entwickler magazin

Alle Infos zur W-JAX » S.75

Deutschland € 6,50 Österreich € 7,00 Schweiz sFr 13,40

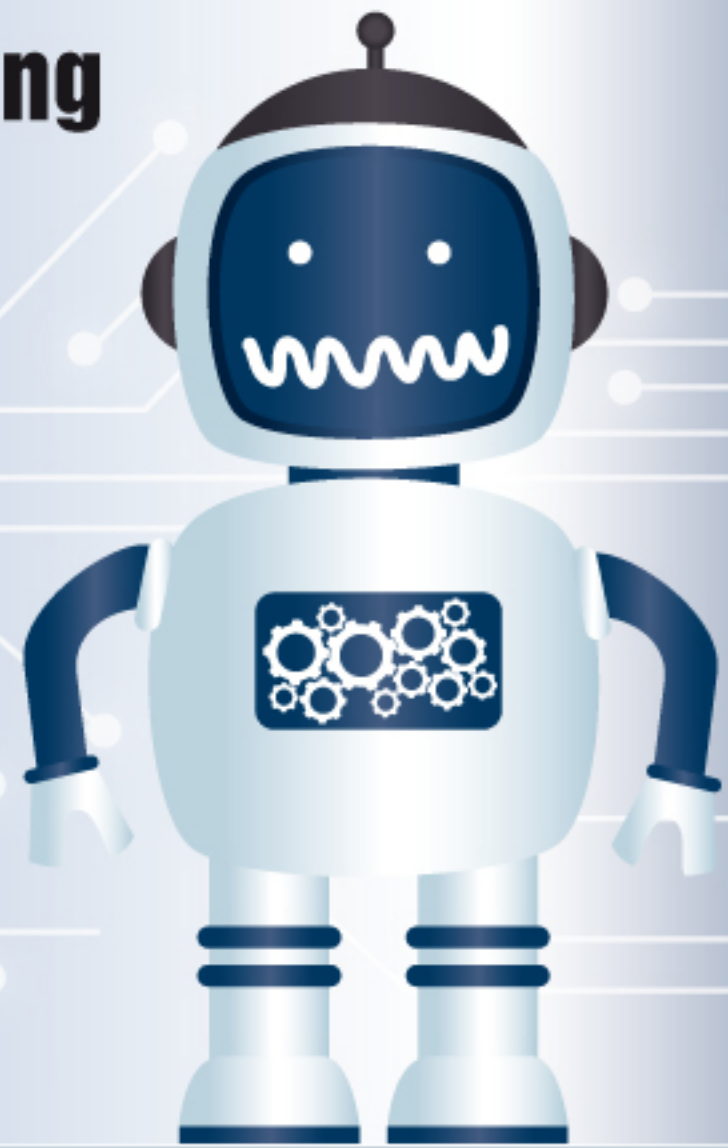
entwickler magazin

www.entwickler-magazin.de

September/Oktober 5.2014

Robotik

Programmierung für Einsteiger



Arduino
Erste Schritte

Agile
Scrum und die Spezialisten

C++
Frameworks für Webprogrammierung

SERIE **PIC**
Mikrocontroller für jedermann

SERIE **OpenGL**
Landschaftsdarstellung

4 194156 606509 05

Erfolgreiche Dependency Injection mit ABAP

ABAP OO flexibler

Seit ABAP Objects ist es möglich, objektorientiere Programmierung in SAP-Systemen zu nutzen. Doch wie lässt sich Dependency Injection in diesem Umfeld realisieren? In diesem Artikel sollen erste Einblicke geboten werden.

