Why Libraries are better than Frameworks

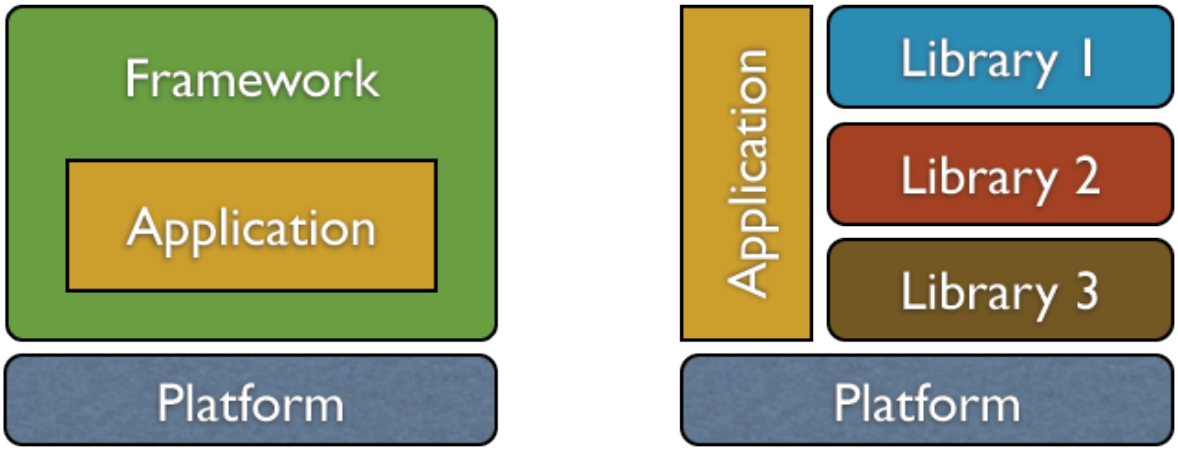
There are two ways to design an application: based a particular *framework*, or using *libraries*. Actually, there are a lot more ways to build applications, including combinations of the two, but let's just focus on those.

First, let's define the word "framework". I want to set it apart from what I call a platform. A platform is some *lowest level* on which to build an application, examples are OS's or virtual machines, e.g.: Linux, OS X, Java, or .NET. These platforms are great; They allow application developers to focus on the specific application logic, reusing all the 'plumbing' like sockets and file systems.

No, what I mean by frameworks, are the things your application (or part of your application) lives in. Examples of such a frameworks are Java's Swing, or ASP.NET MVC.

Differences between a framework and a library

There are, of course, a lot of definitions of frameworks and libraries. I don't want to get into those, rather I'd like to illustrate my view on the differences by means of this picture:



When building an application using a particular framework, the application lives *inside* the framework. This is most noticeable when the application is required to inherit from some framework class, although this is not always the case. From the viewpoint of the application, the framework is the whole world. The framework is the all-powerful environment which can do everything the application would ever want (for some particular domain, at least).

Alternatively, an application can be writing using libraries (and it's always *libraries*, plural, it's never just the one). In that design, a library is just tacked on to the side of the application. The application stands on its own, it has an identity outside of a particular framework, and it just uses libraries to do some part of the work.

Benefits of a framework

On his blog, Martin Fowler describes "Inversion of Control" as perhaps the "defining characteristic of a framework". Wikipedia also mentions inversion of control, along with default behaviour and extensibility, as one of the distinguishing features of a framework.

Inversion of control, also known as the Hollywood Principle "Don't call us, we'll call you", can indeed be a benefit. An example of this is a graphical

(web) application that renders some data. In this example, the framework is responsible for rendering the data to the end-user. Thus the application might not know how much data is required or when to compute it. By having the framework call application code, *it* can ask for data when it is needed, and only ask for as much data as can currently be displayed.

Martin describes this better than I can:

> *Inversion of Control is a key part of what makes a framework different to a library. A library is essentially a set of functions that you can call, these days usually organized into classes. Each call does some work and returns control to the client.*
>
> *A framework embodies some abstract design, with more behavior built in. In order to use it you need to insert your behavior into various places in the framework either by subclassing or by plugging in your own classes. The framework's code then calls your code at these points.*

Downsides of a framework

I've already alluded to some of the downsides I think frameworks have. First of all, frameworks require a lot from client code. In some OO frameworks the application is required to implement an interface, or worse, to inherit from a framework baseclass. An example of this is Swing, the Java GUI framework, where you're supposed to create a subclass of `JComponent`. When you do this, you can call the framework (e.g. `super.getX()`), and the framework can call you back, for example if you override the `paint` method. However, by using this design you've just giving up the only inheritance mechanism you have in Java! Say goodbye to all your nice object oriented designs.

