Software Design 2020
April 21, 2020

City Building: Iteration 2

Community transformation

Programs must be written for people to read, and only incidentally for machines to execute.

Abelson and Sussman

Figure 1: Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.

City Building: Iteration 2

The Catamorphic Ion Transformer...the City Building Project is going well, and your crunching hours haven't been left unnoticed. We don't have enough money yet so don't you dare ask for a raise. We need that money to give the boss a well deserved annual bonus... Anyway, the product has finally taken off, and you may be the sole developer responsible for its ongoing rise to the top of the App Store. The number of users is way past what the board was expecting, so the higher-ups decided to move everything to the cloud^{TM}! That means you have to throw your beloved MVC architecture in the gutter, and transition to a Client-Server architecture. Given that our new focus is to move fast and break things (I may have stolen this motto from somewhere), we need to attract even more people, which means more features for our cherished sources of revenue.

Our citizens want to be more involved in the construction of the city ("want" is a rather strong word, we all know that we don't have free will), which is another way to say that they want to nag us whenever they find a defect. Our main priority now is to find a way to register, centralize and process all the *bugs* in the city. One professional complainer is the city mayor, who wants to search for a place before travelling there. On top of that, we haven't really been mindful about the weather, which could dramatically influence the usage of our system. Luckily, our friends at the Weather Department have already implemented an API for us, so we don't have to do anything major here. And we've now just realised that the aforementioned search bar should behave more like an omnibar, an integrated knowledge retrieval interface. *People/places/concepts* you name it, everything should be easily available through this magic input box.

Unfortunately we cannot reverse entropy yet, so the fate of the planet depends upon your input. Chop chop.

Requirements and food for thought

1. Extract these newly received requirements and merge them with all the unfinished ones that are still in your backlog. As general Eisenhower once said: "plans are useless, but planning is indispensable". These new requirements may have completely changed the course that you were planning for, and that is completely acceptable. What you should learn from all this is that you have to be able to adapt to the ever changing landscape that is software development. As the saying goes, the only thing that is constant throughout time is **change**.

2. Create[1] a[2] persona[3]. Now that we've introduced the concept of a citizen into our system, we need to be more mindful about our users[4].

[1] Personas – A Simple Introduction
[2] The origin of personas
[3] [video] Understanding Personas - An Interview with Alan Cooper
[4] The Inmates Are Running the Asylum by Alan Cooper is a required read if you want to learn more about interaction design

3. Consider the following when you get to link your application with an external API:
   (a) The request might be malformed.
   (b) One of the APIs could be down or busy for the moment.
   (c) Web APIs may refuse to serve a request due to authentication errors.
   (d) Results returned by the web APIs are malformed.

   In general, an event used for communication between contexts will not just be a simple signal, but will also contain all the data that the downstream components need to process the event.

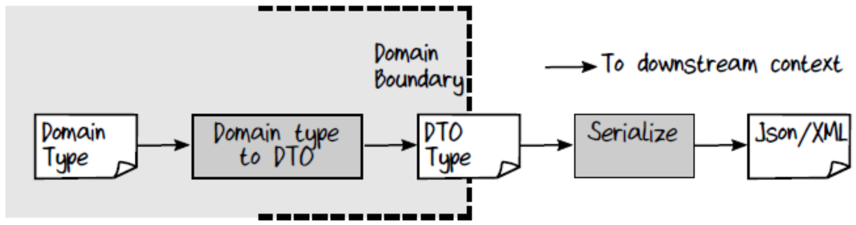

Figure 2: At the boundaries of the upstream context, the domain objects are converted into DTOs, which are in turn serialized into JSON, XML or some other serialization format.

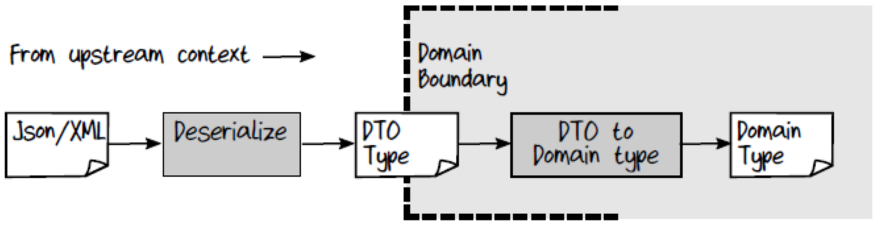

Figure 3: At the downstream context, the process is repeated in the other direction: the JSON or XML is deserialized into a DTO, which in turn is converted into a domain object.

