Kommunikation zwischen entkoppelten Java-Modulen über strukturell typkonforme Objekte

Niels Gundermann

28. 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

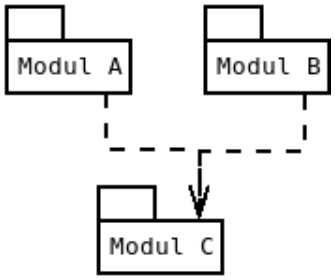

Abbildung 1: Problem: Szenario 1 (abstrakt)

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.

Ein konkretes Beispiel hierzu ist in Abbildung 2 zu sehen.

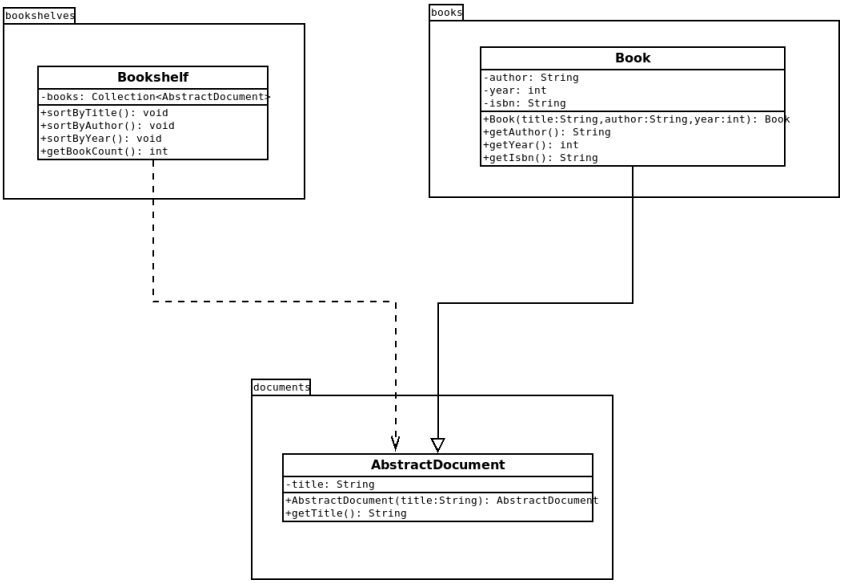

Abbildung 2: Problem: Szenario 1

Hier liegen die Module *bookshelves* und *books* auf einer Architekturebene, analog zu den abstrakten Modulen *A* und *B*. Dementsprechend stellt das Modul *documents* das Pendant zum abstrakten Modul *C* dar. Zu erkennen ist, dass die Klasse *Bookshelf* im Modul *bookshelves* einige Sortier-Methoden enthält. Da die Klasse *AbstractDocument* aus dem Modul *documents* jedoch lediglich einen *title* enthält, wäre die Implementierung eines Algorithmus zur Sortierung nach dem Jahr (*sortByYear*) oder nach dem Autor (*sortByAuthor*) nur schwer umzusetzen. Würde hingegen die Klasse *Book* aus dem Modul *books* in der Klasse *Bookshelf* nutzbar sein, könnten der Entwickler das *year* und den *author* bei der Implementierung der Sortier-Algorithmen verwenden.

Aufgrund der nominalen Typkonformität gäbe es für dieses Szenario folgende 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 3 zeigt den angenommenen Ausschnitt aus einem System.
Hierbei wird von einem Broadcast Serviceaufruf ausgegangen. Sas bedeutet, dass es eine

