Genetic System Architecture

Ilya Mikhaltsou

Contents

Genetic System Architecture

What is Genetic System Architecture

What is Genetic System Architecture? Genetic System Architecture (GeSA for short, alternately Genetic Programming) is a software design architecture (paradigm) that acts around data flow, not execution flow[1]. It is akin to both Functional Programming and Object-Oriented Programming, but unlike those, it has neither a defined functional composition nor a defined class composition. Instead, the explicit decision on what function/class (in GeSA — gene) will be used is made at runtime based on intrinsic and extrinsic properties of the data, match condition specified in the gene, priority of the gene and external conditions of the system (e.g. local or remote gene, availability of computational resources, etc).

While initially it may seem convoluted, wasteful, indeterminate — there are several benefits, that in my opinion far outweigh this. Here are a couple of such benefits:

- **GeSA is agnostic to computational units** and can work as a **microservice architecture**. Beyond having implicit multi-threading and thread-safety, you can have one program split and replicated on any number of completely (hardware) unrelated computational units, till they are networked in any way. For example, once you have developed your software for PC, you can easily just cut some of the genes, and copy them over to a hundred of Raspberry Pi's (with, of course, corrections for target OS and HW architecture). They will interoperate without any additional changes, and you will instantly get all the benefits of microservices without the hassle of designing the system for microservices initially (and the less design decisions you have to make upfront is usually the better).
- **Changing and substituting components is easy**, since each gene is supposed to rely only on its data and never on the existence (or implementation) of other genes. Take this as a same core premise of OOP but even more: your genes don't even know there are other genes, what are interfaces to them, what inputs and outputs they have. So you can easily interject additional genes or replace other genes (even with specific and definite conditions!) anywhere in the data-flow without breaking a thing. Take OOP: to add some specific behaviors to a class, you need to subclass it. Then you have some indeterminate calls to superclass at occasional points where you want to fallback to other functions. You also have to use something like dependency injection or painstakingly replace each call to a new instance of this superclass with your new subclass. And what if it's

[1] As all other paradigms, it is language-agnostic and can be freely implemented on top of any other language, but having a GeSA-language will do it just as much good as existence of OOP and FP languages does for OOP and FP.

in the third-party lib? There are multitudes of design patterns to accommodate that, each adding more design complexity... If we go to FP — it is even worse and changing anything in a strictly defined composition is near impossible without major dives into the code. What to do in GeSA? You just define a new gene with higher priority and specific conditions when it should fire — the system works everything out internally and in a determinate predictable matter. You can even define a lower-priority "a-la superclass" gene — later than "subclass" genes. Even less upfront design decisions, right?

- **Debugging can be terrifically simple**. Each data point that is important for branching of any kind is exposed by design and can be captured, viewed, extracted, substituted or mocked without additional efforts from inside the genes themselves. Execution can be controlled in a manner that shows what have impacted certain decisions, and any gene can be stopped while leaving other genes to continue working (imagine in-app debugging).

When should you use it

So here are some points of when should you consider using GeSA:

- Your system is potentially complex
- Your system may contain multiple joints and interdependencies
- You want your system to be very agile from the inception and till EOL
- You can start with a minimal system and build upon it
- Initial design may differ significantly from later designs
- You may need additional computational resources eventually
- It is hard to draw distinct lines of concern-separation in your system
- You want a specific feature of GeSA for your system (this one is obvious, right?)

Also, here are some points of when it may be a bad idea to use GeSA:

- Your system is small-scaled
- Your initial system design will not change much
- Your system will suffer significantly from poor single-thread performance and/or high execution latency

And, finally, here is the most important factor-test (the first one is also a sign of a well-designed genetic system):

1. If you can split your system in such pieces that you can throw away or omit without significant impact on the operation of your system as a whole.
2. Removing any such pieces in most other architectures will inevitably create lots of dangling references and involve lots of work to remove them (or at least refactoring).

How it is made

Let's start with the two most important components: *Gene* and *Storage*.

Storage is a global data-space where Genes put theirs output and from where it is taken to other Genes. Storage is separated in named compartments (the type of naming system is not important — the language may use symbols, strings, UUIDs or whatever is best suited), compartments can be nested and can contain named data values. Multiple data values of same or different types can share the same name (but doing so too much may impact performance, unless it is intended for parallelization).

Each data value stored is *unique and non-copyable*. Its access is guaranteed to be *exclusive* for the Gene it was given to.

A Gene is defined by two parts: a *match-bind expression (MBE)* and an *executable*. When an MBE of the Gene is successfully matched with some data in the storage, the data is taken out from the storage and bound to that Gene. Right after that, the executable of that Gene will be called, which can produce some output at the end. So basically, a Gene is an object that has a condition (match part of MBE), some variables/properties (bind part of MBE) as an input, some defined transformation (executable part), and some output (return value of the executable).

