5 Komponentenarchitektur: Entkopplung

»...it is almost always incorrect to begin the decomposition of a system into modules on the basis of a flowchart. We propose instead that one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others.«

David Parnas

Nach den ersten Umstellungen im Coding, um Modularität zu erreichen, und den Grundlagen einer modularen Architektur wird der Aspekt der Schnittstellen von Modulen genauer beleuchtet und es wird untersucht, wie Services verwendet werden, um Entkopplung zu erreichen. Direkte Verwendung von Services ist nicht erwünscht, denn das würde eine enge Kopplung nach sich ziehen.

Enge Kopplung muss auf jeden Fall vermieden werden. Schnittstellen und Dependency Injection helfen bei der richtigen Anwendung und sind einen genaueren Blick wert.

5.1 Hintergrund modularisierter Software

An dieser Stelle stehen die Schnittstellen von Modulen, die Services, im Vordergrund und die Art und Weise, wie Serviceinstanzen durch das modulare Laufzeitsystem eingesetzt werden. Hierbei muss das gewohnte Vorgehen eines Entwicklers, die Objektinstanzen direkt zu erzeugen, verworfen werden.

5.1.1 Schnittstelle von Modulen

Services definieren die Schnittstelle der Module. Die entscheidende Frage besteht jetzt darin, wie die Definition eines Moduls mit dem Service verbunden ist. Im Rahmen der Entkopplung sollte die Definition eines Moduls niemals mit der Definition eines Service gekoppelt sein. Die Serviceinformationen sind daher niemals mit den Modulmetadaten verbunden.

Es existiert aber eine andere, weitverbreitete Methode, mit Services umzugehen. Dabei werden die Services genau wie Module deklarativ angegeben. Ein Service ist nicht nur eine Implementierung, sondern besitzt eine Deklaration.

Hier soll wieder auf die serviceorientierte Architektur (SOA) hingewiesen werden. In diesem Bereich ist es gang und gäbe, dass die Services in aller Ausführlichkeit beschrieben werden. Meistens wird SOA mit Webservices und allen daran gebundenen Standards gleichgesetzt. Diese Diskussion soll an dieser Stelle nicht vertieft werden.

Das Entscheidende ist allerdings, dass die Services deklariert und in der Moduldefinition referenziert werden, sodass absolute Klarheit darüber herrscht, was die Schnittstelle eines Moduls ist. In einem weiteren Schritt werden die Verwendungen von Services angegeben, sodass es nicht nur feststeht, welche Services eines Moduls wiederverwendbar zur Verfügung stehen, sondern welche konkreten Instanzen der Services durch das modulare Laufzeitsystem erzeugt und in die verwendenden Module injiziert wurden.

5.1.2 Dependency Injection

Dependency Injection verdient ein wenig mehr Beachtung, denn es handelt sich um ein weitverbreitetes Prinzip in der modernen Softwareentwicklung und gewinnt immer mehr an Bedeutung. Ein anderer Begriff für Dependency Injection ist Dependency Inversion, weil hierbei das Setzen der Abhängigkeiten umgedreht wird. Der Verwender ruft nicht mehr direkt, sondern bekommt den zu verwendenden Service injiziert (Hollywood-Prinzip: »Don't call us, we call you«). Unter Dependency Injection bzw. Inversion versteht man zwei Aspekte der Implementierung:

- Implementierungen in einem Modul sollen nicht direkt auf anderen Implementierungen in Modulen basieren, sondern auf Abstraktionen (Schnittstellen).
- Abstraktionen (Schnittstellen) sollen keine Implementierungsdetails exponieren.

Es ist ziemlich offensichtlich, dass dieses Prinzip bei der Aufteilung der Datenquellen angewendet wurde. Das Servlet-Modul basiert nur noch auf den Interfaces der Datenquellen, nicht mehr auf anderen Modulen, die Implementierungen wie zum Beispiel für Datenquellen enthalten. Darüber hinaus exponiert das Interface keine Details der Implementierung.

Um das Verfahren zu verdeutlichen, bietet es sich an, den Ablauf einer solchen Dependency Inversion zu visualisieren [8].

Der Anfang umfasst Instanzen zweier Klassen, die zunächst einmal unabhängig voneinander sind. Es besteht die Anforderung, dass die Instanz von Class 2 Methoden der Instanz von Class 1 aufrufen soll.

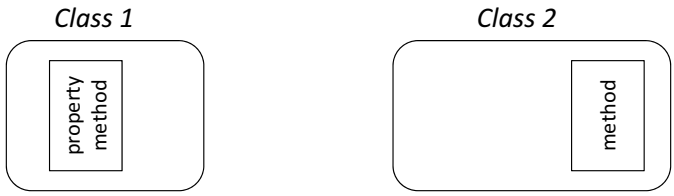

Abb. 5–1 *Problemstellung von Dependency Injection*

Im nächsten Schritt wird die Instanz von Class 1 in Class 2 injiziert. Diese Aufgabe übernimmt normalerweise das Framework, denn wenn Class 1 die andere Klasse kennen würde, dann wäre ja das Ziel der Entkopplung nicht erreicht.

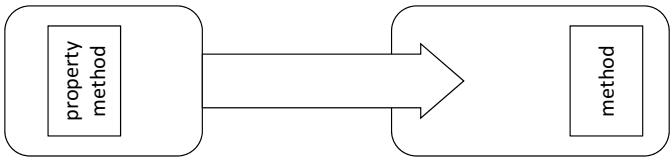

Abb. 5–2 *Injection einer Instanz in Class 2*

Nun ist es möglich, innerhalb der Instanz von Class 2 auf die Instanz von Class 1 zuzugreifen.

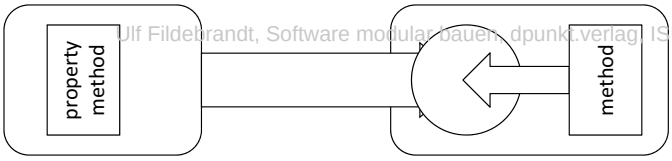

Abb. 5–3 *Zugriff auf Instanz von Class 1*

Methoden oder Properties von Class 1 können jetzt geändert oder aufgerufen werden.

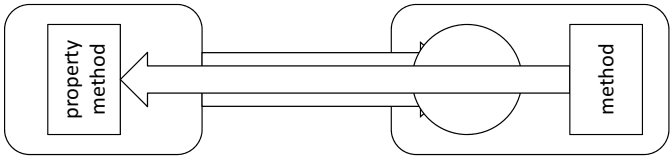

Abb. 5–4 *Aufruf von Methoden*

Auf den ersten Blick scheint dieses Konzept sehr einfach zu sein. Aber wie es mit einfachen Dingen oft der Fall ist, handelt es sich um ein sehr mächtiges Instru-

ment. Die Bedeutung lässt sich alleine daran erkennen, wie viele Frameworks existieren, um den Entwickler bei der Anwendung zu unterstützen.

Meistens wird der Begriff Dependency Injection benutzt, sodass die Frameworks als Dependency-Injection-(DI-)Frameworks bezeichnet werden.

Ganz wichtig ist aber, dass es sich bei DI nicht nur um ein Coding-Prinzip handelt. Jede Architektur drückt sich am Ende im Coding und in einzelnen Methodenaufrufen aus, aber viel wichtiger ist Dependency Injection für das Design von Systemen, wenn Komponenten unabhängig gehalten werden sollen.

Anstelle von Klasseninstanzen hätte man im Beispiel auch sehr große Frameworks innerhalb eines Systems annehmen können, zum Beispiel die Benutzerverwaltung und ein Webservice-Laufzeitsystem. Es ist keine Frage, dass ein Webservice-Laufzeitsystem eine Benutzerverwaltung braucht, aber beide Komponenten müssen sich auf Implementierungsebene nicht notwendigerweise direkt referenzieren. Wenn das Webservice-Laufzeitsystem gegen eine Abstraktion (Schnittstelle) programmiert ist, dann ist es während der Laufzeit möglich, die Benutzerverwaltung zu injizieren. Während der Lebenszeit eines Softwaresystems wäre es auf diesem Wege möglich, die Implementierung der Benutzerverwaltung komplett auszutauschen.

