Kommunikation zwischen entkoppelten Java-Modulen über strukturell typkonforme Objekte

Niels Gundermann

10. Juni 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

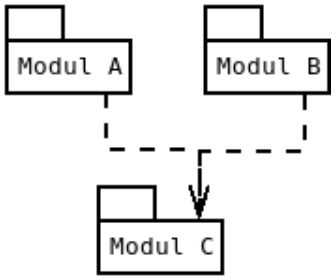

Abbildung 1: Problemszenario 1 (abstrakt)

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.

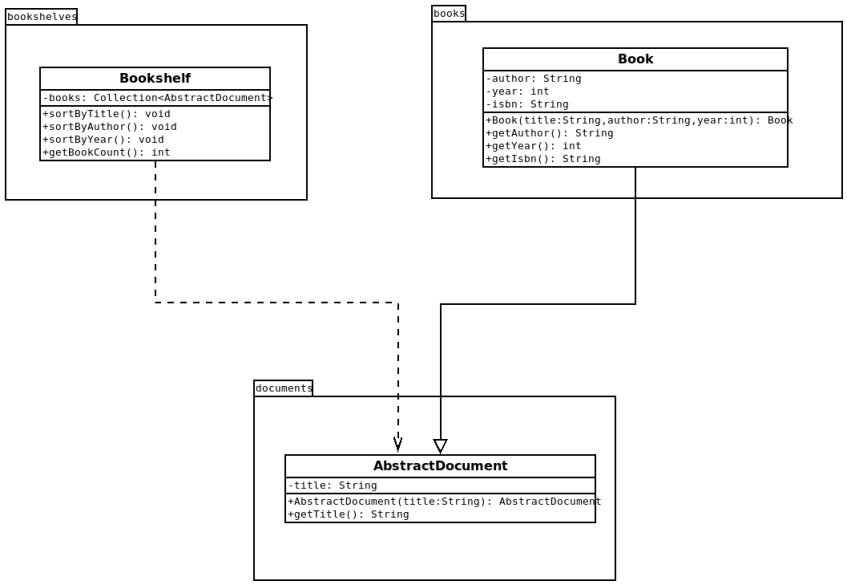

Abbildung 2: Problemszenario 1

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

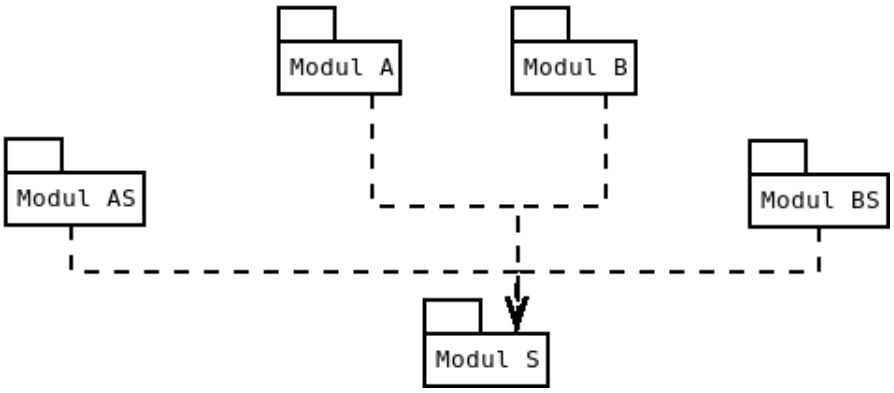

Abbildung 3: Problemszenario 2

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.

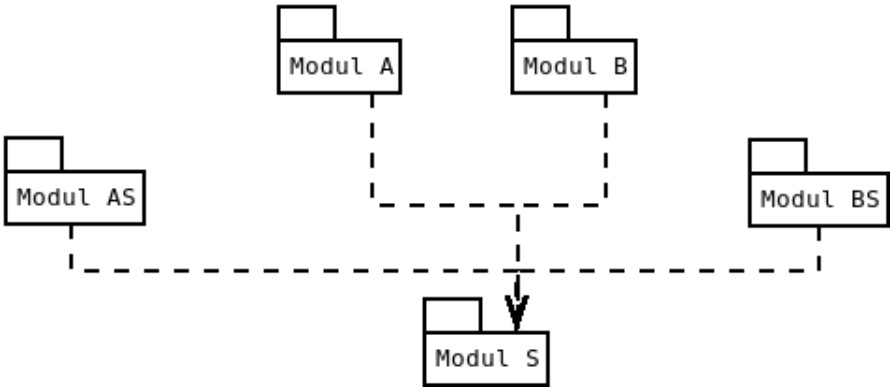

Abbildung 4: Problemszenario 2

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 Zufällige Typkonformität

[9]

3 Softwarearchitektur

[3]

3.1 Schichtenarchitektur

[10]

3.2 Serviceorientierte Architektur

[10]

3.3 Schnittstellen

[3], [5]