In practice, the top-level DTOs that are serialized are typically Event DTOs, which in turn contain child DTOs[5].

Types

Types are sanity checks, little hints to the compiler to help it prove that our reasoning is correct. Programmers cannot fit the entire architecture into their head, which means that our very limited scope hinders our ability to understand the larger system. One of the most powerful problem solving techniques is to search for a suitable combination of submodules to

[5]Domain Modeling made Functional by Scott Wlaschin

reason about at a single point in time. The number of lines of code is also a big impediment in our line of work and, on average, programmers can fit approximately 10k lines of code in their head. These difficulties should nudge you towards more expressive programming languages but more importantly, towards tools that help you reason about bigger and bigger portions of the system.

> *Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.*
>
> — Alan Perlis

This is where *types* come in. These innocuous code annotations represent our way of delegating some of the cognitive load to the compiler. Moreover, it's the best technique we

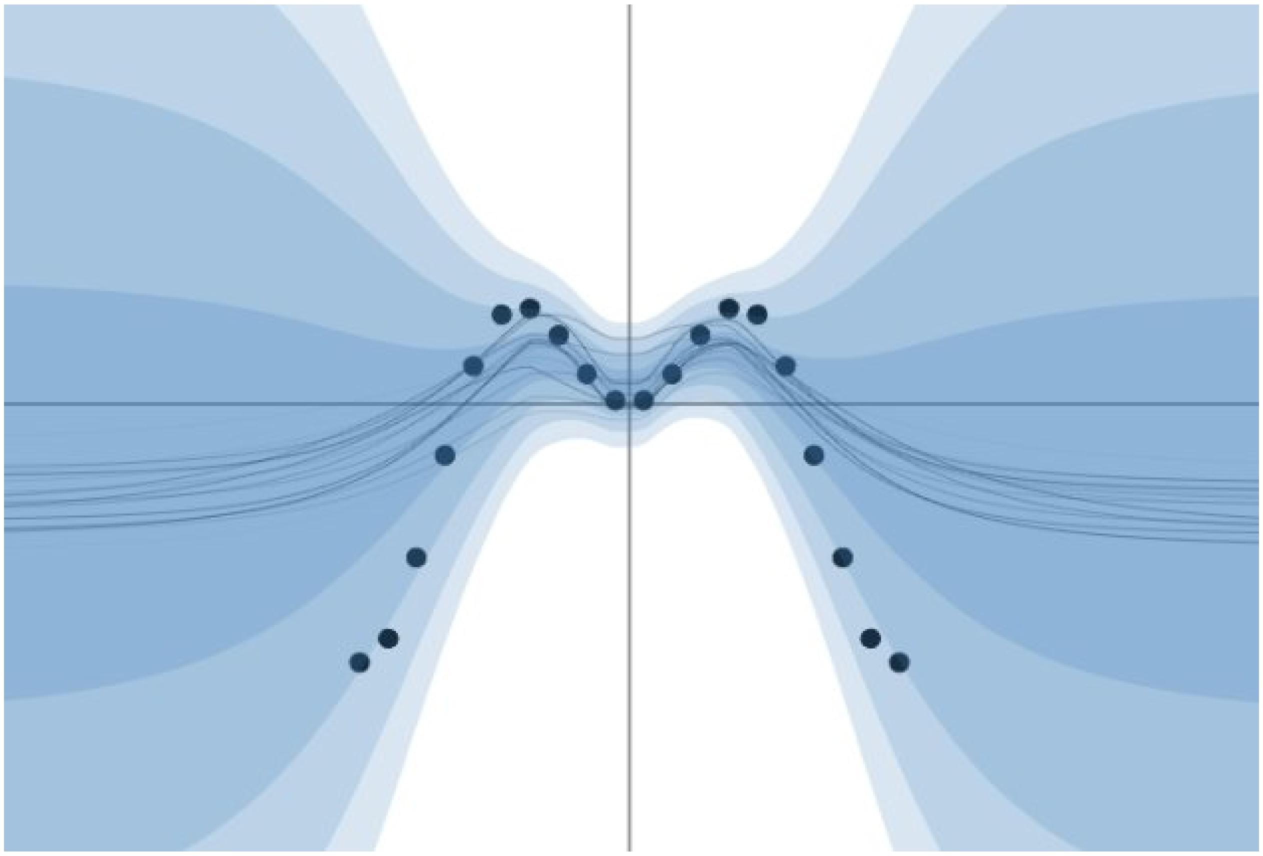

Figure 4: Consider the function with the following type: *String* → *String*. Its functional domain is virtually infinite, since it can accept everything and generate everything. There are no enforced restrictions on what it can do.