> In fact, it is very similar to Actors and the only major difference from Actors is that there is no "sending of messages"; instead, messages find their target Gene.

There are also some additional components which can be added onto Genes to implement specific features. Most of them will be explored below.

A short comparison

So basically GeSA combines:

- Event-Driven programming. Genes basically react to events that happen in the Storage. The main difference is that unlike with events, there is the central Storage, and events are bound to abstract condition definitions that may not be ever present.
- Some Functional programming. Genes are supposed to have no side effects and only transform inputs. Of course, in real app this isn't necessarily the case as some actions would certainly require side effects.
- Aspect-Oriented programming. Genes and Storage can be understood as advices and a centralized join-point collection.
- Data-driven programming. Genes are the combination of input conditions and data transformation. Unlike in Data-driven programming, there is no whole-program input, and so Genes have no differentiation between external input and other Gene's outputs.

Sure, I'm not stating that I'm taking the best of those. Each has lots of downsides to it, and some of them are also relevant to GeSA.

Sort of Q&A

This looks weird

It is! Most programming paradigms are weird, till you become familiar with them. I also felt weird at first, but now I feel all the time I program not in GeSA, "this would look so much better in GeSA".

This looks complex

It is! But also consider how it simplifies further additions and complications.

This looks like it will be terribly slow

Help me create a compiled language for GeSA. Compiling MBEs would help a lot with the speed, and optimizations should even remove most of the matching runtime that is unnecessary. Compiling genetic application will certainly speed this up and still leave it a great paradigm.

This looks like X

See Comparison. And if it's something not there, I would love to hear about it — tell me at ilya.mikhaltsou@gmail.com

Why is it written in Swift?

Swift is a modern and developing language, that has a lot of features that are demanded by modern software development and lacking from other languages.

I would love to base GeSA compiled language on Swift, and so, even though at times Swift obstructs proper development by lacking some important features (e.g. proper generics), I choose it.

Besides, GeSA needs not to be constrained to any single language. Cross-computing can also facilitate communication between different languages, and it shouldn't be hard to port GeSA to any other OOP language. Scala and C++ would be my next choice. Feel free to port if you think it's so great :)

Current state

There is a basic version implemented in Swift — MicroGene. It lacks some of the advertised features, but it provides the necessary foundation to implement software based on Genetic System Architecture. And, most importantly, I use it to develop fully-featured GeSA right now.

Also, it would be awesome to have later:

- GeSA-based language
- GeSA compartment and data ACLs with Gene trust features
- GeSA bindings or reimplementations for other languages
- Better matching algorithms
- Fully-featured MBEs (probably as part of the language)
- Gene-packs as libraries
- GeSA debugger
- GeSA IDE with data visualization and Gene overview
- GeSA-based OS with apps as Gene-packs
- Hardware-based gene-packs

Some basic and advanced patterns to consider

Bridging object instance or OS resource (bridging from OOP)

Put the instance into storage at a pre-defined path. Match by this path on all Genes that require that object. Access is by design thread-safe.

If the object requires a specific execution thread, add this to the Gene specification (should be a feature of a GeSA implementation).

Performing sequential execution (bridging from procedural programming)

Create a dummy data value and put it into a pre-defined path. Match by this path on all Genes that require to be never executed in parallel.

If the object requires a specific execution thread, add this to the Gene specification.

Using GeSA process in a procedural way (Bridging GeSA into other paradigms)

Create an OS lock object and put it into storage. Then put the input data in the storage. Wait on the lock. Have a Gene that matches the final result of the process and the lock created — and unlocks the lock.

Alter the data with a single-run process in the already established process

Match with a higher priority. Add a marker to the output. Never match if the marker is present.

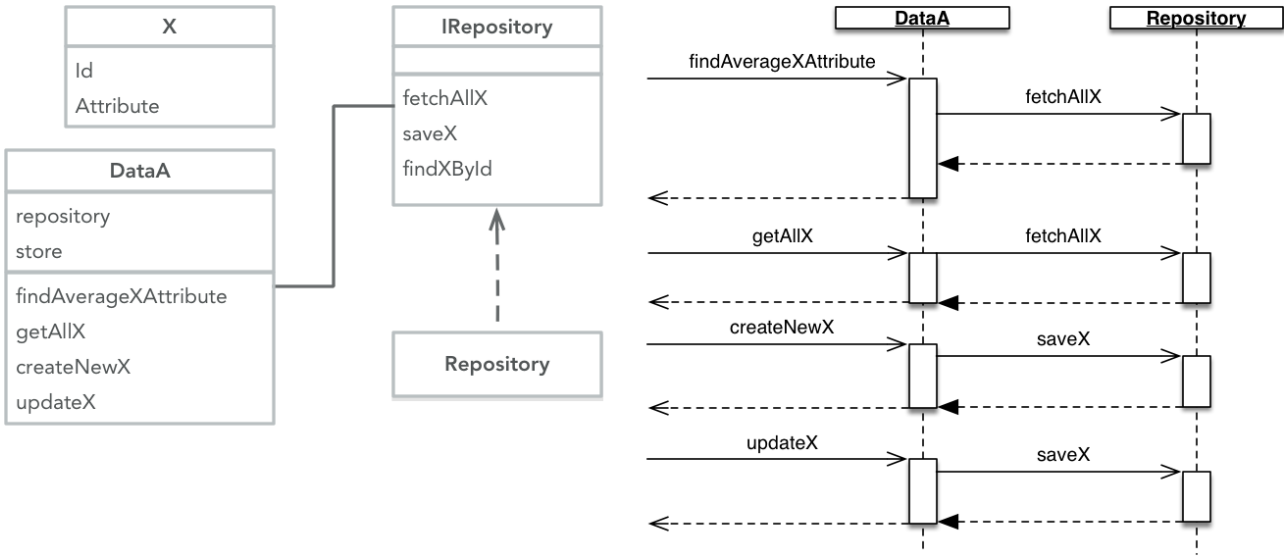
(more later)