4 Lösungsansätze

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 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, das strukturelle Typkonformität erlaubt und zu* X *strukturell Typkonform 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 allen 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. Wann ein Interfaces die sturkutrelle Typkonformität erlaubt, ist in Definition 5 festgehalten:

Definition 5 *Ein Interface erlaubt die sturkturelle Typkonformität, wenn eine der folgenden Bedingungen erfüllt ist:*

1. *Das Interface erweitert ein speziellen Marker-Interface (z.B.* Structural*)*

2. *Das Interface erweitert ein anderes Interface, welches strukturelle Typkonformität erlaubt*

(vgl. [9])

Ausgehend von den Definitionen 1 - 5 können die beiden Problem-Szenarien aus Kapitel 1.1 wie folgt mithifle dieses Ansatzes umgesetzt werden:

Aus den Sortier-Methoden (*sortByTitle*, *sortByYear*, usw.), die von der Klasse *Bookshelf* angeboten werden, kann man schließen, dass die in einem *Bookshelf* verwalteten Dokumente über folgende Informationen verfügen müssen.

- Titel
- Jahr
- Autor

Ausgehend von diesem Wissen, kann ein Typ als Interface bereitgestellt werden, welcher innerhalb der Klasse *Bookshelf* verwendet wird und über die Sepzifikation der im Interface enthaltenen Methoden sicherstellt, dass die o.g. Informationen abgefragt werden können. Abbildung 5 zeigt den Inhalt des Moduls *Bookshelves* mit den notwendigen Erweiterungen.

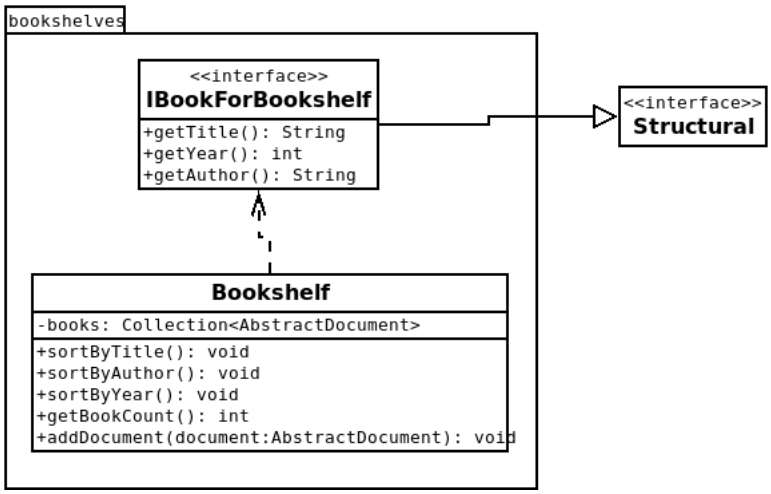

Abbildung 5: Lösung Problemszenario 1 mit erweitertem Compiler

Da sich die Änderungen nur innerhalb des Moduls *Bookshelves* vorgenomment werden sollen, wird das Interface *IBookForBookshelf* darin aufgenommen, welches vom Marker-Interface *Structural* erbt. Weiterhin spezifiziert dieses Interface drei Methoden, über die die oben genannten Informationen angefragt werden können. Die letzte beiden Änderungen beziehen sich auf die in der Klasse *Bookshelf* verwalteten *Collection*. Vorher konnten

wurde dort Objekte vom Typ *AbstractDocument* enthalten sein. Da die Klasse *AbstractDocument* das Interface *IBookForBookshelf* aufgrund der Architektur-Richtlinien nicht implementieren darf, wurde der generische Typ der *Collection* ebenfalls in *IBookForBookshelf* geändert. In der Konsequenz wurde auch die Signatur der Methode *addDocument* geändert, sodass hier ein Objekt vom Typ *IBookForBookshelf* übergeben werden muss.

Laut den Beschreibungen von Läufer et. al. [9] genügen diese Änderungen, damit der Compiler zum Übersetzungszeitpunkt sicherstellen kann, dass der Methode *addDocument* nur Objekte übergeben werden, deren Typ laut Definition 1 konform zum Typ *IBookForBookshelf* ist.

4.2 WHITEOAK

Der Lösungsansatz vom Gil und Maman [8] beschreibt eine Spracherweiterung für die Programmiersprache Java, die es ermöglicht spezielle Typen zu definieren, die explizit für die strukturelle Typkonformitätsprüfung vorgesehen sind. Damit ist dieser Ansatz genauso wie der Ansatz von Läufer et. al. [9] als hybrider Ansatz einzustufen, weil er die Verwendung von struktureller und nominaler Typkonformität innerhalb einer Sprache vereint.

