Paradigm Design

Daniel Nikpayuk

June 11, 2016

Overview

I've been programming for about 10 years now.

At first it was just getting to know one language and how to code in general, but as I've grown and branched out and studied up, it's more recently become all about design theory. Most recently with some experience in my pocket, I've felt programming languages are missing something, but I lacked the personal language to say exactly what that was. That *something* is the content of this article. Alternatively, this essay could be titled *On best practice paradigm design for building a code library.* Such a title is more informative as to the intent here.

I'll add—it should be obvious—that I wouldn't be thinking of such things if I weren't building a library myself, so I will mention that library is in C++, the language to which I am most experienced, but I would say most of the insight offered here is generic to just about any programming language.

philosophy

Paradigms are an *engineering* thing.

Such patterns of design are intended to mitigate the complexity that comes with building large programs of code, or libraries for that matter. Paradigms such as "dispatch by type" or "object oriented programming" or "message passing" etc. They are meant to break up the code into manageable modules, which makes the system as a whole more stable as you can swap out old parts for newer ones without having to change the rest of the system. Thus we speak of making our design *modular*.

Often enough, a secondary concern of nearly the same level of importance is making these modules scalable: This is to say *extensible*. To design our modules such that if we want to add more to them later, it is not unnatural to do so. This way the weight load remains balanced, but narrative dependencies remain clean and clear as well.

The main take away if nothing from this essay is that not only do we need paradigms to make our code modular and extensible, we need to actively design our paradigms themselves to be modular and extensible. As it stands for example, it is known the "functional" and "object oriented" paradigms are in many contexts *not* naturally compatible. And yet both are highly valued to model real world applications. In the long run it is best to design the paradigms themselves to interact in cooperative or at least non-interfering ways.

This is also the content of this essay.

Design

The top programming language designers in the world seem to have a consensus that *type theory* in some variety or another is the best way to specify and implement a programming language. It's the compiler's job not only to translate your code into machine code, but also:

Prevent. Reduce. Detect. Correct.

That's the game here. You try to prevent errors in your code before it compiles. If you can't prevent during compile-time, the next best bet is to reduce errors through active auditing or otherwise. But for those bugs that still slip through or are generated through interactive use, you then need to be able to detect them during run-time, as well as correct them on the fly. You need to *fail gracefully*.

Compiler theory in general statifies its error theory: Starting with a lexer, regular expressions are used to filter out code that doesn't make the first round of cuts. Then the parser using the context-free grammar and its pushdown automata filter out code in the second round. Finally we get to the interesting semantics: Using type checking to filter out the final level of code which does not deduce. Each test increases in level of complexity by the way. All are semantics, though lexers correspond to linguistic *morphology* while parsers to *syntax*.

Before moving on, I should add that in the following examples of code, I won't be using any proper programming language, but instead some kind of psuedo code—and I might even borrow from other languages altogether such as Haskell. It'll be mix and match, but still understandable.

bottom up

Bottom up design here is just a fancy way of saying *implementation side*. Or maybe it's the other way around? I intend to present two main paradigms in this essay, the first one being implementation side, which is to say: A *constructive; prescriptive* design. We assume nothing to begin with and build up from there. This is standard engineering when it comes to mitigating complexity: define primitives; define ways to combine them; define ways to compare—and with that we're able to abstract such combinations to a higher level of complexity. Then we repeat.

highly compressible code

I will say the number one trope within my own best practice paradigms is this idea of *highly compressible* code. This phrase is problematic though. There is no such thing as highly compressible code in a universal sense, it is very much based in an *expressive* context. Take these lines of text for example:

$$\begin{array}{ccc} 1 & + & 2x \\ 1 & + & 3x \\ 1 & + & 4x \\ 1 & + & 5x \\ 1 & + & 6x \\ 1 & + & 7x \end{array}$$

You look at these lines, and you recognize a pattern. Each line is an identical string but for one term. Intuitively it seems as if these lines collectively can be highly compressed into a simpler form—presenting the same content but using less information to do so. But in reality it depends entirely on what tools of compression we have available to us. The trivial example being if we have no tools, then what you already see is exactly as compressed as it gets. On the other hand, if you have the *pattern* of recursion, you could re-express the above as follows:

$$\text{for } (k = 2;\ k \leq 7;\ {+}{+}k) \text{ print } 1 + kx;$$

In this exact example, we may have actually used just as many characters to express our string as the original, and so in reality have compressed nothing, but the attempt is still beneficial because our expression is scalable. We can not only compress the exact code above, but we can compress potentially infinitely many similar such strings.

