



TELIS FINANZ

Talk about Code
2024-11-21

**Dependency Injection und das
Dependency Inversion Principle**

Kenny Pflug
Senior Dev DMS

KENNY PFLUG

- .NET seit 15 Jahren
- Seit 2014 baue ich Web Apps basierend auf ASP.NET (Core) und verteilte Systeme
- Internals wie Memory Management, Threading und Asynchronous Programming, Performanceoptimierung
- Auch ein bisschen Frontend, hauptsächlich React und Angular sowie XAML-basierte Apps

FEATURES ENTWICKELN

...mit prozeduraler Herangehensweise

Neues Feature bitte

- Lese einen Buchstaben von der Konsole ein
- Wenn der Nutzer ESCAPE gedrückt hat, dann beende das Programm
- Ansonsten gib den Buchstaben auf der Konsole aus

LET'S CODE

Feature Request 1
in prozeduralem Stil

```
133 background: url(...);
134 display: inline-block;
135 width: 12px;
136 height: 14px;
137 float: left;
138 margin: 2px 7px 0 0;
139 }
140 .phone{
141   background: url(../img/phoneico.png) no-repeat center;
142   display: inline-block;
143   width: 20px;
144   height: 18px;
145   float: left;
146   margin: 2px 8px 0 0;
147 }
```

foo@private: var/ folders/11/q702vb87ng3c885drcfpam0000gp/T/8c98b21-8e7e-4e01-bb01-8c4e1b88eb11/ctrl/ctrl/style.css/

MEIN PRODUCT MANAGER SAGT

Mach mal noch ein neues Feature

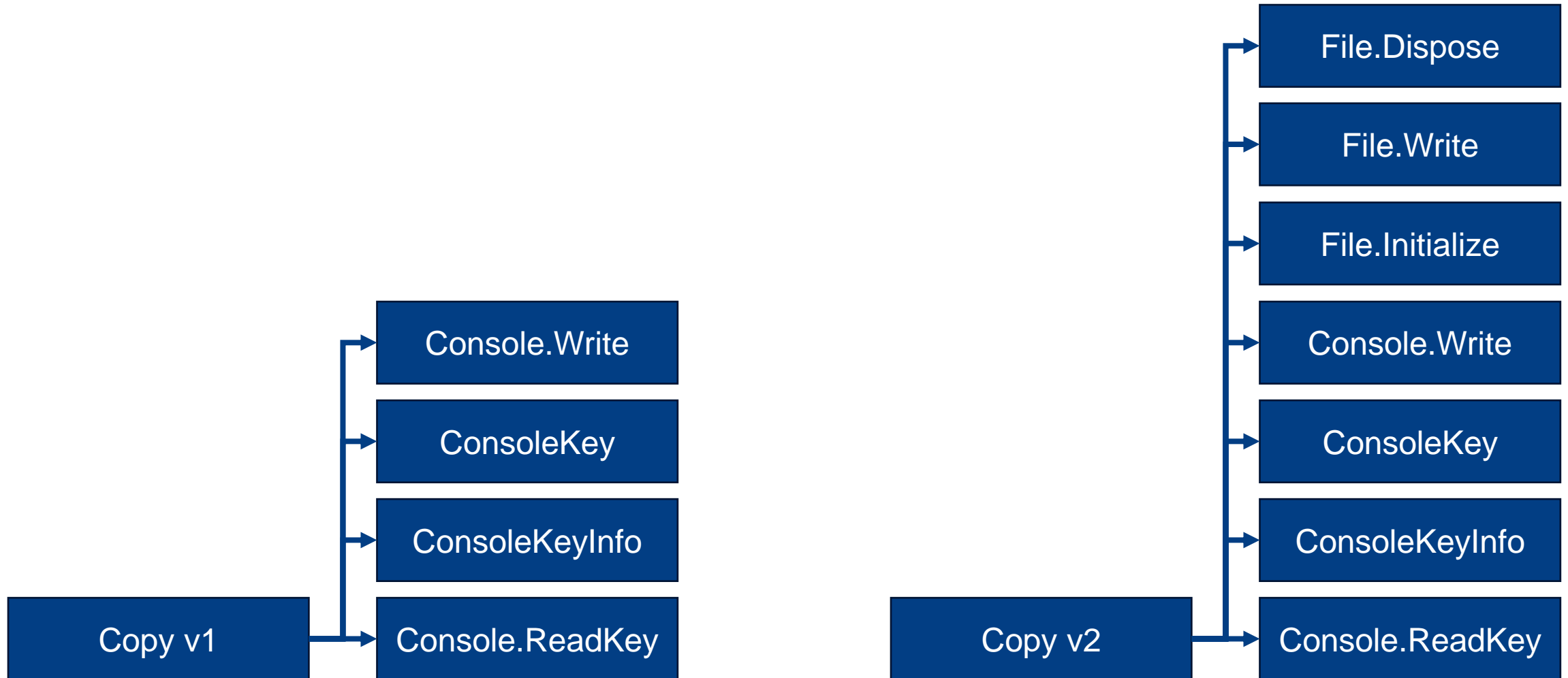
- Erlaube dem Nutzer, in eine Datei zu schreiben anstatt auf die Konsole