Gil und Maman beschreiben in ihrer Arbeit ein neues Schlüsselwort *struct* in der Syntax. Durch dieses Schlüsselwort können strukturelle Typen definiert werden. Diese strukturellen Typen insofern mit Interfaces gleichzusetzen, dass sie Mehrfachvererbung ermöglichen. Dabei wird die Vererbung innerhalb struktureller Typen auf struktureller Ebene deklariert und nicht auf nominaler. (vgl. [8])

Aufgrund der exklusiven Verwendung der strukturellen Typkonfimität bei strukturellen Typen (definitiert mit dem Schlüsselwort *struct*), ist es nicht notwendig, für einen strukturellen Typ eine Bezeichnung zu definieren. (vgl. [8]). Das zeigt, dass das Konzept der strukturellen Typkonformität durch die Spracherweiterung stärker im Vordergrund steht und dem Entwickler aufgrund der Markierung über ein bestimmten Schlüsselwort deutlich gemacht wird, welche Form der Typkonformität verwendet wird. Eine Verpflichtende Bezeichnung für strukturelle Typen und deren daraus folgende zwangläufige Verwendung führt zur Vermischung der Ansätze.

Der Übergang von strukturelle zu nominalen Typen verläuft auf Basis des Grundsatzes, dass ein nominaler Typ *NT*, der strukturellen typkonform zu einem strukturellen Typ *ST* ist, auf den strukturellen Typ gecastet werden kann (siehe Listing 1)

```
1  struct ST { };
2  interface NT { };
3
4  NT nomType = new Object();
5  ST structType = (ST) nomType;
6
```

```
}
```

Listing 1: Strukturelle Typen auf nominale Typen casten

Weiterhin haben Gil und Maman [8] durch ihre Arbeit nicht nur Analogien zwischen strukturelllen Typen und Interfaces festgestellt, sondern auch zwischen strukturelle Typen und abstrakten Klassen. So können strukturelle Typen beispielsweise bestimmte Methode-Implementierungen und Instanzvariablen vorgeben. Eine vordefinierte Methode kann jedoch durch das Objekt eines *Quell-Typs* überschrieben werden. So werden bspw. in dem Szenario, welches in Listing 2 dargestellt wird, von dem strutkurellen Typen ST zwei Methoden spezifiziert und stellt eine vordefinierte Implementierung für diese bereit. Der nominale Typ NT spezifiziert jedoch nur eine dieser beiden Methoden. Eine Impementierung dieser Methode ist ebenfalls in NT definiert. Wird nun ein Objekt des nominalen Typs NT in ein Objekt des strukturellen Typs ST konvertiert, bietet dieses konvertierte Objekt vom Typ ST die beiden Methoden aus ST an. Dabei wird beim Aufrufen der Methode *m1* die Implementierung aus der Spezifikation des nominalen Typs NT verwendet, wohingegen beim Aufrufen der Methode *m2* die Implementierung aus dem strukturellen Typ ST verwendet wird. Dieses Konzept wurde von Gil und Maman als *virtuelle Objekte* bezeichnet (vgl. [8])

```
struct ST {

  String getA(){
    return "A";
  }

  String getB(){
    return "B";
  }

}


class NT {

  String getA(){
    return "a";
  }

}
```

Listing 2: Standardmethoden in WHITEOAK

Innerhalb eines strukturellen Typs ist es jedoch nicht möglich Konstruktoren zu definieren. Das führt zu dem Problem, dass die Instanzvariablen nicht in einem Konstruktor initialisiert werden können. Um dem entgegenzuwirken können Constraints für Konstruktoren definiert werden, durch die zumindest die für den Konstruktor benötigten Parameter definiert werden können. (vgl. [8])

Weiterhin bietet *WHITEOAK* die Möglichkeit diverse Kompositionstechniken wie *Mixins*, *Traits* oder *Delegationsobjekte* zu emulieren. (vgl. [8]) Da diese Konzepte allerdings nicht

im Vordergrund bzgl. dieser Arbeit stehen, wird auf darauf nicht weiter eingegangen.

Mit dem bisherigen Wissen zu *WHITEOAK* ist es möglich, jeweils einen Lösungsansatz für die genantnen Problemszenarien unter Zuhilfenahme von *WHITEOAK* zu skizzieren.

Für das erste Problemszenario wie ähnlich vorgegangen, wie bei der Verwendung des Ansatzen von Läufer et. al [9]. Grundlegend muss auch ein Typ bereitgestellt werden, der entsprechende Methoden anbietet, um die Informationen bzgl. *Titel*, *Autor* und *Jahr* abzufragen. Allerdings muss mit *WHITEOAK* kein Interface im Modul *Bookshelves* bereitgestellt werden, sondern es kann auf einen strukturellen Typ zurückgegriffen werden, der je notwendiger Information eine abstrakte Methode bereitstellt. (siehe Abbildung 6) [2]

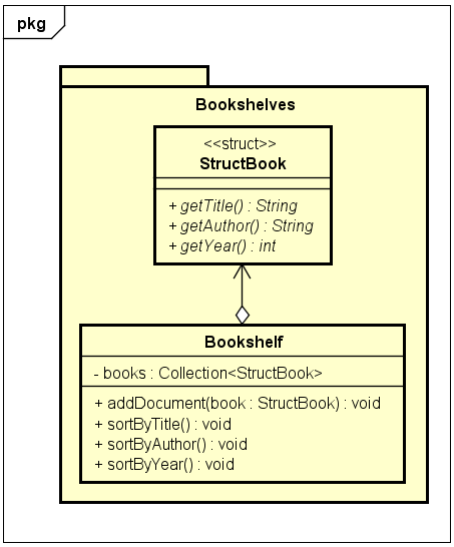

Abbildung 6: Lösung Problemszenario 1 mit *WHITEOAK*[1]

