Design Principles vs. Performance

Wie mein Wissen über Interna und Performance das Design meiner Anwendungen verändert hat – ein anekdotischer Vortrag

Advanced Developers Conference 2021
30.11.2021

Advanced Developers Conference 21
Development for Professionals!

ppedv

Agenda

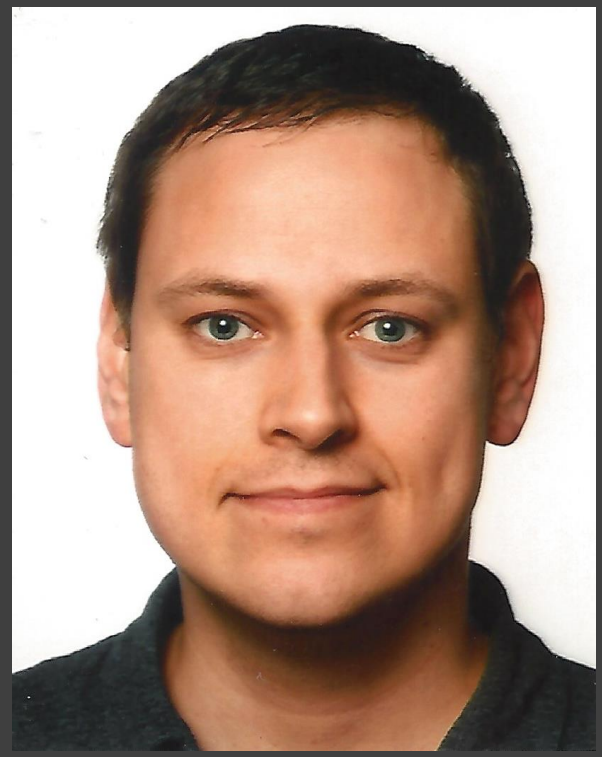
- Wie habe ich bis 2015 meine Software-Applikationen gestaltet?
- Drei Beispiele für Performance und Interna
- Vereinbarkeit mit Prinzipien, Patterns und Best Practices

Über mich

Kenny Pflug

- Tech Lead bei Synnotech

- Twitter: @feO2x
- GitHub: feO2x
- YouTube: youtube.com/c/kennypflug



Anno 2015…

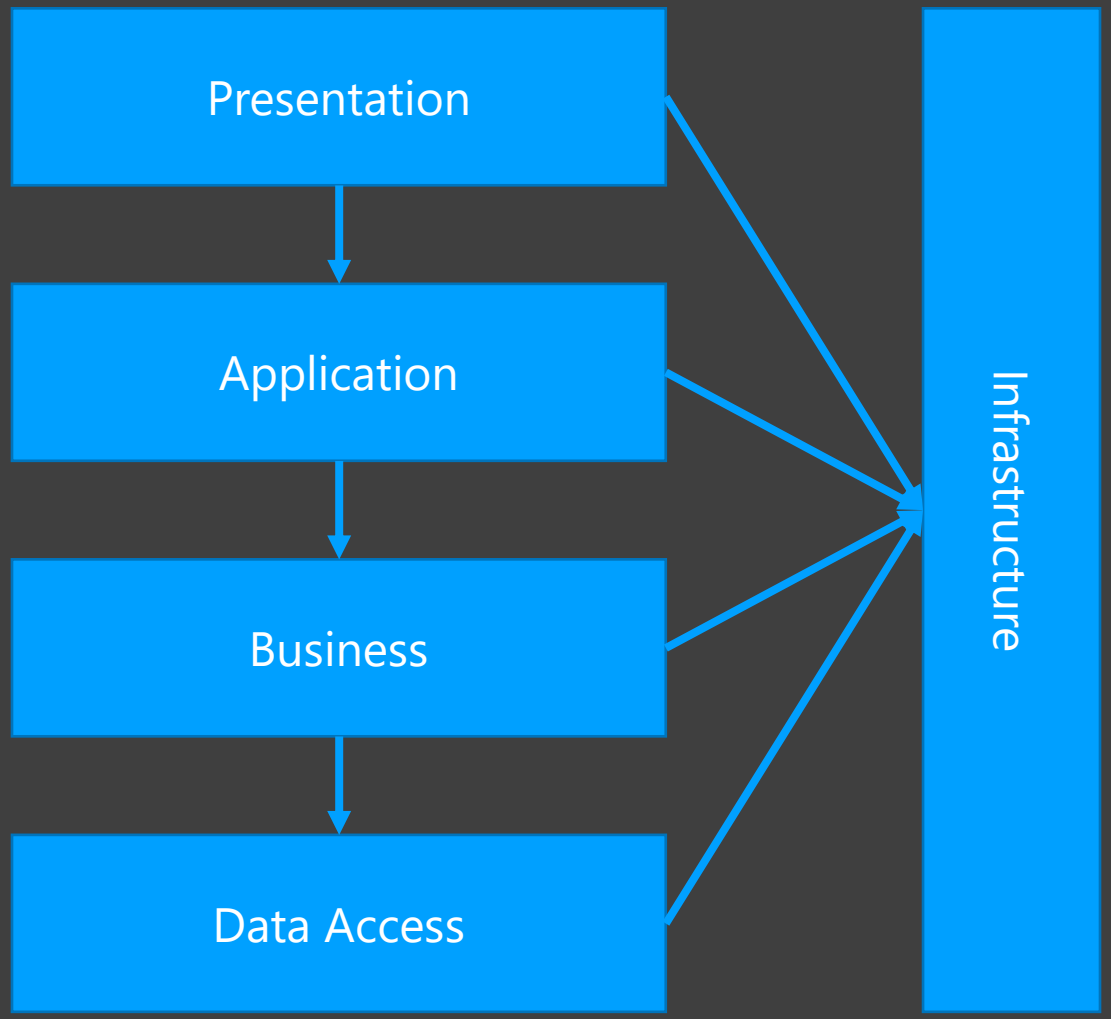
Wie ich bis 2015 meinen C# Code gestaltet habe