LET'S CODE

Feature Request 2
in prozeduralem Stil

```
133 background: url(...);
134 display: inline-block;
135 width: 12px;
136 height: 14px;
137 float: left;
138 margin: 2px 7px 0 0;
139 }
140 .phone{
141 background: url(../img/phoneico.png) no-repeat center;
142 display: inline-block;
143 width: 20px;
144 height: 18px;
145 float: left;
146 margin: 2px 2px 0 0;
147 }
```

DAS PROBLEM MIT DEN ABHÄNGIGKEITEN



A close-up shot of a middle-aged man with a mustache, looking slightly to the right. He is wearing a striped shirt and a grey sweater. He is holding a glass of beer with a thick head of foam. The background is blurred, showing what appears to be a window and some indoor decor.

Es muas a
Abhängkeiten gebn.
Aber es werdn abe
mehra...

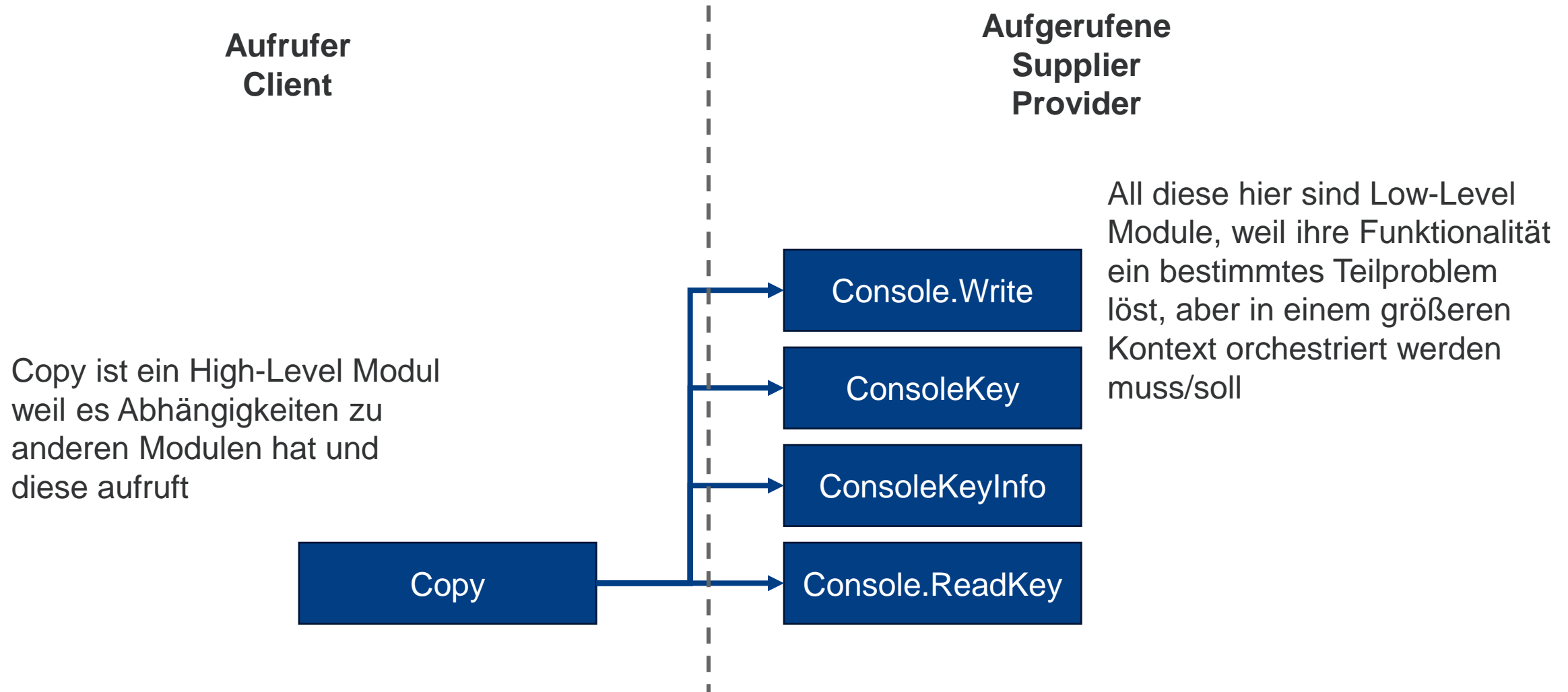
DAS DEPENDENCY INVERSION PRINCIPLE

Wer kennt die Definition?

Das DIP („D“ in den SOLID-Prinzipien) besteht aus zwei Sätzen

1. High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
2. Abstractions should not depend on details. Details should depend on abstractions.

WAS SIND HIGH-LEVEL UND LOW-LEVEL MODULE?



WAS DAS DIP VORSCHLÄGT



In objektorientierten Sprachen benutzen wir hier standardmäßig Interfaces oder abstrakte Basisklassen, weil sie polymorphe Aufrufe ermöglichen. Anderen Arten der Indirektion sind aber auch möglich (z.B. Funktionszeiger, Higher-Order Functions, Delegates).