Weiterhin muss die Klasse *Bookself*, wie in Abbildung 6, etwas angepasst werden. Auch hier muss die Valriable, in der die Dokumente verwaltet werden sollen einen anderen generischen Typ erhalten. Ob hierzu einfach der generische Typ der *Collection* mit dem strukturellen Typ *StructBook* ersetzt werden kann, ist nicht sicher. Der Grund dafür ist, dass aus den Ausführungen vom Gil und Maman [8] nicht auf die Verwendung von strukturellen Typen als generische Typen eingegangen wird. Allerdings wird beschrieben, dass ein struktureller Typ durch den Compiler letztendlich in einen nominalen Typ konvertiert wird (vgl. [8]), womit er auch als ein solcher behandelt werden kann. Daher ist die Anpassung der Instanzvariablen *documents* in der Klasse *Bookshelf* wie in Abbildung 6 zulässig. Ferner muss die Methodensignatur der Methode *addDocument* so angepasst werden, dass ein Objekt vom Typ *StructBook* als Parameter gefordert wird.

[2] In dieser Arbeit werden die strukturelle Typen, die in *WHITEOAK* über das Schlüsselwort *struct* definiert werden, in Klassendiagrammen durch den Stereotypen *struct* gekennzeichnet.

So kann abschließend festgehalten werden, dass sich die notwendigen Änderungen nur auf das Modul *Bookshelves* beschränkten. Die strukturelle Typkonformität zwischen dem neu aufgenommenen Typen und den Typen *Book* und *UnpublishedBook* aus dem Modul *Books* ist wie bei der Verwendung des Lösungsansatzes von Läufer et. al. [9] gegeben.

4.3 Neuer Lösungsansatz

In den folgenden Kapiteln wird auf die Umsetzung der Problem-Szenarien aus Kapitel 1.1 unter der Verwendung des im Rahmen dieser Arbeit erarbeiteten Ansatz beschrieben.

Dieser Lösungsansatz verwendet weder einen erweiterten Compiler noch eine Erweiterung der Sprachekonstrukte. Die Verwendung der strukturellen Typkonformität muss innerhalb der Programms mithilfe einer einzubindenden Bibliothek explitit angestoßen werden. Hierzu steht in der Bibliothek das Interface *TypConverter* bereit, welches die Verwendung der strukturellen Typkonformität ermöglicht. Ein Objekt, welches das Interfaces *TypConverter* erfüllt, kann mithilfe der Klasse *TypConverterBuilder* erzeugt werden. Auf die Parameter, die dem *TypConverterBuilder* mitgegeben werden müssen, wird im späteren Verlauf spezieller eingegangen. Im Allgemeinen dienen sie der Konfiguration der Konformitätsprüfung sowie der Form Konvertierung von strukturell typkonformen Objekten.

Der neue Lösungsansatz geht über den Ansatz von Läufer et. al [9] insofern hinaus, dass sich eine strukturelle Typkonformität nicht nur zwischen Klassen und Interfaces feststellen lässt, sondern auch zwischen zwei Klassen. Weiterhin wird in Bezug auf Interfaces auf die Nutzung von *default-Methoden* eingegangen. Hierbei wird jedoch auf Konzepte aus der Arbeit von Gil und Maman [8] zurückgegriffen.

Allerdings wird es auch in Bezug auf *WHITEOAK* eine Erweiterung geben. Im nueuen Lösungsansatz wird in Betracht gezogen, dass eine vordefinierten Methode in einem *Ziel-Typ* auch explizit im Kontext, in dem dieser Typ verwendet wird, so verwendet werden soll. Das heißt, dass die Verwendung von vordefinierten Methoden, die durch den *Quell-Typen* möglicherweise überschrieben werden, gesteuert werden kann.

4.4 Interfaces als Schnittstellen-Typ

Im ersten Teil wird beschrieben, wie mit Interfaces als sturkturelle Typen umgegangen wird. Da Läufer et. al. [9] in ihrer Arbeit hierfür bereits eine fundierte Grundlage geschaffen haben, werden die Definitionen 1 - 4 auch als theoretische Grundlage für den neuen Lösungsansatz verwendet.

Da die Verwendung der strukturellen Typkonformität im neuen Lösungsanstz explizit angegeben werden muss, ist es anders als im Ansatz aus [9] nicht notwendig, die Interfaces, für die eine sturkturelle Typkonformität Anwendung finden kann, explizit durch Marker-Interfaces zu markieren. Folglich wird die Definition 5 für den neuen Lösungsansatz

nicht benötigt. Ausgehen davon kann auch die Definitionen 5 und 3 für den neuen Lösungsansatz nicht verwendet werden. Allerdings sind nur kleinere folgende Anpassungen notwendig, um Definitionen für die Verwendungen der strukturellen Typkonformität in Bezug auf Interfaces nach dem neuen Lösungsansatz als Basis zu formulieren.

