

# A Tale of two Languages

.. oder wie man aus C++11 eine produktive Anwendungsprogrammiersprache macht.

von Marius Elvert

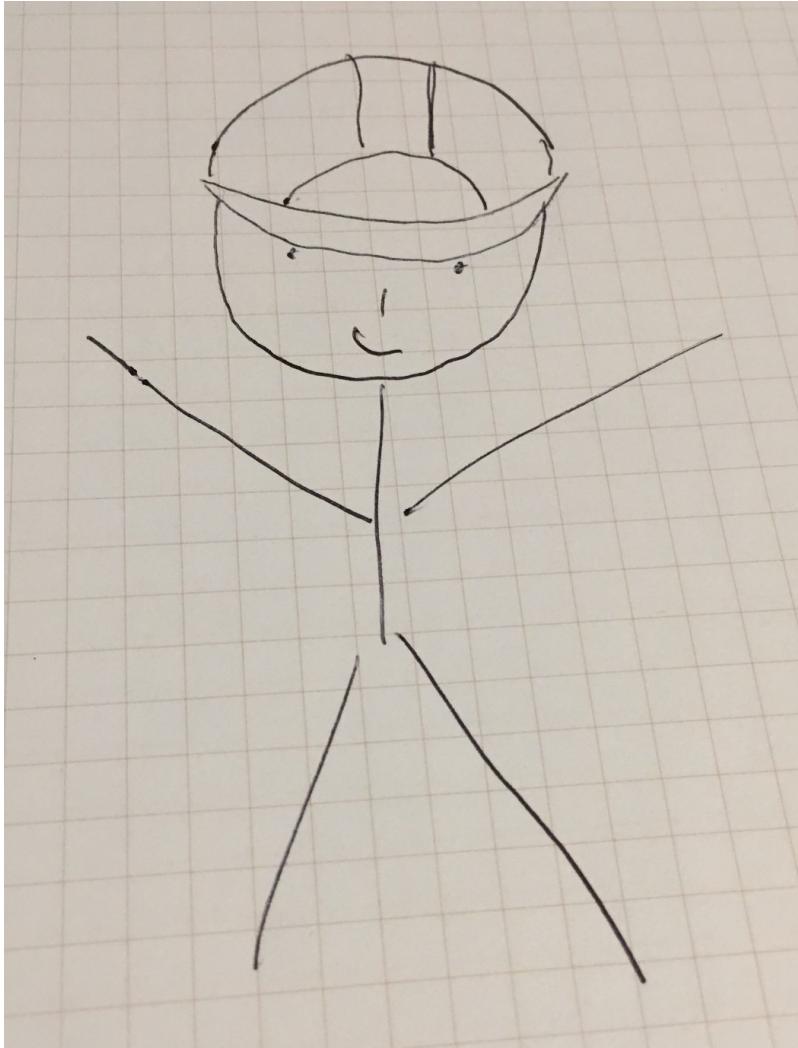
# Wer bin ich?

Hello, ich bin Marius!



Softwareeschneiderei

# Bob



# Was sagen die Experten?

## Stroustrup

**“Within C++, there is a much smaller and cleaner language struggling to get out”**

## Sutter

**“C++ is a fresh new language, simpler than ever.  
Sometimes we are addicted to too much complexity”**

# Problem erkannt?

## Modern C++: Reasonable Default Advice

	Cheap or impossible to copy (e.g., int, unique_ptr)	Cheap to move (e.g., vector<T>, string) or Moderate cost to move (e.g., array<vector>, BigPOD) or Don't know (e.g., unfamiliar type, template)	Expensive to move (e.g., BigPOD[], array<BigPOD>)
Out		X f()	f(X&) *
In/Out		f(X&)	
In	f(X)	f(const X&)	
In & retain "copy"			

Summary of what's new in C++1x:

- ✓ Defaults work better

\* or return `unique_ptr<X>/make_shared_<X>` at the cost of a dynamic allocation

# Noch ein schönes Zitat

**Tony Van Eerd:**

***Like every C++ ?, answer: „it depends“***



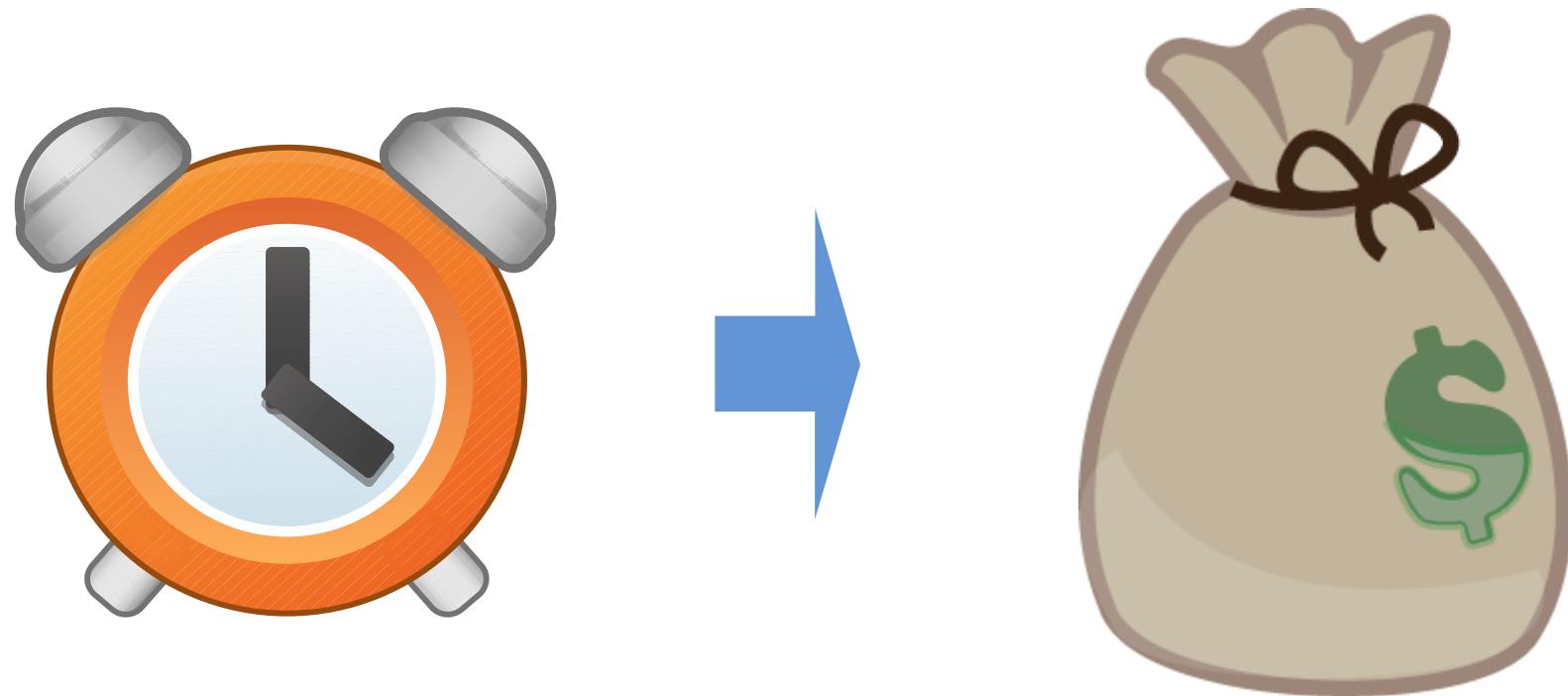
# Jetzt mal ganz objektiv...

„You see every little so and so, trying to be a language expert and discussing the subtleties of r-value references, or something like that, which does not actually matter in most code, unless you are writing a high performance library for use of others. They are all trying to be language lawyers, and are getting lost in technical details“

# Applicayshington



# C++ wirtschaftlich gesehen



# Immernoch zu komplex...

Auch C++11 ist in seiner Gesamtheit noch zu komplex, um bezüglich der Produktivität z.B. bei C#, Java, Python, Ruby und ähnlichen mitzuhalten.

Die Produktivitätsprobleme werden bei Legacy-Projekten, bei großen Projekten und großen Teams eher schlimmer.

# Andere Sprache für High-Level Code

- **Script Hosting**  
Python Extensions in C++ schreiben, und die High-Level Logik in Python.
- **Script Embedding**  
Lua in die eigene Applikation einbetten und dort die High-Level Logik Implementieren.

Kann sich lohnen, aber:

- hohes Investment
- polyglotte Entwickler

# Coding-Guidelines

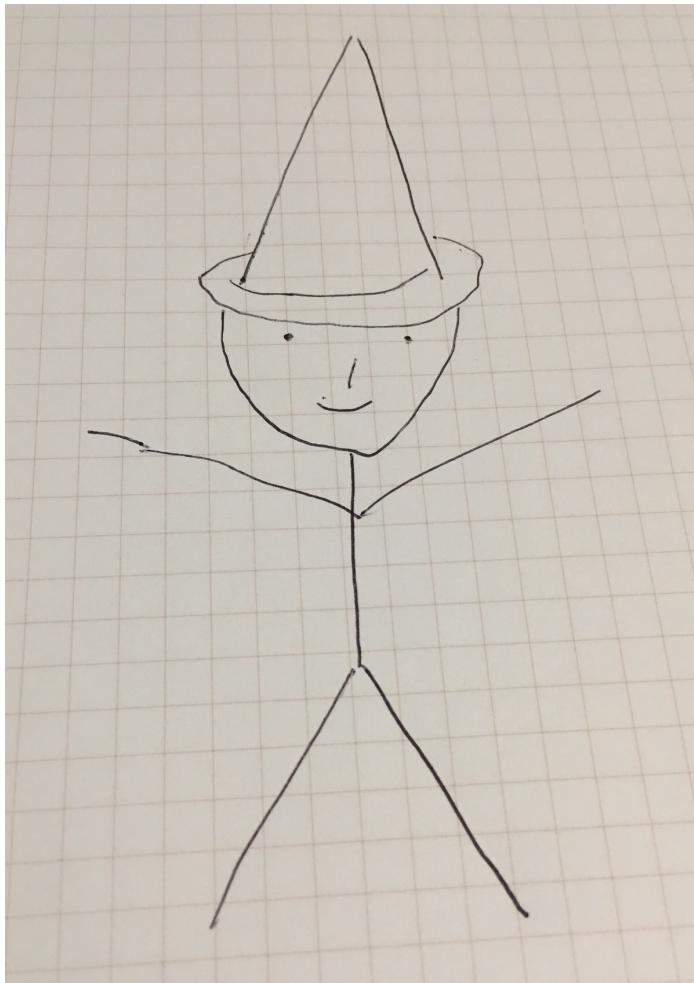
**Strikte Coding-Guidelines, die Komplexität, also z.B. Templates/TMP verbieten.**