von Christian Buschmann

Mit ABAP Objects (OO) hat die SAP AG die objektorientierte Programmierung in SAP-Systemen ermöglicht. Mittlerweile ist OO als Programmiermodell für Neu- und Weiterentwicklungen durch SAP gesetzt. Damit hat der Programmierer die Möglichkeit, die aus OO bekannten Patterns und Programmierprinzipien in seinen ABAP-Applikationen zu verwenden. Das Vorgehen zur Erstellung von OO-Programmen unterscheidet sich allerdings von prozeduralen Anwendungen. Mithilfe der Kapselung von Programmeinheiten in Klassen lassen sich mehrere Instanzen erzeugen, die unterschiedliche Implementierungen und inhaltliche Abbildungen enthalten. Für das Erzeugen und Verwalten von Instanzen gibt es diverse Herangehensweisen. Ein ungünstiges Vorgehen wäre es, die abhängigen Instanzen innerhalb der Fachklasse selbst zu erstellen. Dies wirkt sich nachteilig auf die Flexibilität, Testbarkeit und Wiederverwendbarkeit der Klassen aus. Folgender Artikel erläutert als Alternative das Pattern der Dependency Injection (DI) im Umfeld der ABAP-OO-Entwicklung. Es wird zunächst das Pattern und im Anschluss ein Ansatz zur pragmatischen Umsetzung dargestellt.

Weitere Informationen zu ABAP OO

Ursprünglich ist ABAP eine Sprache für die prozedurale Programmierung, in der die Anwendungsentwicklung mithilfe von Funktionsbausteinen und Formroutinen erfolgt. Mit dem Erscheinen des SAP-Releases 6.4 wurde ABAP um die Eigenschaften einer objektorientierten Sprache erweitert. Die SAP AG entwickelte ABAP Objects kontinuierlich weiter und mittlerweile wird das Programmiermodell in den ABAP-Programmierrichtlinien von SAP für Neuentwicklungen und Weiterentwicklungen empfohlen. Auch durch SAP HANA und die neuen Möglichkeiten in der Anwendungsentwicklung verliert der Application Server ABAP nicht an Bedeutung. Gerade im Zusammenwirken mit SAP HANA hat SAP Möglichkeiten geschaffen, diese neue Technologie effizient mit ABAP zu verbinden.

Das Dependency Injection Pattern

Bei der Dependency Injection [1] geht es um die Entkopplung von abhängigen Programmteilen und des Kontrollflusses bei OO-Anwendungen. Folgendes Beispiel verdeutlicht das Konzept: Eine Klasse, die eine Berechnung durchführt, soll das Ergebnis einer beliebigen Weiterverarbeitung zuführen. Diese kann beispielsweise die Ablage des Ergebnisses in einer Datenbank oder der Aufruf eines Web Service sein. Berechnung und Weiterverarbeitung sind in verschiedenen Klassen abgebildet. Im Rahmen der Dependency Injection wird die Instanz der Weiterverarbeitung nicht durch die Berechnungsklasse selbst erzeugt, sondern über den Konstruktor oder mithilfe einer Setter-Methode der Klasse übergeben. Auf diese Weise hat die Berechnungsklasse keine Kenntnis darüber, wie die konkrete Weiterverarbeitung aussieht. Somit bestimmt nicht die Implementierung in der Berechnungsklasse den Kontrollfluss der Applikation, sondern er wird von außen vorgegeben.

Eine derart gestaltete Anwendung bietet flexible Möglichkeiten, das Laufverhalten zu konfigurieren. **Abbildung 1** verdeutlicht das oben beschriebene Beispiel in Form eines Klassendiagramms. Die Konfiguration der Anwendung geschieht innerhalb einer Fabrik.

Der Vorteil der Dependency Injection ist, dass die Implementierung einer Fachklasse nicht angepasst werden muss, wenn sich die Anforderungen an den Kontrollfluss ändern. Die neue Implementierung reduziert

sich auf den Programmcode, der den fachlichen Zweck der Klasse abbildet. Die Fachklasse lässt sich so gut isolieren, dass Unit Tests problemlos möglich sind.

Hier noch ein Hinweis auf die SOLID-Prinzipien [2]: Sie geben gute Designrichtlinien für Klassen und helfen dabei, ein sauberes Klassendesign zu erzeugen.