What you can do with GeSA

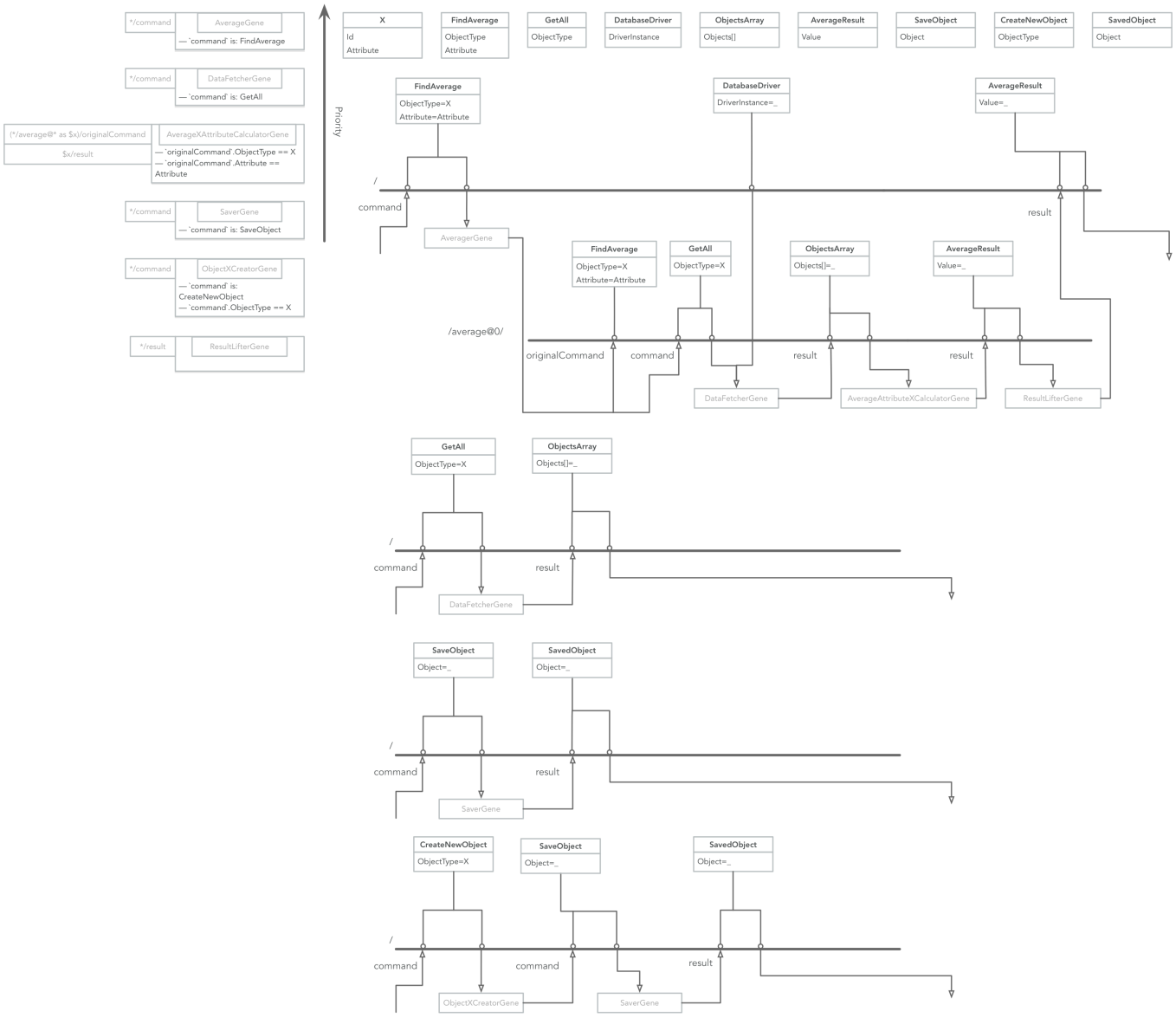
Let's see some examples when it is really great to use GeSA.

