Christian Metzler | ABAS Software AG

Dependency Injection

Über mich

- Christian Metzler
- Studium der Informatik an der Uni Karlsruhe (2000-2008)
  - Webentwicklung während des Studiums (PHP)
  - Selbständige Webprojekte
- Softwareentwickler bei MetaSystems (2009-2013)
  - Delphi 2007/XE2
  - Framework zum Binden und Serialisieren von Objekten
  - Kamerasteuerung
- Softwareentwickler bei ABAS Software AG
  - JAVA
  - SSO und Identity Management

Agenda

- Einführung
- Patterns und Anti-Patterns
- Häufige Probleme mit DI
- „Do it yourself“ DI
- Frameworks
- Zusammenfassung

Einführung

- Was ist Inversion of Control (IoC)?
  - Bedeutet zunächst nur Steuerungsumkehr
  - Bindung üblicherweise zur Laufzeit und nicht zur Compilezeit
  - Beispiele:
    - Factory
    - Strategy
    - Template Method
    - Callbacks
- Was ist Dependency Injection (DI)?
  - DI ist eine Untermenge von IoC
  - DI ist eine Sammlung von Design Prinzipien und Patterns, die es erlauben lose gekoppelten Code zu schreiben
- Die Begriffe werden oft im Austausch verwendet, es sollte aber klar sein, dass es bei DI um die Bindung von Objekten geht.

Einführung

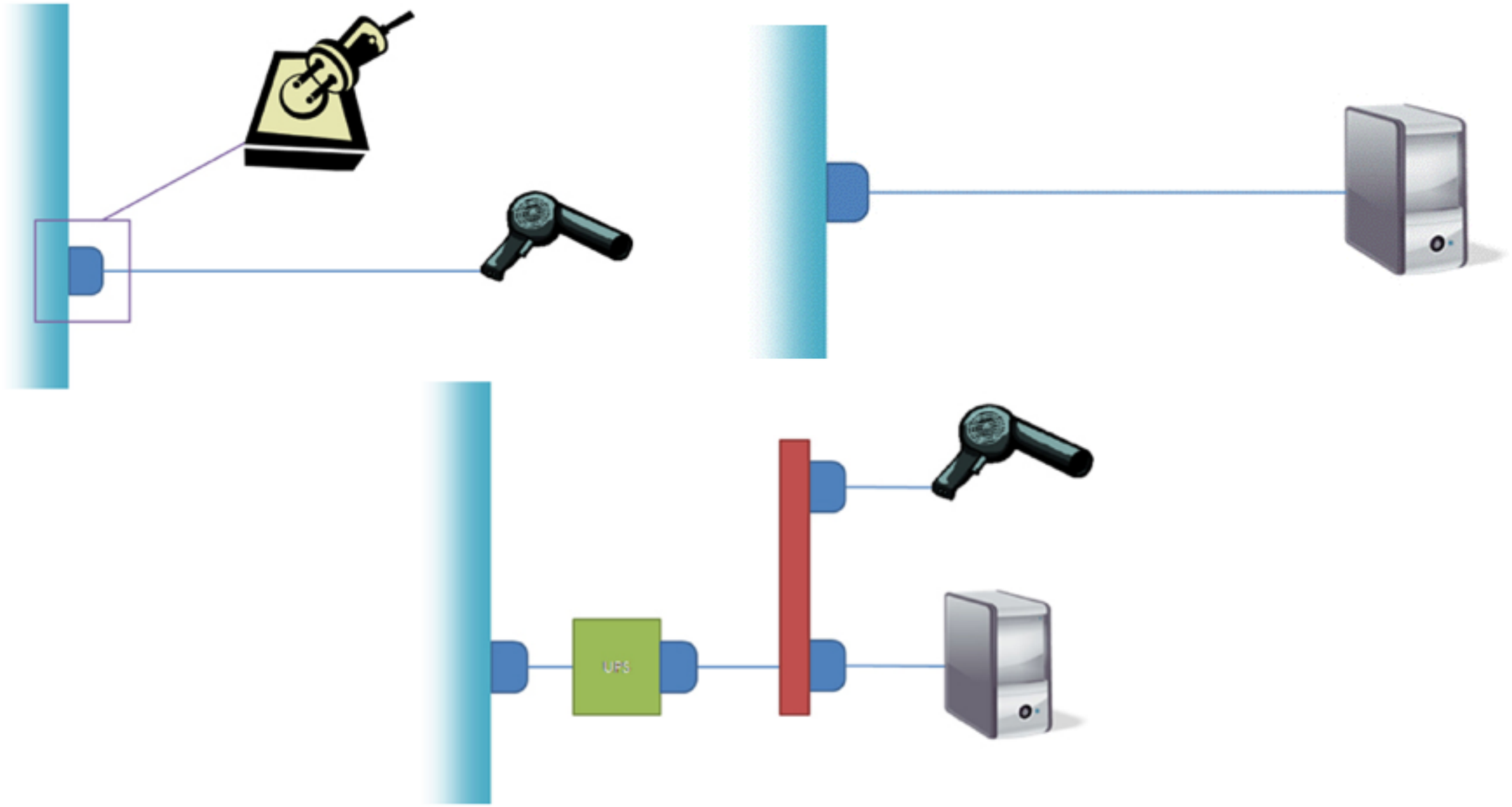
- Was ist das Ziel von DI?
  - Wartbaren Code schreiben
  - Prinzip: Programmieren gegen eine Schnittstelle nicht gegen eine Implementierung
    - Schnittstelle semantisch, nicht syntaktisch gemeint
- Missverständnisse im Bezug auf DI:
  - DI ist nur für die späte Bindung relevant
    - DI ermöglicht späte Bindung, aber das ist nur ein Teilaspekt
  - DI bringt nur was für Unit-Testing
    - Und was wenn keine Unit-Tests geschrieben werden?
  - DI ist eine Art abstrakte Fabrik auf Anabolika (Gott-Fabrik)
    - Verwechslung mit Service Locator
  - Für DI benötigt man einen DI Container
    - Definitiv nicht, aber es macht das Leben einfacher

