Modern Software Over-Engineering Mistakes

By RDX.

Few things are guaranteed to increase all the time: Distance between stars, Entropy in the visible universe, and Fucking business requirements . Many articles say Dont over-engineer but don't say why...

Few things are guaranteed to increase all the time: Distance between stars, Entropy in the visible universe, and **Fucking business requirements** . Many articles say *Dont over-engineer* but don't say why or how. Here are 10 clear examples.

> **Important Note:** Some points below like "Don't abuse generics" are being misunderstood as "Don't use generics at all", "Don't create unnecessary wrappers" as "Don't create wrappers at all", etc. I'm only discussing over-engineering and not advocating cowboy coding.

1 1. Engineering is more clever than Business

Engineers think we're the smartest people around because we build stuff. This first mistake often makes us over-engineer. But if we plan for 100 things, Business will always come up with the 101st thing we never thought of. If we solve 1,000 problems, they will come back with 10,000 problems. We think we have everything under control — but we have no clue what's headed our way.

Figure 1: We think you have everything under control — But we have no clue whats headed our way. Image courtesy http://jamesbond.wikia.com/wiki/Casino_Royale and http://fortune.com/2015/07/06/failed-trump-businesses/ [Image Source]

> TL;DR — The House (Business) Always Wins

Tip: If you don't have time to go through the entire post, then this one point is enough.

2 2. Reusable Business Functionality

When Business throws more and more functionality (as expected), we sometimes react like this:

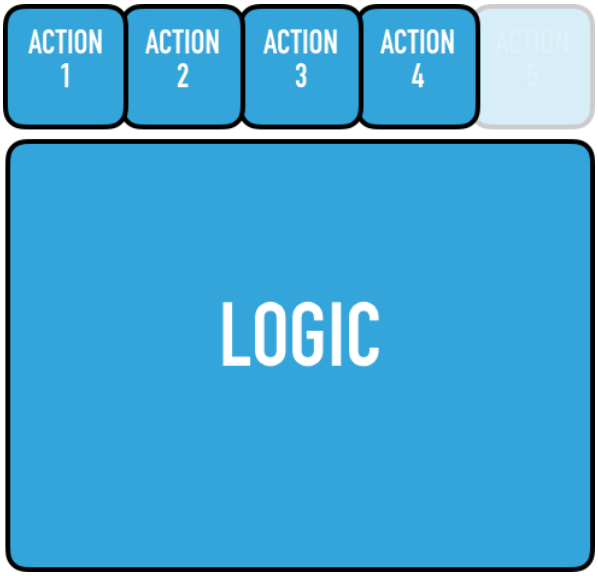

Figure 2: We try to group and generalize logic as much as possible. This is why most MVC systems end up in either Fat Models or Fat Controllers. But as we saw already, business requirements only diverge, they never converge. [Image Source]

Instead, how should we have reacted:

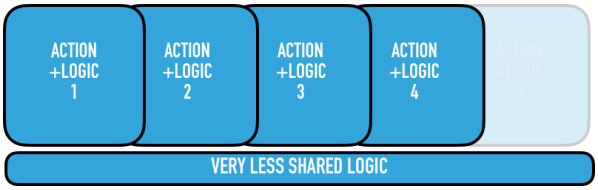

Figure 3: Shared logic and abstractions tend to stabilise over time in natural systems. They either stay flat or relatively go down as functionality gets broader. When the opposite happens, it creates systems that are Too big to fail (leading closer to the dreaded rewrite). [Image Source]

Example: We created a User profile system for a previous client. Started with a CRUD controller with shared functionality because we assumed everything

is going to be similar. But ended up with 13 different signup flows — initial social connection, a long signup form upon first entry, smaller and edit page sections, completely different looking profile page and so on and on — that it made very little sense to share stuff in the end. Similarly, an Order View and Order Edit flow ends up so inherently different from the actual Ordering flow.

Try to vertically split business functionality first before splitting horizontally. This works in all cases — isolated services, trunk-based services, language-specific modules, etc. Also helps to switch from one form to another with ease. Otherwise it becomes increasingly complicated to change parts of the system.

> TL;DR — Prefer Isolating Actions than Combining

Tip: Pick one external-facing action (Endpoint/-Page/Job/etc) in your codebase. How many context switches does someone need to understand what's going on?

