Frog Design Principles

Taken unaltered from the http://garethdavies.wordpress.com blog.

Early in Frog’s history the software was designed pretty much entirely by one person, with odd exceptions. In principle, this is an ideal situation because it creates consistency in approach and ensures that the overall direction follows a single coherent vision.

As the company has grown, however, we now have a wide range of differently skilled people working directly with clients and feeding back customer requests and ideas for additions to the software.

Of late there are an increasing number of people developing a view on exactly how these new features should be integrated into the software. The fact that this is happening is, I think, a testament to the simplicity of the Frog software. The reality, of course, is that the Frog software is so simple on the outside because we’ve gone to so much effort to make it so by doing lots of work on the inside. Going forward, allowing it to be designed by what could be described as a “committee of enthusiastic amateurs”, while great fun for those involved, would be fatal for the future of the software.

Very few people in Frog are aware of the design principles that we use in the development team. It also occurred to me that some of our customers might be interested in our processes, so I’ve decided to serialise it on this blog. These are very specific to the componentised nature of the Frog software, and may not translate into other systems, but the principles we use may be of interest.

I’ve added a category called, “Frog Software Design” to the blog, and I guess you’ll need to start reading from the bottom, or click the links at the bottom of each post to navigate to the next post.

Fluffy Stuff

How do we want people to “feel” when they are using our software? Now there’s a question you don’t very often see!

I started playing around with a friend’s ZX81 and Sinclair Spectrum when I was about 10 years old. Seeing an aptitude my parents bought me a BBC Micro for my 11th birthday. I learned how to program by typing game code in from magazines and then spending hours trying to get them to work. After a while I moved on to inventing my own ideas for games and then building them.

I remember the day that I knew I would work with computers when I grew up. I was 12 years old, sitting at the breakfast bar in the kitchen, hunched over the BBC and scribbling down notes and ideas for a game, when I realised, “Wow, this is great. I can make this thing do anything I want”. To be able to conceive of something in one minute and then invent it from scratch the next is
something that I don’t imagine you get to experience in many walks of life. I remember feeling empowered, a sense of pride and achievement, and of course, a sense of focus in my life.

Part of the Frog journey has been answering the question, “Is it possible to build software that makes other people feel like this, without them having to invest months or years learning how to program software?”

We believe the answer is Yes. We can’t claim that everyone that comes into contact with Frog gets this, but a significant enough number of people in each organisation do, and it turns out that that’s enough to make an impact.

**Are you Serious? Emotions in Software?**

Our software is used by more than 250,000 people and counting. Many of these organisations have 1,000 or more people in them. This kind of scale immediately brings issues of culture change into the frame. It’s fairly common for a computer system to go into an organisation, get used for a short while, and then subside into occasional use by only a handful of people, before sliding off the map altogether. Most large scale computer systems fail not because they are unable to deliver against the stated business requirements, but because people either don’t want to, or are unable to engage with them.

I’m sure you can see where this is going. Let a few people play around with the system, and then add in the feelings of empowerment and achievement around the things they have created. Throw in a few compliments from their peers, “Wow, you did that?” and there is a kernel of a team that is very quickly evangelising about the system, and more importantly, taking ownership. Occasionally, organisations will deliberately plant a well known technophobe in this initial group - “well, if Henry can do it, then I can do it.” (apologies to anyone named Henry, I picked the name at random!)

Other people then start to want to get involved. Some will start with, “Could you make it do ..... for me”. Others will move straight to, “Can you show me how to do .....” Either way, the system will start to creep through the organisation, momentum building as more and more people get their own personal “Wow, I did that” moment.

This doesn’t happen by accident.

**Okay. How?**

There are a number of design metrics used at Frog. I’ll deal with these in more detail later, for now I’ll cover just one or two that illustrate the building of a sense of pride and achievement mentioned in the previous post.

The first point is perhaps obvious. If you want someone to get a sense of achievement for their creations, then they need to be able to **create** something in the first place. This means that the system must be closer related to a toolkit, or a platform than to an application. If it was just an application with set parameters on use then there is no opportunity for creation, and therefore no opportunity to feel good about what you’ve created.
We need to make people feel like they are the developers of the system, and not just the users. No one likes being told what to do, but being told how to do it is particularly upsetting, especially if it’s not in line with the way you’ve been successfully doing things for years.

Secondly, in order to achieve our lofty ambition of allowing “ordinary people to create extraordinary systems”, the platform must focus around a specific set of problems. Frog’s primary focus is web applications. Within that we have a number of sets of components, including e-learning, e-commerce, social networking, collaboration, database applications, and so on. Within each of these areas, our users should be able to build pretty much whatever they need. As an example of something outside of these areas, Frog is not particularly suited to building graphic manipulation applications, whereas other development environments, like Visual Basic, might be. By maintaining a tight focus on what your platform can build, this allows the individual components, or Lego bricks to be similarly focused.

In short, what I’m saying here is that to provide 100% flexibility we would need to start with a completely blank sheet of paper - this would require our users to be professional programmers. Instead we define boundaries about what the system can do, allowing us to code in all the typical use cases, making the system accessible to all, and not just programmers.