Die Entkopplung von Komponenten ermöglicht erst eine wirkliche modulare Architektur. Alle Mechanismen zur Entkopplung, sei es auf Coding-Ebene oder Komponentenebene sind essenziell für eine modulare Architektur. Daher spielt dieses Prinzip eine so große Rolle für die Komponentenarchitektur.

5.1.3 Modularität ohne Schnittstellen

Bisher könnte man annehmen, dass Modularität nur durch eine Trennung von Schnittstelle und Implementierung erreicht werden kann, wobei die Implementierungen jeweils in einzelnen Modulen abgelegt sind.

Wenn man sich aber die Definition eines Moduls als Grundlage für Modularität genauer durchliest, dann ist nur eine klar definierte Schnittstelle eines Moduls notwendig. Diese Unterscheidung ist wichtig, um Missverständnisse zu vermeiden. Eine Trennung zwischen Schnittstelle und Implementierung ist nur dort erforderlich, wo diese Abstraktion erwünscht ist und wenn man die Implementierung später austauschen will. Wie im Modularitätsmodell beschrieben, gibt es verschiedene Ebenen der Modularität. Eine praxisgerechte Modularisierung von Software kann auch mit anderen Mitteln erreicht werden.

Es stellt sich jetzt die Frage, wie eine solche Modularisierung zu erzielen ist. Als Grundvoraussetzung muss die Software in Module unterteilt sein. Diese Module exponieren eine definierte Schnittstelle, die wiederum von anderen Modulen verwendet werden kann.

Der Trick besteht jetzt darin, dass unterschiedliche Module zu einem System oder einer Applikation zusammengestellt werden können. Die Verwendungsbe-

ziehungen gehen gegen eine bestimmte Klasse oder nur gegen eine Funktion. Wie dieses Objekt in dem referenzierten Modul implementiert ist, hängt dann davon ab. Es kann zwei oder mehr Implementierungen in unterschiedlichen Modulen geben. Erst die Zusammenstellung entscheidet darüber, wie das Gesamtsystem sich verhält.

Dieses Verfahren kann man anhand zweier sehr bekannter Beispiele erklären:

- Linken in maschinennahen Programmiersprachen
- Classloading in Java

5.1.3.1 Linken

Der Entwicklungsprozess in C oder C++ sieht immer gleich aus. Zuerst wird das Coding kompiliert, d.h. vom Quelltext in einen maschinennahen Code übersetzt. Verwendungen von Funktionen oder Methoden in anderen Klassen sind dabei nur als Referenzen enthalten. Um ein ausführbares Programm zu erstellen, muss noch eine Link-Operation ausgeführt werden.

Beim Linken werden die symbolischen Referenzen durch die wirklichen Referenzen ausgetauscht, sodass andere Module ab diesem Zeitpunkt für die Ausführung notwendig sind. Es gibt Erweiterungen in den Betriebssystemen, damit das Linken erst zur Laufzeit der Applikation oder des Systems stattfindet.

Im Prinzip lassen sich die einzelnen Module in diesem Entwicklungsprozess beliebig zusammenstellen und durch andere Module ersetzen. Dieses Verfahren bietet nicht die theoretisch beste Modularität, aber in der Praxis hat es sich über Jahrzehnte als ausreichend erwiesen.

Für dieses Verfahren ist es nicht notwendig, die Schnittstelle von der Implementierung zu trennen. Es muss nur sichergestellt sein, dass die Referenz im verwendenden Modul die Funktion im verwendeten Modul aufrufen kann. Die Schnittstellen müssen kompatibel sein, aber nicht getrennt von der Implementierung.

5.1.3.2 Classloading

In Java werden die Klassen in Archiven gruppiert. Sobald eine Laufzeitumgebung gestartet wird, erhält die JVM eine Anzahl von Archiven und alle verwendeten Klassen werden aus diesen Archiven geladen.

Die Menge der Archive ist flexibel, sodass es wieder zwei Archive geben kann, die dieselbe Klasse in Bezug auf die Schnittstelle enthalten, deren Implementierung aber unterschiedlich ist.

Durch das einfache Angeben eines anderen Archivs wird jeweils eine andere Implementierung geladen. Die Zusammenstellung der Menge der Archive bestimmt darüber, wie sich eine Anwendung oder ein System verhält.

Obwohl Java die Möglichkeit bietet, Interfaces direkt in der Sprache zu definieren, könnte eine Implementierung in einem Archiv nur Klassen enthalten.

Durch die Zusammenstellung in Archiven, sprich Modulen, kann zum Startzeitpunkt angegeben werden, welche Funktionen verwendet werden.

5.1.4 Modularity Patterns in diesem Kapitel

In diesem Kapitel werden die folgenden Modularity Patterns anhand von Beispielen beschrieben. Die Formulierung ist abstrakt, nur das Beispiel bedient sich konkreter Frameworks.

- Extensions – Restricting extensibility mechanism for modules
- Module services – Using services as communication means between modules
- Open-closed modules – Using Open Closed principle for modules
- Layer – Using layers in a modular runtime
- Dependency injection – Using dependency injection in a modular runtime

5.2 Grundlagen von Declarative Services

Bevor die Anwendung geändert wird, führt ein kurzer Exkurs in eine weitere Teiltechnologie im OSGi-Bereich. Die Entkopplung der Implementierung soll über Declarative Services erreicht werden. Wie in der Beschreibung der Services in OSGi dargelegt ist, handelt es sich dabei um den Mechanismus von OSGi, um verschiedene Teile der Implementierung miteinander kommunizieren zu lassen.

Normalerweise werden in Java Abhängigkeiten über Import-Anweisungen abgedeckt, aber der Nachteil eines Imports besteht darin, dass es eine direkte Abhängigkeit während der Zusammenstellung des Klassenpfades gibt, basierend darauf, welche Archive gerade zur Verfügung stehen. Daher sollten keine Implementierungen über *import* in einem anderen Bundle bekannt gemacht werden, nur Interfaces.

Die Declarative Services Specification [1] aus dem Service Compendium der OSGi-Spezifikation bietet ein deklaratives Service Component Model auf der Basis von OSGi-Services. OSGi-Services wurden im letzten Kapitel eingeführt, und die Registrierung per Coding wurde kurz erklärt. Declarative Services gehen einen Schritt weiter, indem die Definition der Services nicht mehr per Coding vorgenommen wird, sondern in einer Deskriptordatei.

OSGi Declarative Services sind Teil jedes OSGi-Laufzeitsystems und bauen auf dem OSGi Service Model auf. Dadurch ist die Spezifikation mit dem programmatischen Ansatz vollständig kompatibel.

Durch den deklarativen Ansatz werden die POJOs (OSGi-Service-Instanzen) in Deskriptordateien definiert und vom Laufzeitsystem verwaltet. Referenzen in anderen Deskriptordateien erhalten auf dynamischem Weg die Objektinstanzen, um damit zu arbeiten.

Das ist genau derselbe Mechanismus, der auch bei den bekannteren Dependency-Injection-Frameworks wie Spring (siehe [42]) oder Guice (siehe [43]) eingesetzt wird, aber dazu später mehr bei einem Vergleich dieser Frameworks. Dependency Injection ist ein mächtiges Konstrukt, weil Objekte nicht mehr manuell aufgebaut, sondern komplett deklarativ und vom Laufzeitsystem verwaltet werden. Damit muss der Anwender einer Komponente nichts mehr über die Erzeugung der Instanz wissen.

