Kommunikation zwischen entkoppelten Java-Modulen über strukturell typkonforme Objekte

Niels Gundermann

26. Mai 2020

1 Einleitung

Die Modularisierung ist ein gängiges Mittel zur Beherrschung komplexer Softwaresysteme. Die Kommunikation zweier Module wird dabei durch eine vorab definierte Schnittstelle gewährleistet. Bei der Kommunikation kann es sich lediglich um den Aufruf eines Dienstes handeln, oder um einen Datenaustausch über so genannte Transfer-Objekte.

In der Programmiersprache Java werden diese Schnittstellen im Allgemeinen häufig als Interfaces definiert und gliedern sich somit in die Typ-Hierarchie des Programms ein. Soll ein Modul A mit einem Modul B kommunizieren, so müssen beide Module ein Interface I als Schnittstelle kennen und sind damit abhängig von diesem. Wenn es zu einem Datenaustausch über I kommen soll, so müssen die beiden Module darüber hinaus die Typen kennen, durch die die Transfer-Objekte abgebildet werden (Transfer-Typen).

Die Konformität der Typen (Transfer-Typen und Interfaces) wird in Java auf nominaler Ebene, also auf der Basis der Bezeichnung des jeweiligen Typs, sichergestellt (Nominale Typkonformität). Die dadurch entstehende Abhängigkeit führt zu einer Behinderung möglicher paralleler Arbeiten an diesen Modulen - insbesondere dann, wenn die Schnittstelle im Zuge der Arbeiten angepasst werden muss und die beiden Module im Verantwortungsbereich unterschiedlicher Entwicklerteams liegen.

Ein anderer Ansatz zur Sicherstellung der Typkonformität beruht auf dem Abgleich der strukturellen Eigenschaften von Typen (Strukturelle Typkonformität). Dabei werden die Transfer-Typen und Interfaces, die für die Kommunikation zwischen zwei Modulen (A und B) benötigt werden, innerhalb beider Module definiert, sodass jedes Modul seine eigenen Typen bereitstellt. Die beiden Module wären somit voneinander und von einer gemeinsamen Schnittstelle (I) syntaktisch unabhängig.

Es gab bereits Überlegungen dazu, wie eine strukturelle Typkonformität in der Programmiersprache Java umgesetzt werden könnte (vgl. [8], [9]). Die Arbeit von Läufer et al. ([9]) beschränkt sich dabei jedoch nur auf die Konformität zwischen Klassen und Interfaces und bedingt eine Anpasssung des Java-Compilers. Bei der Lösung von Gil et al. ([8]) handelt es sich um eine Spracherweiterung, was die Integration in bestehende Systeme erheblich erschwert.

Die vorliegende Arbeit befasst sich mit einem Ansatz der einerseits als Java-Bibliothek integriert werden kann und andererseits auch die Konformität zwischen Klassen als Transfer-Typen herstellen soll. Aufgrund der Tatsache, dass die Methoden strukturell typkonformer Objekte, in unterschiedlichen Modulen auch unterschiedlich implementiert werden können, muss entschieden werden, welche der Implementierung letztendlich verwendet werden soll. Auf dieses Problem wird ein besonderer Fokus innerhalb dieser Arbeit gelegt. Dies betrifft natürlich nicht nur Methoden-Implementierungen in Klassen, sondern auch default-Methoden in Interfaces.

1.1 Problembeschreibung

In dieser Arbeit werden zwei Szenaien betrachtet, die unterschiedliche Probleme aufzeigen. In beiden Fällen wird ein Ausschnitt aus einem System beschrieben, dessen Aufbau den Prinzipien einer strengen Schichtenarchitektur folgt (siehe 3.1).