Developing a growing agile system

So you start with a backend service that has a database **A** and some business logic.

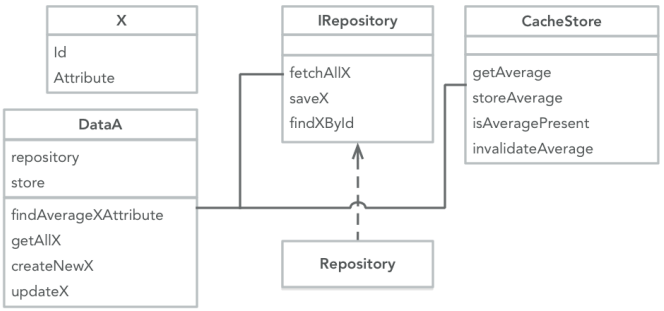


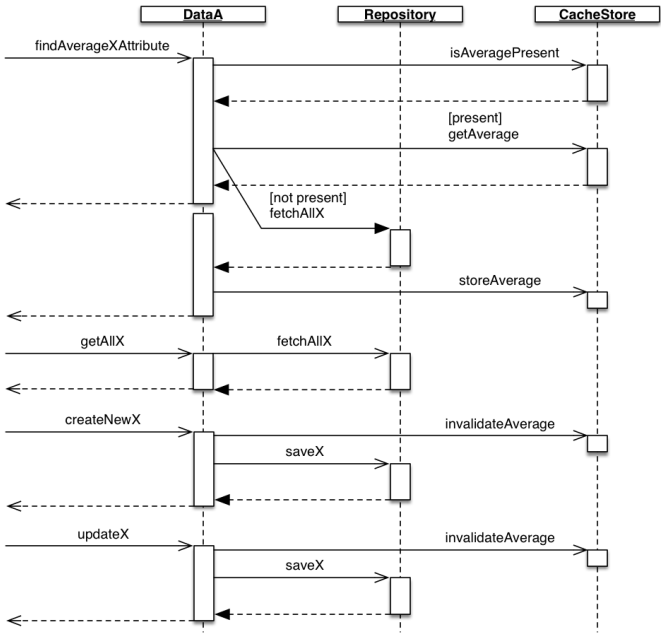
Seems simple enough...



Seems like you never mentioned that GeSA is simple for a reason.

Then you decide you need some caching for the business logic.

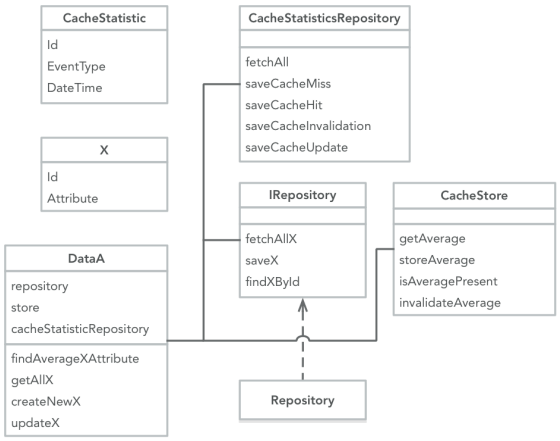


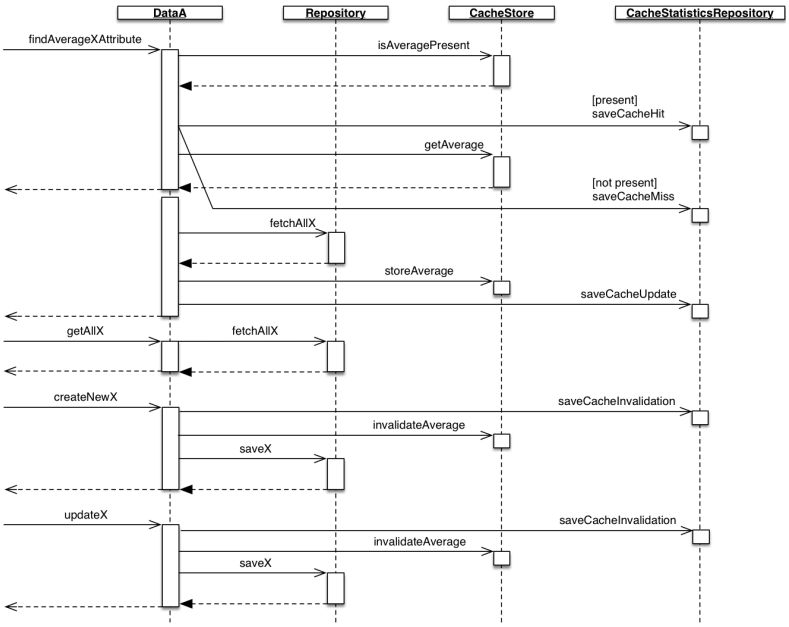


Still OK...