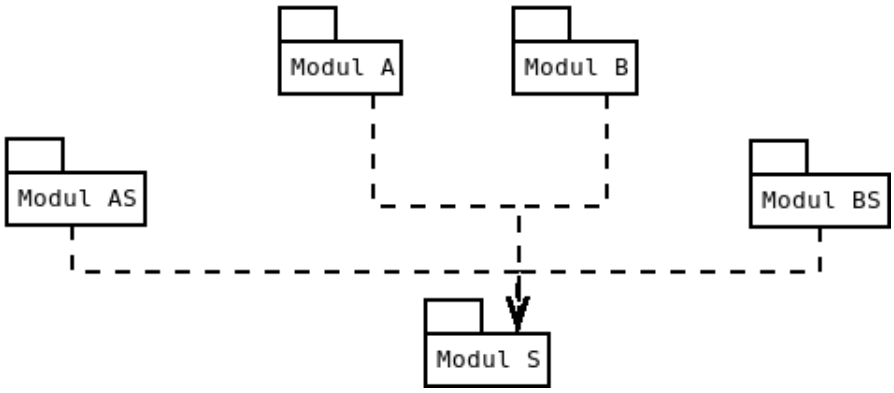

Abbildung 3: Problem: Szenario 2

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.

Abbildung 4 zeigt ein konkretes Beispiel für dieses Szenario.

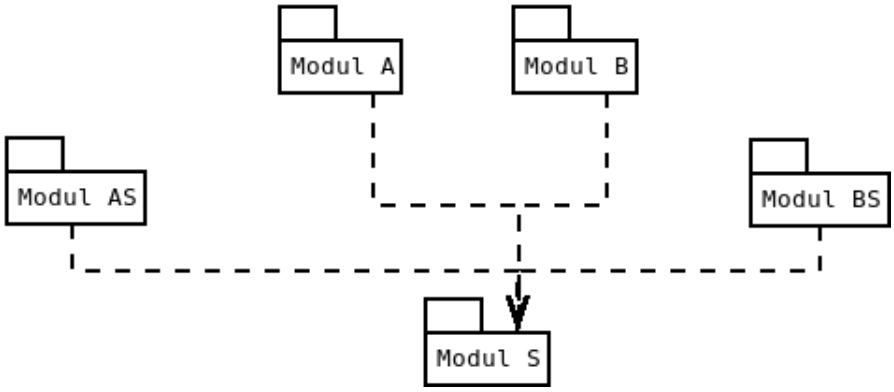

Abbildung 4: Problem: Szenario 2

Im Speziellen ist hier eine Art Notruf-Szenario abgebildet. Dabei gibt es ein Modul *injured*, in dem unterschiedliche Personen, die sich verletzen können bzw. in eine Notsituation geraten könnten, über die Klassen *Person* und *Allergic* abgebildet sind. Die daraus entstehenden Objekte können über einen Service aus dem Modul *MedicalServices* medizinische Hilfe anfordern. Die konkreten Services werden in den Modulen *doctors* und *cardriver* bereitgestellt. (Fachlich gesehen handelt es sich bei den Services also um eine Art Erstversorgung bzw. Erste-Hilfe).

Nun ist aber davon auszugehen, dass ein Allergiker (*Allergic*) mitunter eine andere medizinische Erstversorgung benötigt, als eine Person, die keine Allergien aufweist (*Person*). Weiterhin wäre vorstellbar, dass der Allergiker spezielle Informationen oder Werkzeuge, die für die notwendige Versorgung benötigt werden, bei sich trägt. (Beispielsweise einen Notimpfstoff mit Instruktionen zur Verabreichung.) Um diese zusätzlichen Informationen in den Service-Implementierungen nutzen zu können, gäbe es basierend auf der Tatsache, dass eine nominale Typkonformität im System angenommen wird, folgende Lösungsansätze:

1. Die Service-Schnittstelle wird erweitert.
2. Es wird eine neue Service-Schnittstelle geschaffen, die auf die zusätzlichen Informationen Zugriff hat.

Beide Lösungsansätze erfordern wiederrum erheblichen Aufwand und Koordination zwischen den Entwicklerteams. Dabei ist zu erwähnen, dass der zweite 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

In der Arbeit von Läufer et. al. ([9]) wurde der Java-Compiler so erweitert, dass die Deklaration der implementierten Interfaces an einer Klasse entfallen kann. Die Substituierbarkeit nach dem eines Interfaces und einer Klasse nach dem Liskovschen Substitutionsprinzip wird durch den Java-Compiler zusätzlich auf Basis der Struktur der entsprechenden Klassen und Interfaces festgestellt.