This is a highly compressed expression representing highly compressible code. It is my first best practice. Which is to say: When you're building larger components of code out of smaller components, these smaller code snippets—maybe wrapped up in functions—should always be highly compressed. Replace all highly compressible code with highly compressed expressions, and any piece of code you need which isn't highly compressible, should be clustered and composed through highly compressed expressions. If you're going to make code that you plan to be reusable, do it well!

In contrast, sometimes in your program you turn a chunk of code into a function because it cleans up otherwise messy details, and makes relationships clearer, and that's fine and important, but usually that code isn't meant so much in the way of being *highly reusable*. This is to say: Know the difference between housekeeping encapsulation and highly reusable. Highly reusable requires highly compressed expressions. Why? Efficiency. If some code is not highly compressible it means there are a lot of exceptions to the rule of any one template pattern you're trying to use to compress it. If there are many exceptions, it means during run-time your code is effectively checking many different cases. High volume case-testing such as this becomes cumulative and computationally expensive in the long run.

domain checking

I agree with the general idea of type theory in its use of type checking to prevent errors by deduction. And yet none of the languages I have come across handle the very basic idea of function *domain* checking.

So let's say you have a function $f : A \to B$ where A and B are types. Let's give a practical example:

$$f(a, b) \quad := \quad \frac{a}{b}$$

This is to say: Division.

How do we construct the domain type of this function? As it is a/b we should have a, b as integers. As our function takes two arguments as input, we use the cross product:

$$f : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Q}$$

Easy peasy. But not. It's not quite right. If we pass the value $(1, 0)$ for example there is no rational number returned. We've divided by zero.

So how do we fix this? ...I can think of two possible ways:

The first being that we can add *control flow*, so for example:

$$f(a, b) \quad := \quad \textbf{if } (b == 0) \text{ return error; } \textbf{else } \text{return } a/b;$$

this is fine, but without proper care, it loses its scalability if you're doing this sort of graceful failure in many or all of your functions within a larger program. Why? Because every time you compose functions, it's doing these safety checks, but it's not smart enough to know if it's already done the same safety check more than once. That overhead is like a tax, a toll you have to pay for each and every bridge you try to cross. It does add up, and you do lose efficiency. It's pretty much the same idea above regarding highly compressible code: High volume case-testing becomes cumulative and computationally expensive. You still need it, it's a sunk cost, but at the same time it's also ideal to minimize it as much as possible.

As for the careful way of handling it? I've actually written about that recently in the *Semantic Constraints* essay located in this same github repo if you're interested. Otherwise, a quick summary of the idea is being able to *partition* the instances of a type into "safe" and "unsafe" and handle accordingly.

The second way is theoretically better: We add it directly to the type definition. So we instead construct our domain type as such:

$$\mathcal{D}(f) \quad := \quad \mathbb{Z} \times (\mathbb{Z} \sim \mathbf{0})$$

I've made the *zero* bold here to distinguish it between the instance of an integer (the instance *zero*) and a type which is a subtype of $\mathbb{Z}$ but which has only one instance (the instance *zero*). The $\sim$ operator is like set-theoretic difference, it's just saying we remove this zero subtype—and thus its instance—from our integer type. This way we now have encoded in the type definition itself of the domain of our division function that a denominator of value zero is unacceptable. So in some ideal programming language with this sort of type feature, when the compiler goes and does its type checking it will catch this or any other invalid type input during compile-time as well as run-time.

