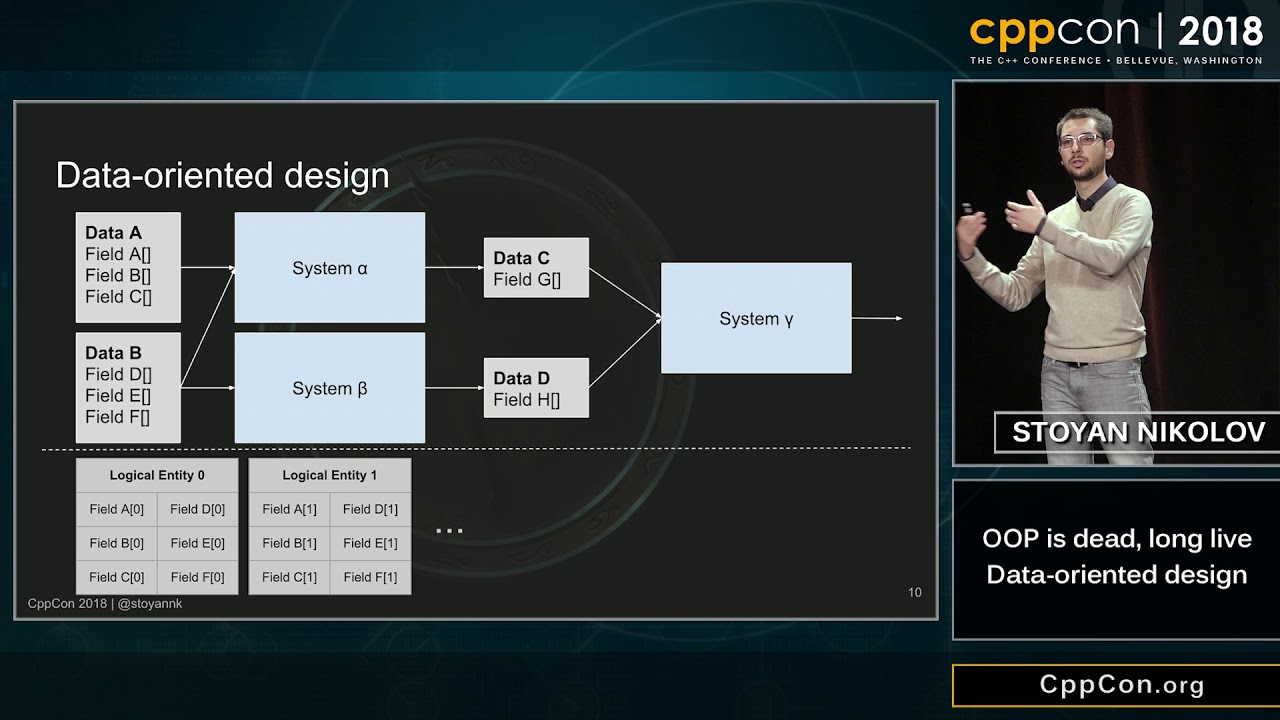
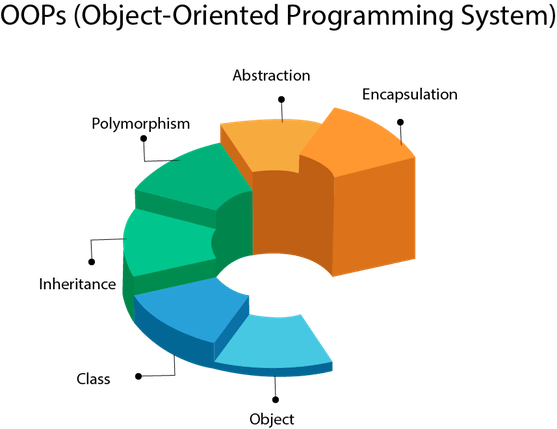
Alternatives to OOP

By Kimon Togrou



Contents

[Introduction 2](#_Toc9169793)

[Problem Definition 2](#_Toc9169794)

[Method 3](#_Toc9169795)

[Planning 3](#_Toc9169796)