Um Declarative Services zu verwenden, existiert im Bundle-Archiv neben dem META-INF-Verzeichnis ein weiteres Verzeichnis OSGI-INF, in dem die Deskriptordateien für Services abgelegt werden. In der Beispielanwendung werden Declarative Services schon für die Initialisierung des Webcontainers verwendet.

```
<?xml version="1.0"?>
<component name="helloworldComponent">
  <implementation class="arch.datadisplay.ui.RegisterComponent"/>
  <reference name="httpService"
    interface="org.osgi.service.http.HttpService"
    bind="setHttpService"
    unbind="unsetHttpService"
    cardinality="0..n"
    policy="dynamic"/>
</component>
```

Listing 5–1 *Injection eines Service in eine Komponente*

Zunächst einmal wird die Komponente definiert, indem die Implementierungsklasse angegeben wird. Die Erzeugung dieser Instanz übernimmt das Declarative Services-Framework. Anschließend wird über die Referenz angegeben, dass eine Instanz des HTTP-Service in die Klasseninstanzen injiziert werden soll. Bei dem String »*org.osgi.service.httpService*« handelt es sich um den Schlüssel des OSGi-Service in der OSGi-Registry. Dieser Vorgang ist bereits aus dem einfachen Beispiel bekannt, wo ein Service bei der Registry per Coding angefragt wurde. Bei Declarative Services funktioniert die Registrierung und Injizierung von Serviceinstanzen nicht programmatisch, sondern per Deskriptordateien.

Der HTTP-Service in OSGi unterstützt die Servlet-API-Spezifikation Version 2.1, was nicht der aktuellen Technologie entspricht, aber für die Zwecke des Beispiels reicht es aus.

Bei den Vorteilen von OSGi (Moduldefinition, Sichtbarkeit, Services, Versionierung) wurde erklärt, dass eine Trennung zwischen Coding und Deklaration der Struktur der Anwendung förderlich ist. Mit Declarative Services ist es nun möglich, diesen Teil von OSGi deklarativ zu definieren und zu verwenden.

5.2.1 Declarative Services als Dependency-Injection-Framework

Alle Charakteristiken eines DI-Frameworks sind bei Declarative Services umgesetzt. Die Abhängigkeiten werden deklarativ definiert, die Instanziierung der Klassen übernimmt das Framework, genau wie das Injizieren der Instanzen in die verwendenden Klassen. In diesem Zusammenhang wird oft vom Hollywood-Prinzip gesprochen (»do not call us, we call you«). Damit soll zum Ausdruck gebracht werden, dass keine direkten Abhängigkeiten benötigt werden, sondern dass die Instanzen gesetzt werden.

Neben den Declarative Services gibt es zumindest noch drei Frameworks, die erwähnt werden müssen.

- Spring:
  Bei Spring handelt es sich um ein Open-Source-Framework, das neben anderen Features hauptsächlich ein DI-Framework enthält. Dort gelten dieselben Regeln, nämlich dass die Referenzen über Namen definiert und gesetzt werden. Spring erfreut sich in der Entwicklergemeinschaft großer Beliebtheit.
- Guice:
  Dieses Framework stammt von Google und ist ebenfalls Open Source. Der Funktionsumfang und die Verbreitung sind nicht ganz so groß wie bei Spring.
- CDI (Context and Dependency Injection):
  Der Java-EE-Standard ist weit verbreitet in der Implementierung von Enterprise-Anwendungen, sodass es in der neuesten Version (Java EE 6) eine Erweiterung für DI gibt (siehe [44]).

Diese Frameworks und ihre Möglichkeiten werden noch genauer analysiert, um während der Implementierung in einem Projekt zu einer guten Entscheidung bezüglich der Frameworks zu kommen.

5.3 Entkopplung in der Beispielanwendung

Nach der Beschreibung der Architektur und Umstellung auf Coding-Ebene geht Harald im Projekt eine Ebene höher und widmet sich in der Webanwendung der Komponentenarchitektur. Das Zusammenspiel der Komponenten innerhalb eines Systems soll basierend auf den Prinzipien der Modularisierung verbessert werden.

Das Management hat eine Reihe von Anforderungen bezüglich Erweiterbarkeit aufgestellt. Die Anwendung soll in der Lage sein, Daten von neuen Quellen zu verarbeiten, neue Datenanalysefunktionen zu unterstützen und die Datenanzeige soll später erweiterbar angelegt sein.

Wenn eine Anwendung oder ein Framework erweiterbar sein soll, dann ist es normalerweise zwingend notwendig, die nötigen Erweiterungsmöglichkeiten in der Anwendung vorzusehen. Als nächsten Schritt sollen die bestehenden Datenquellen und Datenanalysefunktionen auf demselben Weg in die Anwendung ein-

gebettet werden. Es darf nicht unterschieden werden zwischen Teilen der Anwendung, die zuerst vorhanden waren, und neu hinzugefügten.

In der Softwareentwicklung wird dieses Konzept meist als »eat your own dogfood« bezeichnet. Im Hinblick auf die bestehende Anwendung bedeutet das ein Refactoring, denn bisher ist sie nicht wirklich erweiterbar, ohne dass das Coding angepasst wird.

Modularity Pattern (Component Architecture):
Extensions – Restricting extensibility mechanism for modules

Beschreibung

In einem modularen System interagieren viele Module miteinander. Manche dienen als Framework, andere sind nur notwendige Erweiterungen, um eine bestimmte Business-Anforderung zu erfüllen. Die Frameworkmodule bieten die notwendigen Erweiterungspunkte, um die Erweiterungen zu integrieren und aufzurufen.

Es darf bei diesem System keinerlei Unterschiede zwischen den Erweiterungen geben. Es ist nicht akzeptabel, wenn bestimmte Grunderweiterungen innerhalb des Frameworks enthalten sind. Alle Erweiterungen müssen gleich behandelt werden und dieselben Regeln des modularen Laufzeitsystems erfüllen.

Komponenten

- Frameworkmodule beinhalten die Frameworks mit den Erweiterungspunkten. Kein Framework besitzt eine Erweiterung ähnlicher Funktionalität.
- Erweiterungspunkte dienen dem Hinzufügen neuer Funktionalität.
- Erweiterungen registrieren sich bei den Erweiterungspunkten.

Die Änderungen, die hier angestrebt werden, sind der erste Schritt zu einer modularen Architektur, bei der das Softwaresystem nicht mehr aus einem monolithischen Block besteht. Wenn zwischen den Modulen der Anwendung klare Schnittstellen eingezogen sind, dann hat man einen großen Schritt hin zu einem beherrschbaren System gemacht.

Nach dem Prinzip des Refactorings wird die Anwendung Schritt für Schritt angepasst, indem immer wieder ein Teilproblem herausgegriffen und verbessert wird. Das erste Teilproblem war die Struktur der Datenanalysefunktionen.

Ein Problem in viele kleine Teilbereiche zu zerlegen, sodass man für die Teilprobleme einfachere Lösungen findet, ist der einzig verlässliche Weg, wirklich komplexe Probleme zu lösen. Nichts anderes wird bei Softwaresystemen gemacht, wenn die Implementierung in kleinere Teilbereiche zerlegt wird. Die Teile sind einfacher zu beherrschen und zu verwalten.

In der Beschreibung zur Factory-Verwendung klang schon an, dass die Separierung von Schnittstelle und Implementierung ein Problem in der Anwendung ist. Schnittstellen dienen dazu, eine Abstraktion anzubieten, um dahinter die Implementierung auszutauschen oder variabel zu halten. Genau das ist eines der Ziele der Datenquellen. Wie vorher bereits erwähnt, funktioniert die Abstraktion allerdings nur unvollständig. Der erste Schritt, um eine vollständige Entkopplung

zu erreichen, besteht darin, dass ein weiteres Bundle eingeführt wird, das nur die Schnittstellenbeschreibung für die Datenquelle enthält. Jede Datenquelle wird dieses Bundle referenzieren, weil darin die grundlegenden Schnittstellen für eine Datenquelle definiert sind. Somit ist die Schnittstelle in einem Bundle definiert und jede Implementierung wird in einem separaten Bundle abgelegt.

