As Peter Wolf has aptly put it: "wrap and wrap some more".
- Use ONLY your-own-API for the rest of the code (i.e. in the code beyond your Module MyX)
- make sure that everybody on the team knows that you're NOT using API of the 3rd-party module directly
- try to prohibit APIs of the 3rd-party module in your build system
    - In C++ this can be achieved, for example, using pimpl idiom for your own module and prohibiting direct inclusion of 3rd-party header files by anybody-except-for-your-own-engine
- unless you have managed to prohibit 3rd-party APIs in your build system (see above), you should have special *periodic* reviews to ensure that nobody uses these prohibited APIs. It is much much simpler to avoid these APIs at early stages, than trying to remove them

pimpl idiom
also known as an opaque pointer, Bridge pattern, handle classes, Compiler firewall idiom, d-pointer, or Cheshire Cat, is a special case of an opaque data type, a datatype declared to be a pointer to a record or data structure of some unspecified type.
— Wikipedia —

later (which can amount to rewriting really big chunks of your code)

"Dependencies are sneaky, and it takes extreme vigilance to avoid them.

While these rules may look overly harsh and too time-consuming, practice shows that without following them you get over-95%-chance that you won't be able to replace the 3rd-party module when you need it. Dependencies are sneaky, and it takes extreme vigilance to avoid them. On the other hand, if you don't want to do these things – feel free to ignore them, just be honest to yourself and realize that Module X is one of your Absolute Dependencies forever with all the resulting implications.

Summary

TL;DR of Chapter IV:

- DON'T take "re-use vs DIY" question lightly; if you make Really Bad decisions in this regard, it can easily kill your game down the road
- Consider using Engine-Centric approach, but keep in mind that Absolute Dependency (a.k.a. Vendor Lock-In) that you're introducing. Be especially cautious when using this way for Games with Undefined Life Span (as defined in Chapter I). On the other hand, this approach is pretty much inevitable for MMOFPS/MMORPG games. If going Engine-Centric way, make sure that you understand how the engine of your choosing implements those things you need.
- If Engine-Centric doesn't work for you (for example, because there is no engine available which allows to satisfy *all* your Business Requirements), you generally should use "Responsible Re-use" a.k.a. "Modular" approach as described above. If going this way, make sure to read the list of hints listed in ""Responsible Re-use" a.k.a. "Modular" Approach: Looking for Balance" section above.

[[To Be Continued…

This concludes beta Chapter IV from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Stay tuned for beta Chapter V, "Modular Architecture: Client-Side"]]

EDIT: Chapter V(a). Modular Architecture. Client-Side. Graphics has been published.

[–] References

[NoBugs2011] 'No Bugs' Hare, "Overused Code Reuse"

Acknowledgement

Cartoons by Sergey Gordeev from Gordeev Animation Graphics, Prague.

« *Chapter III. On Cheating, P2P, and [non-]Authoritative Server*..

Due to Popular Demand: PDFs of Beta Chapters from "Develop.. »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: Code Reuse, game, multi-player