Einführung

- Was ist der Nutzen von DI?
  - Flexibles erweiterbares Design schaffen
  - Lose Kopplung
- Was ist das Problem von starker Kopplung?

Einführung

- Bessere Lösung



Einführung

- Vorteile von loser Kopplung
  - Späte Bindung: Services können ausgetauscht werden
  - Erweiterbarkeit: Auch für Fälle die vorher nicht eingeplant wurden
  - Parallele Entwicklung: In großen Anwendungen
  - Wartbarkeit: Klassen mit einer klar definierten Verantwortlichkeit sind einfacher zu warten
  - Testbarkeit: Wenn Unit Tests eingesetzt werden
- Ein einfaches Beispiel
  - Hello World mit DI

Einführung

- Welche Abhängigkeiten sollten injiziert werden und welche nicht?
  - Statische Abhängigkeiten sollten nicht injiziert werden
    - Änderungen der Implementierung nicht zu erwarten
    - Beispiel: Klassen aus der Delphi Bibliothek (z.B. TStringList)
  - Unbeständige Abhängigkeiten sollten injiziert werden
    - Abhängigkeit hängt von der Laufzeitumgebung ab
    - Abhängigkeit ist noch in Entwicklung
    - Abhängigkeit hat nicht-deterministisches Verhalten (Testbarkeit)

Einführung

- Um was kann/soll sich DI kümmern?
  - Objektgraphen erstellen
  - Lebenszeit von Objekten kontrollieren
  - Interception
- Wie sollte DI verwendet werden
  - Die Abhängigkeiten sollten an einem zentralen Ort erzeugt werden
  - Und zwar ALLE
  - Composition Root

Patterns und Anti-Patterns

- Patterns
  - Constructor Injection
  - Property Injection
  - Method Injection
  - Ambient Context
- Anti-Patterns
  - Control Freak
  - Bastard Injection
  - Constrained Construction
  - Service Locator

Patterns und Anti-Patterns

- Constructor Injection
  - Garantiert, dass eine Abhängigkeit vorhanden ist
  - Die Abhängigkeit wird als Konstruktor Parameter angegeben
  - Evtl. Guard-Clauses vorsehen
  - Problem: Überladung von Konstruktoren - welcher soll verwendet werden?
- DEMO

Patterns und Anti-Patterns

- Property Injection
  - Schreibbare Property ermöglicht Aufrufer ein anderes Verhalten zu setzen
  - Sollte verwendet werden, wenn es eine sinnvolle lokale Abhängigkeit als Default gibt
  - Am Besten jedoch wenn die Abhängigkeit optional ist
- DEMO

Patterns und Anti-Patterns

- Method Injection
  - Methode nimmt Abhängigkeit als Parameter entgegen
  - Die Abhängigkeit ist unterschiedlich bei jedem Aufruf
  - Enge Verwandtschaft zur Abstrakten Fabrik wenn diese eine Abstraktion als Abhängigkeit entgegen nimmt
- DEMO