1.1.1 Szenario 1

Auf architektonischer Ebene kann das erste Szenario, wie im Abbildung 1 folgt dargestellt werden. Die Module *A* und *B* liegen architektonisch auf der gleichen Ebene und dürfen

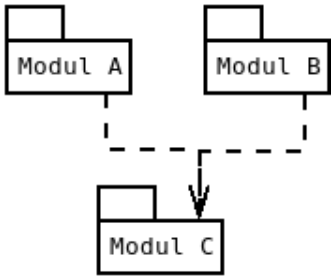

Abbildung 1: Problem: Szenario 1

somit keine direkte Abhängigkeiten aufweisen. Das Modul *C* stellt eine Abstraktionsebene dar, die für das gesamte System verwendet wird. Änderungen an diesem Modul würden demnach nicht nur die Module *A* und *B* betreffen, sondern auch noch weitere Module, die in Abbildung 1 nicht aufgeführt sind. Zudem soll zusätzlich davon ausgegangen werden, dass die Module *A* und *B* im Verantwortungsbereich eines Eintwicklerteams *E1* liegen, während das Modul *C* im Verantwortungsbereich eines Entwicklerteams *E2* liegt.

Nun wird eine abstrakte Klasse aus Modul C in Modul B konkret implementiert. Weiterhin werden Objekte, die in Modul B erzeugt werden in Modul A verwendet. Die Verwendung erfolgt jedoch über die abstrakte Klassendefinition aus Modul C.

Aufgrund einer geänderten Anforderung muss innerhalb von Modul A eine Information genutzt werden, die bereits in der konkreten Implementierung aus Modul B vorliegt, aber nicht in der abstrakten Implementierung aus Modul C zur Verfügung steht. In diesem Szenario gäbe es aufgrund der nominalen Typkonformität folgenden Lösungsvarianten:

1. Die abstrakte Impementierung wird um diese Information erweitert.
2. Es wird eine weitere Abstraktionsschicht zwischen den beiden vorliegenden Schichten eingebaut.

Beide Lösungsvarianten führen zu relativ hohem Anpassungsaufwand, wenn man bedenkt, dass die benötigte Information bereits zur Verfügung steht.

1.1.2 Szenario 2

Das zweite Szenario bezieht sich auf eine Serviceorientierte Architektur (siehe 3.2). Abbildung 2 zeigt den angenommenen Ausschnitt aus einem System.

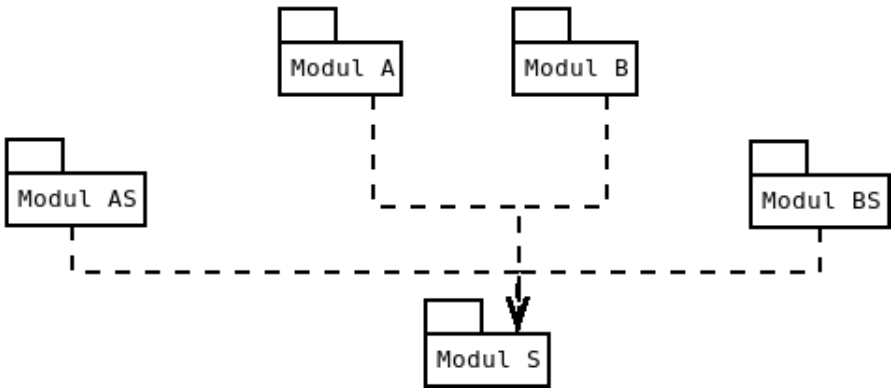

Abbildung 2: Problem: Szenario 2

Hierbei wird von einem Broadcast Serviceaufruf ausgegangen. Sas bedeutet, dass es eine Service-Schittstelle gibt, die in mehreren Modulen implementiert wird. Die Aufrufer liegen in diesem Fall in Modul A und B, während die Service-Schnittstelle in Modul S liegt. Die weiteren Module beinhalten unterschiedliche Implementierungen des augerufenen Services. Weiterhin ist anzunehmen, dass alle Module im Verantwortungsbereich unterschiedlicher Entwicklerteams liegen.

