MASTER THESIS

Thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering at the University of Applied Sciences Technikum Wien - Degree Program Game Engineering and Simulation Technology

Using LLVM/Clang to to maintain the abstraction level of Object Oriented Programming, yet abide SOA rules

How compiler technology could possibly make a link between conflicting programming paradigms

By: Julian Müller, BSc.

Student Number: 1610585015

Supervisor: Stefan Reinalter, DI

Second Supervisor: Prof. Alexander Hofmann, DI

Wien, January 16, 2019

FH University of Applied Sciences TECHNIKUM WIEN

Declaration

“As author and creator of this work to hand, I confirm with my signature knowledge of the relevant copyright regulations governed by higher education acts (see Urheberrechtsgesetz /Austrian copyright law as amended as well as the Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I hereby declare that I completed the present work independently and that any ideas, whether written by others or by myself, have been fully sourced and referenced. I am aware of any consequences I may face on the part of the degree program director if there should be evidence of missing autonomy and independence or evidence of any intent to fraudulently achieve a pass mark for this work (see Statute on Studies Act Provisions / Examination Regulations of the UAS Technikum Wien as amended).

I further declare that up to this date I have not published the work to hand nor have I presented it to another examination board in the same or similar form. I affirm that the version submitted matches the version in the upload tool.“

Wien, January 16, 2019 Signature

Abstract

This work will prospect the possibility of using compiler technology as a mediator between the conflicting programming paradigms *OOP* and *SOA*.
While Object oriented programming is often praised for its benefits on abstraction and maintainability, it encourages programmers to design inefficient datalayouts. Specifically in game engineering, where performance is a constitutive factor for a product's success, data oriented solutions are on the rise. While it is debated, wether or not performant data layouts inevitably entail challenging maintenance, surely the base concepts of *objects* are well observable in a game. Therefore this will be an attempt to make object oriented programming a valid option for ever rising demands on performance.
A prototypical implementation of a source-to-source transformation tool called *COOP* (**C**ache friendly **O**bject **O**riented **P**rogramming), developed in the Clang tooling environment, will determine wether or not compiler technology can be used to achieve a performance optimization on a classically OOP abidant target source code base.

Keywords: Object Oriented Programming, OOP, Structure Of Arrays, SOA, Data Oriented Design, Compiler, LLVM, Clang

Acknowledgements

TODO dankschee

Contents

1 Conflicting Paradigms

Numerous programming paradigms exist for even more general programming languages. Each of them come with different perks and offer different perspectives for a problem. Different programming paradigms, however don't necessarily exclude each other. Languages like *Lisp* or *Scala* for example both combine functional and object oriented programming.
This thesis will specifically concentrate on *Object Oriented Programming* (OOP) and *Data Oriented Design* (DoD). Whether or not data oriented design is even to be considered a programming paradigm is debatable [8, p. TODO], however its fundamental ideas (specifically concerning data layout) conflict with those of existing paradigms like OOP, so for the sake of consistency in this manner of comparison, we will call it that way.
Depending on the domain, certain programmers will have different answers to the question which of both should be preferred. Each party will make compelling arguments to why their choice is mandatory. This is because those paradigms (partially) solve different problems and therefore offer dissenting perspectives on problems and their respective solutions for them. To understand why *Object Oriented Programming* and *Data Oriented Design* are rather cannibalistic to each other, we need to first have a look to what they are trying to solve, independently.

1.1 Object Oriented Programming / AOS

Starting with punch cards, each iteration of new programming generations provided new forms of abstraction for programmers, be it control flow statements, type systems, data structures like native arrays. When machine code started to simplify operations, FORTRAN partly introduced portability as early as 1957; LISP introduced symbolic processing and automated memory management until finally Simula/67 introduced objects in the 1960s [15][22]. Object Oriented Programming could not be called popular until the 1970s or 1980s, when Stroustrup created C++. Originally OOP was meant to be the way to go for creating graphics based applications [12]. This makes sense, since tree like data structures (GUIs), containing entities with shared behavior can easily be described through polymorphism.
OOP shines, whenever the problem to be modeled can be abstracted to one or more base classes, that define shared data and/or behavior. Even though some languages allow for a subclass to "*exclude variables and/or methods which would otherwise have been inherited from the superclass(es)*" [19, p. 4].
OOP quickly established and rooted itself in the industry without solely being used on graphics applications. This is due to the fact, that it represents a world model, we are taught since

elementary school. We are familiar with *is-a* relations ever since we learned that despite dissenting traits, both Labradors and Pugs are dogs. Arguably abstraction is one of - if not - the most important disciplines in programming [9, p. 5]. Since programs oftentimes try to model real life information, OOP delivers an easy to grasp, quick-to-learn approach to do so. That is also the reason, why there are so many OOP programmers around the world. Without trying to evaluate whether or not OOP's way of abstraction is superior to DoD, it is undoubtedly the more prominent one, especially for virtually any other profession than a programmer/computer scientist (see Fig. 1). This is however where OOP and modern computer architectures don't get