[Preface: 4](#_Toc9169797)

[Mike Acton CppCon 2014 Talk 4](#_Toc9169798)

[4](#_Toc9169799)

[Stoyan Nikolov CppCon 2018 talk 6](#_Toc9169800)

[His conclusion: 7](#_Toc9169801)

[Jonathan Blow and Project “Jai” 8](#_Toc9169802)

[AoS vs SoA 10](#_Toc9169803)

[Further Experimentation: 11](#_Toc9169804)

[Visual Studio Features 12](#_Toc9169805)

[Conclusion 13](#_Toc9169806)

[Benefits of DoD: 13](#_Toc9169807)

[Cons: 13](#_Toc9169808)

[Which languages are most suited to DoD? 13](#_Toc9169809)

## Introduction

First off, I choose to write this in English because of all the technical terms that I’m going to be dealing with and the fact that everything I’ll be studying on the subject is in English and partly because I might want to post this online at some point.  
I have chosen this subject because I feel that we get taught to think in a very specific way when it comes to developing software (Object-Oriented), and if I look on the web there is an increasing amount of very prominent and very experienced engineers that are speaking against this way of writing software so I thought that it would be interesting to find out why that is. I want to figure out what we can learn from these people, and if what they’re saying holds any water so to say. I have always been into philosophy and that extends to software development philosophies and as my teacher probably picked up from pretty much day one, performance is a big interest of mine.

## Problem Definition

Although this is a very hard and technical subject, I am hoping to have answered all these questions by the end of these weeks:

* Is Object Oriented Programming slowing down our programs?
  + What is Data-Oriented-Design (DoD)?
  + What are the benefits of DoD vs OOP?
  + Which languages suit themselves best to DoD?
  + What is the difference between Array of Structs (AoS) and Structs of Arrays (SoA) and why is it relevant?
  + Does DoD compromise structure and maintainability for performance?
  + Can you write Data-Oriented in C#?
  + What kind of project/company should use which design method?

## Method

As this is a very theoretical and heavy subject and there isn’t a lot of readily available information on it, most of my work will consist of research although I am hoping to produce at least 1 or 2 code examples.

* I will watch all the talks I can find on the subject (which are about a handful)
* I’ll be making a code example to test AoS vs SoA and see what runs the fastest
* Read forums and study the few practical examples I can find
* I would like to have a program developed in OOP and then make another version where I will try to apply DoD principles to look at any perceivable code and performance differences and code metrics

## Planning[[1]](#footnote-1)

I have four weeks to hand in this assignment, but I have spent first 3 days trying to come up with an angle for this subject and looking for sources so I will be planning the remaining days:

First Week:

* May 1st – 3rd: Watch talks and general reading/research fundamentals

Second Week:

* Write about the fundamentals
* Research benefits of DoD and DoD within C# and other languages

Third Week:

* Write about which languages support DoD best
* Write about which kind of projects/companies fits DoD the best
* Research AoS vs SoA

Fourth Week

* Write a code example for AoS vs SoA
* Write a bigger code example if possible and compare metrics
* Write conclusion

## Preface:

Let’s start by lightly defining Object-Oriented Programming (OOP) first so we have a point of comparison.  
OOP was established as a way of designing software from the idea of world modeling. It’s the presumption, that if we have different kinds objects like three types of “chairs” in our program there must be some defining general characteristics, which we perceive, when we look at it in the real world, that makes a “chair” a “chair”, and therefore we should make a base class that hold the general info that all “chairs” share and all the individual objects should be derived from that base class. Basically, software/code is the platform, and it’s putting more importance on the fact, that we as people can find the “ideal” way to abstractly represent the data than on the fact that computers and more specifically CPU’s want the data laid out in a specific way. There is more to it than this of course, but this will do for now.

## Mike Acton CppCon 2014 Talk

### 

I thought it best to start by watching this talk, because as far as I can tell this talk is where the term Data-Oriented Design gained popularity and it framed this type of conversation for other people to talk about their own view of designing software in alternative ways to OOP.  
  