This second approach is ideal in that it puts more burden on the compiler and less on the coder, and because it refactors this strategy itself into the type system it would even able to optimize this additional burden.

The thing is, it is entirely impractical in existing languages.

Haskell for example, has an amazing type system. Not only does it prevent bugs, but it also feeds into the grammar itself allowing for *pattern matching*. And yet it also doesn't really allow for a great range of expression when it comes to type definitions. This is very unfortunate as it is probably the closest thing we have to an advanced type system compiler.

C++ does not work for this paradigm of coding at all. Object oriented programming interprets types as *classes* which embed both structure and function within. Every time you want to declare a variant type, you more or less have to start from scratch and reimplement everything, even if everything is almost identical to the original to begin with.

Before moving on, I would mention both approaches above (control flow and native type support) have the same underlying theme in their respective solutions: The idea of being able to partition a type (or its instance set). Any other solution generally comes down to this, as a partition is what allows us to dispatch appropriately on input.[1]

[1] You might say we need not dispatch at all with a variant type as domain, but notice the key word there is *variant*. This is to say any language privileges certain types over others to provide native support (builtin types), and any other type we then create from those. Why do you think that is? The builtin types have highest entropy, which is why they're selected over others—this is to say they are highly compressed expressions, and thus any new types introduced by the coder is usually a type of less compressive ability: You're effectively dispatching anyway, just in another manner. There's a kind of isomorphism there.

technology space

So I've identified two important design problems to mitigate when coding for which I do not see existing languages doing an adequate job. The idea of a technology space is my solution.

A *Technology Space* is something I've written about before, actually I believe I have an old intuitive essay by the same name also located in this same github repo if you're interested. Otherwise it effectively is a vehicle for interpreting the design of (ideally) any situation you come across. Basically you break things down into their *context*, *semiotics*, and *media*. I won't go into detail as to what those mean in a broader philosophical discourse, only I will explain how I interpret them here in this one.

The goal then for the technology space paradigm of coding is to add an interpretive constraint to any *type* you construct in your implementation. As stated above, this is to break up your type into a context, semiotics, and media.

What does this mean?

context:

Coding the context of a type is actually not real code, it's documentation. Good documentation is always a best practice, so why not provide native support for it in your pardigm design? Anywho, when creating a new type to add to your library, you start with the context by describing in pseudo-code or natural language legalese what the specification of your type is. What it's intuitively meant to be. Generally that's it for the context, though I would add for any effective library you're going to have to add not only an origin story for your type, but dictionary style reference documentation as well for long-term ease of use.

media:

Context always comes first, but after that we actually privilege media over semiotics even though conceptually, you can think of media being built on top of semiotics—we'll get to that.

Context is just an intuitive specification of our type, but the media space is the actual implementation. The thing thing is, the media space is reserved for what I will call *partitionable* types. As mentioned above, this paradigm is trying to solve two main problems: Domain checking, and highly compressible code. The idea of a "partitionable" type is that we have a known language of expression to partition the type instances—viewed as members of a set—in arbitrary ways which are effective in a computable sense. This is to say, we should be able to describe an arbitrary partition of our instance set, such that any determination of whether or not a given instance of our type belongs to a particular partitioned subtype is recursively computable.

So for example, the type of "finite length strings" is partitionable by this standard, as we have regular expressions to partition it in arbitrary ways, and we can test the membership of any string to a subtype recursively. Another example are the integers bounded by the bit-length of a computer processor. We can partition this type using the standard ordering relations combined with boolean logic: We can test for membership of any integer to any subtype recursively and efficiently.

semiotics:

Okay, so if the media space is where we implement partitionable types, what is the semiotic space for? It is were we implement highly compressed expressions. Intuitively, our media space is where we implement safe code for general use by all, as it should provide a nice user-interface and take care of domain checking for us. It is safe, but the trade-off is it is less compressible and thus less efficient. The semiotic space then takes care of the second problem recognized above: Highly compressible code.