Definition 6 *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 oder mehrere direkte Interfaces.*

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

- X *nominal Typkonform zu* Y *ist, oder*
- Y *ein Interface ist, das zu* X *strukturell Typkonform ist.*

Um zu definieren, wann ein Interfaces oder eine Klasse zu einem anderen Interface strukturell typkonform ist, muss in Bezug auf den neuen Lösungsansatz folgendes vorweggenommen werden. Mit der Java-Version ... wurden sogenannte *default-Methoden* für Interfaces aufgenommen. Diese Methoden erlauben es, ein bestimmtes Verhalten innerhalb eines Interfaces zu definieren. Nach der Definition 3 aus [9] müsste ein zum Interface I typkonformes Interface oder eine Klasse auch die Methoden enthalten, die in I als *default-Methoden* bereits implementiert sind. Die Möglichkeit, Verhalten innerhalb eines Interfaces zu definieren, welches in der implementierenden Klasse nicht erneut definiert werden muss, würde bei diesem Ansatz nach Läufer et. al. [9] jedoch verloren gehen.

In dem neuen Lösungsansatz wird dem Entwickler daher die Möglichkeit gegeben, die Verwendung von *default-Methoden* bzw. implementierten Methoden innerhalb von Klassen (siehe Kapitel 4.5) zu steuern. Das bedeutet, dass der Entwickler zwischen zwei Definitionen bzgl. der sturkturellen Konformität zweier Typen wählen kann. Die erste Definition ist dabei an der Definition 3 aus [9] angelehnt und beschreibt den Ansatz, dass alle Methoden (auch die *default-Methoden*) im *Zieltyp* spezifiziert sein müssen.

Definition 7 X *ist strutkurell typkonform zu* I *genau dann, wenn* X *nominal typkonform zu* I *ist, oder alle der folgenden Bedingungen gleichzeitig erfüllt sind:*

- X *enthält für jede Methode, die in* I *spezifiziert ist, eine strukturell äquivalente Methode und*
- X *ist typkonform zu allen direkten Interfaces von* I.

(vgl. [9])

Dem gegenüber steht die folgende Definition bzgl. der strukturellen Typkonformität, welche eine Spezifikation bereits implementierter Methoden im *Zieltyp* nicht fordert

Definition 8 X *ist strutkurell typkonform zu* I *genau dann, wenn* X *nominal typkonform zu* I *ist, oder alle der folgenden Bedingungen gleichzeitig erfüllt sind:*

- X *enthält für jede Methode, die in* I *spezifiziert aber nicht implementiert ist, eine strukturell äquivalente Methode und*
- X *ist typkonform zu allen direkten Interfaces von* I.

Die o.g. Unterscheidung zwischen der Verwendung von Definition 7 und Definition 8 wird innerhalb des neuen Lösungsansatzes über das Enum *StructureDefinition* gesteuert. Der Wert *ALL_METHODS_NECESSARY* gibt an, dass Definition 6 zu verwenden ist. Hierbei handelt es sich um den Standardwert. Sofern im *TypConverterBuilder* also nicht die *StructureDefinition.ABSTRACT_METHODS_NECESSARY* angegeben wurde, wird die strukturelle Typkonformität immer laut Definition 7 festgestellt. Die Wahl Definition 7 als Standard wird dadurch gebründet, weil sie der allgemeinen Definition der Strukturellen Typkonformität eher entspricht als die Definition 8, da in Definition 8 auf spezifische Elemente der Sprache Java referenziert wird.

Um eine vollständige Grundlage für den neuen Lösungsansatz zu erhalten, muss darüber hinaus noch definiert werden, was unter strukturell äquivalenten Methoden zu verstehen ist.

Definition 9 X *enthält eine Methode* f *(*X.f*), die strukturell äquivalenz zu einer Methode* Y.f *ist, wenn* X.f Y.f *lauf Definition 4 überschreibt, oder alle der folgenden Bedingungen erfüllt sind:*

- X.f *ist von der Sichtbarkeit her nicht stärker eingeschränkt als* Y.f
- X.f *hat denselben Rückgabetyp wie* Y.f
- X.f *benötigt dieselben Parameter in derselben Reihenfolge wie* Y.f
- *Checked Exceptions, die von* Y.f *geworden werden, werden auch von* X.f *geworfen, wobei diese in* X.f *auch als Unterklassen derer, die von* Y.f *geworden werden, umgesetzt sein können*