LET'S CODE

Refactoring mit dem DIP



```
133 background: url(...);  
134 display: inline-block;  
135 width: 12px;  
136 height: 14px;  
137 float: left;  
138 margin: 2px 7px 0 0;  
139 }  
140 .phone{  
141 background: url(../img/phoneico.png) no-repeat center;  
142 display: inline-block;  
143 width: 20px;  
144 height: 18px;  
145 float: left;  
146 margin: 2px 2px 0 0;  
147 }
```

Was sollte ich mir nochmal merken?

1. Zwischen zwei Modulen sollte man Abstraktionen einsetzen, um den Aufrufer vom Aufgerufenen zu entkoppeln.
(Low Coupling vs. High Coupling im Kontext Softwaredesign)
2. Abstraktionen sollen auf den Aufrufer zugeschnitten werden, nicht auf den Aufgerufenen. Low-Level Details haben in Interfaces nichts verloren - ggf. müssen neue Typen zur Datenübergabe erstellt werden.
3. Dependency Injection wird genutzt, um Low-Level Instanzen and eine High-Level-Instanz zu übergeben. High-Level Instanzen sollten keine konkreten Typen kennen, folglich darf von ihnen auch kein Konstruktor im Scope des High-Level-Moduls aufgerufen werden.
4. Daraus folgt: ein High-Level Module kümmert sich nicht um die Lebenszeit eines Low-Level Modules. Dies wird im Composition Root gemacht.
5. Unterschiedliche Arten von Dependency Injection: Constructor Injection, Method Injection, Property Injection (in C#)
6. Der Composition Root besteht aus den Phasen Register, Resolve und Release.