Is that it?
The truth is that there is a lot of evidence to support these fluffy claims, but the reality is that it wouldn’t affect how we approach our software if there wasn’t. It is critical to have a guiding set of consistent principles when designing software, without them a system becomes nothing more than a collection of disjointed features. Without a clear set of principles, the design quickly becomes about “what’s easiest for the developers” rather than “what’s easiest for the users”. These are the principles we chose when we started out, and they happen to work for our customers as well, which is great. We could have chosen a different set. The key is that we have some and we abide by them.

So that’s all there is to it? Well, yes, but it’s not as easy to do this as you might think.

There’s an open source piece of software in our market, called “Moodle” that manages to elicit the same kind of evangelical devotion from it’s users. That’s why there is a lot of excited noise from the Moodle technical crowd, because they too feel this sense of empowerment and achievement. In Moodle’s case, this level of meaningful flexibility is only available to the technical folk though, and does not extend to the ground floor users, so the system becomes part of the technical team’s domain and it struggles to get buy-in from the people at the coal face. Getting everyone to feel this is not easy.

Before we move on to the more detailed “how we do it” bits, I’ll need to give you a bit of an overview of Frog to give it some context.
Frog Structure
Under the skin, Frog is a web based development platform, built from the ground up specifically for web applications. The first tier of the platform contains:

- document storage in a typical directory structure (as per “My Computer”)
- a sophisticated user permissions system, including user profiles (what users can do), group systems and individual user rights
- state management (allows “web pages” to operate more like applications)
- web based, drag-drop content management system
- API’s, web services and developer hooks to allow the platform to be extended

Overlaying this development platform there are a large number of “Frog Bricks”. The stock built bricks include components for e-learning, document management, network management, e-commerce, collaboration, social networking, multimedia, and so on. Users and third party companies are also able to build their own Frog Bricks, typically containing interfaces to third party applications, or mini-applications built for each customers specific needs.

The secret to the Frog design is the way that these individual components are built to:

1. provide 90% of the likely uses with very little configuration, accessible by “ordinary” people (this is achieved by focusing the system around a specific set of problems, as described earlier)
2. to provide the remaining 10% of special cases through the “Advanced” tabs, allowing them to be used in ways that we never predicted (give the power users what they want - “total control”)
3. to interact with each other in a way that allows our users to put together entire systems through different combinations of Frog Bricks

In other words, these Frog Bricks are not independent islands of functionality, this is not a matter of just widgetising an application so you can change how it looks by dragging different pieces around a screen, it’s about creating a system of interconnected bricks that allows complete systems to be built in pretty much any way people want, our users are doing things with it that we never envisaged when designing the software.

This is difficult and time consuming to achieve. We’ll cover that next.
Building for an Unknown Future

Usually, when a company wants to add a new application to their system it will tend be a “bolt on”, typically adding a new menu option so their users can get to it. Where possible, at Frog, we look to not build that application at all. Instead we look at what features we would need to add to the core Frog Bricks in order to allow it to be built from these directly.

The risk with this, of course, is that you could end up just adding application specific capabilities into the Frog Bricks - adding capability so targeted towards a specific need that you don’t actually extend the capabilities of the system beyond this immediate requirement.

It’s also tempting to believe that you can invent these features in a generic way - we’re clever enough to do this, right? No, unfortunately not. It would just end up with lots of bolt on pieces of functionality slapped onto an existing component.

So how do we do it?

We’ve been learning how to do this for nearly 10 years now, and we’ve found a way to get pretty close to the target outcome, but it takes a lot of work. The process we use includes the following steps:

1. Look at it from the user’s perspective
2. Throw more problems into the mix
3. Plan a number of journeys
4. Bring it back down to earth
5. I’ll cover these one by one in the following posts.

The Users’ Perspective

We need absolute clarity on what the users are trying to achieve, and more importantly, WHY. This is standard stuff, to be honest, especially the need to understand WHY people want something (but you’d be amazed how often software developers forget this and jump straight into trying to solve a problem they don’t fully understand).

For clarity, I’m not suggesting that we allow our customers to design our software. It’s very easy to put too much confidence in the requests made by end users, “I just want a button that does this on the software. Yes, just there, on that screen. That’ll do it. Great, thanks.”

There are two very significant issues with this approach:

1. Software, under this model, ultimately becomes a system designed by a “committee of people, with no collective training in software design or any understanding of the underlying principles behind this software.” Any consistency, flexibility or maintaining of the underlying architecture’s integrity is lost. Software becomes affectionately known as “bloatware”, more and more buttons getting layered on top of each other.

2. It presumes that your users fully understand the problem, which typically they don’t, at least not in terms of how to apply technology to it. Software is too often developed to meet a
client need only to get the response, “this is great, but it doesn’t cope with ..... which I didn’t think of at the time, but if you just add some more buttons here, here and here I think that’ll sort it. Okay? Thanks.”

This isn’t the users’ fault. When you go to your local car dealer, you don’t expect him to ask you how you want your car designing - that’s left to the experts, the guys that understand NCAP safety standards, aerodynamics, ride and handling dynamics...