3 3. Everything is Generic

(Sometimes goes together with previous point, but also seen applied individually in separate projects)

- Want to connect to a database? Lets write a Generic Adapter
- Query that database? Generic Query
- Pass it some params? Generic Params
- Build those params? Generic Builder
- Map the response? Generic Data Mapper
- Handle the user request? Generic Request
- Execute the whole thing? Generic Executor
- and so on and so on

Sometimes Engineers get carried away. Instead of trying to solve the business problem, we waste our time trying to find the perfect abstractions. The answer is so simple.

Figure 4: Whats's the perfect abstraction? Image courtesy https://www.pinterest.com/pin/415738609324811773/ [Image Source]

> TL;DR — Duplication is better than the wrong abstraction

Conversely, Duplication is sometimes essential for the right abstraction. Because only when we see many parts of the system share "similar" code, a better shared abstraction emerges. The Quality of Abstraction is in the weakest link. Duplication exposes many use cases and makes boundaries clearer.
Tip: Shared abstractions across services sometimes leads to Microservices ending up as a Distributed Monolith.

4 4. Shallow Wrappers

> This is one of the most hardest points in this whole article. Quick reminder that we're discussing over-engineering.

The practice of wrapping every external library before using it. Unfortunately most wrappers we write are Shallow. We are juggling between delivering functionality and writing a good wrapper. So our wrappers are mostly tightly bound to the underlying library (in some cases being a 1:1 mirror, or doing 1/10th of what the original library does with 10x effort). If we change the underlying library later, usages of this wrapper everywhere usually end up having to be changed as well. Sometimes we also mix up business logic inside wrappers, making it neither a good wrapper nor a good business solution, but some kind of gluey layer in between.
This is 2016. External libraries and clients have improved leaps and bounds. OSS Libraries are fantastic. They have high quality and well tested codebases written by awesome people, who have had **dedicated, focused** time writing it. Most have clear, testable, instrumentable APIs, allowing us to follow the standard pattern of Initialize — Instrument — Implement.

TL;DR — Wrappers are an exception, not the norm. Don't wrap good libraries for the sake of wrapping

Tip: Creating an "agnostic" wrapper is no laughing matter. "Swap out library" comes from a mindset of "Configurability" which is covered in detail in the "<X>-ity" section later.

5 5. Applying Quality like a Tool

Blindly applying Quality concepts (like changing all variables to "private final", writing an interface for all classes, etc) is NOT going to make code magically better.

Check Enterprise FizzBuzz (or Hello World). It has a gazillion code. In the micro-level each class follows SOLID principles, uses all sorts of great Design patterns (factory, builder, strategy, etc) and coding techniques (generics, enums, etc). It gets high Code quality ratings from CQM tools.

But if we take a step back, **this prints Fizz Buzz**.

TL;DR — Always take a step back and look at the macro picture

Conversely, automated CQM tools are good at tracking Test coverage, but can't tell whether we are testing the right thing. A benchmark tool can track performance, but can't tell whether stuff runs parallel or sequential. Only a Human has to look at the big picture.

Which takes us to...

5.1 5.1. Sandwich Layers

Lets take a concise, closely bound action and split it into 10 or 20 sandwiched layers, where none of the individual layers make any sense without the whole. Because we want to apply the concept of "Testable code", or "Single Responsibility Principle", or something.

In the Past — this was done by a chain of Inheritance. A extends B extends C extends D and so on.

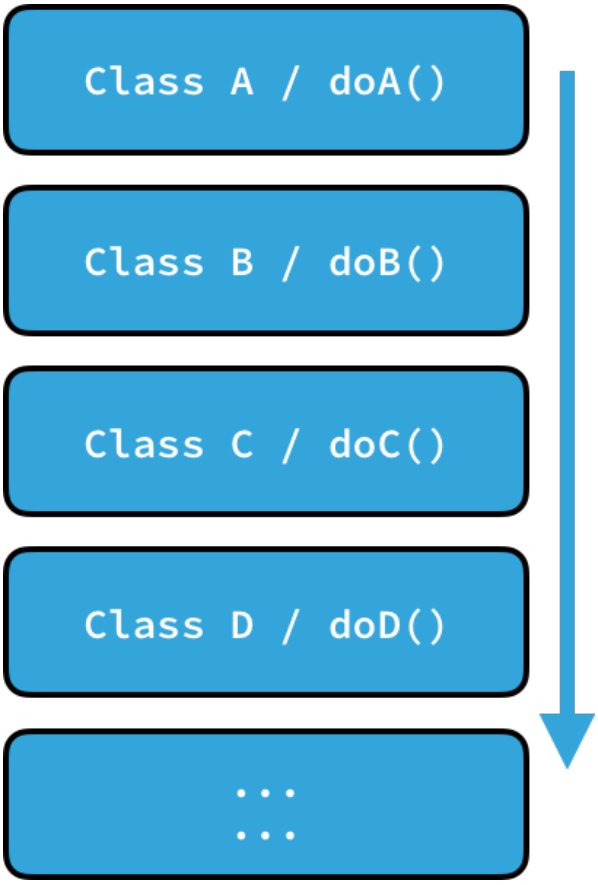

Figure 5: Now — People do the exact same thing, except they make each class have an interface/implementation and inject it into the next layer, because duh SOLID. [Image Source]