- Viele kleine Klassen, welche jeweils genau eine Aufgabe übernehmen
- Interfaces / abstrakte Basisklasse zwischen Aufrufer und Aufgerufenen
- Objektgraphen werden über Dependency Injection (DI) aufgelöst – normalerweise mithilfe eines DI Containers
- If-Else- oder Switch-Blöcke werden ersetzt durch Objekte mit Abstraktion
- Test Driven Development
- Einsatz etablierter Design Patterns

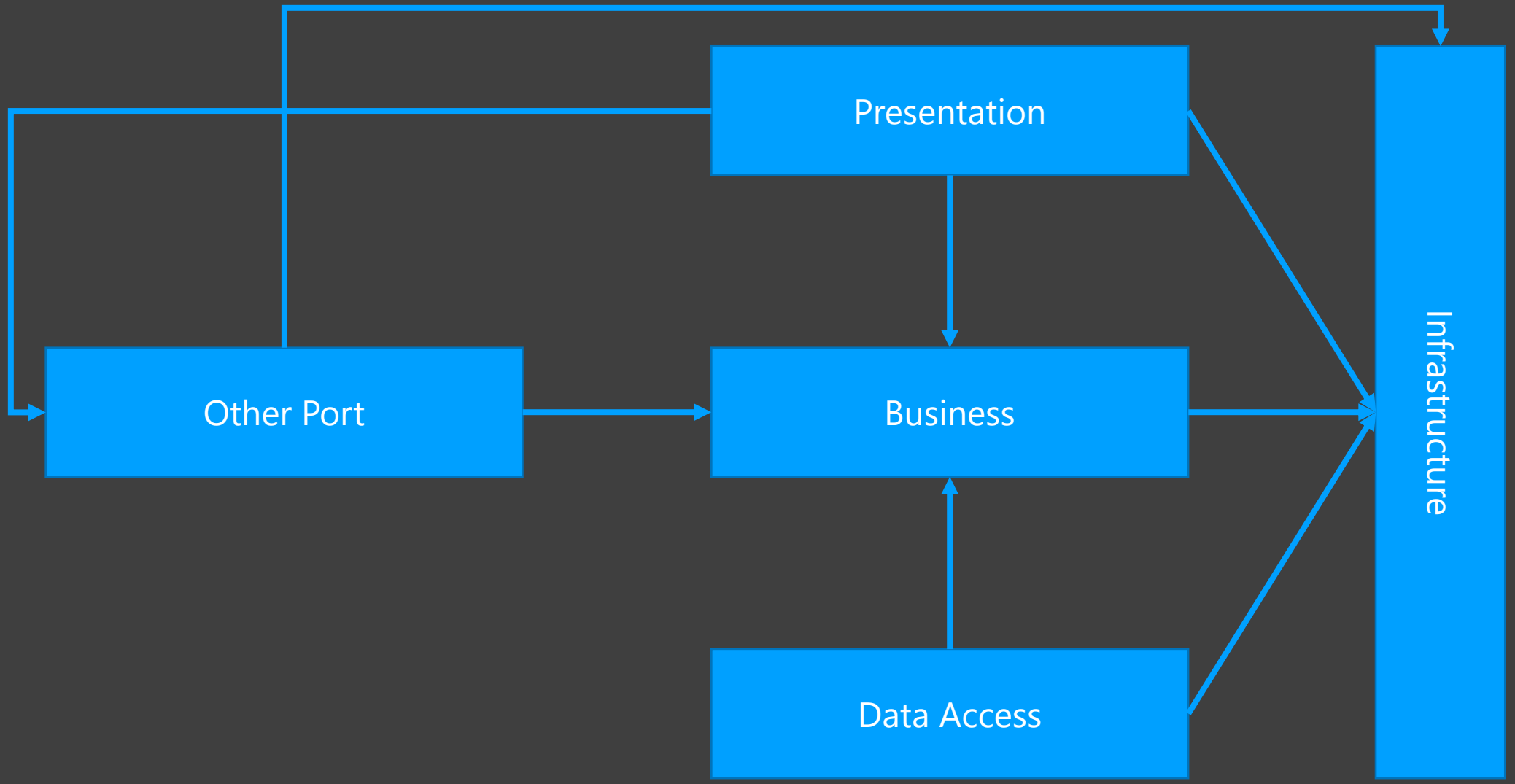
2015 – Design Patterns