Patterns und Anti-Patterns

- Ambient Context
  - Abhängigkeit soll überall verfügbar sein wo sie benötigt wird, jedoch nicht als Parameter gesetzt werden (Übersichtlichkeit)
  - Für Cross-Cutting-Concerns
  - Sollte nur in Ausnahmefällen verwendet werden
  - Abgrenzung zu Interception (typisch für Interception: Logging)
- DEMO

Patterns und Anti-Patterns

- Control Freak
  - Alle Abhängigkeiten werden von der Klasse selbst erzeugt
  - Indiz: Aufruf von Konstruktoren, Abhängigkeit zu implementierenden Units
  - Achtung: Eine Factory zu verwenden löst das Problem nicht!
  - Lösung: Constructor Injection
- Bastard Injection
  - Ein Konstruktor um Abhängigkeiten zu injizieren
  - Ein Default Konstruktor, der den anderen mit konkreten Implementierungen aufruft
  - Lösung:
    - Constructor Injection
    - Property Injection

Patterns und Anti-Patterns

- Constrained Construction
  - Vorraussetzung eines bestimmten Konstruktors (z.B. Standard Konstruktor)
  - Meist gefordert für späte Bindung
  - Lösung: Abstrakte Fabrik
- Service Locator
  - Ein Service Locator wird von überall aufgerufen, wo eine Dependency benötigt wird
  - Eine Art Gott-Fabrik, die überall bekannt ist
  - Problem: Statt der Abhängigkeit zu einer konkreten Klasse, jetzt Abhängigkeit zu einem konkreten Service Locator
  - Lösung: Eines der DI-Patterns verwenden, je nach Anforderung