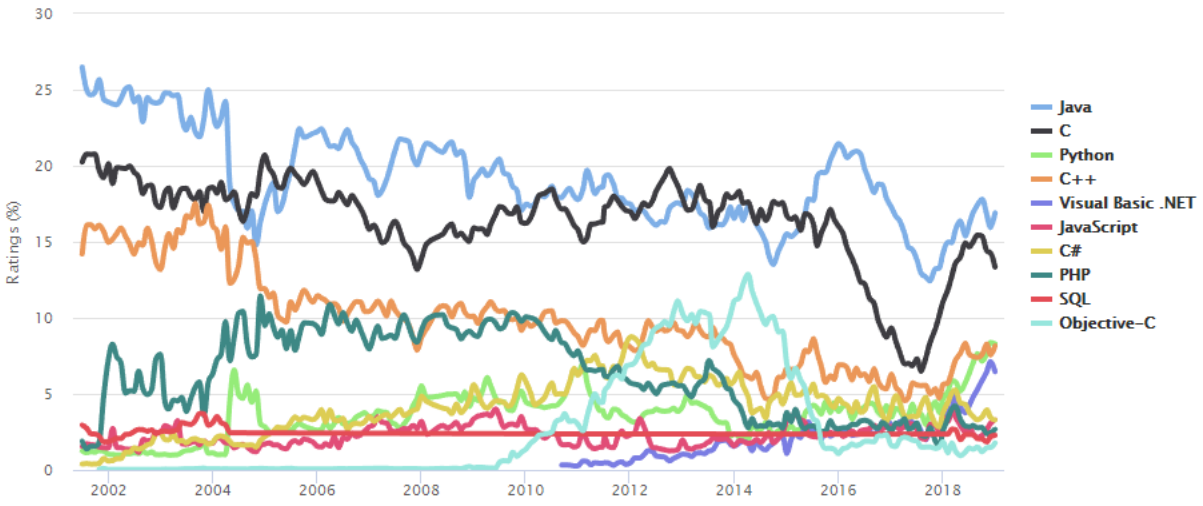

Figure 1: Most popular programming languages throughout the years (Source: [1]).

along, so to say. OOP ovvers an elegant way to intuitively model an issue into code, but doing so it encourages us to implement our data in an inefficient way. Inefficient because creating monolithic models of our data usually lack cohesion, which means, that the classes' members tend to not be related/dependent [2] - at least in terms of computational order.
In a home office application, juggling a few dozens of entities every other minute, this will not appear as a problem. And in this case it might be preferable to program such application in a strict object oriented way, since the development can be done fast and reliably even by a novice. On the other hand and especially in the game development industry OOP has proven to result in poorly performing software, due to inefficient data layouts. Because especially for games: data is performance [21, p. 272].
The abbreviation *AOS* stands for *Array Of Structures*[5] and it describes, what usually happens with Object Oriented Programming.

```
struct Obj {
   float xyz[3];
   float vel[3];
};

struct Human : Obj {
   char *name;
   int age;
};

struct NPC : Human {
   int mood;
};

NPC npc_arr[3];
```

Code 1: Example of some hierarchical POD class definitions

Considering the rather arbitrary C++ class definitions in Code 1 the *npc_arr* will occupy memory according to Fig. 2 (disregarding any *padding*). This is quite literally an array of structures -

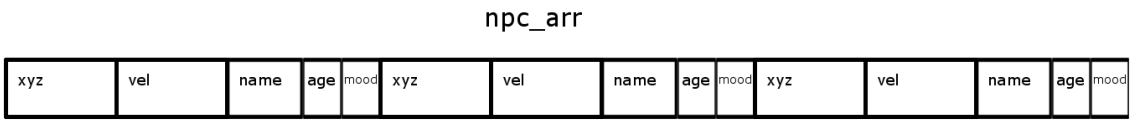

Figure 2: Visualization of how a *npc_arr* will exist in memory

hence the notation *AOS*. The following sections 1.2 and 1.3 will elaborate on *why* exactly this way of thinking/way of abstraction is inefficient.