have for reducing the realm of possibility[6]. Suppose that every function in your codebase

[6]Parse, don't validate. The following images are meant to convey a visual understanding of the concept. Their actual meaning is related to uncertainty in Machine Learning.

accepts a *String* as input and generates another *String* as output, as seen in Figure 4. We're not talking about Language Oriented Programming (LOP) here, but a generic system that implements any business process you can imagine. The chosen static language is pretty much irrelevant, since you've removed its main advantage over the dynamic class of languages: the alignment of types. The domain reduction of the static checking mechanism is a bit masochistic, since we want to catch as many errors at compile time, rather than watch them manifest in production - at run time. Put another way, we want to prevent the code from compiling, by adding evermore restricting type annotations to our functions.

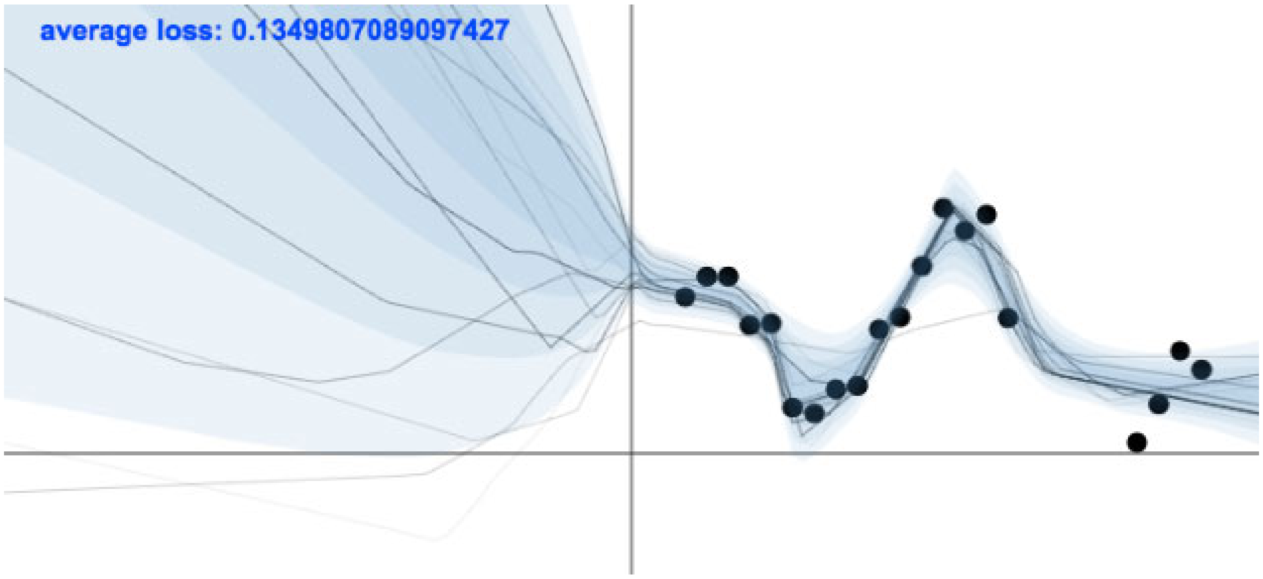

Figure 5: Consider the function with the following type: $(Int, Int) \rightarrow Int$. Suppose that this function is equivalent to the integer $\div$ operator applied on the first two arguments. If the second argument is 0, this function throws an exception (I'm ignoring the Haskell Maybe monad for the sake of this example), otherwise it returns the result.

