Ausarbeitung

Typisierung

Proseminar
Effiziente Programmierung

vorgelegt von

Thomas Schnieders

Fakultät für Mathematik, Informatik und Naturwissenschaften
Fachbereich Informatik
Arbeitsbereich Wissenschaftliches Rechnen

Studiengang: Informatik
Matrikelnummer: 6800924

Betreuer: Dr. Michael Kuhn

Hamburg, 08.07.2018

Inhaltsverzeichnis

1 Einleitung

In der vorliegenden Arbeit geht es um das Thema Typisierung in Programmiersprachen. Typisierung ist zeichnet sich nicht nur dadurch aus, dass je nach Programmiersprache, eine Variable durch einen bestimmten Datentypen gekennzeichnet ist oder nicht. Vielmehr ist der Umgang mit den Typen, zum einen durch die Sprache, zum anderen aber auch durch den Programmierer, durch verschiedene Konzepte und Verhaltensweisen, gekennzeichnet. Diese verschiedenen Konzepte werden in der vorliegenden Arbeit, angereichert durch Codebeispiele in unterschiedlichen Sprachen, genauer betrachtet. Die Arbeit beginnt mit einigen grundlegenden Inhalten, wie z.B. was sind Typen, und der Unterscheidung von Typen. Je nach Programmiersprache kann eine unsachgemäße Verwendung von Typen zu Fehlern in der Software führen. Um solche Typfehler zu vermeiden, sollte man sich mit dem Thema Typsicherheit beschäftigen, worauf im Anschluss eingegangen wird. Um Typsicherheit herzustellen, bieten die Programmiersprachen explizite und implizite Typumwandlungen an, die je nach Programmiersprache unterschiedlich implementiert sind.

Im zweiten Teil der Arbeit werden die unterschiedlichen Typisierungskonzepte genauer beleuchtet. Da wäre zum einen die Unterscheidung in statische und dynamische Typisierung sowie die Einordnung der Programmiersprachen in stark und schwach typisierte Sprachen. Der Leser soll am Ende der Arbeit in der Lage sein, sich mit den Methoden zur Herstellung von Typsicherheit auseinandersetzen zu können und anhand der Typisierungsmerkmale einer Programmiersprache diese zu unterscheiden.

2 Typen

Durch das Zuweisen eines Datentyps bekommt eine Bitfolge eine Bedeutung. Diese Bitfolge stellt einen Wert oder ein Objekt repräsentiert durch eine Variable dar. Das Verknüpfen einer Bitfolge mit einem Typ überträgt diese Bedeutung auf die programmierbare Hardware, um ein symbolisches System zu bilden, das aus dieser Hardware und einem Programm besteht [Wik18a].

In Programmiersprachen kommen dabei unterschiedliche Typsysteme zum Tragen. In objektorientierten Sprachen werden Informationen in der Regel durch Objekte repräsentiert. Jedes dieser Objekte ist dabei von einem bestimmten Typ. Der Typ gibt dabei an, welche Art von Informationen im Speicher abgelegt werden. Wir können diesen Objekten Namen zuweisen, sodass diese benannten Objekte durch Variablen im Quelltext eines Programms identifizierbar sind.

Der Typ definiert dann den Wertebereich und die zugehörigen Operationen/Methoden (für ein Objekt).

```
1 Integer a = 3; // klappt, da 3 im Wertebreich
2 Integer b = 2147483648; // Fehler: out of Range
```

Listing 2.1: Wertebereich

In diesem Beispiel kann die Variable `a` den Wert 3 annehmen, da dieser im Wertebereich des Integer-Objektes liegt. Wenn jetzt b vom Typ `Integer` den Wert 2147483648 zugewiesen bekommen soll, dann geht das nicht. Glücklicherweise meldet der Java-Compiler dann, dass 2147483648 nicht im Wertebereich des zugewiesenen Typen liegt (dieser endet nämlich genau bei 2147483647[1]) und bewahrt den unachtsamen Programmierer vor diesem Fehler.

```
1 Integer a = 3;
2 int c = a.intValue(); // klappt, da Methode zulaessig
3 int b = a.length(); // Fehler: Methode kann auf a nicht
    ↪ angewendet werden
```