- Factory, Abstract Factory
- Builder
- Singleton
- Prototype
- Adapter
- Composite
- Decorator
- Facade
- Proxy
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- Strategy
- Visitor
- Immutable Objects
- Arrange – Act – Assert (– Cleanup)
- Dummy, Stub, Spy, Mock
- Model – View – View Model
- Model – View – Controller
- Object Pooling
- und viele mehr…

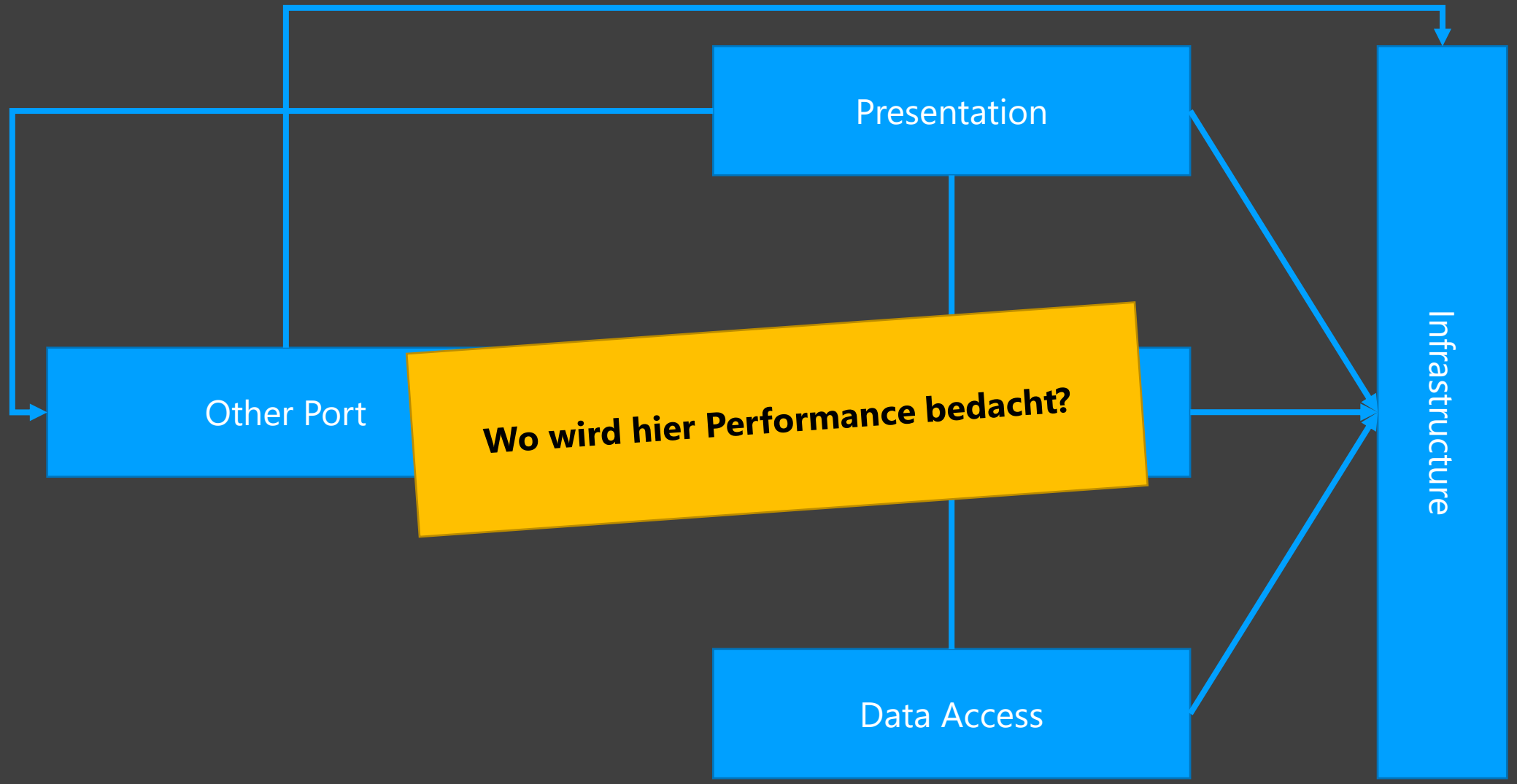
2015 – Software Architectures: N-Layer / N-Tier