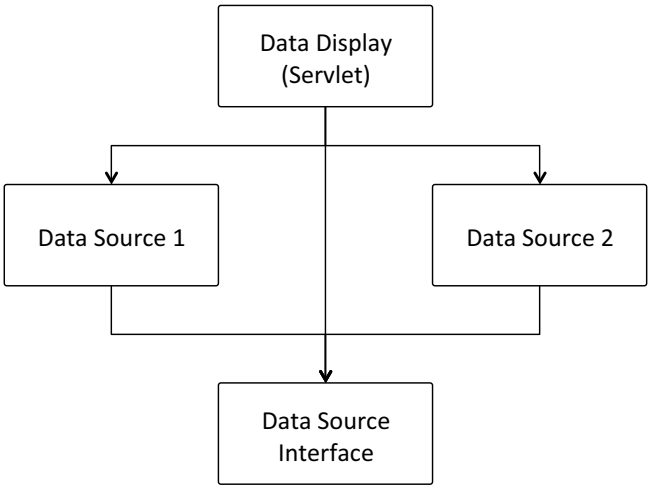

Abb. 5–5 *Struktur der Bundles für Schnittstellen und Implementierung*

Genau diese Aufteilung wie in Abbildung 5–5 implementiert Harald. Er erzeugt das Bundle mit der Schnittstellendefinition. Danach referenziert jede Implementierung die Schnittstelle im separaten Bundle. Im Moment hängt die Webanwendung von der Schnittstelle und der Implementierung in den jeweiligen Bundles ab, denn an diesen Abhängigkeiten wurde nichts verändert.

Allerdings ist die Schnittstelle einer Datenquelle schon in einem separaten Bundle klar definiert. Gerade im Rahmen von OSGi ist das ein gut anwendbares Pattern, um die Separierung zwischen Implementierung und Schnittstelle zu erreichen. Die Implementierungen in anderen Bundles sind dadurch modular.

```
public interface IDataSource {
    public List<IDataItem> getData();

 public String getArea();
}
```

Listing 5–2 *Definition des Interface der Datenquellen*

Es sollte an dieser Stelle nicht verschwiegen werden, dass man bei diesem Pattern mit Augenmaß vorgehen sollte, denn jede Schnittstelle in ein separates Bundle zu extrahieren würde zu einer undurchschaubaren Struktur führen. Hier sollte man die logisch zusammengehörenden Schnittstellen in ein Bundle separieren.

Bei diesem Design ist allerdings das Gegenteil zu bedenken. In der Softwarearchitektur ist das als Interface Segregation, beschrieben in [5] und [3], bekannt. Ein Anwender sollte nicht gezwungen sein, ein Interface zu referenzieren, das zu viel Funktionalität anbietet, die er nicht benötigt. Keines der beiden Extreme, sowohl zu feine Granularität als auch zu grobe Granularität, ist die ideale Lösung, sodass man einfach im Projektverlauf immer wieder prüfen muss, ob die aktuelle Trennung so noch sinnvoll ist. Im besten Fall führt man ein Refactoring durch und teilt ein Bundle mit mehreren Schnittstellen auf oder fügt zwei Bundles zusammen.

5.3.1 Entkopplung der Datenquellen

Bisher funktioniert die Benutzung von Datenquellen komplett per Coding. Der Anwender instanziiert die Datenquellen direkt. Das bedeutet leider, dass der Anwender im Detail wissen muss, welche Implementierungen existieren.

```
DataSource1 datasource1 = new DataSource1();
DataSource2 datasource2 = new DataSource2();
```

Listing 5–3 *Direkte Abhängigkeit zur Datenquellenimplementierung*

Es kann keine weitere Datenquelle hinzugefügt werden, ohne dass das Coding des Anwenders angepasst wird. Das Ziel der Modularität kann so nicht erreicht werden, denn dazu darf der Anwender nichts über die konkreten Implementierungen wissen, sondern kann die Datenquellen nur über die Schnittstelle benutzen.

Eine Factory leistet für solche Zwecke im Allgemeinen sehr gute Dienste, aber in diesem Fall ist der Anwender nicht daran interessiert, eine Instanz zu erhalten, sondern eine Liste aller Datenquellen, sodass die Daten aggregiert werden können.

Es wird eine Instanz benötigt, die nach allen Datenquellen gefragt werden kann. Diese Instanz kann nicht beim Anwender liegen und auch nicht in einer Datenquelle. Das Bundle, in dem die Schnittstellendefinition für Datenquellen liegt, würde sich dazu optimal eignen. Es geht darum, allgemeine Services für die Verwendung von Datenquellen in einem Bundle zusammenzufassen.

Harald nimmt sich der wirklichen Erweiterbarkeit auf Datenquellenebene an. Das Ziel besteht darin, dass die Webanwendung nur noch eine Referenz auf das Bundle enthält, in dem die Schnittstelle definiert ist. Alle anderen Datenquellen registrieren sich selbstständig beim Laufzeitsystem, sodass keine Änderung der Anwendung mehr notwendig ist für eine neue Datenquelle. Das Versprechen des OSGi-Laufzeitsystems, dass ein neues Bundle während der Laufzeit automatisch hinzugeladen wird und verfügbar ist, soll eingelöst werden. Die neue Struktur der Anwendung zeigt Abbildung 5–6.

Dazu implementiert Harald im Interface Bundle einen Datenquellen-Provider. Bei diesem Provider registrieren sich alle Datenquellen, und eine Anwendung kann abfragen, welche Datenquellen vorhanden sind. Zur Implementierung greift Harald auf Declarative Services zurück.

Der Provider wird das folgende Interface implementieren:

```
public interface IDataSourceService {
      public void addDataSource(IDataSource dataSource);
      public List<IDataSource> getDataSources();
}
```

Listing 5–4 *Definition des Service zur Datenquellenverwaltung*

Im Falle des Datenquellen-Providers implementiert Harald den Provider als OSGi-Service. Sowohl die Registrierung im Interface Bundle als auch die Registrierung der Datenquellen wird deklarativ vorgenommen. Das OSGi-Declarative-Services-Laufzeitsystem instanziiert zunächst einmal ein Objekt vom Typ des Datenquellen-Providers. Das wird durch die Angabe der Implementierungsklasse in der Beschreibungsdatei *service-description.xml* im Verzeichnis OSGI-INF erreicht.

```
<scr:component xmlns:scr=http://www.osgi.org/xmlns/scr/v1.1.0
        name="Data Source Service">
   <implementation class="arch.datasourceinterface.DataSourceServiceImpl"/>
   <service>
      <provide interface="arch.datasourceinterface.IDataSourceService"/>
   </service>
</scr:component>
```

Listing 5–5 *Exponieren eines Service*

Wenn das Objekt des Datenquellen-Providers im OSGi-Laufzeitsystem existiert, werden die Deskriptordateien der Declarative Services analysiert und dabei stellt das Laufzeitsystem fest, dass eine Referenz auf den Provider jeweils in jede Datenquelle injiziert werden soll. Bei diesem Verfahren handelt es sich um Dependency Injection [6], diesmal als Architekturprinzip eingesetzt, um die Entkopplung von Datenquellen und Anwendungen zu erreichen. Da die Datenquellen den Datenquellen-Provider kennen, registrieren sie sich selbstständig.