Diese Grundlage unterscheiden weiterhin wie folgt von derer aus dem Ansatz von Läufer et. al. [9] wie folgt. Zum einen ist es nicht mehr notwendig, dass ein Interface die strukturelle Typkonformität erlauben muss, um eine strutkurelle Typkonformität zu gewähleisten. Weiterhin können sich die Methoden zweier strukturell konformer Typen durchaus vom Namen her unterscheiden. Die Definition 9 erlaubt dabei zwei Wege, um das *Enthalten* einer strukturell äquivalenten Methode in einer Klasse zu gewährleisten. Der erste Weg besteht im Überschreiben der Methode gemäß Defintion 4. Der zweite Weg fordert hingegen nicht, dass die Bezeichnungen der Methoden gleich sein müssen. Diese Herangehensweise kann jedoch zu Problemen führen. So kann die strukturelle Typkonformität zwischen einem Interface (I) und einer Klasse (C), wie in Abbildung 7, zwar festgestellt werden, eine Konvertierung und damit Nutzung des Konzepts ist aber nicht möglich. Die Klasse C ist auf Basis der o.g. Definitionen strukturell Typkonform zum Interface *I*. Bei der Konvertierung muss jedoch festgestellt werden, welche Methode innerhalb der Klasse das strukturelle Äquivalent zur Methode *getName* aus dem Interface

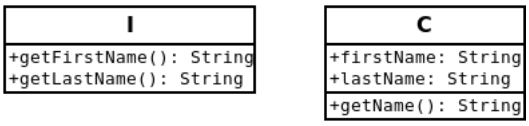

Abbildung 7: Mehrdeutige Methoden ohne Namensbeachtung

darstellt. Da die Methodensignaturen abgesehen vom Methodennamen aller Methoden gleich sind, sind auch alle Methoden sturkturell äquivalent.

Für dieses Problem wurde im *TypConverterBuilder* die Möglichkeit geschaffen, einen der beiden Wege zur Ermittlung der strukturell äquivalenten Methoden explizit anzugeben. So kann der Entwickler selbst entscheiden, welchen Ansatz er verwendet und muss in der Konsequenz die Klassen und Interfaces bzgl. der Methoden dementsprechen so entwerfen, dass der gewünschte Ansatz verwendet werden kann.

Daher muss beim Erzeugen eines Objekte vom *TypConverterBuilder* eine *ComformityCheckingBase* angegeben. Diese kann genau zwei Werte annehmen, wobei der erste (*Name*) die Ermittlung der strukturell äquivalenten Methoden nach Definition 4 angibt und die zweite (*SIGNATURE*) die Ermittlung auf Basis der Signatur, als ohne Beachtung des Methodennamens, angibt. In Abhängigkeit der angegebenen *ComformityCheckingBase* werden unterschiedliche Objekte erzeugt, die das Interface *TypConverter* erfüllen (siehe Abbildung 8). Innerhalb der

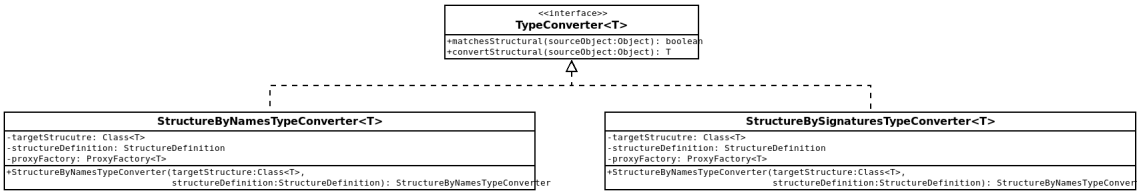

Abbildung 8: TypeConverter

konkreten Implementierungen *StructureByNamesTypeConverter* und *StructureBySignaturesTypeConverter* sind die unterschiedlichen Wege zum Ermitteln der strukturell äquivalenten Methoden umgesetzt.

Bezogen auf das erste Problemszenario (siehe Kapitel 1.1.1) sähe eine Lösungsvariante über ein strukturell typkonformes Interfaces wie folgt aus. Der Typ, der in dem Modul *Bookshelves* die Elemente abbilden soll, die innerhalb der Objekte der Klasse *Bookshelf* verwaltet werden, könnte innerhalb des Moduls, wie in Abbildung 9 spezifiziert werden. Hierbei ist zu bemerken, dass es sich bei den Methoden im Interface *IBookFromBookshelf* um genau die Methoden handelt, die in den Sortier-Methoden der Klasse *Bookshelf* Verwendung finden. Bei genauerer Betrachtung der Klasse *Book* aus dem Modul *Books* fällt auf, dass diese Klasse strukturell typkonform zu dem neune Interface *IBookFromBookshelf*

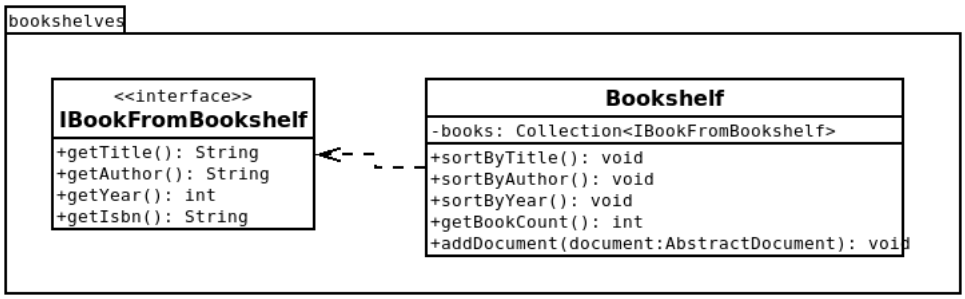

Abbildung 9: Lösungsansatz: Interfaces - Problemszenario 1

ist. Die Frage die offen bleibt ist, wie ein Objekt der Klasse *Book* in ein Objekt vom Typ *IBookFromBookshelf* konvertiert wird.