Mit Blick auf andere Programmiersprachen wird das Konzept der Dependency Injection mit Frameworks wie Spring und Guice [3] in Verbindung gebracht. Diese unterstützen die oben dargestellte Fabrik insofern, als dass sie das Erzeugen, Konfigurieren und Versorgen von Instanzen automatisieren. Zudem schafft und ermittelt das Framework notwendige Abhängigkeiten einer Klasse. Über welche Regeln diese in die Klassen injiziert werden, legt der Entwickler deklarativ fest. Ein DI-Framework mag für wenige Klassen nicht notwendig sein – verwaltet aber eine Fabrik eine große Anzahl von Klassen, dann erhöht es die Übersicht und vereinfacht die Konfiguration.

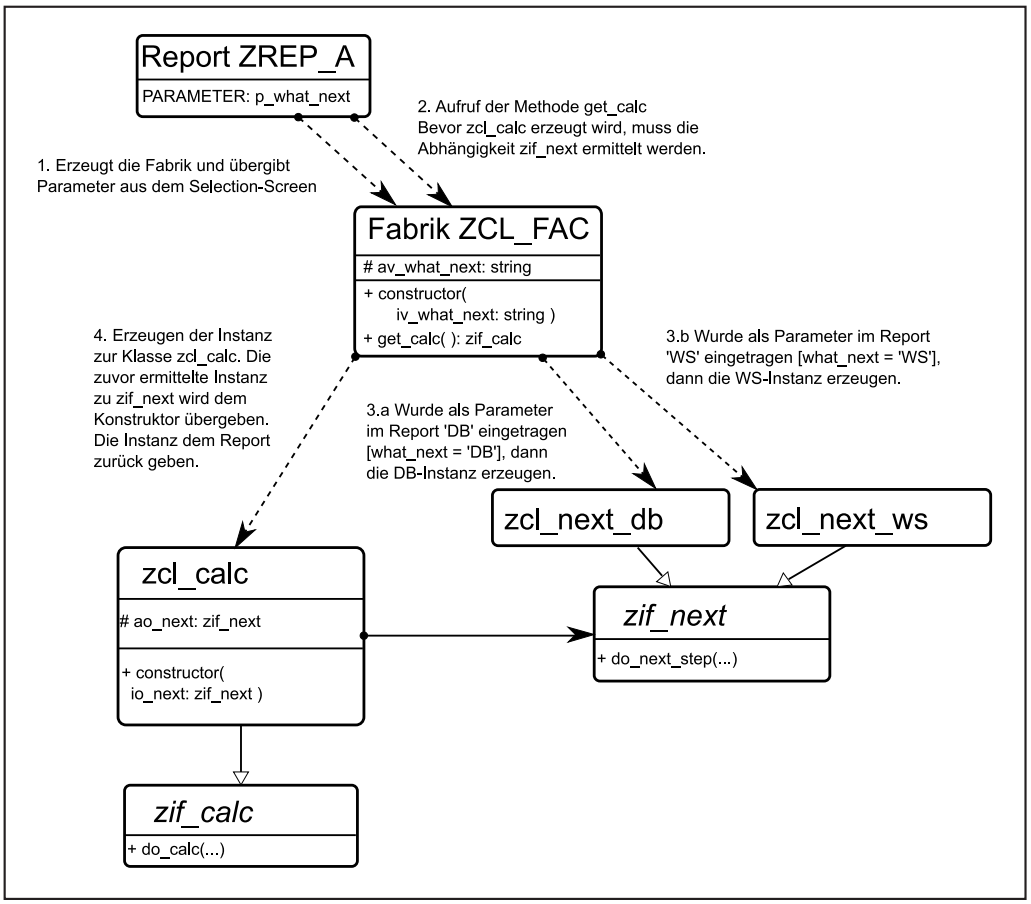


Abb. 1: Beispiel einer einfachen Dependency Injection

Da SAP kein eigenes DI-Framework anbietet, hat die innobis AG eines geschaffen. Dessen Funktionsweise wird im Folgenden erläutert.