2015 – Software Architectures: Ports and Adapters / Onion Architecture

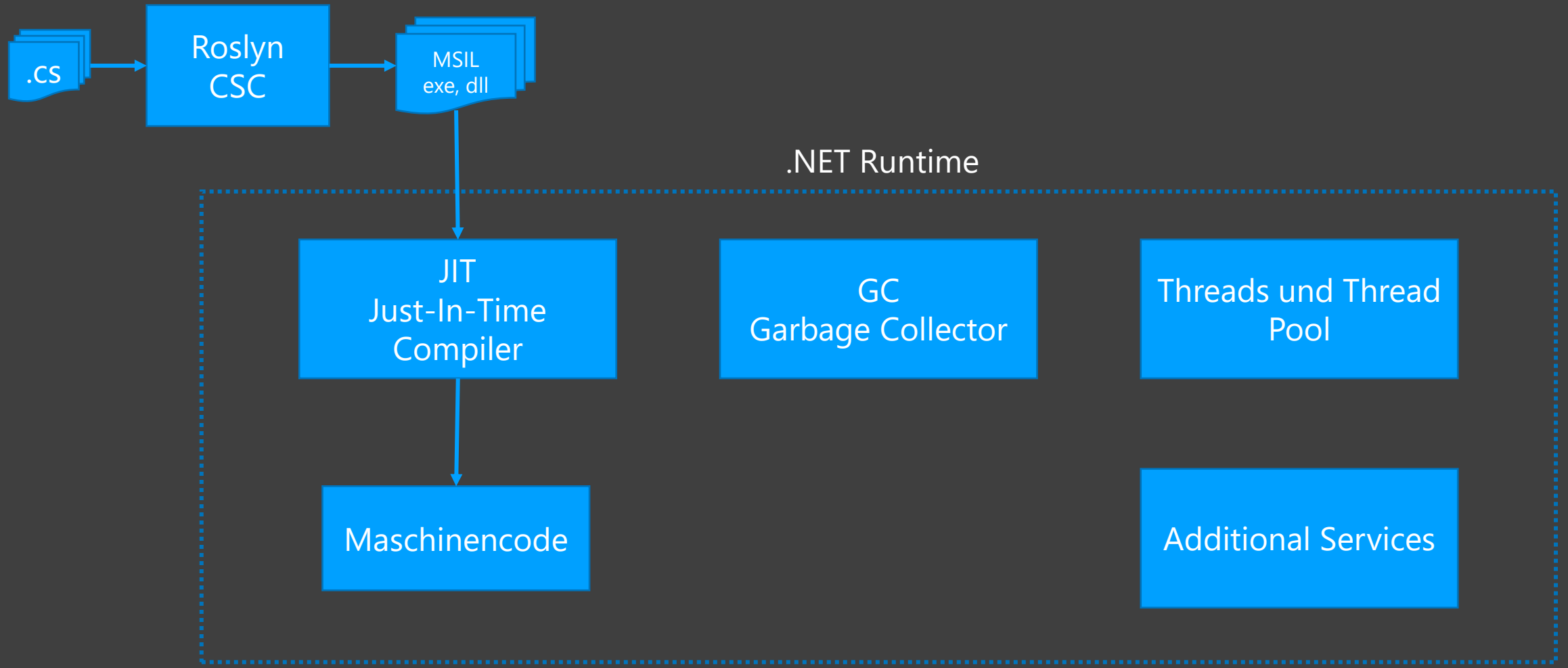


2015 – Software Architectures: Ports and Adapters / Onion Architecture

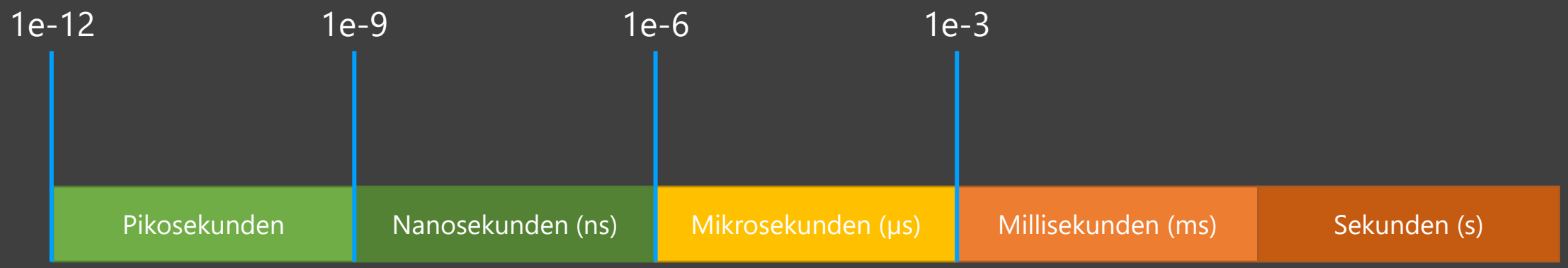


Performance of Everyday Things

Noch ein paar Worte zur .NET Runtime

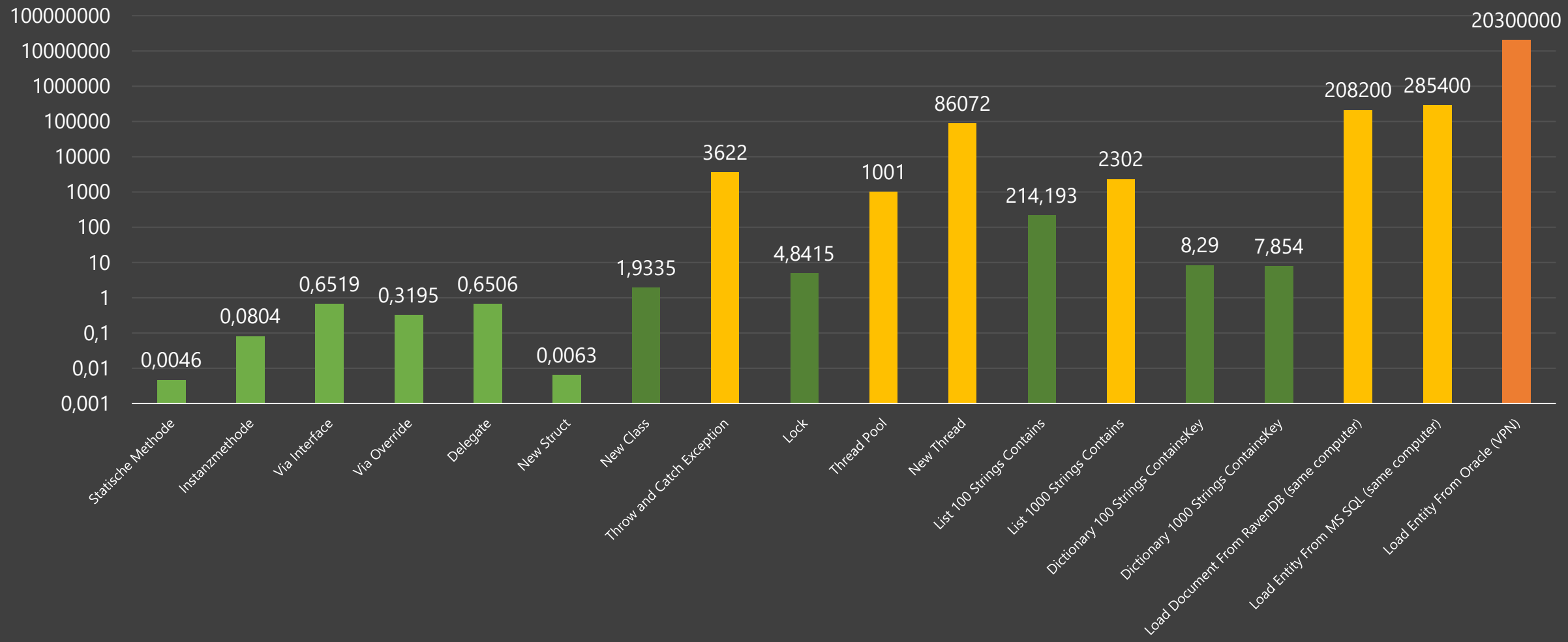


Die Zeit, die Zeit...



Ein paar Zeiten zum Vergleich (Werte in Nanosekunden)

BenchmarkDotNet=v0.13.1, OS=Windows 10.0.19044.1348 (21H2)
AMD Ryzen 9 5950X, 1 CPU, 32 logical and 16 physical cores