In practice, the exact history of code development under this paradigm will end up being a co-evolution between the media and semiotic spaces, but pedagogically, you can think of it like this: We specify our type in the context, then we code its partitionable implementation in the media space. Once done though, we go back and look at how we can refactor this implementation. We modularize the domain checking code from the highly compressible (and thus reusable code) and factor out the reusable—efficient but usually unsafe—code and place it into the semiotic space. Then we return to the media space and reimplement the under-the-hood operations through use of the semiotics. The semiotics are our signs, our signifiers, signifieds, representations: Incomplete but useful constructs for interacting with a more complete world.

———

That is the general approach to a technology space for arbitrary programming languages, but there is still a more specific problem—an extension of the above—that arises specifically for C++:

When declaring and defining a type as an object oriented class in C++, as mentioned above, the methods—the functions—we tend to overload and define within the scope of the class itself. The problem with this is it makes our function code for those types less reusable. Why? Take the *linked list* and a *string* types for example. A string effectively is a linked list, and so some functions overlap: Some methods could be shared between the two, but given the grammatical nature of C++ classes, we would

have to reimplement and thus double-up the same code for both classes. That's inefficient.

So for C++ specifically, when we create a new type, our technology space paradigm is thus extended to split the type into its *procedural* code, as well as its *structural* code. The procedural code comes first, and you can effectively think of it as generic functions similar to that from the functional programming paradigm and functional languages. The point is, we make these reusable functions first and we make them independent of the classes they're intended to belong, and so when we implement within classes, we *call* these generic functions instead of defining them there. This way if you have two similar types that otherwise would share generic function code, they now can.

summary

To summarize, our technology space paradigm mitigates two complexity issues: Domain checking as the next iteration in error checking, as well as highly compressible code for maximum code reuse. It does so by splitting a type into:

1. its specification documentation as context;
2. its partitionable implementation as media;
3. its highly compressed implementation as semiotics.

For C++ specifically, we orthogonally but additionally split up our type into its procedural method code along with its structural type definition.

Finally, note that there is nothing contradictory about creating types which have no safe (partitionable) implementation. This is to say we can add unsafe but efficient types to our code library even if they have no safe equivalent—it just means there will be some types which have an effective semiotic space but an otherwise empty media space.

top down

The bottom up design is the more involved aspect of this essay. Bottom up design is more about *typology*: What types actually exist (and how best to implement them). Top down is more about specification side, it's more about *ontology*: What types *can* exist within our system in the first place. It's about what's possible. Boundaries of the library itself. In life in general I'm willing to believe anything is possible, but for the purpose of a library—even one which is modular and extensible—it should be expected to be more restrictive than *anything and everything*. Your goal is to mitigate complexity, not to build something equivalent to it!

So top down design is about ontology: Another way of saying this is it's more like specifying what "genres" of type we're willing to work with.

In that regard, my paradigm is much simpler: To make genres of type which will stand the test of time; which will scale; which are modular and extensible; I will categorize the types within my code library as abstract types of *hardware*. Why? *indexing*! The exact details of hardware changes all the time, but hardware types remain relatively stable, for example a screen is a screen, a mouse is a mouse, a keyboard is a keyboard.

Quite a lot of hardware exists to interface humans with computers to communicate, or computers with computers to communicate, and given the slow nature of human evolution, as well as the best practices of the science of communication (electromagnetic spectum, etc.) the specifications of these protocols remain relatively stable. Thus it's reasonable to index our ontological inflation to these stable forms. They may change at some point, but it's expected there is enough foresight and a buffer period of time to update our code to match the new realities whatever they may be.

To that end, I have created the general "genre" categories:

graphic, interic, kinetic, literic, numeric, phonetic.

So, "graphic" is for anything graphical such as the screen; "interic" is for inter-computer communication such as the internet; "kinetic" is for anything motion oriented such as the track-pad, the mouse; "literic" is for anything literal, which is to say the keyboard, and in particular character and text manipulation code—it's all about the textual world; "numeric" is for the brain of the computer, and core concepts of *memory*: So the processor, registers, cache, random access memory, file system and hard drive, etc.. Finally for the existing category, I have "phonetic" which corresponds to anything audio related like the sound card, or interacting with speakers, headphones, microphones.