It should be noted that Mike Acton is a game developer, and so he is very much coming at this from a perspective of squeezing as much out of the hardware as possible, but he also notes throughout the talk that his opinion wouldn’t change if he was going to develop other types of programs.

Mike Acton’s point is that, since the purpose of all programs is to transform data from one form to another, the way we write programs should reflect the most efficient ways of doing this. It’s the presumption that because all we are doing is transforming data from one form to another, that if we don’t understand the cost of solving that specific problem, we don’t understand the problem. In other words; the hardware is the platform. We have a finite range of hardware constraints, and so we should make the most of it. From his point of view world modeling leads to huge programs that are so separated and abstract with unrelated data structures and transforms that tries to idealize the problem rather than solving the very specific data transformation that is required in the best/fastest way. He believes the programmer’s responsibility is for the data and not the code, the code is simply a tool to transform one set of data to another.

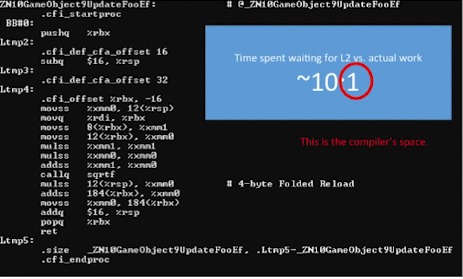
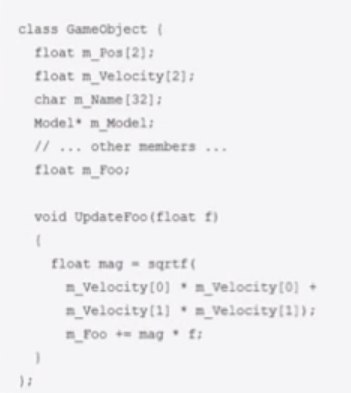
Make Acton believes that OOP perpetuate 3 lies:

1. Software is a/the platform
2. Code should be design around a model of the world
3. Code is more important than data

And that this philosophy leads to:

* Poor Performance
* Poor Stability
* Poor Testability
* Poor Concurrency
* Poor Optimization options

Another very interesting thing that he talks about is that compilers can’t reason well about the data in many situations. In a very simple example, he showcases the compiler spent roughly 90% of the time waiting for the CPU cache instead of doing actual work.



## Stoyan Nikolov CppCon 2018 talk

Stoyan Nikolov is also a game developer, which I think is largely going to be an ongoing pattern in my research as game developers out of absolute necessity must be worried about performance and deadlines.

His starting assertion is that OOP marries data with operations and that this is not a very appropriate way to structure a program if you are concerned about:

* Performance
* Scalability
* Modifiability
* Testability

Nikolov says that you should start building your program with your most common operation in mind.

He has some much-appreciated practical guidelines, he lays out, that I have slightly modified for clarity.

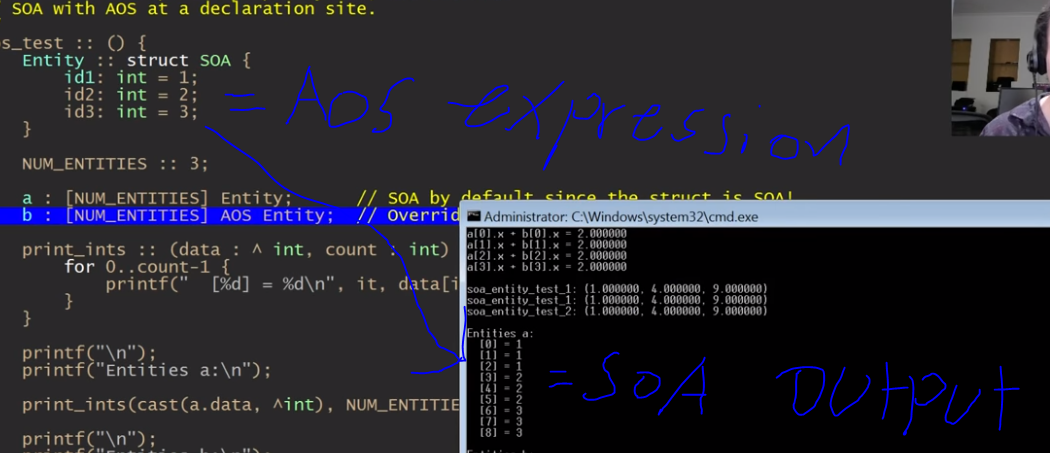
* Separate data from logic
  + Structs and functions live independent lives
  + Data is regarded as information that must be transformed
* The logic embraces the data
  + Does not try to hide it
  + Leads to functions that work on arrays/linear data structures
* Recognizes data according to its usage
  + If we aren’t going to use a piece of data, there is no reason to pack it together with the data we need to access a lot
* Avoid “hidden state”
  + Avoid making a lot of unnecessary states, and separate collections that are based on active/inactive states
* No virtual calls
  + There is no need for them (v-tables can slow things down because you’ll be waiting for your function reference as you will get cache misses)