Listing 2.2: Operationen

[1] ein `Integer` ist 64 `bit` lang und hat daher als Minimum-Wert -2^{31} und als Maximum-Wert $2^{31} - 1 = 2147483647$

Auf `a` kann die Methode `intValue()` angewendet werden, nicht aber die Methode `length()`, da Erstere eben für Objekte des Typen Integer zulässig ist, die Zweite aber nicht.

Ein konkretes Objekt repräsentiert den Speicherbereich eines Wertes zu einem entsprechenden Typen. Der Wert ist dann nur noch eine Folge von Bits im Speicher, die nun entsprechend seines Typen interpretiert werden [Str10].

Je nach Programmiersprache wird dann zwischen unterschiedlichen Typgruppen unterschieden. So bietet zum Beispiel Java zwei Arten von Typen an:

- Primitivtypen:

  Dies sind die einfachen, eingebauten Datentypen `byte, short, int, long, float, double, boolean, char`.

- Referenztypen:

  Damit lassen sich Objektverweise auf Zeichenketten und Datenstrukturen realisieren [Ull18].

In C++ ist die Unterteilung, ähnlich zu der von Java, in Grundtypen (`int, short, bool, char`, usw) und die daraus ableitbaren oder zusammengesetzten Typen.

2.1 Typsicherheit

Typsicherheit beschreibt zunächst das Ausmaß, in dem eine Programmiersprache Typfehler verhindert. Wenn keine Typverletzungen auftreten, kann also davon ausgegangen werden, dass die Datentypen gemäß ihren Definitionen in der benutzten Programmiersprache verwendet werden. Ein Typfehler äußert sich dabei durch unerwünschtes oder auch fehlerhaftes Programmverhalten, bis hin zum Programmabbruch. Hervorgerufen wird ein Typfehler eben durch die fehlerhafte Zuweisung von Variablen und Werten oder durch die Verwendung von Operationen auf Werten, die nicht den geeigneten Typen aufweisen (vgl. Listing 2.2).

Typsicherheit herzustellen bzw. zu prüfen ist je nach Sprache dann Aufgabe des Compilers (Statisch Typisiert 3.2.1) oder des Interpreters (Dynamisch Typisiert 3.2.2). In bestimmten Fällen kann aber allein die Sprache selbst dazu beitragen, dass Typfehler vermieden werden, in dem z.B. kompatible Typen umgewandelt werden. Diesen Vorgang nennt man dann Typumwandlung (explizit oder implizit). Das ermöglicht dem Programmierer eine gewisse Flexibilität beim Schreiben der Software.

2.2 Typumwandlung

Als Typumwandlung wird der Vorgang bezeichnet, einen Datentyp in einen anderen Datentypen umzuwandeln. Insbesondere um Typverletzungen zu vermeiden, die z.B.

durch die Zuweisung inkompatibler Typen auftreten. Diese Typumwandlung kann dabei implizit oder explizit erfolgen. Implizit in dem Sinne, als dass die Umwandlung nicht im Quelltext erscheint, sondern in die Semantik der Sprache eingebunden ist. Bei expliziten Typumwandlungen wird diese ausdrücklich in den Quelltext hineingeschrieben. Da unterschiedliche Datentypen oftmals verschiedene Wertebereiche haben, können bei zulässiger Typumwandlung Typerweiterungen (Listing 2.3 - Vergrößerungen des Wertebereichs) oder Typeinschränkung (Listing 2.6 - Verkleinerung) vorkommen.