Frameworks are also notoriously hard to replace. Precisely because the framework is the application's whole world. If you're using an encryption library you don't like, you can replace it by some other one. In frameworks, this is not the case, you can't just replace a framework. The best you can hope for is to just *add* a new encryption library, bypassing the capabilities build-in to the framework. Granted, replacing a library might be some work, since the API's for the different libraries can be quite different, but it's probably doable.

The same, hard-to-replace argument can be used against platforms, but there I think it's less applicable. An application usually is better, the more native to the platform it feels (and is). For example, a native OS X application usually works a lot better than something that barely uses OS X's capabilities. An application that uses more of a framework doesn't feel more native, it just feels more default.

The biggest problem I have with frameworks is the tight coupling between the application and the framework. The application calls the framework to do certain things, and the framework calls the application back. This tight coupling goes directly against well known software design principles, and it is the ultimate source of the problems described above.

Alternative to a framework

Ok, so that are some of the benefits and downsides to frameworks, but what is a good alternative to a framework? Well, libraries of course!

Here we come to the issue of *programming languages*. If your language is powerful enough, you can build libraries with the same capabilities as frameworks. By using **higher-order functions**, you can pass in code to the library to execute when it needs it. Some might say that such a library is now

a framework, but I wouldn't call the `map` function a whole framework. I certainly don't *live* in `map`.

So, using higher-order functions you can replace some of the capabilities of a framework in a library, but that still doesn't feel right. If your library uses *a lot* of higher-order functions, you might get the same functionalities as a framework, but now your code is all written inside-out. The solution to this issue is **lazy evaluation**.

In a lazy-by-default language like Haskell, an application can just pass in a (potentially) giant data structure to a rendering library. The library will only consume as much of the data structure as it needs, and no further evaluation will be done. In other words; Lazy evaluation uncouples the data producer from the data consumer. Sure there's still the need to communicate about how much data is needed, but this is taken care of by the runtime system. There doesn't need to be an intimate dialog between a framework and the application about which data is needed and when.

The first example some people give when explaining lazy evaluation is "now you can create an infinite list of prime numbers!", but who needs that? I think lazy evaluation brings something that's way more important, that is a fundamental part of software design; Lazy evaluation gives you composability.

Composability

The silver bullet in software development is *composability*. Or, if you don't belief in silver bullets, let's just agree its a step in the Right Direction. The old idea of building

software out of separate parts, like Lego blocks is still appealing.

"LEGO" photograph, by Jez Page.

However, the fact that there are so many frameworks out there shows that there is a need for something more than "blocks". I think the parts we need to build up larger and more complex applications are not shaped like rectangular blocks. The "components" of the future will make heavy use of lazy evaluation and higher-order functions to allow for better composability.

By writing libraries from the core up to make extensive use of lazy evaluation and higher-order functions, we get reusable, composable, loosely coupled components. These libraries are way more reusable than any framework, and they *will* lead to better designed applications.

Conclusion

In conclusion, I think the benefits frameworks have over traditional libraries **do not** outweigh the costs. Using sufficiently powerful programming languages, we can create better libraries that use lazy evaluation and higher-order functions.

The only downside that properly designed libraries have to frameworks is that they are *way harder to build*. A framework can basically be created by taking any old application, removing the content, putting in some hooks to call back to client code, and now you have a framework! (That's perhaps a bit of an oversimplification, but I like to beat up on frameworks ;-).) Libraries

on the other hand need to be designed, they need a good and clean API, otherwise they're unusable.

On the upside, while frameworks often need the be "learned", this is less so the case with libraries. People read whole books on how to use a particular framework, but they usually just start programming against a library.

Again, I think the costs of a framework are bigger than the benefits it has. Especially now that more and more libraries begin using lazy evaluation and higher-order functions. Particularly in languages like Haskell. The future of software components is libraries.

Or, to put this article in other words: *Haskell people, please stop building "frameworks", we're better than that!*

 Tom Lokhorst / September 23, 2010 / Architecture, Haskell

---

6 thoughts on "Why Libraries are better than Frameworks"

Dave Neuer

September 24, 2010 at 21:26

Excellent article. When can I expect the followup article, "Why Frameworks are Necessary Anyway?"

I agree with most of what you said about the advantages (particularly WRT flexibility) of libraries over frameworks, but AFAICT the majority of software development is bespoke software development for businesses. The typical

case in this environment is (too) large teams of developers of widely divergent skill, under ridiculous deadlines. In situations like that, freedom and flexibility are not necessarily good things. The key benefit of *software* frameworks in such an environment is that they provide structure: they take the place of the missing mental frameworks that you can get by with in smaller, better-organized projects.