**Versuche, die Anzahl der Abhängigkeiten stabil zu halten.
Halte deine Abstraktionen zu Low-Level Modulen stabil.**



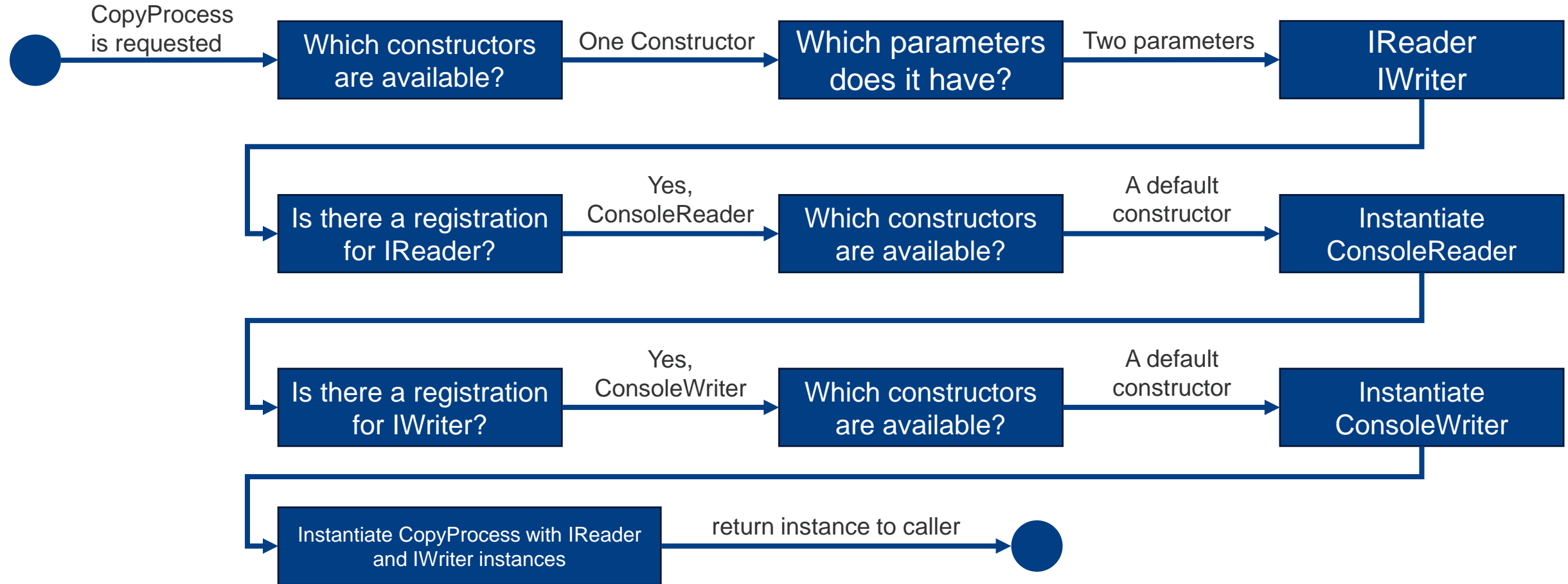
DEPENDENCY INJECTION CONTAINER

LET'S CODE

DI Container im Composition Root nutzen

```
133 background: url(...);
134 display: inline-block;
135 width: 12px;
136 height: 14px;
137 float: left;
138 margin: 2px 7px 0 0;
139 }
140 .phone{
141   background: url(../img/phone1co.png) no-repeat center;
142   display: inline-block;
143   width: 20px;
144   height: 18px;
145   float: left;
146   margin: 2px 2px 0 0;
}
```

WIE LÖST EIN DI CONTAINER EINEN OBJEKTGRAPH AUF?



LEBENSZEITEN VON OBJEKTEN

1. **Transient:** Bei jedem Request wird eine neue Instanz erzeugt
2. **Scoped:** Innerhalb eines Scopes wird nur eine einzige Instanz erzeugt
3. **Singleton:** Über die gesamte Lebenszeit des DI Containers wird nur eine Instanz erzeugt

Verschiedene DI Container Implementierungen können weitere Lebenszeiten unterstützen.