.NET SDK=6.0.100
[Host] : .NET 6.0.0 (6.0.21.52210), X64 RyuJIT
Job-FTZBOG : .NET 6.0.0 (6.0.21.52210), X64 RyuJIT



Welche Schlüsse ziehen wir daraus?

- Multithreading macht erst ab einer gewissen Anzahl Operationen Sinn
- Neue Threads sind teuer, der Thread Pool verwaltet automatisch diese für uns
- I/O ist deutlich teurer als In-Memory-Operationen
- Absolute Schnelligkeitswerte sind an die jeweilige Hardware und Plattform gebunden, wichtig sind die relativen Ergebnisse zueinander

Wie gut skaliert mein Code?

Skalierbare Services

Was ist Asynchrones Programmieren?

Asynchrones Programmieren bedeutet, dass man an bestimmten Stellen in seinem Source Code Funktionen aufruft, **deren Ergebnis** (Rückgabewerte oder Seiteneffekte) **beim Rücksprung zum Aufrufer noch nicht fertigberechnet** sind. Das Ergebnis wird dem Aufrufer später mitgeteilt (typischerweise über einen Event-Mechanismus). Währenddessen kann der **aufrufende Thread andere Berechnungen durchführen**.

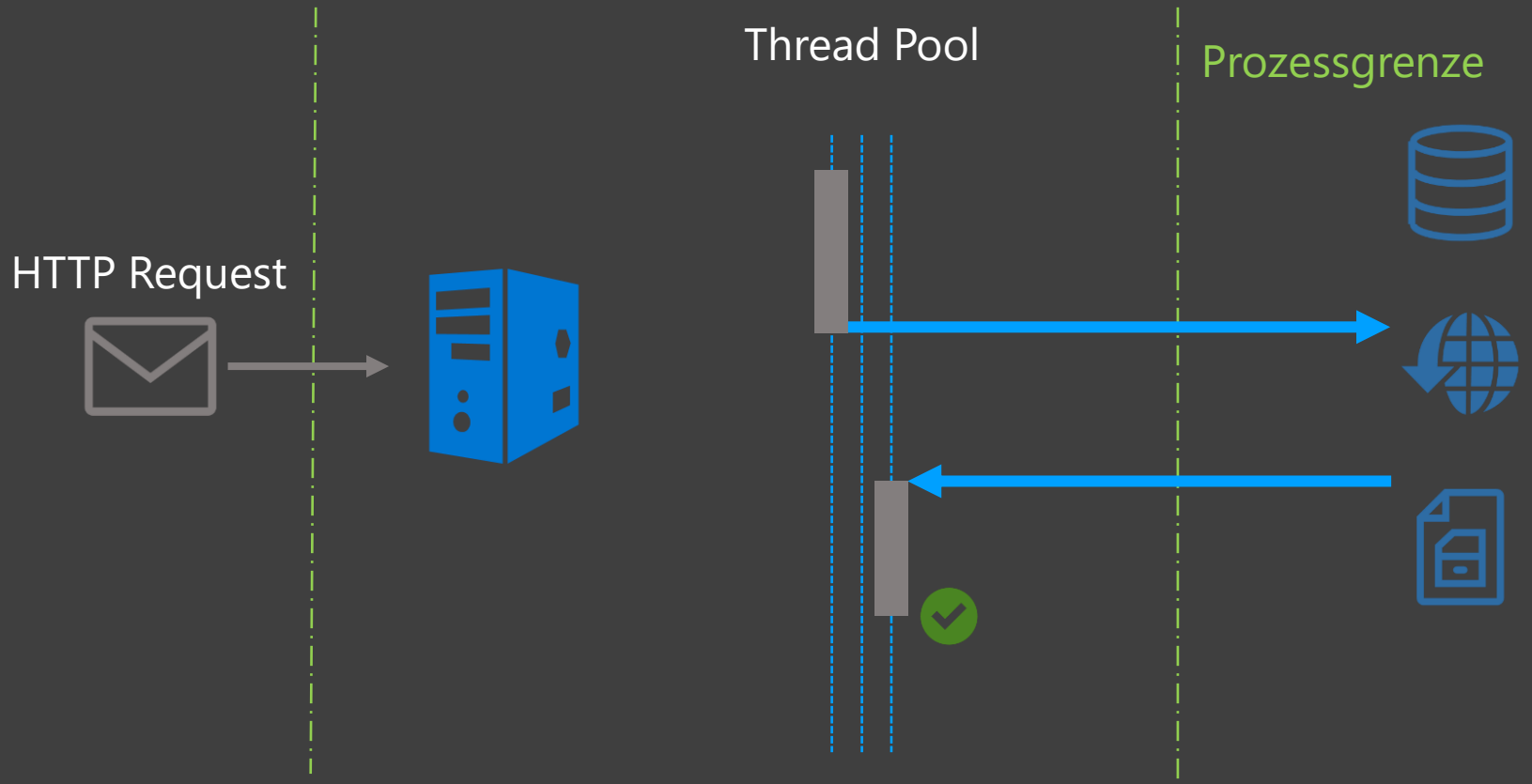
Async Multithreading
(CPU-Bound)