### Nikolov’s conclusion:

* Multithreading in DoD is way easier since the data is separated more clearly in that you separate object states into their own collections depending on the state
* Testing is easier since you don’t have to write a lot of mock code to write the unit test because of multiple states and so on, all you must do is test your functions which are now simple input and out functions with very simple data
* It’s no longer as hard to modify your program, as you don’t have these base classes that are interconnected in very abstract and complicated ways.
* DoD is a tool and has it’s downsides like every other tool
  + Correct data separation is hard to accomplish
  + Quick data modifications can be hard because you must think about if the introduction of a new field/property is going to mess up your performance where in OOP you just add a new one
  + It’s hard for newcomers to learn to think in this way when they are accustomed to OOP
  + Currently popular languages are not encouraging it
* What to keep from OOP?
  + Third party libraries
  + Some API systems might need it
  + It can be good for higher level systems
* Neither philosophy is a silver bullet
  + They both have their places

## 

## Jonathan Blow and Project “Jai”[[2]](#footnote-2)



Jonathan Blow is a game developer who in the last couple of years have been making his own language, as he has grown increasingly tired of using C++.

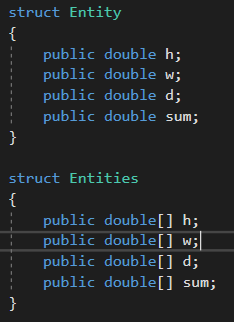
He makes a lot of the same points as others I’ve mentioned so far, but he adds that many languages are made in the pursuit of high-level expressivity and abstraction to make it simpler to program as everything is well separated. Blow believes that this often creates a paradigm in which the program becomes way more complex as your project progresses than it has to be.

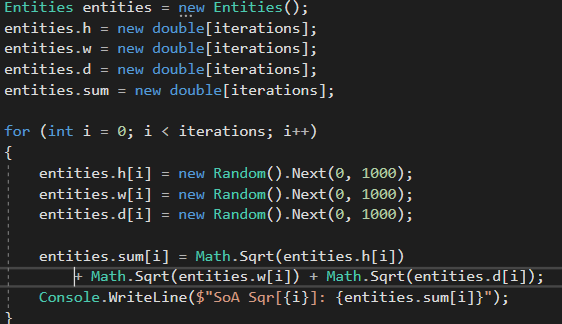
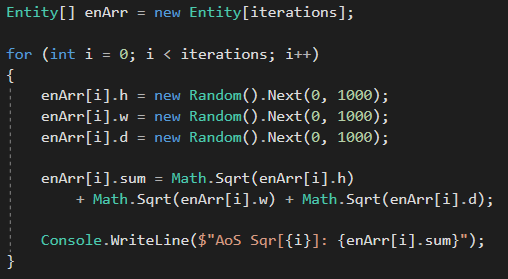
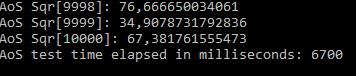
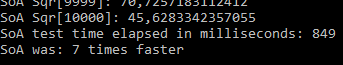
He describes his language as “a language for games”, but it is general purpose language.