Hierzu musste definiert werden, was unter Typkonformität innerhalb der Sprache Java zu verstehen ist. Dabei wurden beachtet, dass in Java sowohl Klassen als auch Interfaces als Typen fungieren. Allgemein wurde folgende formale Definition hinsichtlich der Typkonformität aufgestellt:

Definition 1 *Sei* I *ein Interface und* X *sowie* Y *jeweils eine Klasse oder ein Interface. In der Sprache Java hat jede Klasse mit Außnahme von* java.lang.Object *eine direkte Oberklasse. Jede Klasse und jedes Interfaces hat keine ider mehrere direkte Interfaces.*

X *ist konform zu* Y *genau dann, wenn:*

- X *nominal Typkonform zu* Y *ist, oder*
- Y *ein Interface ist, dass strukturelle Typkonformität erlaubt und* X *strukturell Typkonform zu* Y *ist.*

(vgl. [9])

An Definition 1 ist zu erkennen, dass die nominale Typkonformität in dem Ansatz von Läufer et. al [9] nicht ausgeschlossen wurde, sodass die Konformität zweier Typen sowohl auf nominaler als auch struktureller Ebenen definiert ist. Die nominale Typkonformität wird in diesem Ansatz dabei wie folgt definiert:

Definition 2 X *ist nominal typkonform zu* Y *genau dann, wenn:*

- X *ist identisch zu* Y*, oder*

- *die direkte Oberklasse von* X*, sofern sie existiert, ist nominal typkonform zu* Y*, oder*
- *ein direktes Interface* I *von* X *ist nominal typkonform zu* Y

(vgl. [9])

Die strukturelle Typkonformität wird in dem Ansatz aus [9] wie folgt definiert:

Definition 3 X *ist strutkurell typkonform zu* I *genau dann, wenn* X *nominal typkonform zu* I *ist, oder alle der folgenden Bedingungen gleichzeitig erfüllt sind:*

- I *ist ein Interface, dass strutkurelle Typkonformität erlaubt, und*
- X *überschreibt jede Methode, die in* I *spezifiziert ist, und*
- X *ist typkonform zu alles direkten Interfaces von* I.

(vgl. [9])

Zur Vollständigkeit der Definitionen muss weiterhin definiet werden, wann eine Methode eines Interfaces in einer Klasse oder Interface überschrieben wird. Das Überschreiben einer Methode wird dabei wie folgt definiert:

Definition 4 X *überschreibt eine Methode* Y.f*, die in* Y *spezifiziert ist, genau dann, wenn es eine Methode* f *in* X *gibt (*X.f*), die folgende Bedingungen erfüllt:*

- X.f *ist von der Sichtbarkeit her nicht stärker eingeschränkt als* Y.f.
- X.f *hat dieselbe Methodensignatur wie* Y.f
- *Checked Exceptions, die von* X.f *geworfen werden, sind Unterklassen oder von derselben Klasse, die auch der Checked Excpetions zugrundeliegen, die von* Y.f *geworden werden.*

(vgl. [9])

Da der Ansatz aus [9] eine Hybride Variante bzgl. der Feststellung der Typkonformität darstellt (Verwendung von nominaler und struktureller Typkonformität) musste festgelegt werden, welche From der Typkonformität als Standardvariante verwendet wird. Anderenfalls würde die Gefahr der *versehentlichen Typkonformität* bestehen. Aufgrunddessen wurde festgelegt, dass die nominale Typkonformität als Standardvariante verwendet wird und die strukturelle Typkonformität einer speziellen Erlaubnis bedarf. (vgl. [9]) Dieser Fakt ebenfalls bei genauerer Betrachtung der Definitionen 1 und 3 klar. In diesen Definitionen ist die Rede davon, dass ein Interface die strukturelle Typkonformität erlaubt.

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.