Keep in mind this is not an exhaustive list, but the idea is that this genre approach is modular and extensible. I've demonstrated existing modules, and we can always add more later. In practice, in the real world, when trying to classify types within these genres, what happens if there's genre overlap?[2] You could always classify your type as both (or however many

[2]In the real world, things tend to be non-linear.

genres it belongs to), but if you want to keep things simple, you can use implementation dependencies and similar such narrative details as your tie-breaker.

Usually it's more a matter of where you would implement it first within your library for practical reasons, and so you categorize it there. If there's no way to resolve, flip a coin. The main point is that the code itself is located in only one location. Really, you could take the hard drive / filesystem paradigm: The content is uniquely located (the hard drive), but in the dictionary documentation (the filesystem; the simulation) you can hyperlink your type's page and categorize in as many genres as you'd like.

narrative design

For each of these top down genre categories, you will need to extend a narrative design. Generally if types are within the same genre of hardware, you're likely building more complex types from simpler types within that same hardware space. The reason I do not go into best practice details here is each hardware type will have its own narrative design. I will give one example though, which is more natural to C++ though not incompatible with other languages: The numeric types.

With the numeric types we start with the *bit types*. A bit type has two instances, the most natural example being the *bool* type with instances {*true, false*}. I say bit "types" in the stead of "type" because it is not unreasonable to model many applications which have binary states in the real world, and thus there would be many such bit types. On top of the bit types though, you can actually build *bit iterators*. These are names which point to bit types as well as recursively to each other, and thus allow you to link bits together and navigate them.

Building on the bit types and their iterators, the next natural step would be the *register types*. Bit iterators allow you to link bits, while with register types you actually do it.

As an aside, when implementing the unsafe version of a register type as an integer in C++ using the technology space paradigm, we would implement it in the semiotic space, but in reality it's already implemented within C++ grammar as a builtin type. As such, though conceptually we locate its implementation there, in effect we just pretend like we have (or probably alias it under the new namespace for design consistency).

This way the existing types even fit snuggly into our larger design. Notice the existence of an unsafe version also implies the existence of a safe version? The *int* type itself is clearly safe in C++, but the associated builtin functions such as addition and multiplication are not. Why? Because they don't test for arithmetic *overflow*, and so do not fail gracefully, even though they are partitionable as a type and so it's not unreasonable to do so.

Built on top of our register types would be the *register iterators*. Notice a theme in the design of these numeric types: The paradigm is self-similar, it is recursive, it is extensible. The narrative of the paradigm itself is highly compressible.

Register iterators allow one to link together many virtual registers—and though in practice it's subtlely different—we can from a *highly compressed narrative* point of view claim this is how we build our *ram types*. Granted the exact technology of "random access memory" is not the only variety out there, for example there's flash, but as a representative example of an equivalence class of types we'll use the name *ram*. Our type now is sufficiently complex it's worth creating subtypes. Obvious subtypes of the natural ram type would be vectors and linked lists, for example.

Anyway, I think you get the idea.

summary

To summarize the top down paradigm: Categorize types into hardware genres indexing them for ontological and even sociological inflation, and for each genre determine more specific modular and extensible narrative paradigms.

Conclusion

That's about it.

In reality, in practice, when building a code library, there's gonna be many more subtleties that come your way—one needs to think out good systems for handling errors (not just passing them) for example, as well being able to print your types to the screen for debugging. Streams of design such as these tend to be sown throughout the design of a larger library. For the *nik* C++ library I'm building myself, there are many paradigm details I've left out from this essay—for example the exact approach I've taken to inventorizing generic functions requires me to scale up short-forms for deeply nested namespace names. I have otherwise left out these details for the simple reason such strategies really are very very specific to C++ grammar not to mention specific to the intentions of my library.

Otherwise, if you're building your own library, regardless of language, this essay offers my best wisdom in how to do so. Thank you. Pijariiqpunga.