Everything in a sufficiently large codebase should be optimized for readability and clarity. The very type of a function should nudge your thinking towards its actual behaviour, without having even read the code. There are a lot of things that the function in Figure 5 can still do, but its output domain is so restricted that the cognitive load is greatly reduced.

These concepts don't really apply to dynamically typed languages[7]. They can be enforced using conventions and community guidelines, but conventions are made to be broken. Remember, everything that your tools allow you to do will eventually end up in your codebase. That is the main reason why serious large scale systems that are designed for resilience are almost never built using dynamic languages.[8] An even further reduction of the input domain

[7]Python 3 users may want to use its rather new typing mechanism.

[8]Dropbox's use of Python 2 is a notable exception. However, they've been gradually transitioning to Go and Rust for their core infrastructure

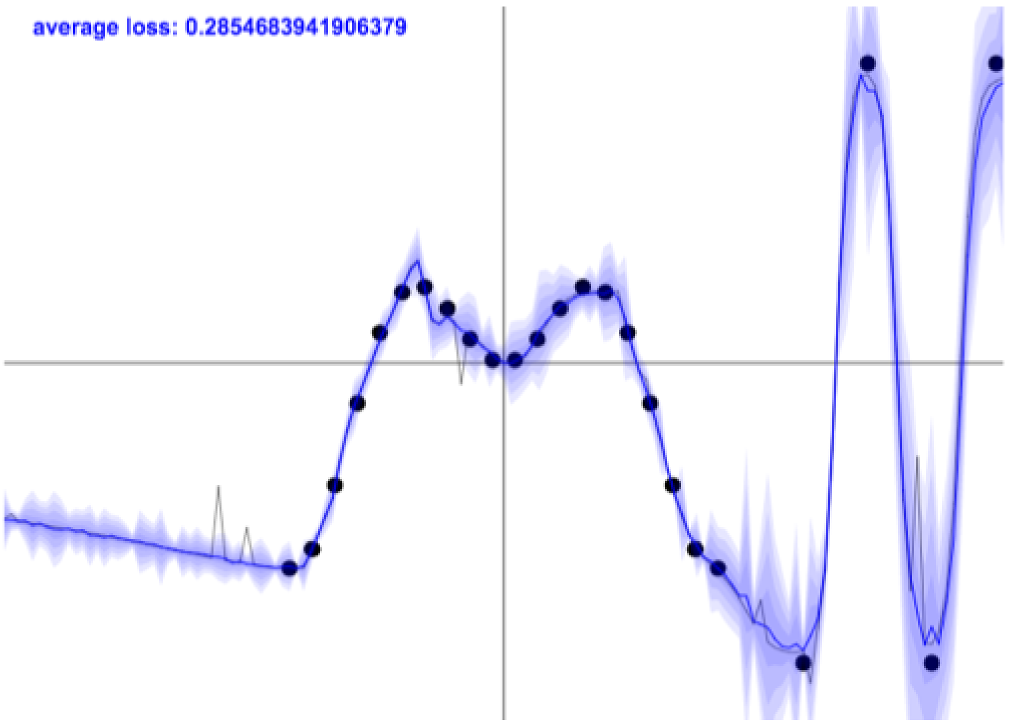

Figure 6: Consider the function with the following type: $(Int, \; NonZero \; Int) \rightarrow Int$.

results in an even smaller collection of acceptable functions that could implement such a type restriction. The stricter the types, the clearer the behaviour of the function becomes. The function from Figure 6 doesn't throw an exception, because it will never ever accept a 0 in its denominator; it's mentioned right there, in the types. If you violate this contract anywhere else in the code, the compiler will know and will fail. This very process just transformed a possible run-time exception into an unrepresentable state, something that won't ever be possible in our system. Types define the legality of the business logic. As a rule of thumb, you should keep your validation logic at the boundaries of a context. The stricter you are about this discipline, the cleaner your code becomes at the core of a context, where it generates the actual value.

This is the heart and soul of software engineering (and DDD as a footnote): mapping the infinite possibilities of the problem domain to a restricted solution space, whose boundaries are defined by its associated types (implicit or explicit).