Im letzten Schritt wird der Datenquellenservice in die Webanwendung injiziert. Die Referenz ist immer gesetzt und jedes Mal, wenn die Anwendung die Daten holen will, wird der Datenquellen-Provider nach der Menge aller Datenquellen gefragt.

```
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0" name="Servlet
Component">
  <implementation class="arch.datadisplay.ui.DisplayServlet"/>
  <reference bind="setDataSourceService"
        interface="arch.datasourceinterface.IDataSourceService"
        name="dataSourceService" unbind="unsetDataSourceService"/>
  <service>
     <provide interface="arch.datadisplay.ui.DisplayServlet"/>
  </service>
</scr:component>
```

Listing 5–6 *Injizieren des Datenquellenservice*

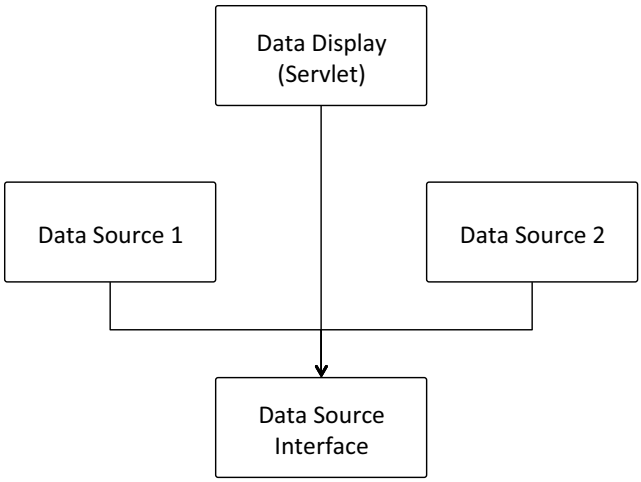

Abb. 5–6 *Struktur mit Datenquellenservice*

Hier zeigt sich jetzt der Vorteil der Abstraktion über eine Schnittstelle für Datenquellen. Alle Datenquellen werden gleich behandelt, und auf die verschiedenen Implementierungen in den Bundles kann einheitlich zugegriffen werden.

Die Verwendung des Service, um Datenquellen bereitzustellen, wird angepasst:

```
ArrayList<IDataSource> list = dataSourceService.getDataSources();
```

Listing 5–7 *Verwendung des Datenquellenservice*

Wenn eine weitere Datenquelle hinzukommt, existiert einfach ein weiterer Eintrag in der Liste. Die Verwendung ist komplett unabhängig von den Implementierungen der Datenquellen, und mithilfe des OSGi-Laufzeitsystems ist es möglich, Datenquellen dynamisch hinzuzufügen oder während der Laufzeit zu entfernen.

Modularity Pattern (Component Architecture):
Module services – Using services as communication means between modules

Beschreibung
Services sind ein integraler Bestandteil eines modularen Laufzeitsystems. Es ist die Intention, dass Wiederverwendung über Modulgrenzen hinweg über die Services in der Service Registry passiert. Daher sollte ein Entwickler Funktionalität in einem anderen Modul niemals direkt aufrufen, sondern immer nach dem entsprechenden Service fragen. Damit sind die dynamischen Aspekte eines modularen Laufzeitsystems beachtet und der Service nimmt an der Versionierung teil.

Komponenten

- Anbieter der Funktionalität stellt die Logik bereit.
- Service Registry stellt das Verzeichnis aller wiederverwendbaren Funktionalität dar.
- Serviceinterface ist die Abstraktion der Funktionalität. Über dieses Interface wird in der Service Registry die Funktionalität zugreifbar gemacht.
- Serviceinstanz ist das Serviceobjekt, das in der Service Registry abgelegt wird für beliebige Anwender.
- Anwender fragen die Service Registry nach einer Serviceinstanz und benutzen die Funktionalität dann über das Interface.

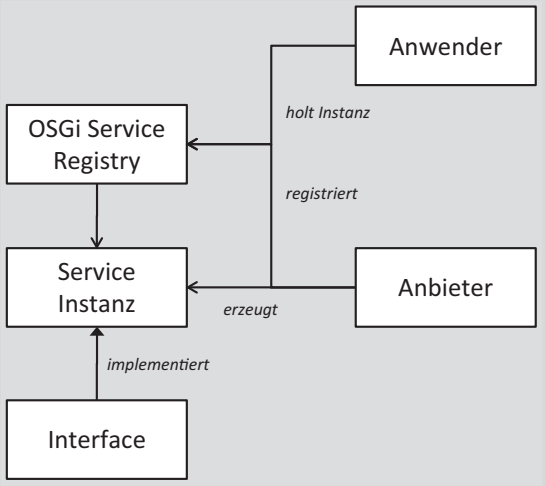

Abb. 5–7 *Beziehung der Komponenten zu Services*

5.3.2 Nachstellen des Beispiels

Die Quellen des Beispiels finden sich auf GitHub unter:

- *git@github.com:ModularityExamples/Session2.git*.

Um das Beispiel nachzustellen, importiert man das Git-Repository in einen Eclipse Workspace und erzeugt die Projekte.

5.3.3 Laufzeitdynamik von Datenquellen

An dieser Stelle des Projektes ist die Anwendung das erste Mal so flexibel aufgebaut, dass die Laufzeit dynamisch angepasst werden kann. Durch die Separierung der Datenquellen können bestehende Datenquellen entfernt oder neue hinzugenommen werden. Harald führt diesen Vorteil in einer kurzen Demo im Projektmeeting vor.

Dazu startet er das OSGi-Laufzeitsystem und greift von der Konsole darauf zu. Die Steuerung des Laufzeitsystems ist eine gute Möglichkeit, um die Flexibilität zu testen. Harald lässt sich daher als Erstes eine Liste der laufenden Bundles ausgeben (siehe Appendix C):

```
34    ACTIVE    DataSourceInterface_1.0.0.build20120412
37    ACTIVE    DataSource3_1.0.0.build20120412
38    ACTIVE    DataSource1_1.0.0.build20120412
39    ACTIVE    DataSource2_1.0.0.build20120412
40    ACTIVE    DataDisplay_1.0.0.build20120412
```

Wie man sieht, existieren im Moment drei Datenquellen, das Interface Bundle und das Servlet Bundle. Alle Bundles sind aktiviert, und die Ausgabe erfolgt wie in Abbildung 5–8.

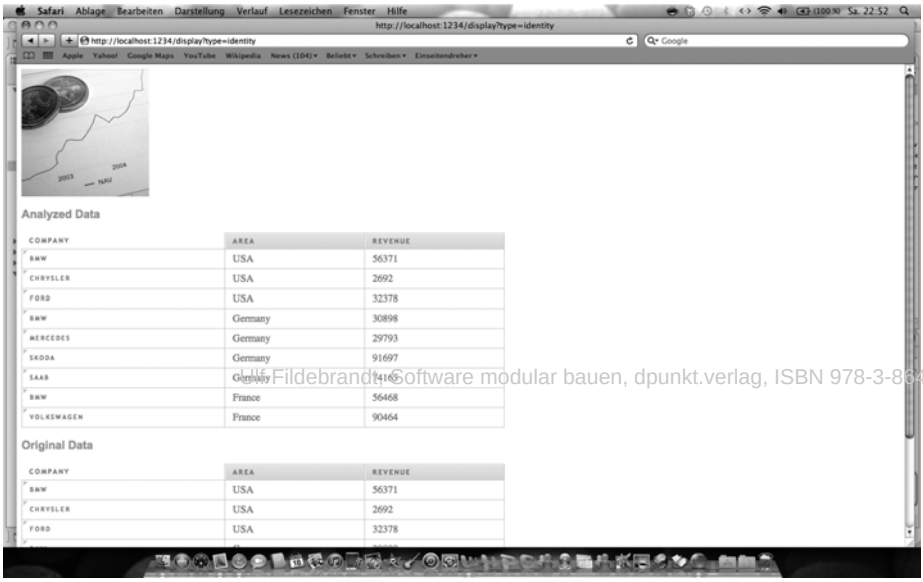

Abb. 5–8 *Ausgabe mit allen Datenquellen*

Um die Dynamik zu demonstrieren, stoppt Harald eines der Datenquellen-Bundles. Damit ändert sich die Ausgabe der aktiven Bundles auf der Konsole wie folgt:

```
34    ACTIVE    DataSourceInterface_1.0.0.build20120412
37    RESOLVED  DataSource3_1.0.0.build20120412
38    ACTIVE    DataSource1_1.0.0.build20120412
39    ACTIVE    DataSource2_1.0.0.build20120412
40    ACTIVE    DataDisplay_1.0.0.build20120412
```

Eines der Datenquellen-Bundles befindet sich im Zustand resolved. Damit ist in OSGi gemeint, dass das Bundle zwar geladen ist, aber nicht läuft. Es kann jederzeit gestartet werden.