**Positive Effekte und negative Effekte.**

- Zu großzügig
  - wenig Aussagekräftig
- Zu restriktiv
  - behindert bei der Arbeit

**Meistens treffen sie *die Linie* nicht.**

# Tim

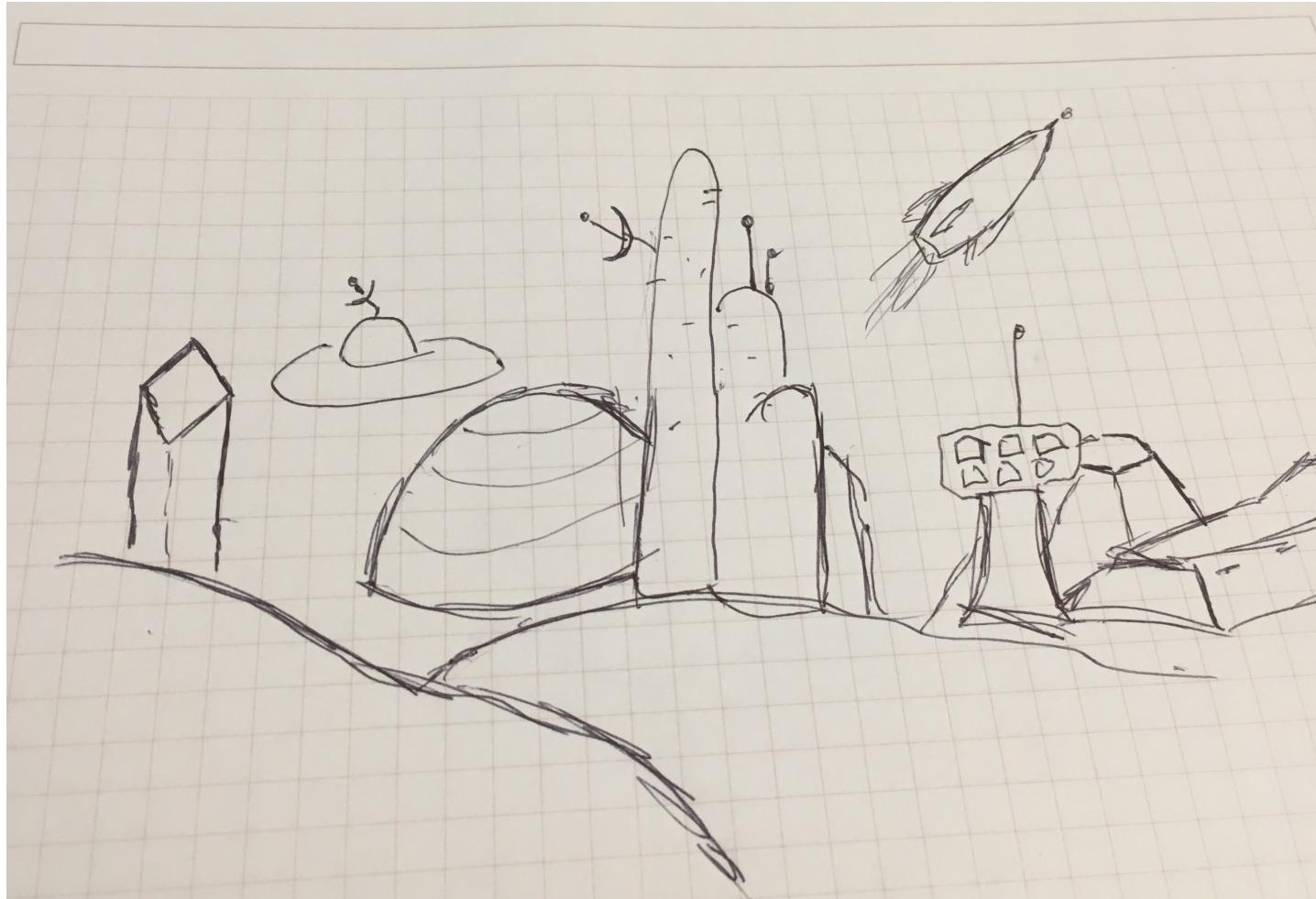


# A Federation of Languages

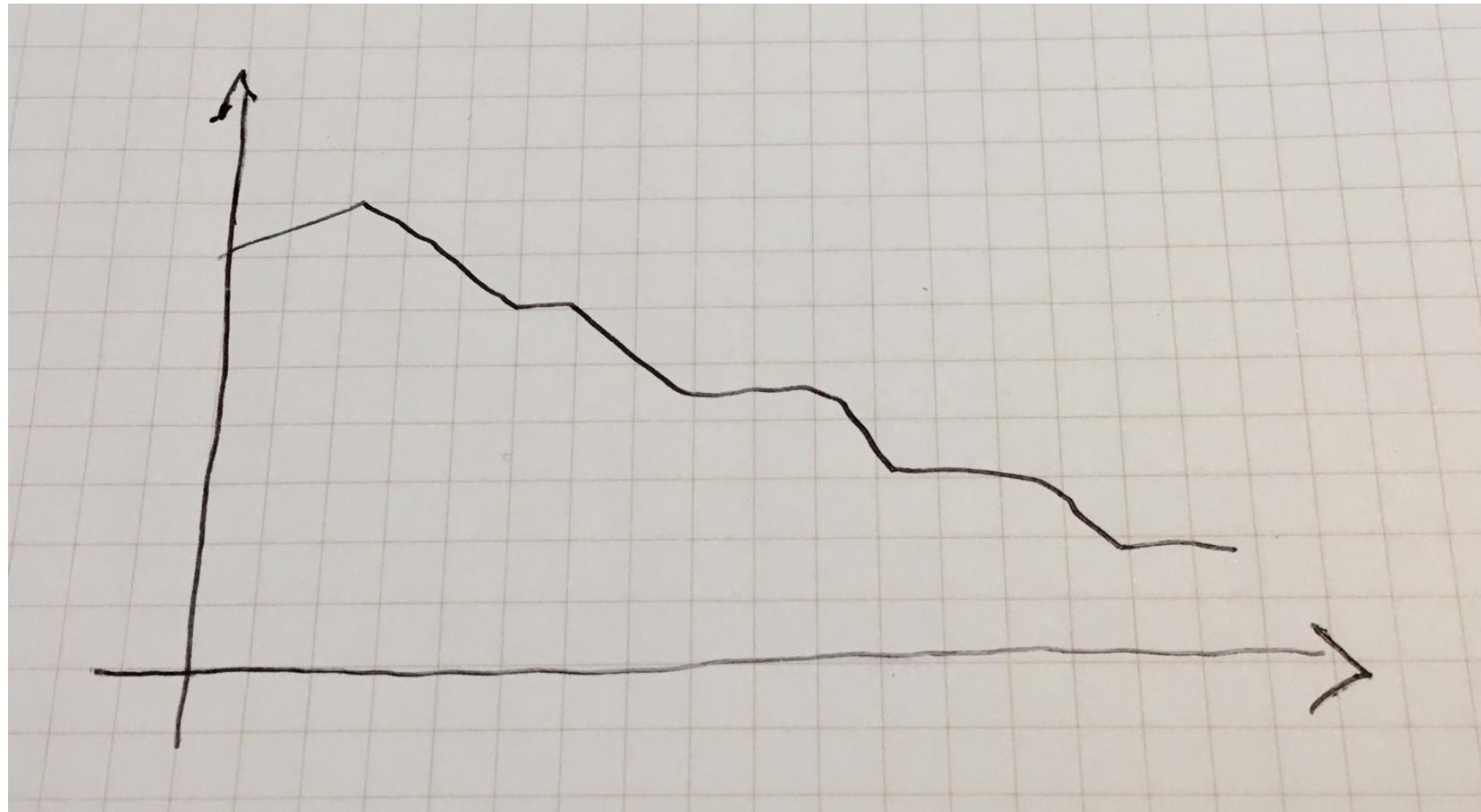
## C++ als 4 verschiedene Sprachen:

- C
- OO C++
- Template C++
- STL

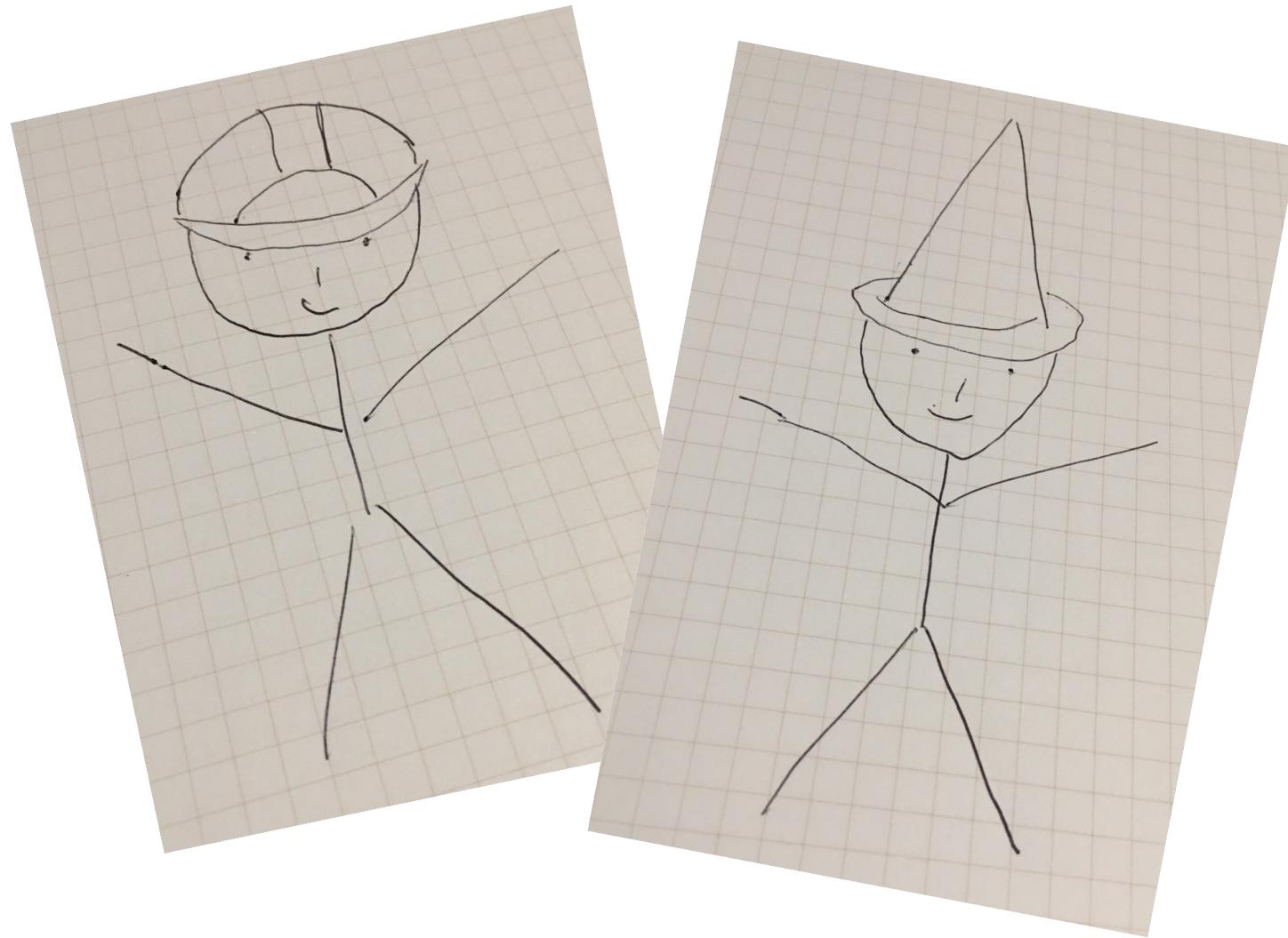
# Librararia



# Regression



# Joint Venture



# Alle 3 Zusammen

- Andere Sprache für High-Level Code
- Coding-Guidelines, die gut passen
- Eine Föderation von Sprachen

2 Sprachen-Modes

# Die Mission!

Macht den Teil der Sprache, in dem ihr eure Applikation schreibt, so ~~dumm~~ einfach wie möglich.



..aber benutzt die volle Macht von C++, um das zu erreichen.

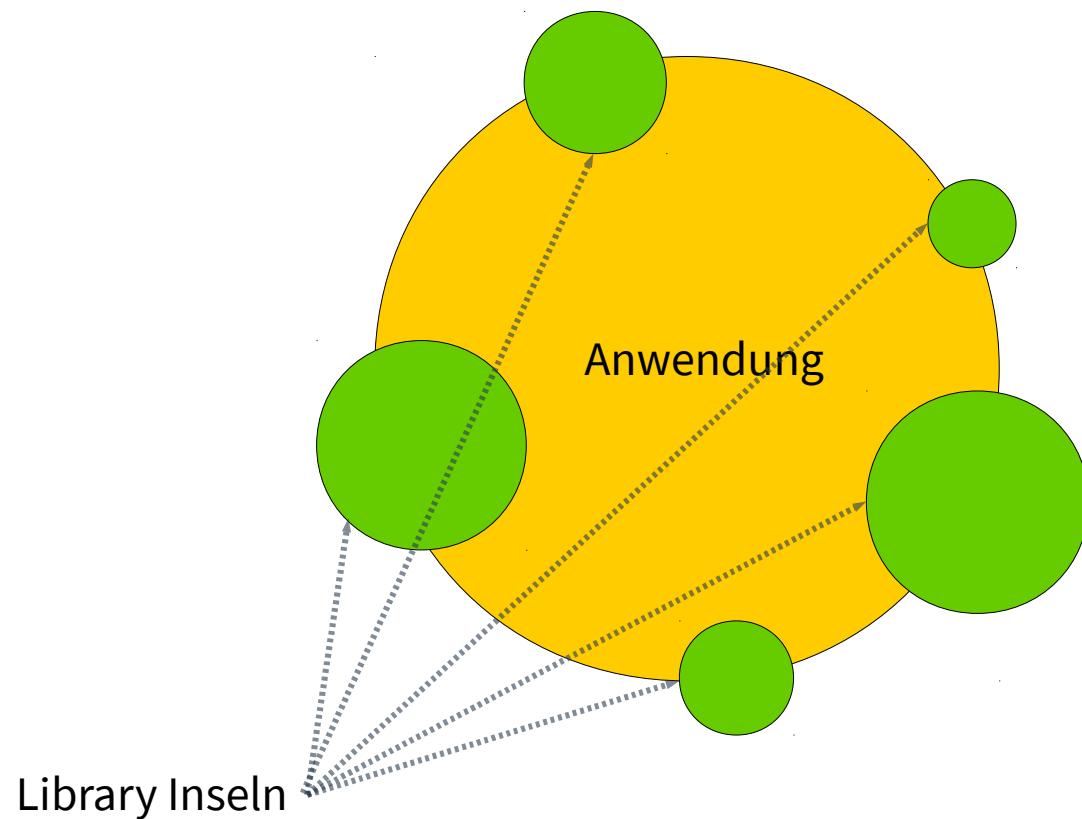
## Application Mode

### Schreib die Anwendung!

## Library Mode

Mach “Application Mode” einfacher, besser

# Strukturell



# Application Mode

## Agilität!

**Vermeidet alles, was euch als Gruppe über lange Sicht langsamer macht.**

## Einfacher Code!

**Erweitert die Sprache im Application Mode, damit dort die Probleme gelöst werden können.**

# Übersicht

	Library Mode	Application Mode
1	Alle Sprachfeatures	“gekapseltes” OO
2	ValueTypes und Funktional	Mechanismen, Protokolle, Referenzen
3	Flexible physikalische Struktur	Rigide physikalische Struktur
4	Benutzbarkeit	Erforschbarkeit
5	Stabil	Knetbar
6	Minimale Oberfläche	Minimale Komplexität
7	Erweiterung	Einheitlichkeit
8	Spezialisten u. Experten	Generalisten u. Anfänger