Grundsätzlich kann die Konvertierung in diesem Fall nur im Modul *Bookshelves* stattfinden, da nur dort der Zugriff auf den *Ziel-Typ* - in diesem Fall *IBookFromBookshelf* - gewährleistet ist. Zu bemerken ist, dass innerhalb den Moduls zwar ein Objekt der Klasse *Book* zu verwenden ist, der konkrete Typ des Objektes jedoch für das Einleiten der Konvertierung irrelevant ist. Somit kann zur Not das konkrete Objekt einer Klasse aus einem anderen Modul - wie in diesem Fall *Book* aus dem Modul *Books* - mit dem Typ *java.lang.Object* verwendet werden. In diesem speziellen Fall wäre es jedoch von Vorteil, die bestehende Signatur der Methode zum Hinzufügen von Elementen in ein *Bookshelf* beizubehalten, da der dort benötigte Typ aus einer Abstraktionsebene stammt, auf die beide Module - *Books* und *Bookshelves* - zugreifen dürfen. Somit können die Methoden der Klasse *Bookshelf* von der Signatur her für diesen Lösungsweg unverändert bleiben, was den Entscheidenen Vorteil mit sich bringt, dass die verwendenen Klassen nicht angepasst werden müssen.

Der Entwickler muss nun beim Einlagern der Elemente in ein *Bookshelf* die Konvertierung zu einem Objekt vom Typ *IBookFromBookshelf* einleiten. Hierzu ist, wie oben bereits erwähnt der *TypConverterBuilder* zu verwenden. Dabei muss entschieden werden, wie die Struktur der Methoden des *Ziel-Typs* definiert werden soll (siehe Definition 9). In diesem Beispiel soll die Variante verwendet werden, die auch von Läufer et. al. [9] verwendet wurde. Daher wird der *TypConverterBuilder* mit der *ComformityCheckingBase.NAMES* erzeugt. Die Methode zum Einlagern eines Elements in das *Bookshelf* könnte demnach wie in Listing 3 aussehen.

```
1 public void addDocument(AbstractDocument dokument) {
2   TypConverter<IBookFromBookshelf> structConverter = TypConverterBuilder
3     .create(ConformityCheckingBase.NAMES)
4     .build(IBookFromBookshelf.class);
5
6   if(structConverter.matchesStructural(dokument)){
7     IBookFromBookshelf buchAusBuecherregal = structConverter
8       .convertStructural(dokument);
```

```
9       books.add(buchAusBuecherregal);
10    }
11
12 }
```

Listing 3: addDocument

Welcher konkrete *TypeConverter* dabei erzeugt wird, ist ist dem Sequenzdiagramm in Abbildung 10 zu entnehmen.

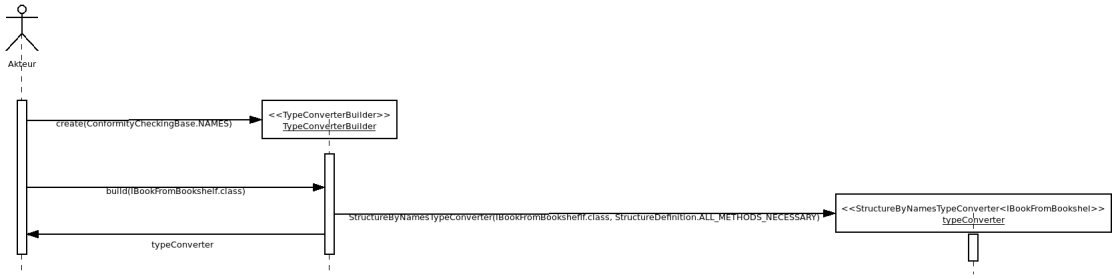

Abbildung 10: Interne Aufrufe im TypeConverterBuilder

Zu bemerken ist, dass bei dieser Implementierung direkt beim Hinzufügen eines *AbstractDocument* geprüft wird, ob der konkrete Typ des Objektes strukturell konform zum benötigten Typ ist. Wäre dies nicht der Fall, würde das übergebene *AbstractDocument* nicht hinzugefügt werden. Dieses Verhalten kann unter Umständen zur Verletzung diverser Vertragsbedingungen führen, auf die an dieser Stelle nicht weiter eingegangen wurde.