Nun ist davon auszugehen, dass innerhalb von Modul A eine weitere Information durch den Service bereitgestelt werden soll. Hierbei gibt es folgende Lösungsansätze:

1. Die Service-Schnittstelle wird erweitert.

2. Es wird eine neue Service-Schnittstelle geschaffen, die künfitg nur von Modul *A* verwendet wird.

Beide Lösungsansätze erfordern wiederrum erheblichen Aufwand und Koordination zwischen den Entwicklerteams. Daber ist zu erwähnen, dass der zweiter Lösungsansätz etwas weniger Aufwand erfordert, da die Entwicklerteams, deren Service-Implementierungen ohnehin in Modul *A* keine Verwendung finden, nicht beteiligt sind.

2 Typen und Typkonformität

[11]

2.1 Typen in Java

2.2 Typkonformität in Java

[9], [4]

2.3 Formale Definition struktureller Typkonformität in Java

[9]

3 Softwarearchitektur

[3]

3.1 Schichtenarchitektur

[10]

3.2 Serviceorientierte Architektur

[10]

3.3 Schnittstellen

[3], [5]

4 Lösungsansätze

4.1 Bestehende Lösungen

In den folgenden Kapiteln wird auf die Lösungsmöglichkeiten der in 1.1 beschriebenen Szenarien mit den bestehenden Lösungen nach [9] und [8] eingegangen. Die hier beschriebenen Lösungsansätze basieren lediglich auf den theoretischen Ausführungen bzgl. der allgemeinen Ansätze. Es wurde kein praktischer Nachweis in Form einer Implementierung erbracht,

der die theoretischen Grundlagen aus [9] und [8] bestätigt. Das Ziel dieses Abschnittes der Arbeit ist es, grundlegende Konzepte, die in den nachfolgenden Lösungsansätzen enthalten sind, aufzunehmen und weiterzuentwickeln.

4.1.1 Erweitung des Java-Compilers

[9]

4.1.2 WHITEOAK

[8]

4.1.3 REST/SOAP

4.2 Interfaces als Schnittstellen-Typ

4.2.1 Umsetzung mit dynamischen Proxies

[2]

4.3 Klassen als Schnittstellen-Typ

4.3.1 Umsetzung mit cglib

[1]

5 Diskussion

5.1 Vergleich mit bestehenden Lösungen

5.2 Verwendung definierter Methoden in Transfer-Objekten

6 Fazit

Literatur

[1] cglib 2.0beta2 api - class enhancer. Webseite, 2003. Online erhältlich unter `http://cglib.sourceforge.net/apidocs/index.html` abgerufen am 23.04.2020.

[2] Java™ platform standard ed. 7 - class proxy. Webseite, 2017. Online erhältlich unter `https://docs.oracle.com/javase/7/docs/api/java/lang/reflect/Proxy.html` abgerufen am 23.04.2020.

[3] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice 3. Edition*. Addison-Wesley, 2013.

[4] Martin Büchi and Wolfgang Weck. Compound types for java. In *OOPSLA '98 10/98*, Vancouver, 1998.

[5] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern orientated Software Architecture: A System of Patterns*. John Wiley Sons, 1996.

[6] Gilles Dubochet and Martin Oderski. Compiling structural types on the jvm. In *ICOOOLPS '09*, Genova, Italien, 2009.

[7] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[8] Joseph Gil and Itay Maman. Whiteoak: Introducing structural typing into java. In *OOPSLA'08*, Nashville, Tennessee, USA, 19-23.10.2008.

[9] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for java. Technical report, Computer and Information Science Department, Ohio State University, 17.06.1998.

[10] Guido Oelmann. Modulare anwendungen mit java: Tutorial mit beispielen. Webseite, 26.06.2018. Online erhältlich unter `https://www.informatik-aktuell.de/entwicklung/programmiersprachen/modulare-anwendungen-mit-java.html` abgerufen am 12.04.2020.

[11] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.

[12] Julian R. Ullmann. How do apis evolve? a story of refactoring. *Journal of Software Maintance and Evolution: Research and Practise*, pages 1–26, 2006.