(—insert drawing here—)

Even later, you add database **B** that stores some performance statistics from caching.

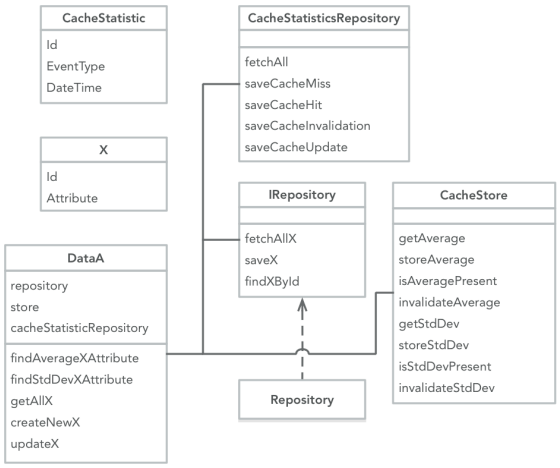


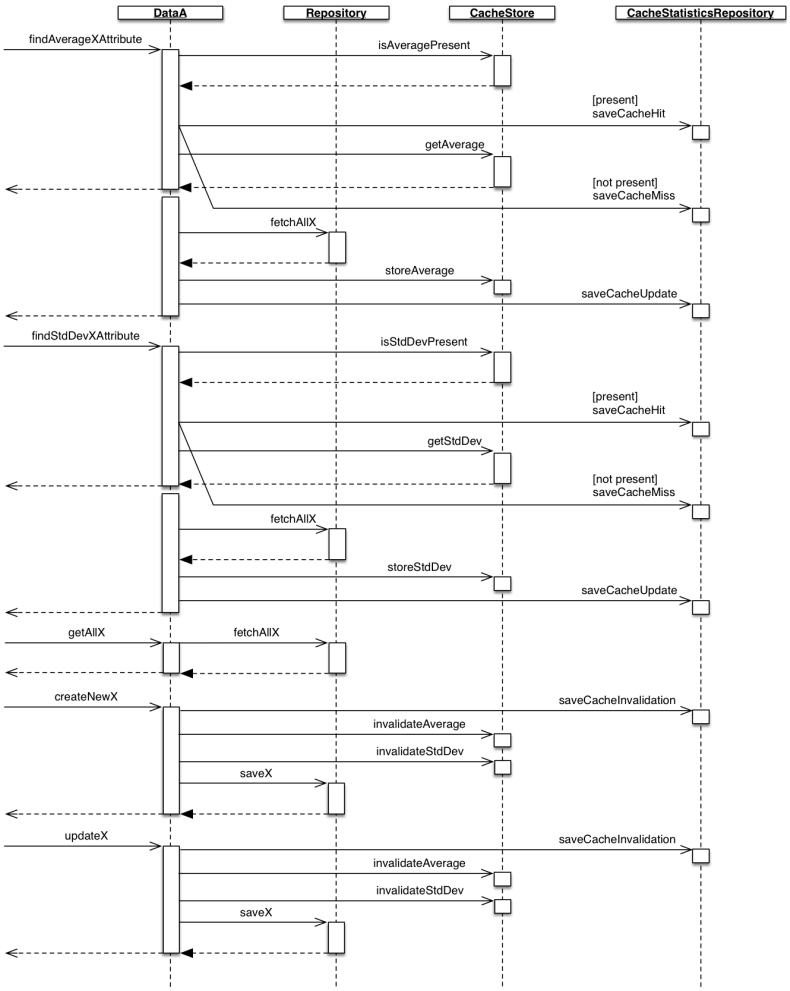


Maybe it's time to think about some refactoring.

(—insert drawing here—)

And also add another value for business logic and cache it as well.