DI-Framework – ein einfaches Beispiel

Im Hinblick auf ABAP ergaben sich einige besondere Anforderungen an das Framework. So sollten die Parameter der Konstruktoren nicht nur mit Objektreferenzen versorgt werden können sondern mit beliebigen ABAP-Datentypen. Außerdem sollte der Verwendungsnachweis auch solche Klassen ermitteln, die über das Framework erzeugt werden. Aufgrund der dynamischen Instanzerzeugung der Klassen ist dies nicht möglich. Die Funktionsweise des Frameworks ist transparent und nachvollziehbar. Es ist offengelegt, welche Klassen mit welchen Instanzen versorgt werden, um mögliche Fehler aufzudecken. Die folgenden Anforderungen sind ebenso relevant:

- Ein einfaches Set-up muss möglich sein.
- Das Framework erfordert keine Implementierungen in den Fachklassen.
- Die Instanzerzeugung erfolgt über Konstruktoren oder Fabrikmethoden.

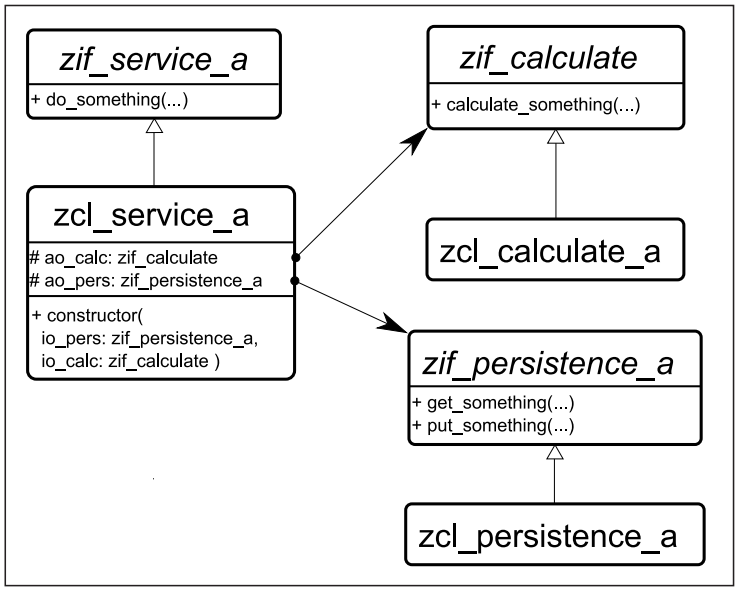


Abb. 2: Einfaches Fallbeispiel

- Die Parameter der Konstruktoren und Setter-Methoden beschreiben die Abhängigkeiten einer Klasse, welche das DI-Framework automatisch ermittelt und versorgt.
- Erzeugte Instanzen müssen mehrfach verwendbar sein (Singleton-Muster).
- Ein modularisierter Aufbau zur Unterstützung einer komponentenbasierten Architektur ist notwendig.