Chris Done

September 26, 2010 at 01:17

Good post, I'm glad you wrote it. I see you and Chris Eidhof are in agreement on this topic.[1] In my recent post[2] to that list I tried to give a concrete example of where libraries do just fine, and not only fine but better because, as you say above, the libraries exist side by side with the application, so I can pick and choose what HTTP and routing library I want, what templating library I want, syntax highlighting, database, etc.

Dave, granted, frameworks take control away from the developer and enforce certain ways of doing things at the software level. Technically a library can do this too, but it mostly cares about how you use it directly, and rarely about how it uses you.

Also I think it's worth always building parts of your program with the assumption that this component will be replaced by a library some day, or separated into a library by yourself and shared with the world! E.g. a tiny Codepad library I wrote today.[3]

So yes, PLEASE, Haskell people: stop building frameworks!

[1]: http://www.haskell.org/pipermail/web-devel/2010/000085.html
[2]: http://www.haskell.org/pipermail/web-devel/2010/000445.html
[3]: http://github.com/chrisdone/amelie/blob/master/src/Web/Codepad.hs

---

Mike

September 26, 2010 at 01:52

I've been working on something which lives on a LAMP stack which is somewhere between a framework and a collection of libraries.

I've been calling it a 'site kit' because it supplies structure to get up and running, but is not restrictive.

It's also disgustingly early alpha – at least as far as generating sites go.

Take a look: http://www.yasitekit.org

Tom Lokhorst

September 26, 2010 at 11:47

Thanks for your comments Chris and Dave.

Yes I'm definitely in agreement with Chris Eidhof. A lot of thinking in this post came from discussions I've had with him. I like Johan Tibell's reply to Chris where he mentions the `text` library as a great example of a "small library".

Dave, I've haven't yet planned to write "Why Frameworks are Necessary Anyway", although this may come if I become disillusioned with

libraries ;-). However, you do make some great points. Frameworks bring some structure that libraries might lack. (The word “structure” is a bit more positive than “anti-freedom” / “anti-flexibility”.)

I think frameworks bring several levels of structure; Some of them are the technical, architectural and social levels of structure, there might be more.

On the technical level, frameworks give you a way to implement inversion of control. This is nice, as this is definitely needed, although I think higher-order functions and lazy evaluation can be a better way to implement this.

Architecturally, frameworks provide (or force) a way to set-up your application. Not all frameworks do this, but some do, and those I have issue with. The frameworks that force a tight coupling between the application and the framework might provide some value, but at a very high price.

Finally, socially, frameworks can provide structure to an application. Now I’m going to reverse my opinion from the previous paragraph, but I really like that frameworks impose this structure. When working in a team of developers you need a lot of agreements and conventions to build a working application. Human conversations are very expensive. I much rather spent them on talking about features and complex technical problems, than discussing naming conventions and directory layout. So, while some (arbitrary) imposed structure from a framework might not be the ideal one, just having the convenience of the whole team use the same directory names might be worth it.

So I think the good parts of a framework (inversion of control) can be implemented in libraries. The bad part (too tight coupling), we should not replicate. As to the social part, or the “mental framework” as you call it; The benefit of being able to find code that some team member wrote, because

she uses the same framework structure. This is useful, I don't know yet how to replicate that in the "small libraries design".

Dave Neuer

September 27, 2010 at 22:14

Tom,

Yes, you're on to part of what I was getting at w/ the social aspect of frameworks.

However, even the part that you say is bad and shouldn't be replicated, is good in some environments. E.g., corporate development in the Java language. If my team of 30-50 developers aren't stopped by the compiler from doing "new RespondToJSFLifecycleEventsButAlsoRadomlyFireZeMissilesBean()," odds are one of them is going to do it, and there aren't enough hours available in the day to catch it w/ code review.

Of course, Haskell is nicer. It makes it easier to enforce invariants, program w/ callbacks, etc. But not everyone gets to use Haskell and if it ever got popular enough where most people did, my guess is we'd still run into the same issues on the same kinds of projects. On small teams w/ good processes, you can avoid frameworks even w/ crappy languages. On large teams w/ poor processes, I suspect you need frameworks even w/ great languages.

Tom Lokhorst

September 27, 2010 at 23:30

Dave,

People will always find ways to write bad code. You can't fix bad programmers with good tools. As you say, if Haskell were used by many people, it would also be much more abused by bad programmers.

Neither incompetence nor inexperience can be completely fixed by technology. I think the biggest stab you can give it is with a language. E.g. Haskell where the type system enforces pure functions.

I guess it depends on how you look at it, maybe a framework can be even more restrictive than a type system. In that sense it would force people more in the correct direction. But I don't think the tight coupling aspect is fixing anything.

Perhaps we should be building more Domain Specific Languages. Of course you can still mess up in a DSL like CSS, but at least you can't `FireMissiles()`;

Comments are closed.