What is a good application?

I recently applied for a position at my current corporation and one of the questions I was asked was “what is a good application?”.

I never thought of it before. Therefore, it was a really good exercise to formalize my own vision of a good software application. It was a technical position so my answer was from a developer point of view. Since the time I gave my answer, I have thought about it again and here is my vision of a good application.

A simple application

When I design applications, I always think « the simpler, the better ». Developers should only **focus on the requirements and just the requirements**. I really hate:

- over-design (with over use of design patterns)
- over-optimization
- over-development
- “flexible” architectures

Nothing beats **something simple than can easily be mastered**. Most of the time, the over-complex architectures designed for potential future needs will never be used at 100% (or even 50%) because:

- the needs have changed,
- the expectations were too high and/or too far away (who can predict what will happen in 5 years?) ,
- the budget of the project has decreased a lot,
- the corporate strategy has changed.

That’s why I prefer something simple that will require a big refactoring **if** the architecture is not enough for the future requirements.

For example, why would you need a Big Data cluster for an application that has just started and therefore deals with very little data? As a Big Data developer, most candidates I have interviewed were working on Big Data projects dealing

with a (very) few millions of data. When I asked them why they were using Big Data, some told me “because there are a lot of data” and others “because in the future the application could deal with more data”. A few years ago, I was working on a large bank and we were dealing with hundreds of millions of data with “only” relational databases. Between a mastered relational database and a buggy Hadoop cluster (buggy because of the little knowledge of people, the complexity of a distributed platform, and the evolving technology itself) the choice is simple. For example, Criteo (a retargeting ads startup) started with a Microsoft SQL server. They only switched to Hadoop during summer 2011 and they now have the biggest Hadoop cluster in Europe.

For me, more complex means:

- more expensive (because it’s more difficult to understand and therefore it takes more time)
- more bugs (again, because it’s more difficult to understand and therefore to test)
- more difficult to monitor (again, because it’s more difficult to understand)

Of course, if the future needs are 100% sure (for example, when designing the future referential of a very big corporation which will be fully used in 3 years), designing a complex architecture is worth it.

A readable code

A key of a good application is a readable code. Since an application will be written and read by many developers, it’s important that they **don’t spend too much time understanding legacy code**.

In object programming, this means having a code with **short functions** (with little cyclomatic complexity) and **classes with few responsibilities**. I’m a big fan of **loose coupling** which increases the isolation of the components and therefore helps to share the work among developers.

Another key point is the **code coherency**. On big projects, hundreds of developers modify the source code. If anyone codes as he likes, the code quickly becomes a total nightmare with many styles of codes. It’s like reading a book

written sometimes in English, sometimes in French, sometimes in German, sometimes in Russian … The development team must converge to a unique code convention and the code must be owned by anyone. I really like **self-explained code**: instead of writing tones of comments, using explicit names for functions, classes and variables is most of the time enough. Of course, it's still necessary to write documentation (I prefer to directly write it in the class and the function).

One of the best examples I have read is the Spring Framework: it has more than a million lines of code but the few parts I read (on Spring Core, Spring Batch and Spring Data) where coherent and the naming conventions were very comprehensive (but some people don't like the verbosity of the code conventions used by Spring). Moreover, the Spring documentation is awesome. One bad example is a very famous C library: libavcodec. This library is used by most applications to deal with compression, decompression and on the fly transcoding of audio and video streams. The work done by this library is very impressive but, when you're a **beginner** and try to understand it, it's a total nightmare. I spent approximately 15 hours to understand some parts of it in order to make an audio streaming server on my free time but I dropped it: there was not enough documentation and the code conventions were not explicit enough (for me) to be **quickly** understood without documentation.

A tested code

When I started working, a coworker told me about unit and integration tests and I was thinking "man, why would I need tests, most of the time my code works at the first time". To be honest, a part of me still thinks the same … But my vision has changed a lot.

When I code I always think: "is it unit testable?". It's a very good exercise because it forces me to always (when I have enough time) apply **loose coupling**. Moreover, having integration and acceptance tests allows you to **refractor** code and modify someone else's code without the fear of modifying the behavior of the application.

But ok, let's say you have **unit tests**, **integration and acceptance tests** with a very good **code coverage** and on top of that the tests are **automated** (with Jenkins for instance). But **are the tests relevant**? I'll give you two bad (and real)

examples:

- In a previous project, people where faking most of the tests just to increase the code coverage so that it looks like a very good project. They didn't have the time to write real tests and having high code coverage was mandatory.
- In another project, I made an audit of someone else's code and saw that the guy was not testing the behavior of the code but the behavior of the mocks used in its tests.

In both scenarios the code coverage was high but the application wasn't really tested.

Relevant tests are only possible if the development team really understands the client's needs and the concepts behind testing. Of course, it also requires having enough time.

An exploitable application

So great, you have a good application, but:

- Is it working well? Will it still be the case in 3 months?
- Are you being hacked? Have you been hacked?
- Are you sure?

In order to answer you need to be able to **monitor your application**.

To do so, you must have a **good logging system** with different levels of loging so that you can "switch" your production application in a more verbose mode if you really don't understand a bug.

Moreover, you need to **log what matters when a problem happens**. I've dealt many times with crashes in production and sometimes the only thing I got was "ERROR". Great, but what caused this error? What were the variables and the use case just before this crash? I had hard times understanding some root causes (and it made me want to kill the developer who made this piece of ####)

Another aspect is to write **technical and functional indicators** to monitor the

application behavior. For instance the memory usage, the number of current process, the number of customers/contracts/whatever created on daily basis… These indicators might help you to detect an abnormal behavior. Maybe the memory and CPU are overused because you're being hacked. Maybe there are too many customers created since the last installation because you have a new bug that duplicates creations. Here is an example of a bad exploitable application: On a previous project (on a big corporation), we discovered we were being hacked after 2 weeks of hacking just because the hackers became too greedy and our clustered servers collapsed under the load. With a good monitoring system, we could have detected the intrusion much faster.

All these logs and indicators needs to be stored somewhere. Again I like simple things. For a simple application, using just flat-files is enough. But, if your application has many servers, using a centralized logging solution is useful (like logstash/kibana/elastic search). I worked on a project with more than 30 servers in production and no centralized system, let me tell you that reading each log to see if everything is ok was a huge waste of time.

Customer satisfaction

I've seen many developers who were only driven by technologies. But **an application is not about technologies, it's about answering the needs of a client**.

I love technologies (especially the new and shiny ones), but sometimes I choose an "old" technology I don't like because it fits exactly the client's needs.

Moreover, **most of the time knowing the specificities of a language is useless but understanding the business domain** you're working on **is useful**. I have never understood developers who don't care about the business domains and only care about technologies (often only a few technologies). For example, on a project in a bank (the project dealing with hundreds of millions of data) that dealt with Basel II norms (which are European norms created to avoid bank bankruptcy), many developers in this project had never read an article about Basel II nor had tried to understand the basics of these norms. They could have worked on a nuclear plant or web crawler it would have been the same: they were only implementing what they were asked without understanding the use of their

developments (even a little bit).

For me this point is the most important. Even if your code is a total nightmare, if what you deliver makes the client happy, that what matters. And to do that, you need to understand the business domain of the client.

Customer satisfaction should always be the top priority.

A few words

You've just read **my vision** of a good application which is very close to the AGILE philosophy. I didn't spoke about ergonomics or usability because there are many types of applications (user-interfaces, batches, web-services …).

It's a very subjective and therefore controversial subject. So, what do **YOU** think is a good application?