So, we might not be able to get all the information we need from our users, so we’ll just make a few assumptions and hope for the best? No, not ideally. The best course of action is to get fully engaged with the issue itself. We use the concept of “dry runs” regularly at Frog. What we’re doing here is a combination of role play and drawing lots of computer screens on scraps of paper - pressing the paper buttons with our fingers, with someone pretending to be each user, fulfilling each role in the process. All assumptions made are checked back with our users.

The critical part of this stage is to COMPLETELY IGNORE OUR CURRENT TECHNOLOGY, OR ANY IDEA OF HOW WE MIGHT IMPLEMENT IT. We need to start from a perfect world, otherwise the conversations quickly deteriorate back to what’s easiest for the developers. The end result of this exercise may be extremely uncomfortable for the development team, often resulting in comments like, “but that’ll take ages”, or “but that’s not possible, it won’t work”.

Some of our greatest innovations have come from the constraints we’ve applied to a problem. Without constraints it’s easy to just add a button. With severe constraints on what the end result needs to be, all sorts of exciting things start happening in the development process. Our Secure Gateway product was born of an original vision, “we need our customers to be able to access and use any software their organisation has a licence for, from home, without installing anything, just by clicking a link on a web page.” It turned out that it was possible after all ;)

The most difficult part of this step is resisting the temptation to “solve” the problem, especially for us blokes! We’re not looking for a solution at this point, we’re looking to understand the problem thoroughly, from as many different directions as possible.

**Throw more Problems into the Mix**

Because we’re not building a specific, hard coded application, we next look at how to modify our existing components to provide the kind of capabilities we might need to build this application using the toolkit. We are looking here for a variety of different approaches, creating as many options as possible - and resisting the temptation to just “solve” the problem. Now we have a few possible routes, we look for other customer requests, or ideas on how we might use this additional functionality, “What else could we do if we extended the Frog Bricks in this way? What about if we did it this way?” This more often than not leads to a Eureka moment, something along the lines of “Oh hold on, if we did it that way and then did this and this instead, we’d be able to use this to solve that request we got last week as well, but look what else we could do with it, ........”

This sounds easy, but it isn’t. There are too key areas of difficulty with this approach:
1. preventing everyone from trying to “solve” the problem as quickly as possible (I know I’ve said this 3 times now, but there’s a reason for that ;)

2. getting everyone to invest meaningful thought in the other cases that could be added to the equation (it’s very easy to say, “I can’t think of any, let’s just do this, there’s too much on at the moment”)

To get around this we try to start thinking about the design of systems a long time in advance of building them. I’m a great believer that the best ideas come in the shower, metaphorically speaking - when we’re relaxed, or when other events in our life trigger associations and ideas. Let things grow and develop naturally. Hunched over a computer banging a specification together with a deadline to meet, or running a committee meeting full of people looking for quick fixes, is not, in my experience, conducive to building strong software (but this is how it’s usually done).

**Plan a Number of Journey’s**
The next stage is to take these options, get a group of people together that understand the system inside out (preferably including people from outside of the development team, although those with adequate knowledge will be few and far between). Outline the possible additions, the options, the favourites and then start dreaming! We’re looking for a number of possible journey’s that these development could go on. What other capabilities might we want to add? Where could this go if we had infinite resource and concentrated completely on it? What other cool things would we want to add to it?

There will almost never be a situation where this exercise doesn’t influence the way we choose to implement the additional capabilities. It will help us select the preferred option and shape the way it’s put together.

In principle, we are growing the scope of the project as much as we can in as many different ways as we can. The key to this step is that it forces us to think ahead and consider how our enhancements could be used in the future, reducing the chance that we’ve just invented a dead end that only adds value to the specific issue in front of us.

**Bring it Back Down to Earth**
Once we’ve grown the scope out as much as possible, the next step, obviously, is to pull it back to the immediate task in hand.

There will always be one or two little pieces of magic that are found during the scope growing process - relatively small developments that will add exponential value to the product. Put them in. No-one’s asked for them, but a lot of people are going to get a lot out of them. Put them in.

One word of warning. It’s very easy to get excited about some of the possible journey’s discovered. Resist the temptation to talk about them openly. Before you know it the customers will get wind and start asking for a release date. I’ll admit that I’m also the most guilty one...
**Not Enough Data?**
Occasionally there simply isn’t enough data to get to where we need to get to. In this case, we will take the data we have and put together a plan as big as we can that has no unfounded assumptions in it. It may not fully meet the immediate user need - it will fall short of rather than miss the target. In this case, we’ll put the technology out, let the users play with it, amass more data from this, and then try again.

It’s easy to presume that this is just a matter of taking a guess and seeing what happens, but it’s imperative that the criteria of “make no assumptions” is met. It’s easy to add capability to a software system, but try changing or removing features and users, quite rightly, will get very annoyed. Taking guesses could also create huge issues for data migration, depending on the circumstances.

Make no assumptions, and apologise to your customers for the lack of capability at the moment. They’ll thank you later.