Eine andere Herangehensweise, die ein Vertragsmodel wiederrum unterstützen würde, wäre die Aufnahme einer weiteren Methode (*isDocumentSuitable*) in der Klasse *Bookshelf*. Diese Methode könnte prüfen, ob ein Objekt überhaupt für dieses *Bookshelf* geeignet ist. Hierbei könnte der Sachverhalt, der in Listing 3 in *addDocument* geprüft wird, innerhalb der neuen Methode geprüft werden. Der Entwickler kann das Vertragsmodel dann so anpassen, dass er davon ausgehen kann, dass der Methode *addDocument* nur Objekte übergeben werden, die laut der Methode *isDocumentSuitable* für das *Bookshelf* vorgesehen sind. Die angepasste Klasse *Bookshelf* sähe dann wie in Abbildung 11 aus. Diese Erweiterung ist jedoch in Bezug auf den Anpassungsaufwand, sofern eine solche Methode nicht ohnehin bestand, kritisch einzustufen. Der Grund dafür ist, dass durch die Notwendigkeit der Aufrufs der Methode *isDocumentSuitable* vor der Methode *addDocument* alle Aufrufer angepasst werden müssten.

In Bezug auf das Problemszenario 1 (siehe Kapitel 1.1.1) sollte das Ziel sein, die unterschiedlichen Bücher aus dem Modul *Books* an ein Objekt der Klasse *Bookshelf* aus dem Modul *Bookshelves* zu übergeben, sodass die Bücher einsortiert werden. Auf der Grundlage der Implementierung aus Listing 3 ist das jedoch nur für die Objekte der Klasse *Book*

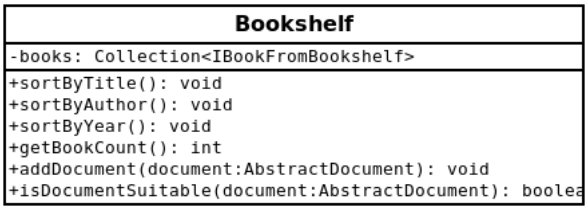

Abbildung 11: Erweiterte Klasse *Bookshelf*

möglich. Ein Objekt der Klasse *UnpublishedBook* würde die Prüfung in *Bookshelf::addDocument* nicht bestehen, da die Klasse die Methode *getIsbn* nicht enthält.

Diesem Problem kann mit der zu Beginn dieses Kaptiels beschriebenen alternativen Definition bzgl. der Strukturellen Typkonformität (siehe Definition 8) Abhilfe geschaffen werden. Hierzu sind zwei Schritte notwendig:

1. Die Methode *getIsbn* im Interface *IBookFromBookshelf* muss als *default-Methoden* spezifiziert und implementiert werden.
2. Im *TypeConverterBuilder* muss die *StructureDefinition.ABSTRACT_METHODS_NECESSARY* gesetzt werden.

Eine sinnvolle default-Implementierung für die Methode *getIsbn* können wie in Listing 4 aussehen, sodass eine fehlende Isbn einfach als Leerstring definiert wird und somit der Sortieralgorihmus in *Bookshelf::sortByIsbn* problemlos damit umgehen kann.

```
1 default String getIsbn(){
2   return "";
3 }
```

Listing 4: Default-Implementierung getIsbn

Die Methode *addDocument* in der Klasse *Bookshelf*, in der der *TypConverter* erzeugt wird, muss wie in Listing 5 angepasst werden, damit bzgl. der Strukturellen Typkonformität die Definition 8 Anwendung findet.

```
1  public void addDocument(AbstractDocument dokument) {
2    TypConverter<IBookFromBookshelf> structConverter = TypConverterBuilder
3      .create(ConformityCheckingBase.NAMES)
4      .withStructureDefinition(
5        StructureDefinition.ABSTRACT_METHODS_NECESSARY)
6      .build(IBookFromBookshelf.class);
7
8    if(structConverter.matchesStructural(dokument)){
9      IBookFromBookshelf buchAusBuecherregal = structConverter
10       .convertStructural(dokument);
11     books.add(buchAusBuecherregal);
12   }
13
14 }
```

Listing 5: addDocument

Mit dieser Implementierung ist es nicht nur möglich Objekte der Klasse *Book* einem Objekt der Klasse *Bookshelf* hinzuzugeüfen, sonder auch Objekte der Klasse *UnpublishedBook*, da beide Objekte (*Book* und *UnpublishedBook*) gemäß Definition 8 strukturell typkonform zu *IBookFromBookshelf* sind.

Bisher wurde nur die Grundlage für die Konvertierung zwischen stukturell konformen Typen beschrieben und wie die Klassen in der Bibliothek des neuen Lösungsansatzes zu verwenden sind. In folgendem Kapitel wird darauf eingeganen, wie die Konvertierung innerhalb der Bibliothek vonstatten geht und welche Konsequenzen sich daraus für die weitere Verwendung von Interfaces als *Ziel-Typen* ergeben.

4.4.1 Umsetzung mit dynamischen Proxies

[2]

4.5 Klassen als Schnittstellen-Typ

4.5.1 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.