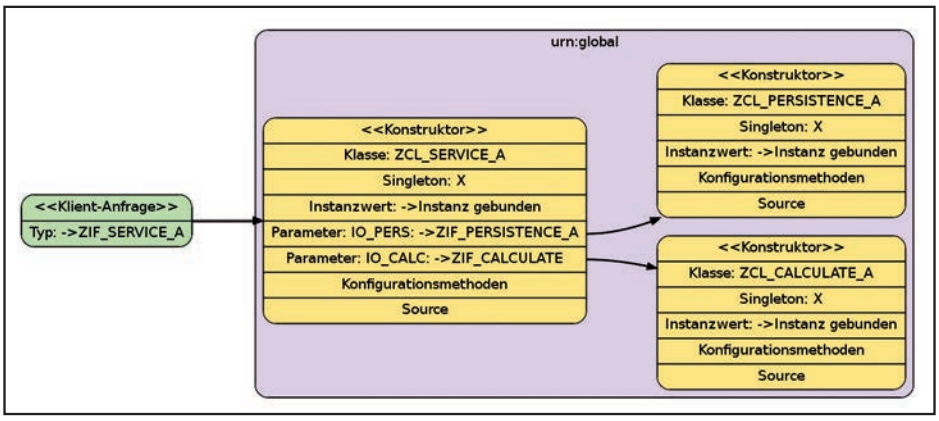


Abb. 3: Trace der erzeugten Konfiguration des Containers

Abbildung 2 zeigt ein Klassendiagramm, das durch das anschließende Beispiel führt. Die ersten Buchstaben der Bezeichnungen weisen Interfaces/Schnittstellen (*zif_**) beziehungsweise Klassen (*zcl_**) aus.

Zu sehen ist eine Serviceklasse (*zcl_service_a*), die von zwei weiteren Objekten abhängt. Die Klasse *zcl_calculate* führt eine Berechnung durch und die Klasse *zcl_persistence_a* speichert das Ergebnis des Service. Wird eine Instanz der Klasse *zcl_service_a* erzeugt, dann sieht dies im ABAP aus wie in Listing 1.

Der Klasse *zcl_service_a* wird jeweils eine Instanz zu *zif_persistence_a* und *zif_calculate* injiziert. Die Dependency Injection geschieht hier manuell.

Dies ist ein sehr einfaches Beispiel, das im Rahmen von Fabrikmustern Anwendung finden könnte. Je mehr Klassen eine Komponente aber enthält, umso aufwändiger ist die Zusammenstellung – insbesondere beim Verwenden vieler kleinteiliger Klassen. Zudem ist die Konfiguration statisch in der Fabrik vorgegeben und kann nicht ohne Duplizieren des Fabrikquellcodes angepasst werden.

Das DI-Framework von innobis unterstützt dabei, Instanzen zu konfigurieren und zu erzeugen. Die Konfiguration wird vom Entwickler deklarativ vorgegeben, und das Framework übernimmt die Aufgabe, die Instanzen gemäß der Vorgabe zu erzeugen und automatisiert zu konfigurieren. Die Idee ist, im ersten Schritt eine Menge von Klassen zu bestimmen und einem Container zuzuordnen, womit die Deklaration abgeschlossen ist. Im zweiten Schritt erfragt ein Klient eine Instanz eines bestimmten Typs aus dem Container an. Das DI-Framework ermittelt nun eine Klasse, die zu dem erfragten Typ passt. Bestehen für diese Klasse über den Konstruktor oder über Setter-Methoden weitere Abhängigkeiten, so ermittelt der Container diese rekursiv. Sind alle Abhängigkeiten ermittelt, so wird eine Instanz zu dem erfragten Typ erzeugt und dem Aufrufer zurückgegeben. Der Code in Listing 2 zeigt dies beispielhaft.

In dem Container lässt sich nun eine beliebige Anzahl von Klassen registrieren. Beim Aufruf der Methode *get_instance_value* errechnet der Container eine Konfigurationslösung und erzeugt aus dieser eine Instanz. Beim Aufruf von *register_classname* kann alternativ eine Instanzvariable mitgegeben werden. Diese bleibt ungebunden (UNBOUNDED) und dient als Prototyp. Das Framework registriert dann die an der Variablen hinterlegte Klasse. Der SAP-Verwendungsnachweis kann so die Klasse weiterhin finden, was beim String-Literal nicht möglich ist.

Bei einer großen Anzahl von Klassen geht hier allerdings schnell mal die Übersicht verloren, welche Lösung der Container errechnet hat und ob diese vom Entwickler gewollt ist. Für eine erweiterte Analyse gibt es die Möglichkeit, über einen Trace die Konfiguration zu visualisieren (**Abb. 3**).