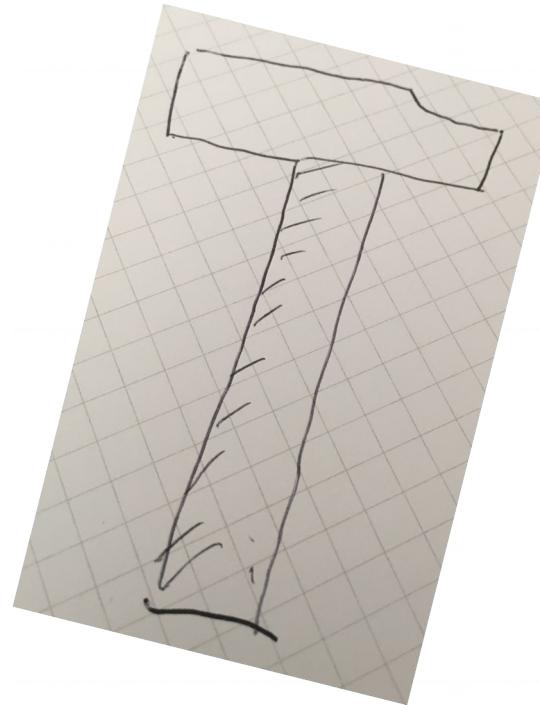
# #1 Features

## Library Mode: Features

# Spracherweiterung selbstgemacht

Der volle C++ Hammer.

Oft Template-heavy.



# Application Mode: Features

## Schreibt keine Subtilitäten

**Keine templates**

**Kein inline**

**Keine Überladung**

**OO im Sinne von C#, Java, Python etc..**



## #2 Taxonomie

# Was sind ValueTypes?

`vector<T>`

`string<T>`

`int`

`vec2, vec3, vec4`

`Image`

`Waveform`

→ Verhält sich in den wichtigen Aspekten wie ein built-in.

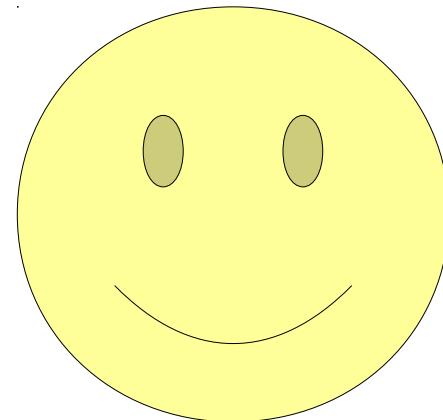
# ValueTypes Nachteile

- Koppeln extrem stark.
- Schwer zu erstellen.
- Hoher nicht-Code Overhead.



# ValueTypes Vorteile

- Selbsterweiterung.
- Hoher Produktivitätsgewinn.
- High-Performance.



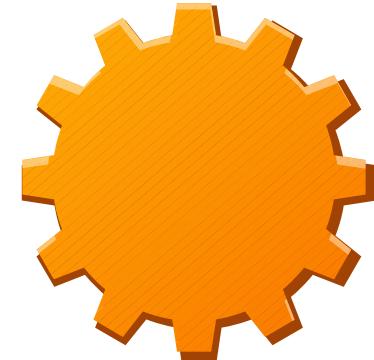
# Andere Dinge im Library Mode

Funktionales	Algorithmen, Hilfsfunktionen...
Container	Priority queues, Hash-container,
Value-Wrapper	optional, outcome, any, variant
DSLs	Metaparse, spirit, expression templates..
Metaprogramming	Hana, MPL, Fusion, Loki
Resourcen Manager	Locks, Netzwerkzugriff, Dateizugriff

Die Standardlib erledigt schon ziemlich viel...

# Application Mode: Mechanisms

- **Nicht-abstrakt**
- **Nicht-wert**
- **Das Fleisch auf den Knochen eurer Applikation.**
- **Use-Cases, Services, Patterns ...**
- **Kopierbarkeit wird in der Regel nicht benötigt**



# Protocols

- **Abstrakte Klassen / Interfaces**
- **Durch Mechanismen implementiert.**
- **Compiler Firewall.**
- **Im Test: Mocks, Fakes oder Spies.**
- **Programmiert wird gegen Protokolle und Libraries**