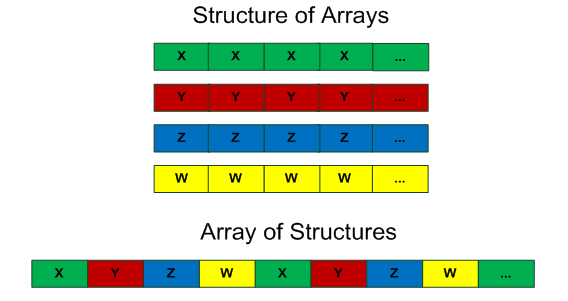
* Primary goals of the language
  + Friction reduction
  + Joy of programming
  + Performance
  + Simplicity
  + Designed for good programmers
* Sub goals
  + Fast compilation
  + Very specific error messages
* What it shouldn’t be
  + Big agenda language
    - A language that is trying to apply one goal to every area of its scope
  + Focus on the fact that bad programmers exist
    - Make everything safe at the cost of freedom and performance
  + RAII (Resource Acquisition Is Initialization)
    - Constructors and destructors and so on causes friction

After looking at some of his demos, I have seen some very interestingly executed ideas. He made it so that you can you do a lot with memory access whilst retaining a lot of high-level expressivity. The most exciting thing for me so far is his use of “AoS/SoA” and “using”. The language supports you writing a construct as AoS but using “SoA” as a keyword for the compiler that then makes the memory access pattern SoA which he says improves performance in most areas without compromising the pattern that we are very used to in OOP. That can be done on the fly too. You can have an AoS setup until the point you need to do something with it, like looping through an array of structs if need be. He uses the “using” keyword in multiple ways, for one, he has made it so that you at any time/scope of your program can use it to pull members into your namespace to shorten how much you have to write to access certain properties or functions within a given context. He also uses the keyword as a way to do polymorphism in a less expensive way, so basically what you can is put the keyword and the struct you want the parameter to “inherit” from as an argument in a function and then you can put anything that is “using” that struct into your function as an argument. It retains a lot of the core features of inheritance without actual inheritance, so the benefit would be that you get to avoid vtables which cause a lot of cache misses.

## AoS vs SoA

  
When we are taught to write OOP, we usually start by creating an object, maybe a person. This object has some attributes and so we type in the properties, a name maybe, perhaps an age. After this it’s quite natural to go on to creating a list of these objects, and then loop trough them to look at average age or some other statistical calculation. This usually leads us to structure most of our code in Array of Structs/Objects. The adherents to DoD talk a lot about the Structure of Arrays pattern.

I tried to come up with a simple example in C# to try to understand what the difference is. I started by declaring two different structs of course, one in AoS and another in SoA.   
The Idea was to loop through these two 10000 times doing the same mathematical operation and output the result, and then use a stopwatch to see how they compare.   
  
  
  
I’m taking three variables and generating a random number for each one and then calculating the square root of each, adding them together and putting it into the sum variable and outputting that. Here are the results of that:  
  
(Typically anywhere from 6-9x faster, 7,5 average I would say)  
 

The core difference between these two patterns is the way they are stored in memory. In AoS it lays out the variables in the way you might imagine objects next to eachother would look like, so if you have 4 variables per object it lays out each set of those 4 variables. If we only needed to do math with two of the variables, in AoS we would go through all of this unneccesay data in each set as seen in the picture. Furthermore it’s heavier on the CPU to run through the data when it’s not next to eachother because of the way CPU’s are structured. With SoA data gets layed out contiguously as a set of each data type. This for one means that we don’t loop through data that we don’t need. There are also very specific CPU bonuses associated with SoA:

* Using the whole cache line
  + As explained above
* Prefetching
  + Fetching data from slower memory and pulling it into faster memory before it is needed
* SIMD (Single instruction, multiple data)
  + Parallelization on one core

### Further Experimentation:

I then started to try with different amounts of iterations and found that there is a huge difference in the performance gain. When I tried with 100/1000 iterations it was more like 3-4x faster with SoA. My initial thoughts are that I am accessing different caches(L1, L2, L3) depending on the initialization size of my variables, which would make sense, because the bottleneck we are dealing with is not the mathematical operation, but memory access speed. Thus, the more time consuming it is to access the data the more benefit we will get from SoA, as it leads to less or zero cache misses. This means we don’t have to travel over to the given cache to access the data more times than we absolutely need to.   
My expectation after seeing the result was that the difference should get larger the more data we need to access, because the more data there is to loop through the further back in the hierarchy of memory-based hardware we must go. The fastest (L1 cache) has the least amount of memory, whereas the slowest (Ram) has the largest amount and in between we have L2, L3. So, if we have to go to the L3 cache a lot then SoA is said to have a bigger impact than if we only needed to access L1.