1.2 CPU Caches and why they don't *fit* objects

The data layouts typically found when programming with objects and classes, are not inefficient because they lack logic. "*Each human - including students - will have positional traits*", is semantically correct. In fact it seems rather unfortunate, that modern computer architectures can't deal well with an abstraction that fits our perception of the world. But the hardware is certainly not to blame here. The problem with a monolithic class definition is much more that of common coding- or data access patterns.

1.2.1 Common data access patterns vs. Monolithic class definitions

Numerous coding best practices teach us to write simple, modular code.

> *Functions should do one thing. They should do it well. They should do it only.* [17, p. 35]
>
> *Keep it simple and smart* (KISS principle). [20, p. 77]
>
> *[...] cohesion is an important object-oriented software quality attribute.*[2]

Just like we want our class definitions to share a common responsibility or task, we want the set of instructions that iterate and probably transform a set of data to be as simple and modular as possible. So usually we try to not write monolithic *for-loops* handling every single aspect of a set of data.
For example in Fig. 2 we would not want a loop that handles each and every single member of an NPC. This would not only result in a big set of instructions, that hide the individual purposes of each expression, but also make it hard to maintain/change the code. Not to mention, that

different data often demands change at different times at runtime. Requirements can change quickly. Breaking up responsibilities that were coupled and forced to coexist change not so quickly.[17]
Exemplary if Code 1 was the model for a game, our game loop could at one point iterate over all the elements in the *npc_arr* to update their position and velocity for each frame. The *NPC's mood* could just as well be updated frequently in a separate function, that only encompasses the information relevant for the calculation of the updated mood. Their *Human::name*s however will most likely not change so frequently - if ever - so the instructions to modify that data will most likely depend on user input and exist in yet a whole other routine. This modularization of code is commonly referred to as *Separation of Concern* and has proven to improve the code's maintainability [13, p. 85]. *This* keeping the objects in some sort of set, then iterating over it for each routine, that manages a subset of the object's data, is a common access pattern that is applied on objects in OOP.
The interim conclusion here is, that even only for maintainability reasons, it is desirable for programmers to process logically related subsets of their data separately - but then why is the resulting software so slow compared to the same idea implemented with a *Data oriented Design*?
The Object Oriented Programming paradigm is exactly doing what it promises - providing a sort of abstraction, that programmers can intuitively apply to their problem definition. Consequently OOP programmers quickly adapt the habit of developing against their abstraction because it is intuitive. What is lost in the process is the concern of developing against the hardware. *This* is probably the fundamental difference between OOP and DoD.
So whats our hardware's deal? Why do objects don't get along with it. Why can't we have super high speed machinery, that makes hardware concerns obsolete? Why can't we have anything nice?

1.2.2 A brief history of memory

To answer section 1.2.1's concluding question: We do have high speed memory units at hand we just can't afford them. Modern computer systems rely on a variety of different memory units each differing in access latency and other properties. The intention behind this complex hierarchy of memory layers is obviously speed and is the result of an evolving cost-benefit calculation.

> *The task the computer designer faces is [...] design a computer to maximize performance while staying within cost[...].*[11, p. 8]

Originally

> *memory access latency and instruction execution latency were on roughly equal footing. [...] register-based instructions could execute in two to four cycles, and a main memory access also took roughly four cycles.*[10, p. 189]

This proportion changed significantly. While it is relatively cheap to produce high speed CPUs the same is not true for memory units. So whats happening, is that today's PCs/consoles are equipped with CPUs that are way faster than the greater parts of their available memory units. Due to increasing tick rates and *Instruction Pipelining* what used to be four cycle RAM reads are now several hundred cycles. Not Because RAM became slower - the opposite is the case - but because CPUs became that much faster in relation. This trend was thoroughly observed and documented by John L. Hennessy and David A. Patterson (see Fig. 3).