SCOPES AM BEISPIEL VON SERVICES



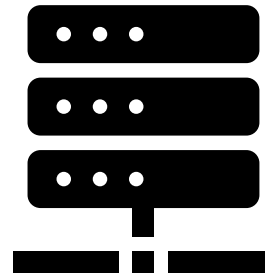
Request A



Request B



Request C



Service-Instanz

Pro Request wird ein DI Container Scope erzeugt. Damit ist es möglich, Objekte einmalig pro Request zu erzeugen und in verschiedene Services zu injizieren (z.B. Unit-of-Work für Datenbankzugriff)



WAS ABSTRAHIERE ICH DENN NUN?

MAN KANNS AUCH ÜBERTREIBEN



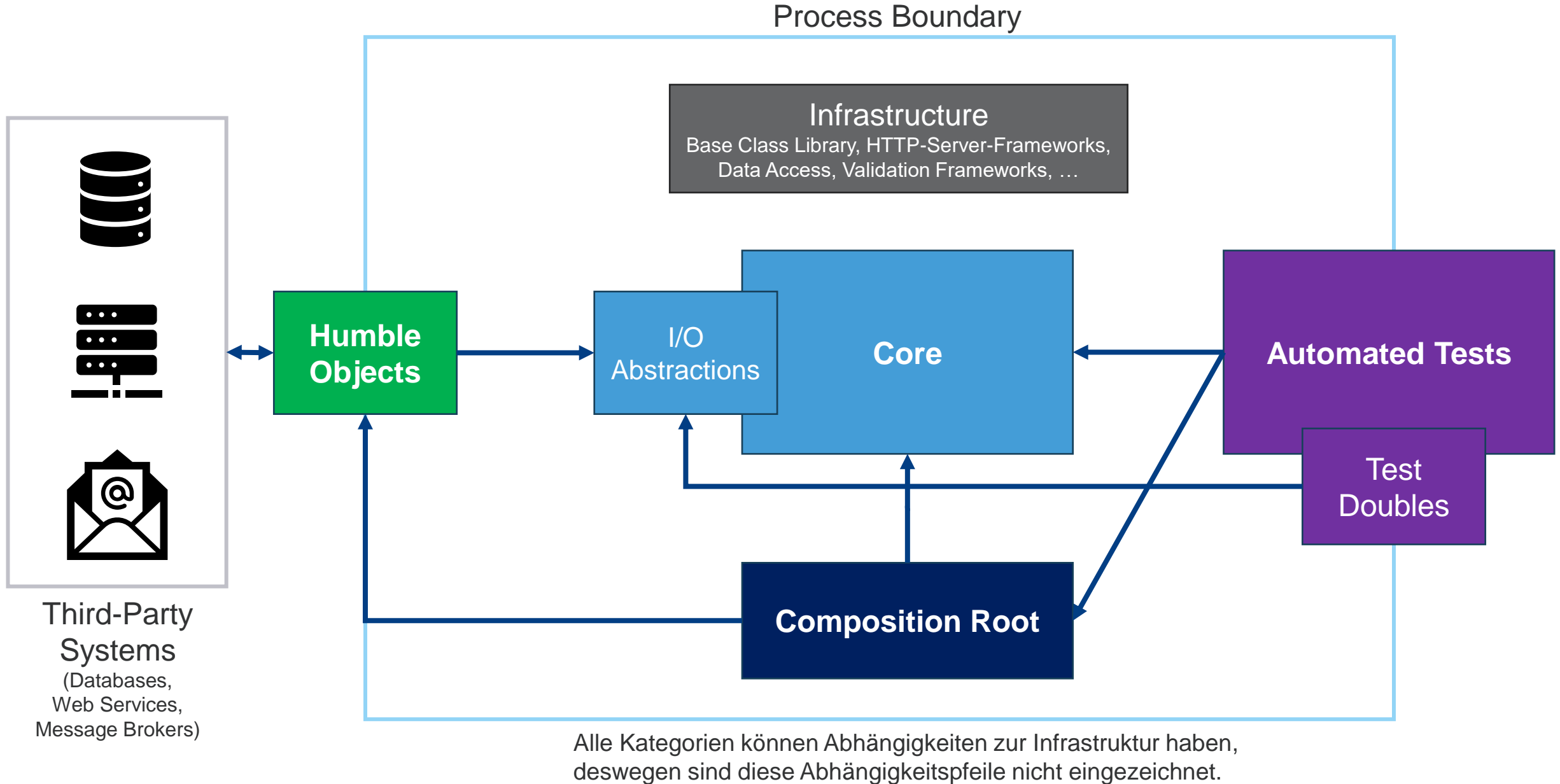
- Tausende Zeilen Code in einer Datei
- es findet keinerlei Abstraktion statt, innerhalb eines Scopes werden SQL-Befehle zusammengebaut, Business-Logik ausgeführt und UI-Elemente manipuliert
- Aufgrund der engen Kopplung sind einzelne Teile des Codes nicht Unit-testbar ohne Refactoring