Da es aber nicht aktiv ist, ändert sich die Ausgabe der Anwendung wie in Abbildung 5–9 gezeigt.

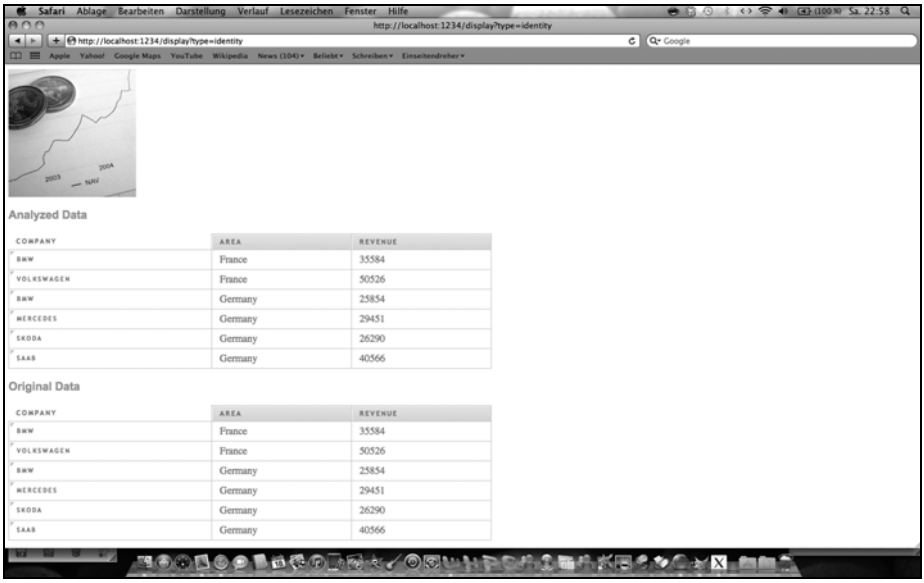

Abb. 5–9 *Ausgabe mit einem Bundle weniger*

Es soll an dieser Stelle nicht verschwiegen werden, dass für dieses Ergebnis bei der aktuellen Implementierung das Durchstarten (Stoppen, anschließendes Starten) des DataSourceInterface-Bundles notwendig ist, da dort statische Referenzen gehalten werden. Das ist kein optimales Design, aber es war Ziel, die Anwendung möglichst einfach zu halten. OSGi bietet durchaus die Möglichkeit, alle Referenzen zu verwenden und Serviceverwendungen korrekt zu handhaben.

Gerade Declarative Services vereinfachen das Handling. Um die Dynamik korrekt zu implementieren, muss die Registrierung und Deregistrierung eines Service in der richtigen Weise vonstattengehen.

Das alte Coding der Registrierungsklasse sah so aus:

```
public class RegisterComponent {
   protected void setDataSourceService(
                IDataSourceService dataSourceService) {
      dataSourceService.addDataSource(new DataSource1());
    }

   protected void unsetDataSourceService(
                IDataSourceService dataSourceService) {
   }
}
```

Listing 5–8 *Einfache Registrierung*

Beim Service *DataSourceService* wird die neue Datenquelle bekannt gemacht, aber in der zweiten Methode, die aufgerufen wird, wenn das Bundle gestoppt wird, wird diese Serviceinstanz nicht mehr aus der Menge der Datenquellen entfernt. Das führt dazu, dass in der Liste des Datenquellenservice immer noch eine Referenz auf die Datenquelle gespeichert wird, sodass der Garbage Collector in Java die Serviceinstanz nicht abräumen kann.

Um dieses Problem zu beheben, muss die Serviceinstanz aus der Liste gelöscht werden.

```
public class RegisterComponent {

   private DataSource1 dataSource = null;

   protected void setDataSourceService(
                    IDataSourceService dataSourceService) {
   dataSource = new DataSource1();
   dataSourceService.addDataSource(dataSource);
   }

protected void unsetDataSourceService(
                    IDataSourceService dataSourceService) {
   dataSourceService.removeDataSource(dataSource);
   dataSource = null;
   }
}
```

Listing 5–9 *Korrekte Registrierung und Deregistrierung*

Declarative Services liefern mit den beiden Methoden bereits alles, um die Laufzeitdynamik, die OSGi bietet, auf einfachem Wege auszunutzen.

Es kann nicht genug betont werden, wie wichtig diese Eigenschaft ist, denn eine modulare Architektur macht nur Sinn, wenn auch Vorteile aus der Flexibilität gezogen werden können. Ausgehend von den Anforderungen haben die Geschäftsexperten diese Art der Dynamik durchaus verlangt. Die Ausnutzung der Dynamik im modularen Laufzeitsystem ist daher kein Selbstzweck, sondern erfüllt eine Anforderung aus dem Geschäftsbereich.

5.4 Exkurs: Zwei weitere SOLID-Prinzipien

Nach der Umsetzung der Datenquellenseparierung nimmt sich Harald in dem Projekt ein wenig Zeit, um die Prinzipien aufzuschreiben und im nächsten Projektmeeting zu erklären.

Zu Anfang wurde auf die Prinzipien in SOLID verwiesen [3]. Auf eines der Prinzipien wurde schon näher eingegangen, das Single-Responsibility-Prinzip. Im Verlauf dieses Kapitels wurde Dependency Inversion (Dependency Injection) angewendet. Zwei der SOLID-Prinzipien sind auch für Module sinnvoll:

- Single-Responsibility-Prinzip
- Open-Closed-Prinzip

Mehr Details über diese Prinzipien zu wissen, hilft, die Implementierung der Module zu verbessern.

5.4.1 Single-Responsibility-Prinzip für Module

Angewendet wird das Single-Responsibility-Prinzip meist auf der Coding-Ebene. Aber über die reine Anwendung auf Coding hinaus können die Prinzipien auch auf größere Bausteine einer Systemarchitektur angewendet werden. Man stelle sich nur vor, dass die Komponenten eines Systems keine klar umrissene Aufgabe besitzen. Eine Komponente könnte zwei Aufgaben übernehmen. Für gewöhnlich würden die Aufgaben miteinander vermischt und keine Aufgabe würde richtig bearbeitet werden. Auch der Aufruf solcher Funktionen wäre sehr aufwendig, denn es müsste immer angegeben werden, welcher Teil der Funktionalität gerade aufgerufen wird.

Auf OSGi angewendet würde das bedeuten, dass ein Bundle immer eine klar umrissene Aufgabe haben sollte. Sofern ein Bundle mehrere Aufgaben übernehmen soll, sollte man über die Trennung der Funktionalität nachdenken. Als erster Hinweis auf eine unklare Aufgabe eines Bundles kann wieder die Tatsache dienen, dass eine Diskussion über den Namen des Bundle im Projekt stattfindet. Das deutet wie bei Klassen oder Methoden darauf hin, dass der Anwendungsbereich nicht eindeutig ist.