Async I/O
(I/O-Bound)

Warum ist Async I/O wichtig?

- Wenn Sie I/O über synchrone APIs ausführen, blockiert der aufrufende Thread, bis das Ergebnis da ist.
- Wenn der Thread Pool blockierte Threads sieht, erzeugt er neue
- Threads sind teuer

Threading in Services in .NET



Overhead von async await

- Der genaue Overhead ist schwierig zu bestimmen.
- Wenn man eine Methode async macht und diese tatsächlich Async Compute oder Async I/O ausführt, ist man aber mindestens im Bereich µs.

Method	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated
Increment	0.2666 ns	0.0204 ns	0.0190 ns	1.00	0.00	-	-	-	-
IncrementAsync	1,181.3835 ns	18.4261 ns	20.4805 ns	4,465.72	356.49	0.0534	-	-	256 B

WPF – Internals

Collection Views in WPF

Principles, Patterns & Practices vs. Performance & Internals

Achtung: subjektiv

Was wir aus den vorherigen Abschnitten lernen sollten

- Eindeutig unterscheiden zwischen I/O und In-Memory Operationen
- I/O sollte asynchron ausgeführt werden, um UI Freezes und unnötige Thread-Pool-Allokationen zu vermeiden
- Unnötigen I/O vermeiden
  - Mehrere Abfragen in eine zusammenfassen, falls möglich
  - Fail-Fast-Prinzip
- Unnötige Objektallokationen vermeiden – Indirektion nur dann einsetzen, falls notwendig
- Auch stark gekoppelter Code kann (leicht) automatisiert getestet werden, sofern dieser ausschließlich In-Memory läuft
- Speicherabbilder helfen beim Design von Mengengerüsten. Wichtig: Messen!

Neues Programmierprinzip: LTI

Learn-The-Internals Principle (LTI)