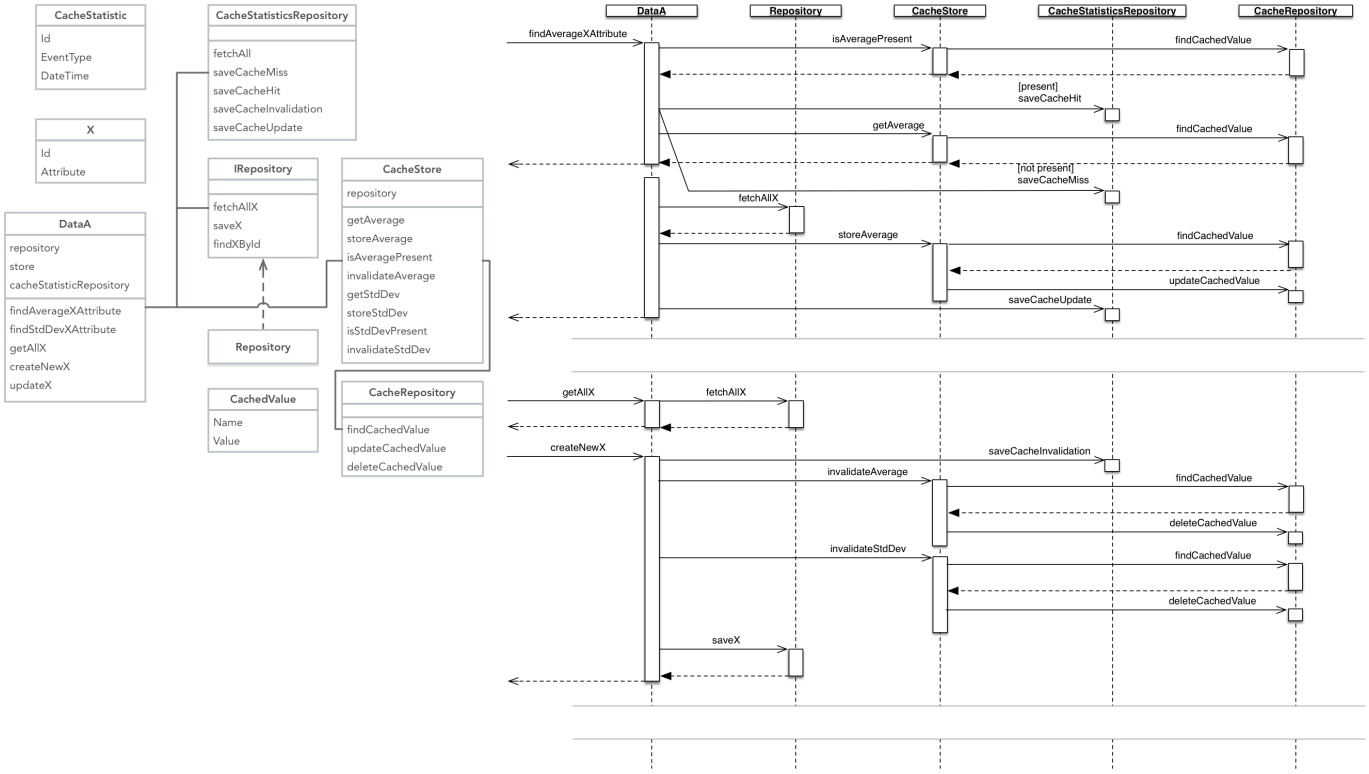




So, you like copy-pasting, eh?

(—insert drawing here—)

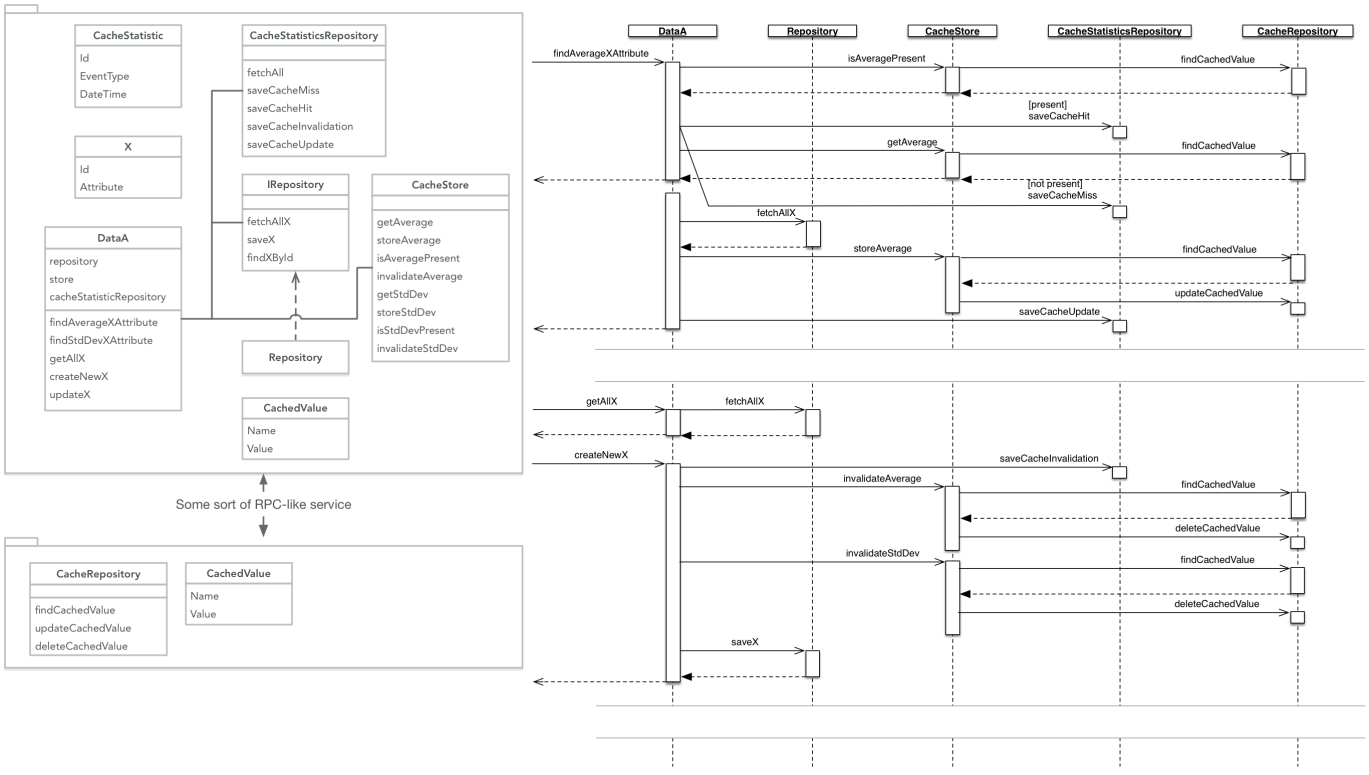
And database **C** that makes some of the caching persistent.