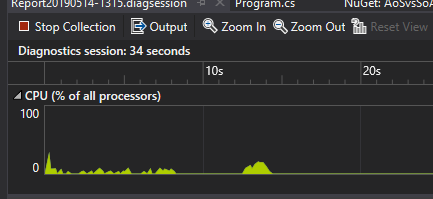
But then something strange in the AoS test is the fact that, if I start the program multiple times in quick succession with smaller amounts of iterations then the performance boost seems to diminish. The same thing happens if I put in any number over 10000 iterations. If I put in 20000 it seems that after the first 10000 iterations, the speed of looping through the remaining data seems exponentially faster. It looks as if the compiler is finding a pattern or maybe there is something in memory that helps.

It turns out that the time it takes to do 10000 calculations is about the time it takes for the compiler to get going, so I found out that if I run an AoS test before running the timed one, the performance boost seems trivial, this makes me to think that there is something wrong with the way I’ve setup the experiment which makes the calculation the bottleneck.

Googling a lot and reading forum posts about SoA in C# lead me to the conclusion that the way Garbage Collecting works in C# is somehow influencing the experiment. Maybe revisit write it in C++

## Visual Studio Features Needs structure

I tried get code metrics to work on .Net Core applications through plugins, but ultimately it was too much of a hassle and so I just created a separate .Net Framework project that I can copy paste files into and view the metrics from there.

Trying to figure out the best way of viewing memory access and CPU cache cycles in Visual Studio has been a very time-consuming challenge, it seems as though people really aren’t concerned with what is happening behind the hood when running their code and so googling left me with very little to go on. The easiest feature to find was the performance profiler:  
  


This doesn’t really tell me at lot that I can’t just output through my code. It shows me some graphs, but what I was looking for was more of a text-based representation of what the compiler is doing, what/how is data being stored in memory? How many cycles did it take to read something? Basically, some information that can help me reason about the data, to see if I’m doing something expensive, and if there might be better ways of structuring functions. Just knowing that a function or line of code is heavy doesn’t tell me much about how to change it for the better.

This was my last-ditch effort to see if I could diagnose the AoS vs SoA example, without being able to see

## Conclusion

### Benefits of DoD:

* It’s all about the data, which means you’re know you’re dealing with the actual problem of transforming data efficiently
* Cache utilization, DoD makes you lay out the data in a contiguous and homogeneous way which is what the CPU ultimately wants
* Parallelization becomes much easier because you localize the problems with small functions that have input and output data which is easy to get running on multiple threads
* Testing becomes a lot simpler too for the same reason. When you are largely dealing with low level data transformations like small functions that have an input and an output, it’s easy to check if the data transformation went as it should without having to create a bunch of mock code to simulate all of the dependencies and status heavy scenarios that OOP encourages
* Modularity is a big bonus as there aren’t as many dependency problems, like classes that are interconnected in very complicated and abstract ways, this makes it so you can move pieces around more freely if you need to experiment

### Cons:

* There are not really any good sources on the subject and that makes it hard to learn
* It’s very difficult to “unlearn” the OOP way of thinking about software although I am not sure how much harder if at all it would be to learn OOD if you are a beginner

## Which languages are most suited to DoD?

1. I couldn’t figure out a productive way to set up a scrumboard as this is a very research heavy subject and felt it would be too vague. [↑](#footnote-ref-1)
2. This is based on a series of videos from his YouTubes channel in which he describes his ideas and demos his new language – references at the end of the document [↑](#footnote-ref-2)