- Jede Funktionalität wird in eine eigene Klasse gegossen inkl. Abstraktion (überall lose Kopplung)
- Kein Einsatz von statischen Methoden
- Ein Heer von Interfaces und Objekten
- Man muss mindestens 10x zur Definition springen, um die relevanten Zeilen Code zu finden

CHUCK IST DIE LÖSUNG



ICH MEINE CHAC



Core, Humble Objects, Automated Tests, Composition Root

Das wichtigste Prinzip:

Mache eine klare Trennung zwischen In-Memory- und I/O-Operationen

- 1. Der Core umfasst alle In-Memory Operationen.**
- 2. Wenn vom Core I/O-Aufrufe ausgehen, werden diese abstrahiert und in Humble Objects implementiert (eins pro Drittsystem, das angesprochen wird).**
- 3. Humble Objects machen ausschließlich I/O-Aufrufe und Mapping zu Datentypen, die der Core versteht.**
- 4. Für Unit-Tests, die komplett In-Memory laufen sollen, werden Humble Objects durch Test Doubles (Dummies, Stubs, Spies oder Mocks) ersetzt.**
- 5. In Integrations- und End-to-End-Tests sowie im Produktivbetrieb werden die tatsächlichen Humble Objects eingesetzt. Der Composition Root setzt (häufig unter Einsatz eines DI Containers) die einzelnen Objekte zusammen.**

Wenn du ein neues Feature entwickelst...

1. **Schreibe alle Statements auf, welche dein Feature umsetzen.**
2. **Identifiziere die Statements, welche I/O-Aufrufe durchführen.**
3. **Abstrahiere diese I/O-Aufrufe. Baue eine Abstraktion und eine Humble-Objekt-Implementierung pro Drittsystem, welches in deinem Feature aufgerufen wird.**
4. **Damit hast du die Business-Logik komplett von I/O getrennt. Du kannst jetzt Unit Tests schreiben, welche die Prozessgrenze nicht verlassen und damit keine Drittsysteme benötigen. Ersetze Humble Objects durch Test Doubles in deinen Unit Tests.**
5. **In Integrationstests kannst du sowohl Core als auch Humble Objects einsetzen.**
6. **Abstrahiere innerhalb deiner Core-Logik nur da, wo du einen Mehrwert siehst. Standardmäßig ist Tight Coupling OK.**
7. **Für deinen Produktivcode setzt du Core-Code und Humble Objects im Composition Root zusammen, typischerweise mit einem DI Container.**

- [Dependency Injection Principles, Practices, and Patterns](#) - Steven van Deursen, Mark Seemann - Manning
- [The Dependency Inversion Principle](#) - Robert C. Martin - The C++ Report
- [.NET Dependency Injection](#) - Microsoft Learn

VIELEN DANK