# Application Mode: Beste Freunde

```
namespace boost
{
    class noncopyable;

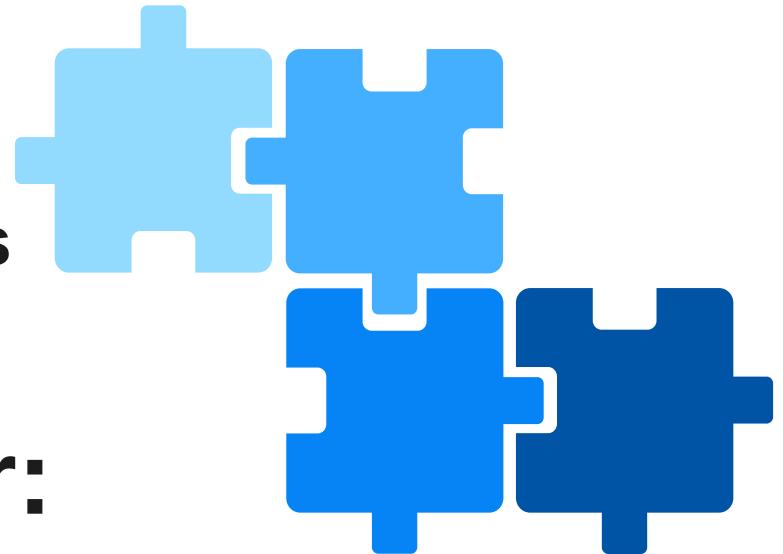
template <typename T> using Ptr = std::shared_ptr<T>;
```

## #3 Physikalische Struktur

# Flexible physikalische Struktur

**Logische Struktur:**

Klassen, Funktionen, Namespaces



**Physikalische Struktur:**

Dateien, Ordner, Artefakte (Libraries, Executables)

# Rigide physikalische Struktur

**1 Klasse**

**1 Header**

**1 Implementierungsdatei  
(1 Testdatei)**



**Gleicher Name für alle!**

## #4 Fokus

# Application Mode: Erforschbarkeit

- **lokale Erfassbarkeit**
- Namen sind essentiell
- Dokumentiert durch Code



# Library Mode: Benutzbarkeit

## Erweiterte Sprachfeatures

→ weniger erforschbar

Was hilft:

- Contracts !
- Kommunikation
- Dokumentation
- Reviews

## Contracts

**Spezifizieren:**

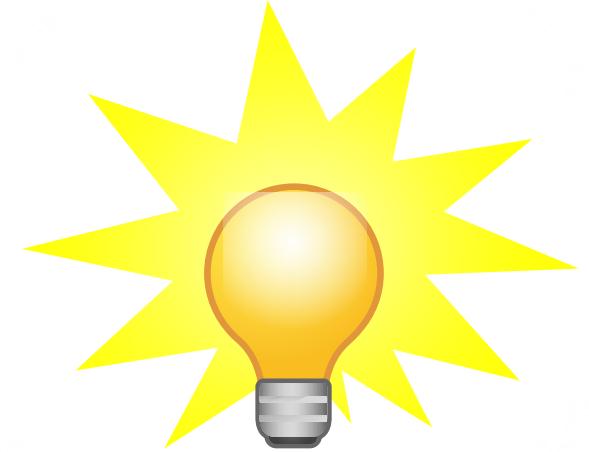
- Pre- und Postconditions
- Kopierverhalten
- Moveverhalten
- Laufzeitverhalten



## #5 Änderungsrate

# Library Mode: Stabiler Code

**Der stabilste Code ist im Library Mode.  
Viele nicht-funktionale Anforderungen.  
Hoher Lernaufwand.**



**Definiert gemeinsame Sprache.**

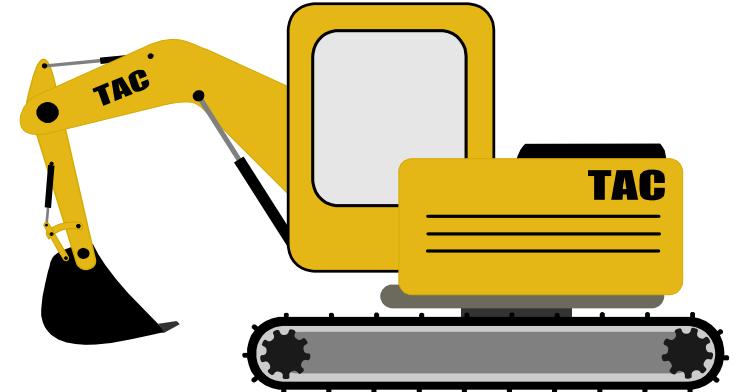
**Trade-off: Nur lokal benutzte Library**