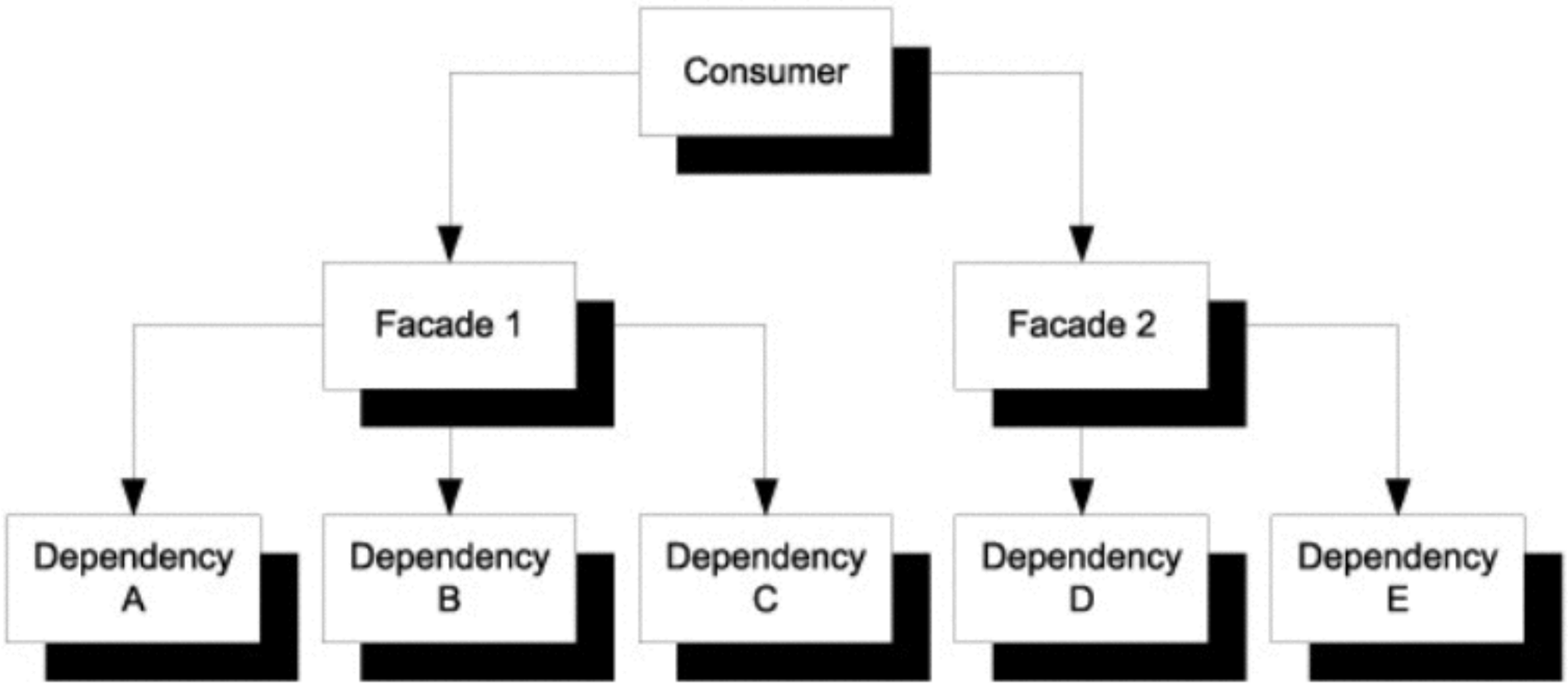
Häufige Probleme mit DI

- Problem Abhängigkeiten zur Laufzeit
  - Wie können Abhängigkeiten zur Laufzeit aufgelöst werden?
  - Beispiel: Strategie zur Berechnung einer Route in einem Navigationssystem
  - Mögliche Lösungen
    - Unterscheidung zwischen „newable“ und „injectable“ Dependencies
    - Einführung einer abstrakten Fabrik
- DEMO

Häufige Probleme mit DI

- Constructor Over Injection
  - Der Konstruktor enthält (zu) viele Parameter
  - Zitat von @punycode bei Twitter: „Für Entwickler die einen Constructor mit 25 Parametern und 1080 Zeichen erschaffen, gibt es eine spezielle Hölle“
  - Nachdenken über das Single Responsibility Prinzip!
  - Refactoring zum Beispiel in eine Fassade
  - Constructor Injection hilft solche Probleme zu erkennen

- Constructor Over Injection



„Do it yourself“ DI

- Möglichkeit 1: Poor Man’s Dependency Injection
  - Alle Abhängigkeiten werden von Hand aufgelöst
  - Create über Create über Create über…….
  - Für eine kleine Klassenstruktur machbar, aber lästig
- Möglichkeit 2: Ein selbst gestrickter DI Container
  - Der Code zum Erzeugen von Objekten wird ausgelagert
  - Im DI Container steht nur einfachster Code
  - Aber: Für jede neue Abstraktion muss eine Methode ergänzt werden
  - DEMO

DI Container

- DI Container bieten ein Framework, um DI einfach zu gestalten
- Folgen dem Register Resolve Release Prinzip
  - Register: Registrieren der Implementierung von Abstraktionen (Services und Komponenten)
  - Resolve: Genau einmal wird der komplette Objektgraph erzeugt
  - Release: Freigeben des Objektgraphs mit allen Abhängigkeiten
- Übernehmen das Lebenszeit Management
  - Singleton: Genau eine Instanz
  - Transient: Jedes mal eine neue Instanz
  - PerThread: Für jeden Thread eine eigene Instanz
  - Scope: Für einen bestimmten Scope eine Instanz

DI Container

- Möglichkeiten der Registrierung
  - Konfigurationsbasiert (z.B. XML)
  - Code-Basiert
  - Konventionen
- Komfortabel durch Generics
  - Register<Service>.ImplementedBy<Component>
  - Resolve<T>

DI Container

- Welche DI Container gibt es in Delphi
  - Spring4D: Aktivstes Projekt
  - Emballo: Features zum Linken von DLLs
  - Ambrosia: Beginn einer Portierung von Castle Windsor (C#, .NET)
- Problem:
  - Finden eines geeigneten Composition Roots
    - Wie werden Formulare erzeugt?
    - Einfachste Lösung in der Main-Form
    - Besser vor dem Application.Start
  - Kompliziert beim Releasen von Interface Referenzen (Reference Counting)

DI Container

- DEMO

Zusammenfassung

- Dependency Injection fördert die lose Kopplung
- In Delphi sind die DI Container bisher eher selten genutzt und noch nicht voll ausgeprägt
- In anderen Programmiersprachen deutlich häufiger anzutreffen
  - JAVA (Spring Framework, Google Guice)
  - C# (Structure Map, Castle Windsor)
- Erfordert ein Umdenken
  - „Das kann man bei unserem Projekt nicht machen“
  - „Das ist viel zu kompliziert und nicht praktikabel"
- Entwickeln Sie mit!

Literatur

- Dependency Injection in .NET (Mark Seemann)
- Self Made DI (Misko Hevery)
  - http://misko.hevery.com/2010/05/26/do-it-yourself-dependency-injection/

- Quellen:
  - https://github.com/coco1979ka

Vielen Dank für die Aufmerksamkeit