5.4.2 Open-Closed-Prinzip für Module

In der Theorie besagt das Prinzip, dass eine Implementierung offen für Erweiterungen, aber geschlossen für Modifikationen sein soll. Dieses Prinzip wurde angewendet bei der Umsetzung der Datenquellenerweiterbarkeit.

Zunächst einmal sind für externe Anwender nur Schnittstellen verfügbar. Nach den Sichtbarkeitsregeln von OSGi sind die Implementierungen jeweils verborgen und extern nicht verfügbar. Schnittstellen sind für zwei Fälle gedacht:

- API (Application Programming Interface):
  Die Schnittstellen sind als Zugriffsschicht vorgesehen wie im Fall des Datenquellenservice, um alle Datenquellen zu ermitteln.
- SPI (Service Provider Interface):
  Die Schnittstellen können verwendet werden, um dem Framework eigene Erweiterungen bekannt zu machen. Diese SPI-Definition wurde bei den Datenquellen angewendet.

Dieses Vorgehen entspricht der normalen Entwicklung mit Java. Es ist ein Prinzip, das zunächst einmal auf der untersten Architekturebene, der Coding-Ebene, angewendet wird. Das Verfahren soll an einem kleinen Beispiel demonstriert werden.

```
public final class ModuleOperations {
      public ModuleOperations() {
      }
      public void createModule(final String name) {
      }
      public void deleteModule(final String name) {
      }
}
```

Listing 5–10 *Alle Operationen in einer Klasse*

Die Klasse *ModuleOperations* ist nicht offen für Erweiterungen, denn wenn eine neue Operation hinzugefügt werden soll, muss die Klasse geändert werden. Offen für Erweiterungen meint, dass eine Implementierung ohne Modifikation an neue Anforderungen angepasst werden kann.

Die Lösung besteht darin, dass jede Operation einzeln betrachtet wird.

```
public abstract class Operation {
      public Operation() {
      }
      abstract public void execute();
}

   public class CreateModuleOperation extends Operation {
      public CreateModuleOperation() {
      }
      @Override
      public void execute() {
      }
}

   public class DeleteModuleOperation extends Operation {
      public DeleteModuleOperation() {
      }
      @Override
      public void execute() {
      }
}
```


Listing 5–11 *Nach Anwendung von Open-Closed- und Single-Responsibility-Prinzipien*

Die zweite spezielle Anwendung des Prinzips besteht in der Ausnutzung der Sichtbarkeitsmechanismen von OSGi. Jedes Bundle hat nur die absolut notwendigen Klassen und Interfaces zu exportieren. Alle anderen Objekte befinden sich in Packages, die nicht exportiert werden. Vor allen Dingen bezieht sich das auf die Implementierungsklassen. Keine Implementierung ist außerhalb eines Bundles verfügbar.

Von daher ist die Implementierung der Datenquellen geschlossen für Modifikationen, aber durch die Möglichkeit, neue Datenquellen einzuhängen, offen für Erweiterungen. Die Sichtbarkeitsverwaltung von OSGi sollte man auf jeden Fall ausnutzen während der Implementierung.

Modularity Pattern (Component architecture):
Open-closed modules – Using Open Closed principle for modules

Beschreibung

Offen ist eine Implementierung in der modularen Welt, wenn die Komponente Schnittstellen für Erweiterungen bereitstellt, die von Anwendern in anderen Komponenten implementiert werden können. Diese Schnittstellen befinden sich deshalb immer in den exportierten Packages eines Moduls.

Die Sichtbarkeitsregeln des modularen Laufzeitsystems definieren, dass Packages, die nicht exportiert werden, für andere Anwender überhaupt nicht sichtbar sind. Aus diesem Grund sind die Klassen in den nicht exportierten Packages geschlossen gegenüber Modifikationen. Niemand hat Zugriff auf sie.

Somit zeigt sich, dass für die Umsetzung des Open-Closed-Prinzips die Sichtbarkeitsregeln des modularen Laufzeitsystems ausgenutzt werden müssen.

Komponenten

- Exportierte Packages eines Moduls enthalten alle Interfaces und Klassen, die offen für Erweiterungen sein sollen.
- Verborgene Packages eines Moduls sind für die Geschlossenheit gegenüber Modifikationen verantwortlich.

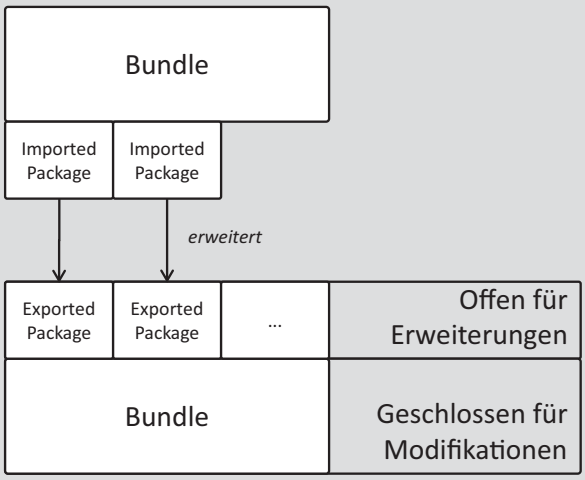

Abb. 5–10 *Open-Closed-Prinzip-Beziehung der Komponenten*

5.5 Hierarchien in der Beispielanwendung

Nachdem Harald die erste Änderung an der Struktur der Anwendung gemacht hat, kommen die Projektmitglieder und fragen, warum er es so kompliziert macht und die Funktionalität des Datenquellen-Providers nicht einfach in das Bundle für die Webanwendung integriert. Es wäre sicherlich möglich, Schnittstellendefinition und Datenquellen-Provider in dieses Bundle zu packen, sodass die Datenquellen einfach nur auf die Webanwendung zeigen und sich dort registrieren. Diese Lösung würde dieselbe Anforderung nach Erweiterbarkeit erfüllen wie die jetzige Implementierung.

Harald antwortet ihnen, dass er hierbei nach dem Prinzip von Separation of Concerns, einem weiteren Architekturprinzip, vorgegangen ist. Zwar kennen seine Projektmitglieder dieses Konzept, aber wie schon eingangs erwähnt fällt das Mapping auf eine konkrete Implementierung bzw. ein konkretes Produkt schwer.

Daher beginnt Harald die Schichten der Anwendung wie in Abbildung 5–11 an eine Tafel zu malen. Dabei entsprechen die einzelnen Bausteine im Diagramm jeweils einem Bundle in OSGi. Ganz unten fügt er die Definitionsschicht für Datenquellen ein. Darin sind die Schnittstellendefinition und die allgemeine Registrierungsfunktion enthalten. Das sind allgemein wiederverwendete Funktionen, sodass sie für alle verfügbar sein sollten. Darüber fügt Harald jetzt eine Schicht ein, in die er alle Datenquellen-Bundles einfügt. Er hat sie darüber gemalt, um anzudeuten, dass Abhängigkeiten nur von oben nach unten gehen dürfen, was hier auch der Fall ist. Die letzte Schicht, die Harald einfügt, besteht aus einer Webschicht oder allgemeiner aus einer User-Interface-Schicht.