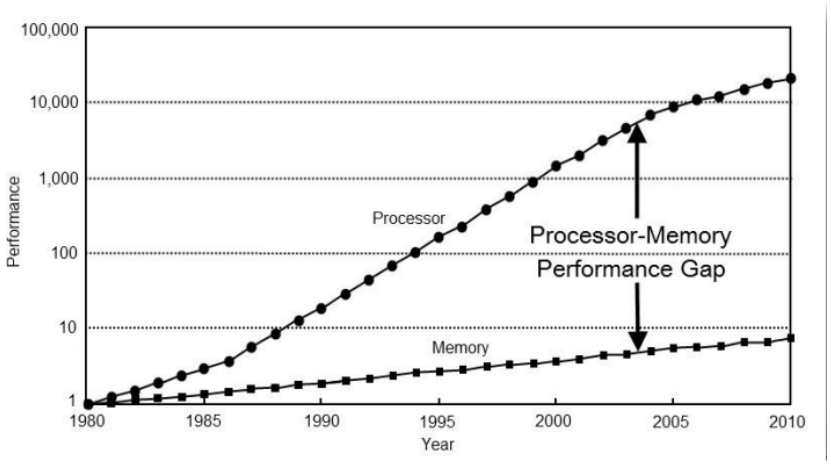

Figure 3: "***Starting with 1980 performance as a baseline, the gap in performance between memory and processors is plotted over time****. Note that the vertical axis must be on a logarithmic scale to record the size of the processor-DRAM performance gap. The memory baseline is 64 KB DRAM in 1980, with a 1.07 per year performance improvement in latency (see Figure 5.13 on page 313). The processor line assumes a 1.25 improvement per year until 1986, and a 1.52 improvement until 2004, and a 1.20 improvement thereafter*" (Source: [11, p. 289])

To solve the issue of ever diverging CPU/memory performances (commonly referred to as the *memory gap [10]*), specifically to reduce the latency of references to main memory, smaller but significantly faster (and more expensive) memory units are placed between the CPU and the main memory. These modules are called *Caches* - first named by C.J. Conti [4] in 1969. Originally cache technology was mentioned as *buffers*[6]. Not considering their complex, modern modalities and policies this is a fitting notation.

The basic idea behind a fast buffer interconnected between the CPU and the main memory is to create local copies of referenced data-chunks, in order to provide faster access on subsequent calls to the same *AU* (Adressable Unit) as well as the ones deemed likely to be accessed soon [10, p. 191]. This principle originated in the work on *Virtual Memory* [6, p. 15] and is today much more sophisticated.

So we actually do use high speed memory in our common computer architecture, we just don't have lots of it.

1.2.3 Cache modules and types

In today's PCs/consoles typically each CPU core has its own hierarchy of cache modules (see Fig. 4). Closest to the core (on-chip) is the *L1* (Level 1) cache. Accessing an AU in the Intel®

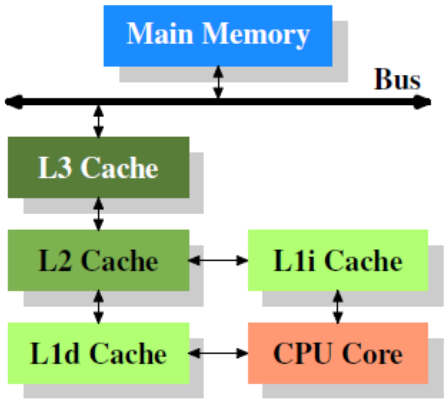

Figure 4: Exemplary, simplified model of a CPU core and its several cache modules (Source: [7, p. 15])

Core™ i7 Processor's L1 cache for example is almost as fast as accessing it in the very CPU's register - 4 cycles (2.1 to 1.2 ns) [14]. For reference access to main RAM "*can take on the order of 500 cycles[...]*" [10, p. 189]. Another cost unrelated reason, why we don't have lots of *L1 D* cache is that more memory literally means more physical space is occupied. Having more cache memory equals more *cache hits* (see section 1.2.4) but as soon as the cache won't fit on-chip anymore, there is yet again additional latency. Thats why the L1 and sometimes L2 cache modules are kept comparably small but on-chip. Other L3 and L4 caches are each bigger and slower than their preceding counterparts respectively, but for the most part share the same ideas slowly converging latency times to the common main memory RAM.
CPU cores can share one or more cache modules (usually starting with L2 or L3), effectively accessing the same local copies. This entails synchronization issues, that will be mentioned in the context of this work later on (TODO).
Modern cache hierarchies include data caches as well as instruction caches, usually both in an L1 cache. However there are different takes on how to implement this. Harvey G. Cragon lists [6, p. 17]:

- instruction cache - holds only instructions
- data cache - holds only the data stream
- unified cache - holds both instructions and data
- split cache - two-cache system, one for instructions and one for data

The scope of this masters thesis will omit instruction-cache related subjects, because the upcoming attempts and techniques to achieve performance optimizations will focus on improving the data layout of a given target source code.