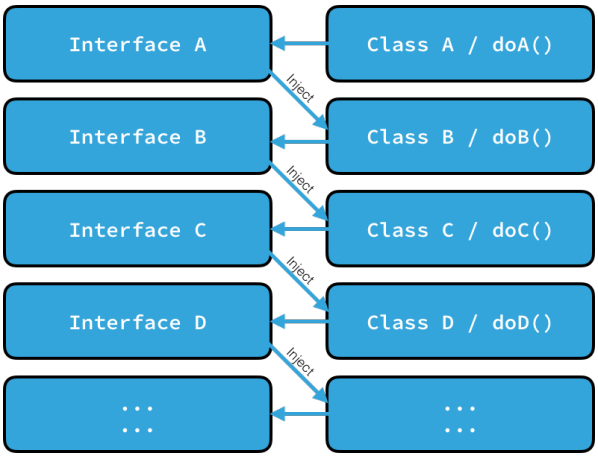

Figure 6: Concepts like SOLID came up in response to abuse of Inheritance and other OOP concepts. Most engineers are unaware of where/why these concepts came from, but just end up following the memo. [Image Source]

TL;DR — Concepts need shift in Mindset. Cannot be applied blindly like tools.

Learn a different language and try the other **mindset** of doing things. That makes a fundamentally better

developer. Pouring old wine in a new labelled bottle doesn't work for concepts. We never have to tear apart a clear design in the name of applying a concept.

6 6. Overzealous Adopter Syndrome

> Discovered Generics. Now even a simple "HelloWorldPrinter" becomes "HelloWorldPrinter<String,Writer>".

Don't use Generics when its obvious that a problem handles only specific data types, or when normal type signatures are enough.

> Discovered Strategy Pattern. Every "if" condition is now a strategy.

Why?

> Discovered how to write a DSL. Gonna use DSLs everywhere.

I don't know...

> Used Mocks. Gonna mock every single object I'm testing.

how to even...

> Metaprogramming is awesome, let me use it everywhere

describe why...

> Enums/Extension Methods/Traits/whatever are awesome, let me use it everywhere

this is wrong.

> TL;DR — TL;DRs should not be used everywhere

7 7. <X>–ity

- Configurability
- Security
- Scalability
- Maintainability
- Extensibility
- ...

Vague. Unchallenged. Hard to argue against. FUD.
Example 1: Lets build a CMS for our forms for "Extensibility". Business people can add new fields easily.
Result: Business people never used it. When they had to, they would have a developer sit right beside them and do it. Maybe all we needed was a simple Developer guide to add a new field in few hours, instead of a point-and-click interface?
Example 2: Lets design a big Database layer for easy "Configurability". We should be able to switch database in a single Magic file.
Result: In 10 years, I've seen only one business make serious effort to completely swap a fully vested database. And when it happened, the "Magic file" did not help. There was so much operational work. Incompatibilities and gaps in functionality. And the client asked us to switch "one half" of our models to the new NoSQL database. We tore our hair apart — our Magic toggle was a single point of change, but this was cross-cutting.
In today's world, we're well past a point where there is **no way** to design a single configurable layer for modern document/KV stores (e.g. Redis/ CouchDB/ DynamoDB/ etc). Not even SQL Databases like Postgres/ HSQLDB/ SQLite are compatible for that matter. Either you completely dumb down your data layer (and struggle with delivering functionality), or acknowledge the database as part of your solution (e.g. postgres geo/json features) and throw away configurability guilt. Your stack is as much part of your solution as your code. When you let go of this vague X-ity, better solutions start to emerge. e.g. You can now break data access vertically (small DAOs for each action) instead of horizontally (magic configurable layer), or even pick and choose different data stores for different functionality [micro] services style.
Example 3: We built an OAuth system for enterprise clients. For the Internal administrators — we were asked to use a secondary Google OAuth system. Because Security. If someone hacks our OAuth, business didn't want them to get access to admin credentials. Google OAuth is more secure, and who can argue against "more security" at any point?
Result: If someone really wanted to hack into our system, they don't have to go through the OAuth layer. We had many vulnerabilities lying around. e.g. they could have just done privilege escalation. So all that effort of supporting two different OAuth user profiles and systems everywhere had little to no returns in securing our system, compared to properly securing our base first.