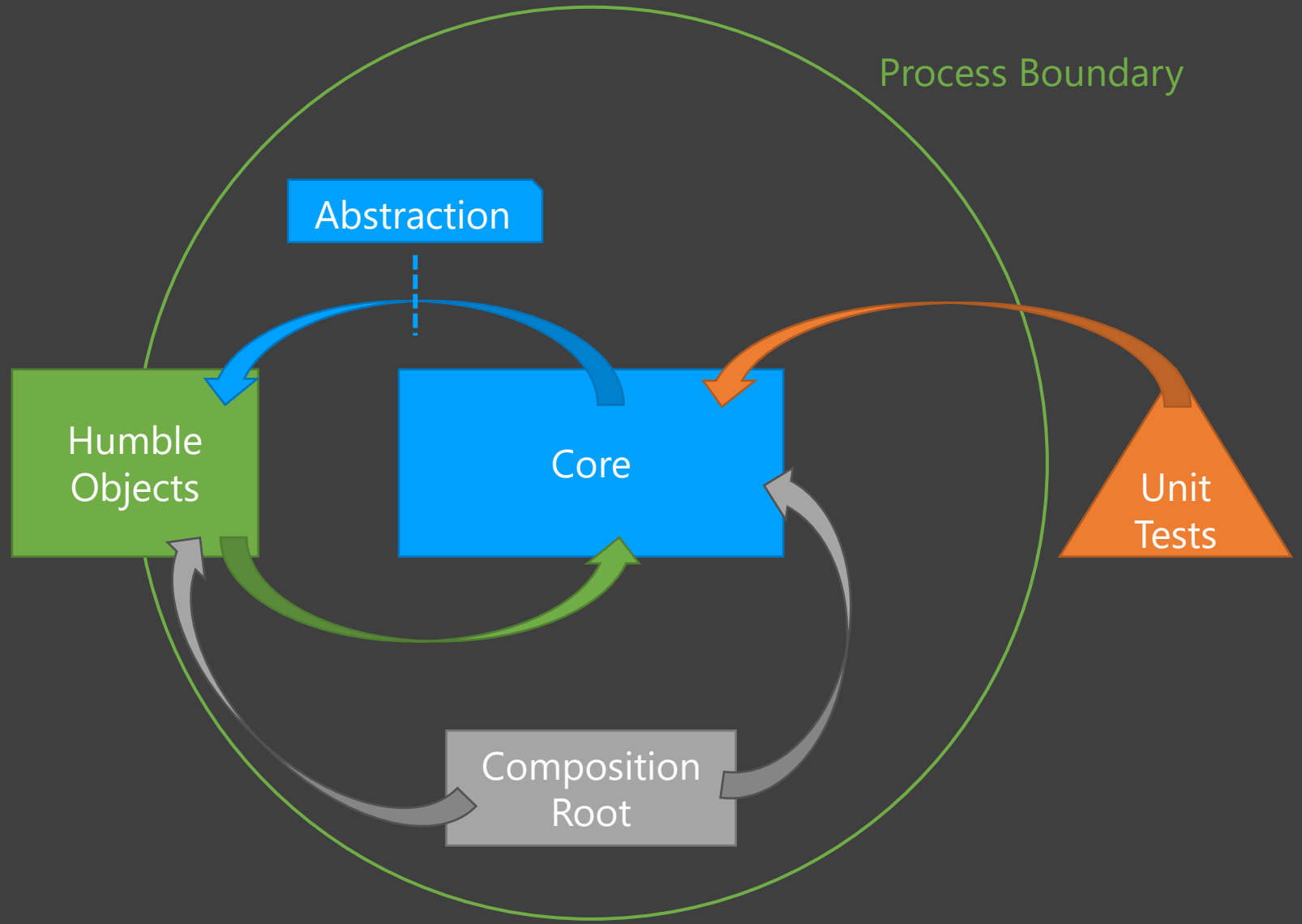
Setze dich mit den internen Mechanismen der Runtimes und Frameworks / Bibliotheken, die du einsetzt, auseinander und verstehe, wie sie wiederkehrende Probleme lösen (zumindest auf der obersten Abstraktionsschicht). Mache ausfindig, welche Aufruf-Muster Probleme erzeugen können und stelle sicher, dass diese im aufrufenden Code vermieden werden.

Neues Programmierprinzip: RPB

Respect-the-Process-Boundary Principle (RBP)

Unterscheide zwischen In-Memory Operationen und I/O Operationen, da letztere deutlich höhere Ausführungszeiten haben. I/O Operationen sollten über asynchrone APIs umgesetzt werden, um blockierende Threads zu vermeiden.

CHUC: Core – Humble Objects – Unit Tests – Composition Root



Quellen

- Matt Waren: The 68 things the CLR does before executing a single line of your code (*)
- Konrad Kokosa: Pro .NET Memory Management
- Jeffrey Richter: Advanced Threading in .NET
- Daniel Palme: IoC Container Benchmark - Performance comparison
- Joseph Albahari: Threading in C#
- John Skeet: Asynchronous C# 5.0
- Robert C. Martin: Agile Principles, Patterns, and Practices in C#
- Andrey Akinshin: Pro .NET Benchmarking – The Art of Performance Measurement
- Martin Fowler: Patterns of Enterprise Application Architecture
- Mark Seemann: Dependency Injection in .NET

Vielen Dank!

Haben Sie Fragen?