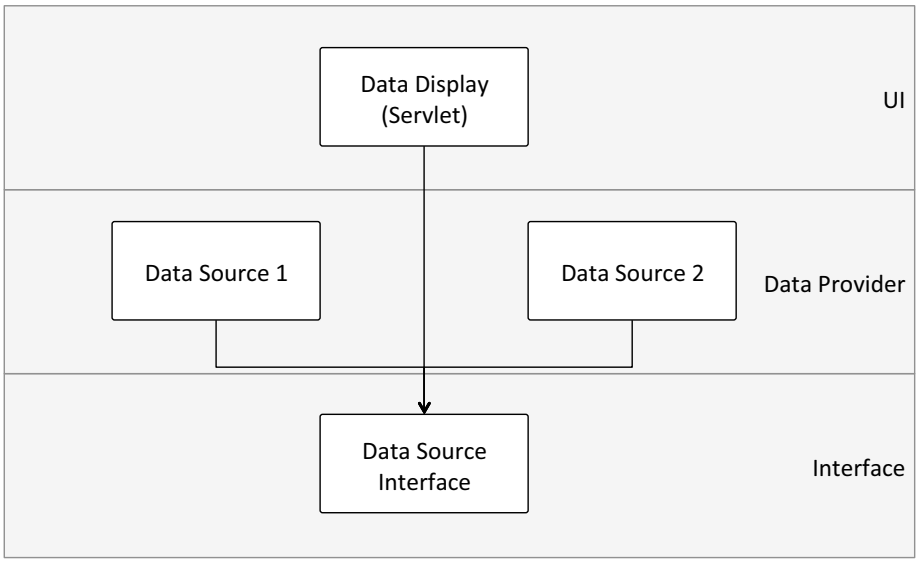

Abb. 5–11 *Schichten der Anwendung*

Jede Schicht besitzt ihren eigenen Zweck. Die Funktionalität der Schichten zu vermischen würde zu dem führen, was allgemein als Spaghetticode bezeichnet wird, das wäre das Ende für jede Weiterentwicklung eines Projektes. Am Ende plaudert Harald noch ein wenig aus dem Nähkästchen und berichtet, dass er schon viele Projekte gesehen hat, auch erfolgreiche, bei denen die Schichtenarchitektur verletzt wurde, das heißt, dass eine Abhängigkeit von einer tieferen Schicht zu einer höheren geht. Solche Abhängigkeiten entstehen meist in der Weiterentwicklung eines Projektes. Die Projektmitglieder implementieren eine Abhängigkeit, weil es einfacher ist und weniger Zeit kostet. Im ersten Moment ist keine negative Wirkung auf das Projekt feststellbar. Später wird es ungleich schwieriger, neue Funktionalität in diesem Projekt zu implementieren, weil man nicht mehr genau feststellen kann, wer welche Funktionalität verwendet, und es an allen Ecken und Enden Seiteneffekte geben kann. Umfangreiche Testkataloge sind meist der Ausweg der Projektleiter aus diesem Dilemma, aber nichtsdestotrotz wird es für die Entwickler sehr schwer, denn die Architektur des Projektes ist »erodiert«.

Modularity Pattern (Component Architecture):
Layer – Using layers in a modular runtime

Beschreibung
Die Schichten in einer modularen Laufzeit werden durch die Zuweisung der Module zu den Schichten definiert. Ein modulares Laufzeitsystem wurde gerade aus dem Grunde eingeführt, um eine gröbere Struktur der Modularisierung zu ermöglichen im Vergleich zu einfachen Klassen oder Packages. Abhängigkeiten sind nur erlaubt von einem Modul in einer Schicht zu einem Modul in einer tieferen Schicht. Verwendungen von Modulen von einer tieferen Schicht zu einer höheren sind strengstens verboten.

Genau wie bei der Implementierung sollten bei Modulen keine zyklischen Abhängigkeiten eingeführt werden. Die Module sind demnach ein ebenso gut strukturierter Ansatz wie die Klassen auf Coding-Ebene.

Komponenten
- Schichten (Layer) sind semantische Gruppen gleicher Funktionalität. Zyklische Abhängigkeiten zwischen Schichten sind nicht erlaubt.

Module werden in die Schichten einsortiert. Hierbei wird zur Definition von Verwendung und öffentlicher Schnittstelle wieder auf die importierten und exportierten Packages zurückgegriffen. →

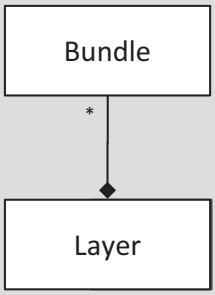

Abb. 5–12 *Beziehungen der Komponenten in den Schichten*

5.6 Schnittstellen in Modulen

Neben den Design Patterns und Konzepten, die für eine Modularisierung notwendig sind, soll noch kurz auf die Umsetzung des einfachen Beispiels zur Selbstähnlichkeit der Architektur eingegangen werden.

In der Coding-Architektur war die Umsetzung einfach, denn es wurde direkt das Sprachkonstrukt verwendet. Ein wenig komplizierter ist der Fall schon bei einem modularen Laufzeitsystem. Wie wird hier eine Separierung von Schnittstelle und Implementierung durchgeführt?

In der Beispielanwendung hat Harald die Lösung schon umgesetzt. Die Antwort liegt im Refactoring der Datenquellen. Hierbei wurde ein Bundle eingeführt, das nur die Schnittstellendefinition beinhaltet. Die Implementierung der Datenquellen wurde in separate Bundles ausgelagert. Demnach ist die Separierung von Schnittstelle und Implementierung in einem modularen Laufzeitsystem in OSGi umgesetzt, indem ein Bundle definiert wird, das die Schnittstelle beinhaltet, und ein oder mehrere Module für die Implementierung.

Im Gegensatz zum Coding, wo direkt ein Sprachkonstrukt verwendet wurde und die Unterscheidung zwischen Schnittstelle und Implementierung sofort ersichtlich war, musste bei Bundles erst ein Mapping ausgeführt werden. Sowohl die Schnittstelle als auch die Implementierung landen in demselben technischen Typ, einem Bundle. Nur durch deren Semantik wird die Unterscheidung offensichtlich. Es handelt sich hierbei um die Anwendung eines Patterns.

Abschließend kann man festhalten, dass sich das Pattern im Coding auf die nächsthöhere Ebene der Architektur übertragen lässt. Allerdings ist hierbei die Verwendung der technischen Konstrukte dieser technischen Ebene notwendig: Bundles.

Modularity Pattern (Component Architecture):
Dependency injection – Using dependency injection in a modular runtime

Beschreibung
Dependency Injection ist ein wichtiges Konzept für die Entkopplung der Implementierung, denn hierbei wird die Implementierung in kleinere, beherrschbare Teile aufgeteilt, ohne dass sie voneinander abhängen.

In modularen Laufzeitsystemen sollte es eine Umsetzung von Dependency Injection geben, um die Entkopplung zu erreichen. Hierbei bietet es sich in einem modularen Laufzeitsystem an, Serviceimplementierungen als injizierte Objekte zu verwenden.

Die Services werden nicht per Coding exponiert, sondern das Service-Laufzeitsystem entnimmt die Instanzinformationen aus der Deskriptordatei. Alle Serviceinstanzen werden erzeugt, in die Service Registry exponiert und in die anderen Module injiziert. Die Kommunikation zwischen den Modulen erfolgt komplett über Services .

Komponenten
- Serviceinstanz: erzeugendes Modul definiert die Service-Instanzen.
- Serviceinstanz: empfangendes Modul definiert alle Verwendungen von Service-Instanzen. Hier sind nur Abhängigkeiten auf Schnittstellen erlaubt.
- Service-Laufzeitsystem: erzeugt die Serviceinstanzen und injiziert sie in die empfangenden Module.

Abb. 5–13 *Dependency Injection – Beziehungen der Komponenten*

5.7 Zusammenfassung

In diesem Kapitel wurde die Komponentenarchitektur der Anwendung umgestellt. Es ging nicht so sehr um das Coding an sich, sondern viel mehr darum, in welcher Granularität Komponenten angelegt werden sollten. Dabei wurde die Frage untersucht, unter welchen Umständen Funktionalität in einer Schnittstelle und einem dazugehörigen Bundle zusammen angelegt oder aufgrund der Aufgaben besser getrennt gehalten werden sollte.

Einen großen Raum nahm die Beschreibung von Dependency Injection als Prinzip für das Coding, aber auch für das Komponentendesign eines Systems ein. Im Beispiel wurde es auf die Datenquellen und die Datenanalysefunktionen angewendet.