Listing 1

```
DATA:
  lo_persistence TYPE REF TO zif_persistence_a,
  lo_calculate TYPE REF TO zif_calculate,
  lo_service TYPE REF TO zif_service_a.

# Instanz erzeugen
  CREATE OBJECT lo_persistence TYPE lcl_persistence_a.
  CREATE OBJECT lo_calculate TYPE lcl_calculate.
  CREATE OBJECT lo_service TYPE lcl_service_a
    EXPORTING
      io_pers = lo_persistence
      io_calc = lo_calculate.

# Instanz verwenden
  lo_service->do_something( ... ).
```

Listing 2

```
DATA lo_container TYPE REF TO zif_di_container.
DATA lo_service TYPE REF TO zif_service_a.

# DI-Container erzeugen
lo_container = zcl_di_container=>create_instance_default( ).

# Menge an Klassen im Container registrieren
lo_container->register_classname( iv_classname = 'ZCL_SERVICE_A' ).
lo_container->register_classname( iv_classname = 'ZCL_PERSISTENCE_A' ).
lo_container->register_classname( iv_classname = 'ZCL_CALCULATE_A' ).

# Instanz aus dem Container ermitteln
lo_container->get_instance_value(CHANGING cv_target_value = lo_service ).

# Instanz verwenden
lo_service->do_something( ... ).
```

In **Abbildung 3** ist ein weiteres Feature erkennbar: Mithilfe von Namespaces lässt sich der Container in mehrere unabhängige Bereiche unterteilen. Die gezeigten Klassen sind im Namespace *urn:global* registriert. Die Ermittlung einer Instanz aus dem Container ist stets bezogen auf einen Namespace. Nur innerhalb eines Namespace werden die Instanz und deren Abhängigkeiten ermittelt. Dadurch lässt sich vermeiden, dass bei der Auflösung der Abhängigkeiten Mehrdeutigkeiten auftreten. So lassen sich mehrere Klassenimplementierungen zu einem Interface registrieren, die in unterschiedlichen Kontexten zum Einsatz kommen. Jeder Kontext erhält seinen eigenen Namespace. Zwischen den Namespaces kann der Entwickler typbezogen Verbindungen definieren. Dies ermöglicht es, Instanzen aus einem fremden Namespace im eigenen zu verwenden.

Fazit

Die bisher gezeigten Beispiele sind recht einfach gehalten. Für den Einsatz in einer größeren komponentenbasierten Architektur mussten dem Framework noch weitere Funktionen hinzugefügt werden. So lassen sich Namensräume nutzen, um innerhalb eines Containers mehrere isolierte Bereiche für die Klassenregistrierung zu schaffen. Zudem wurde ein Pattern zur Modularisierung erdacht, um das DI-Framework innerhalb von Fabrikklassen einsetzen zu können. Als Besonderheit lassen sich diese automatisch über die „Service Implementation Workbench“ (SIW) von SAP generieren. Der Entwickler muss dann lediglich die vorgenerierten Methodenrümpfe der Fabrikklassen befüllen.

Die Verwendung der Dependency Injection führt dazu, dass Unit Tests für kritische Klassen erzeugt werden können. Zudem unterstützt sie die Flexibilität der Anwendung.

Christian Buschmann (36), Diplom-Wirtschaftsinformatiker (FH), ist seit 2005 bei der innobis AG tätig und in seiner Funktion als Senior Consultant unter anderem verantwortlich für Softwareentwicklungsprojekte im APAB/SAP-Umfeld.

Links & Literatur

[1] http://www.itwissen.info/definition/lexikon/Dependency-Injection-dependency-injection-DI.html

[2] http://de.wikipedia.org/wiki/Prinzipien_objektorientierten_Designs#SOLID-Prinzipien

[3] https://code.google.com/p/google-guice/wiki/Motivation

ENTWICKLER³

Jetzt abonnieren!
www.entwicklermagazin.de

Jetzt 3 TOP-VORTEILE sichern!

1

- Alle Printausgaben frei Haus erhalten
- Intellibook-ID kostenlos anfordern (www.intellibook.de)

2 Abo-Nr. (aus Rechnung oder Auftragsbestätigung) eingeben

3 Zugriff auf das komplette PDF-Archiv mit der Intellibook-ID

S&S Media Group

www.entwicklermagazin.de