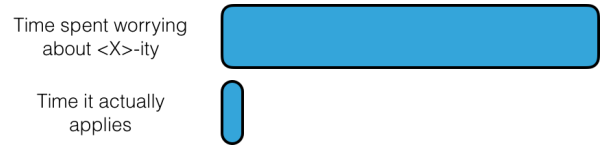

Figure 7: TL;DR — Don't let -ities go unchallenged. Clearly define and evaluate the Scenario/Story/Need/Usage. [Image Source]

Tip: Ask a simple question — "What's an example story/scenario?" — and then dig deep on that scenario.

This exposes flaws in most <X>-ities.

8 8. In House "Inventions"

It feels cool in the beginning. But these are most common sources of Legacy in few years. Some examples:

- In-house libraries (HTTP, mini ORM/ODM, Caching, Config, etc)
- In-house frameworks (CMS, Event Streaming, Concurrency, Background Jobs, etc)
- In-house tools (Buildchains, Deployment tools, etc)

Things that are missed:

- It takes a lot of skills and deep understanding of the problem domain. A "Service runner" library needs expertise of how daemons work, process management, I/O redirection, PID files and so on. A CMS is not just about rendering fields with a datatype — it has inter-field dependencies, validations, wizards, generic renderers and so on. Even a simple "retry" library is not so simple.
- There is a constant effort required to keeping this going. Even a tiny open source library takes a lot of time to maintain.
- If you open source it, nobody cares. Except the original starters and people paid to work on it.
- The original starters will eventually move away with the "Inventor of X" tag in their résumé.
- Contributing to existing frameworks takes up more time NOW. But creating an "invention" takes up considerably more time going forward.

> TL;DR — Reuse. Fork. Contribute. Reconsider.

Finally, if **really** pushed to go ahead, do it only with an Internal OSS mindset. Fight with existing competition. Work to convince even internal people to use this. Don't take it for granted since you are an insider.

9 9. Following the Status Quo

Once something is implemented in a certain way, everyone implicitly starts building on top of it. Nobody questions the status quo. Working code is considered "the right way". Even in cases where it was never intended, people go all the way around to even slightly fit into what's existing.
A Healthy System churns. An Unhealthy system is additive-only. Areas of code that don't see commits for a long time are smells. We are expected to keep every part of the system churning. Here is a wonderful article explaining this in detail.
How teams iterate vs How they should, Every single day:

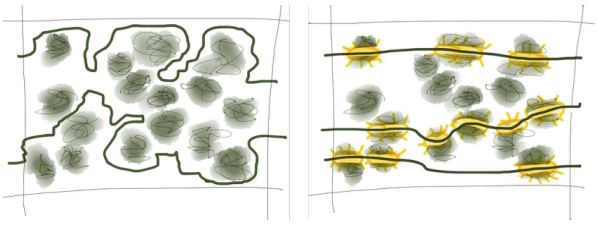

Figure 8: Image courtesy http://ronjeffries.com/xprog/articles/refactoring-not-on-the-backlog/ [Image Source]

10 10. Bad Estimation

Frequently we see really good teams/coders end up producing shit. We see their codebase and wonder "WTF, was this really developed by that team/person I thought was awesome?"
Quality needs time and not just skill. And smart developers frequently overestimate their capability. Finally they end up taking ugly hacks to finish stuff on a self-committed suicide timeline.

> TL;DR — Bad Estimation destroys Quality even before a single line of code is written

If you made it this far, thanks! Reminder that I'm only discussing over-engineering and not advocating cowboy coding. Further links:

- Is Design Dead? — Martin Fowler
- Write code that is easy to delete, not easy to extend
- Prefer duplication over the wrong abstraction (Ruby, YouTube video)
- Refactoring — Not on the backlog (and other great XP articles)
- Mindsets: XP, YAGNI, LSD, KISS

Published on 2016-07-20 23:13:57 730 in Medium / @rdsubhas
https://medium.com/@rdsubhas/10-modern-software-engineering-mistakes-bc67fbef4fc8.