I'll tell you what it doesn't do... It certainly doesn't make things simpler.

(—insert drawing here—)

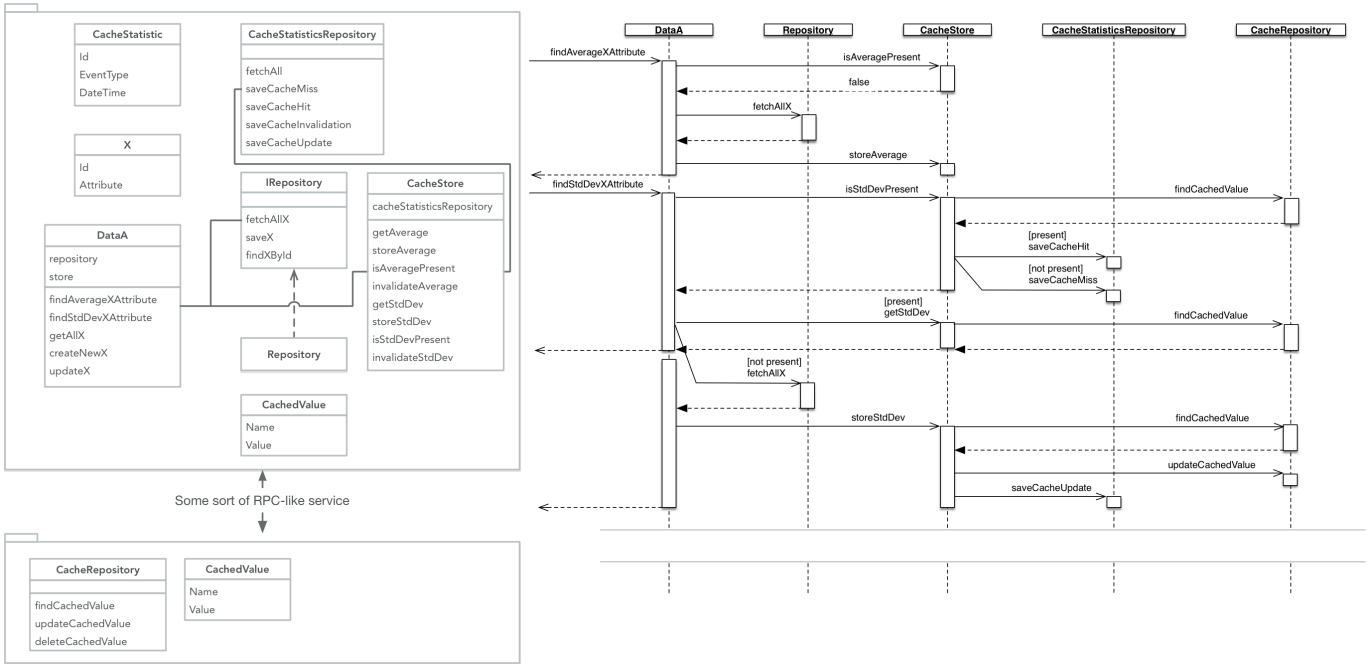
And then you decide that caching needs to be done on a separate server (with database C)...



I hope you didn't want microservices. And please, do use REST at least.