1.2.4 The CPUs cache utilization

A programmer will rarely ever directly interact with a cache module (though there are mechanisms for manual prefetching/clearing). The underlying idea for it was to be transparent to the programmer. However understanding the CPUs cache utilization enables one to tailor the data layout to it, resulting in faster access, less waiting and consequently higher throughput. This is what *Data oriented Design* aims to do - because it develops against hardware concerns.
As mentioned before the basic idea of the cache is to provide local copies of data at a faster rate and prefetch data segments, that are likely to be used soon. This works by directing each main memory access to go through the cache. Main memory access means also going through virtual address translation, address and data buses and depending on the main memory crossbar switching logic [10, p. 190]. Whenever the CPU requests access to a certain AU before saddling the horses and going on a journey to main memory, the cache will check whether or not the requested AU is currently present inside its buffer. If so it is referred to as a *cache hit*, otherwise a *cache miss*. For a modern L1 D cache this buffer, to be more specific, consists of several cache lines each of 64 bytes. Overall cache- and line sizes vary between different architectures but are standardized mostly due to Intel's work.
The simplest and least efficient implementation to determine a cache *hit/miss* is to iterate each cache line comparing fitting criteria. To prevent this nowadays caches implement a certain associativity technique. This way each individual physical main memory address can be mapped to one or more specific cache lines. Doing so addresses are converted to and managed by metadata consisting of: [10, p. 193]

- Offset - Offset to the actual referenced Byte inside the cache line.
- Cache Line Index - Which cache line/s would hold the AU.
- Tag - Which cache sized block in main memory holds the original data.

In the case that each physical main memory address, has exactly one counterpart it is called a *direct-mapped* cache [10, p. 194]. In this case since the cache can by far not encompass the whole extend of the main memory, a lot of physical addresses will be mapped to the very same cache line, effectively extruding each other out of the cache. This is called *eviction* and in unlucky cases will behave like all references are cache misses [6, p. 97]. To prevent this modern caches map a physical address to *n* cache lines. This is called *n-way associativity* - typically implemented as *8-way* or *16-way* caches depending on the level.
There is a lot more to cover about caching technologies and policies like: Replacement-/Write-/Coherency(MESI; MOESI; MESIF) policies and for further reading Harvey G. Cragon's *Memory Systems and Pipelined Processors*[6] as well as Jason Gregory's *Game Engine Architecture*[10] are highly recommended reads. However for the purpose of this work a few specifics are most interesting for us.
How can a data layout affect *hit ratios* and reduce calls to main memory? As mentioned before there are common data access patterns in software and the caches actually accommodate

us by adapting their builtin prefetching mechanisms to it, following a set of *locality concepts*. Harvey G. Cragon counts three of those concepts: [6, p. 16]

- *Temporal locality* - Information recently referenced by a program is likely to be used again soon.
- *Spatial locality* - Portions of the address space near the current locus of reference are likely to be referenced in the near future.
- *Sequential locality* - A special case of spatial locality in which the address of the next will be the immediate successor of the present one.

At first glance these concepts are very straight forward, but their respective implementation for automatic hardware prefetching is more complex than one would think. Prefetching basically means:

> "*[...] bringing data in the data (or mixed instruction-data) cache before it is directly accessed by a memory instruction [...].*" [3, p. 610]

There are however different strategies to decide which bytes should be faithfully loaded into the cache. Tien-Fu Chen and Jean-Loup Baer list two categories for prefetching strategies: [3, p. 610]

- *Spatial* - access to current block is basis for prefetching.
- *Temporal* - lookahead decoding of instruction stream is implied.

There are simple approaches like: whenever block *i* is accessed, prefetch block *i*+1, called the *One Block Lookahead*; stride based approaches storing previously referenced addresses in a table and calculating a stride based on current and previous addresses; combinations of both and many more [3]. Sometimes data is prefetched into the cache, sometimes into separate stream buffers. There are many different hardware prefetching methods to find and while data cache prefetching is considered to be more challanging [18, p. 4] it is still best practice to rely on spatial locality when modeling data to play into the cache's hand.
Compilers already make use of software prefetching in certain cases.