# Application Mode: Knetbarer Code

**Guter Code fühlt sich an wie Knete**

**Änderung hat nur lokale Auswirkung**

- Auch radikale, destruktive Änderungen
- Alles muss leicht zu entfernen sein

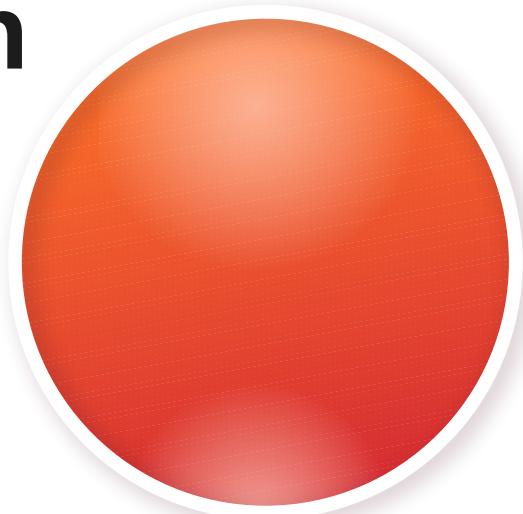


**Der konstante Aspekt liegt hier in der Architektur!**

# #6 Minimierung

# Library Mode: Minimiert Oberfläche

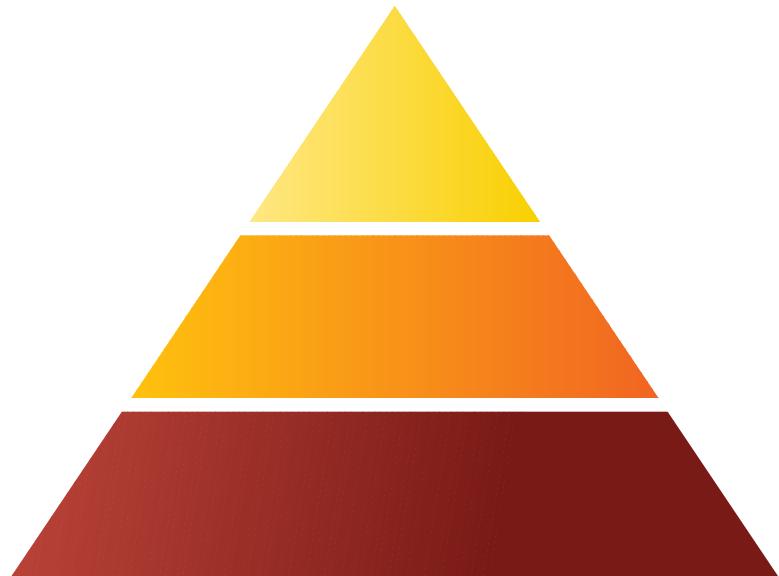
- Kleine scharf-definierte Interfaces
- Mächtige Funktionalität dahinter
- Für den 95% Fall designen



# Library Mode

## Fassaden

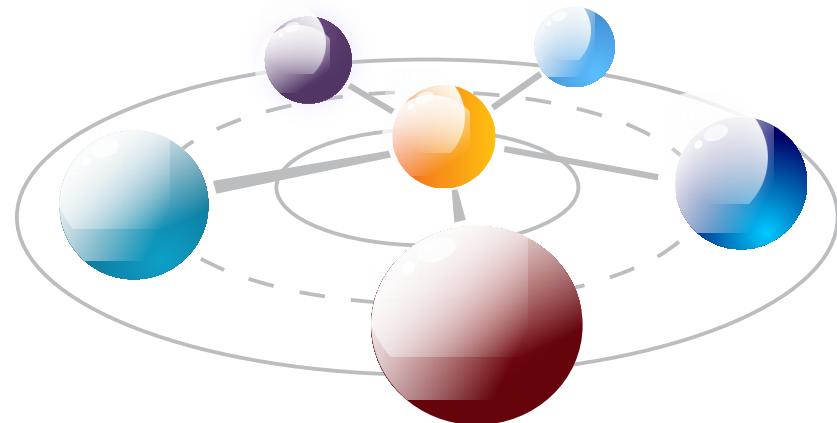
- kristallisieren sich aus
- optimieren, aber nicht kapseln
- gehören in den Library Mode



# Application Mode: Minimiert Komplexität

**Viel Inhalt, aber mit geringer Komplexität.**

- Passende Architektur
- Vermeidet Tiefe



**Kleinere “Treppen” im Abstraktionsniveau.**

## #7 Vokabular

# Application-Mode: Einheitliches Vokabular

## Einheitlichkeit

→ Dumb it down!

- Ausdrucksstarke Namen
- Vereinheitlichtes Vokabular
- Kleines Vokabular
- Stringente Namenskonvention
- Langweilig ist gut!



# Library Mode: Vokabular Erweitern

**Wortschöpfung ist erlaubt**

→ Aber Zweckgebunden!

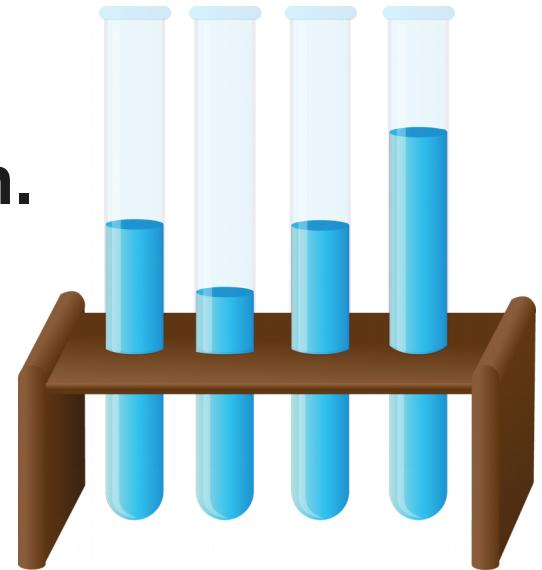
- **push**
- **vector**

**Konsistenz muss es sein!**

## #8 Qualifikation

# Library Mode: Expertensache

- Spezialisten sind gefragt.
- Code darf komplizierter sein.
- Schnittstelle darf auch komplizierter sein.
- Anwender nicht vergessen.
- Busfaktor nicht vergessen.
- Spider-Man Prinzip.



Library-Implementer haben Vorbild-Funktion!

# Application-Mode: Almende

- Schnelles Onboarding
- Hohe Agilität
- Hohe Produktivität
- Produzieren Arbeit für den Library Mode
- Problem: Tragik der Almende



The End  
Fragen?