(—insert drawing here—)

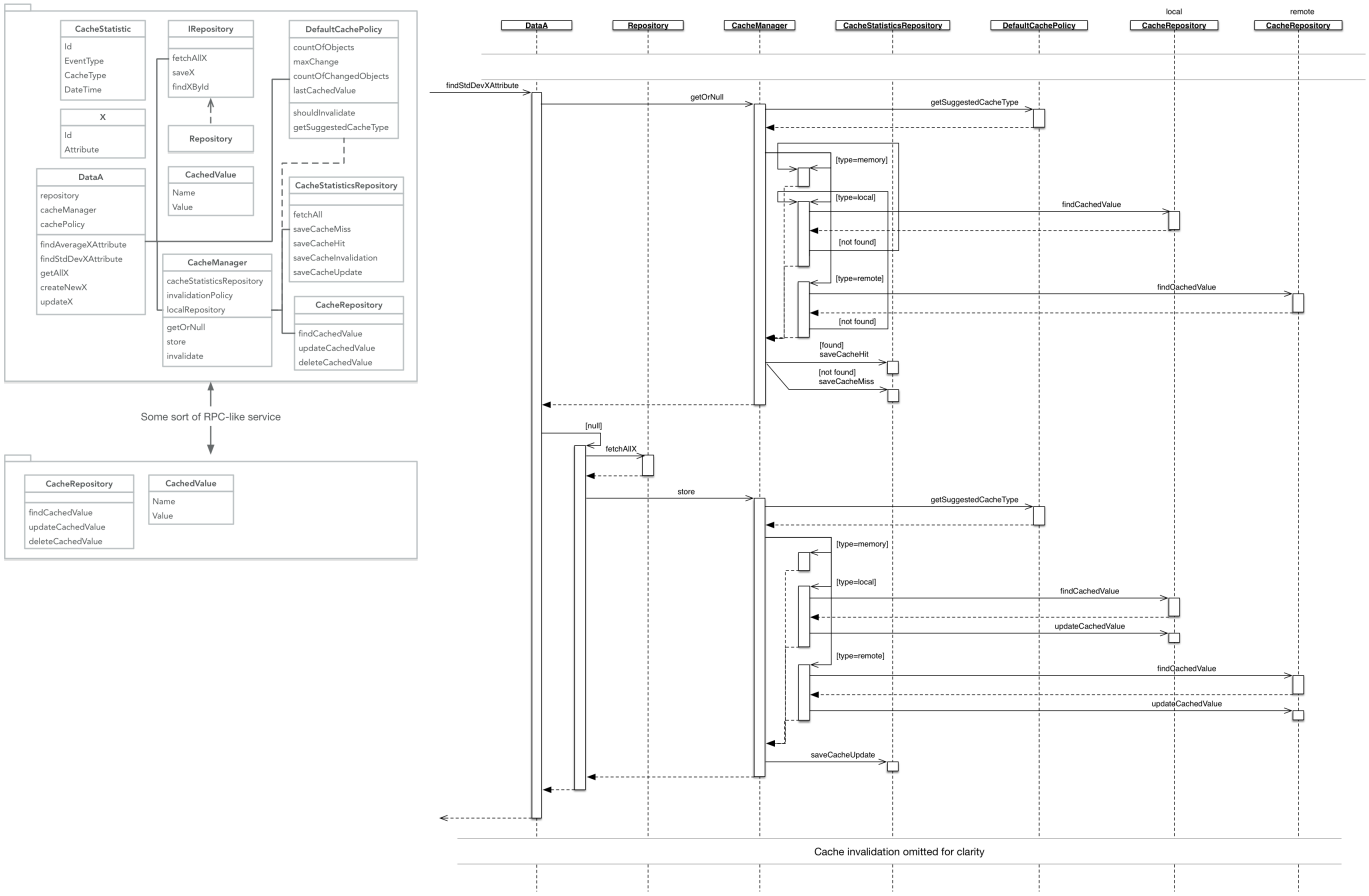
And you also decide you don't need some of the values to be cached. And that seems weird to do cache profiling in business logic — so you refactor it.



Doesn't seem like it became simpler after refactoring.

(—insert drawing here—)

Now you want to cache the other value on the second server only when the count of objects is ☒500. You also want it done only when the count of changed/added objects is ☒5 or when change of the value in any of the objects is more than (cached average)*10%/(count of changed objects). If the count is [100; 500), you want to cache the value on the first server with the same conditions. If the count is <100, you want to cache, but only in memory.



Do you like it? Do you like it?! Do YoU lIkE iT?!!1 Actually, seems quite good.

(—insert drawing here—)

And that profiling? You feel it is ineffective to do that on the first server for the caching that is done on the second server. You want to make it async and replicate it on both servers.

Seems like such change would require even more redesign in conventional architecture — so we skip that.

(—insert drawing here—)

But you also want to use only database B.

(—insert drawing here—)

A dynamic-input system (game)

Let's model a simple open-world RPG in both GeSA and conventional OOP.

(—insert drawing here—)(—insert drawing here—)

And let's add a DLC with a motorcycle.

(—insert drawing here—)(—insert drawing here—)

> GeSA is very much a state-machine on steroids. So whenever you need state-machine-like functionality but with steroids — GeSA may be the best choice)

Another dynamic-input system (SCADA/IoT)

...