> *For array-based applications, the compiler can use locality analysis to predict which dynamic references to prefetch, and loop splitting and software pipelining to schedule prefetches.* [16, p. 223]

So we can already deduce that for an efficient data layout it would be beneficial to rely on arrays, or more generally, concepts compilers can 'comprehend' and optimize. Also even though the concept of *sequential locality* is only a special case, it is the one we can utilize best to derive adaptations for our data layout, since hardware prefetching has adapted best to it. How exactly can we convert this information into a generic set of rules/best practices, a manifesto of efficiency? We don't it already exists.

1.3 Data Oriented Design / SOA

2 CPU caches and objects

2.1 Cache technology

3 Quotes

3.1 hennessy

3.1.1 principle of locality

The most important program property that we regularly exploit is the principle of locality: Programs tend to reuse data and instructions they have used recently. A widely held rule of thumb is that a program spends 90% of its execution time in only 10% of the code. An implication of locality is that we can predict with reasonable accuracy what instructions and data a program will use in the near future based on its accesses in the recent past. The principle of locality also applies to data accesses, though not as strongly as to code accesses. Two different types of locality have been observed. Temporal locality states that recently accessed items are likely to be accessed in the near future. Spatial locality says that items whose addresses are near one another tend to be referenced close together in time. [11, p. 38]

3.2 drepper

3.2.1 cache layout

3.2.2 Lc Ld

Even though most computers for the last several decades have used the von Neumann architecture, experience has shown that it is of advantage to separate the caches used for code and for data [7, p. 14]

Hier wird auf die Formel 1 verwiesen.

$$x = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q} \tag{1}$$

$$x = -\frac{p}{2} \pm \sqrt{\frac{p^2}{4} - q} \tag{2}$$

Bibliography

[1] *TIOBE Programming Community Index*.

[2] AL DALLAL, J.: *A design-based cohesion metric for object-oriented classes*. International Journal of Computer Science and Engineering, 1(3):195–200, 2007.

[3] CHEN, T.-F. and J.-L. BAER: *Effective hardware-based data prefetching for high-performance processors*. IEEE transactions on computers, 44(5):609–623, 1995.

[4] CONTI, C.: *Concepts for buffer storage*. IEEE Computer Group News, 2(8):9, 1969.

[5] COORPORATION, I.: *Intel 64 and IA-32 architectures optimization reference manual*, 2009.

[6] CRAGON, H. G.: *Memory systems and pipelined processors*. Jones & Bartlett Learning, 1996.

[7] DREPPER, U.: *What Every Programmer Should Know About Memory*, 2007.

[8] FABIAN, R.: *Data-oriented design: software engineering for limited resources and short schedules*. 2018.

[9] GHEZZI, C., M. JAZAYERI and D. MANDRIOLI: *Fundamentals of software engineering*. Prentice Hall PTR, 2002.

[10] GREGORY, J.: *Game engine architecture*. AK Peters/CRC Press, 2014.

[11] HENNESSY, J. L. and D. A. PATTERSON: *Computer architecture: a quantitative approach*. Elsevier, 2011.

[12] HUANG, P. J.: *A Brief History of Object-Oriented Programming*.

[13] LAPLANTE, P. A.: *What every engineer should know about software engineering*. CRC Press, 2007.

[14] LEVINTHAL, D.: *Performance analysis guide for intel core i7 processor and intel xeon 5500 processors*. Intel Performance Analysis Guide, 30:18, 2009.

[15] LOUDEN, K. C. and P. ADAMS: *Programming Languages: Principles and Practice, Wadsworth Publ*. Co., Belmont, CA, 1993.

[16] LUK, C.-K. and T. C. MOWRY: *Compiler-based prefetching for recursive data structures*. In *ACM SIGOPS Operating Systems Review*, vol. 30, pp. 222–233. ACM, 1996.

[17] MARTIN, R. C.: *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[18] MITTAL, S.: *A survey of recent prefetching techniques for processor caches*. ACM Computing Surveys (CSUR), 49(2):35, 2016.

[19] NELSON, M. L.: *An Introduction to Object-Oriented Programming*. Techn. Rep., NAVAL POSTGRADUATE SCHOOL MONTEREY CA, 1990.

[20] NIEMANN, K. D.: *Von der Unternehmensarchitektur zur IT-Governance*. Springer, 2005.

[21] NYSTROM, R.: *Game programming patterns*. Genever Benning, 2014.

[22] REBELSKY, S. A.: *A Brief History of Programming Languages*, 1999.

List of Figures

List of Tables

List of Code

List of Abbreviations

ABC Alphabet

WWW world wide web

ROFL Rolling on floor laughing

A Anhang A

B Anhang B