2.2.1 Implizit

Die implizite Typumwandlung ist eine automatische Typumwandlung die vom Compiler übernommen wird. Implizite Typumwandlungen können dabei auf mehrere Arten durchgeführt werden. Der Typ einer Variable kann bei der Zuweisung umgewandelt werden. Es ist aber auch möglich, unterschiedliche aber kompatible Typen, in ein und demselben arithmetischen Ausdruck zu verwenden. Je nach Programmiersprache unterscheiden sich die Möglichkeiten der impliziten Typumwandlung. So können in C++ z.B. Zeiger, ganze Zahlen oder Gleitkommawerte in boolsche Ausdrücke umgewandelte werden, [Str10, vgl S. 1056] während dies in Java nicht möglich ist.

Betrachten wir jetzt folgendes Beispiel:

```
1 int a = 3;
2 double b = a;
```

Listing 2.3: int zu double in Java (Typerweiterung)

Da ein `int`-Wert in den Wertebereich des `double`-Typen passt, kann diese Zuweisung problemlos ausgeführt werden, da der Compiler hier nicht mit Fehlern rechnen muss. Umgekehrt wäre diese Zuweisung in Java nicht zugelassen, wobei hingegen in C++ die Umwandlung sehr wohl möglich ist (vgl. Listing 2.6).

Neben der direkten Zuweisung, gibt es je nach Sprache auch noch die Möglichkeit der Typumwandlung im Rahmen eines arithmetischen Ausdrucks. Zum Beispiel bieten Java und C++ beide diese Möglichkeit.

```
1 int a = 3;
2 double b = 4.7 ;
3 double c = a + b;
```

Listing 2.4: arithmetische Typumwandlung

Damit der Ausdruck in Zeile 3 ausgewertet werden kann, wandelt der Java Compiler den Typen von `a` in den Typ `double` um. In Java, wie auch in C++, passt der Compiler den Typen des Wertes rechts vom Zuweisungsoperator dem Typen der Variablen auf der

linken Seite an.

Eine weitere Möglichkeit der impliziten Typumwandlung ist die Typumwandlung im Funktionsaufruf. Werden im Funktionsaufruf z.B Argumente mit einem algebraischen Datentyp erwartet, dann kann der Typ im Aufruf, wie bei der Zuweisung, in den Typ des entsprechenden Parameters konvertiert werden.

```
1 public void func(long a){ ... } // eine Funktion
2 int b = 1000;
3 // ...
4 func(b);     // der Aufruf
```

Listing 2.5: Typumwandlung im Funktiosaufruf in Java

Die Funktion verlangt einen Parameter vom Typen `long`. In Zeile 4 wird ihr im Aufruf aber ein Argument vom Typ `int` übergeben. Dieser Aufruf funktioniert nun trotzdem, da der Java Compiler im Aufruf den Typen von `b` zu `long` konvertiert.

Bei der impliziten Typumwandlung muss man sehr vorsichtig sein, da sie mitunter zu Informationsverlust führen kann. Während in Java nur dann implizite Typumwandlungen durchgeführt werden können, wenn kein Informationsverlust auftritt, gibt es in C++ auch unsichere, implizite Typumwandlungen, wie das folgende Codebeispiel anschaulich zeigt.

```
1 double a = 3.3;
2 int b = a;
```

Listing 2.6: double zu int C++ (Typeinschränkung)

Die Variable `b` vom Typ `int` bekommt die Variable `a` vom Typ `double` zugewiesen. Wenn eine Gleitkommazahl in eine ganze Zahl umgewandelt wird, dann wird der Teil hinter dem Komma einfach ignoriert, sodass in diesem Beispiel b nun den Wert 3 annimmt. Dadurch wurde ein Informationsverlust provoziert.

2.2.2 Explizit

Das Vorgehen im letzten Beispiel ist nun auch in Java möglich, allerdings kommt dort das Konzept der expliziten Typumwandlung zum Tragen. Dazu bieten verschiedene Sprachen den Cast-Operator `(typ)` an.

Syntax: `(typ)` Ausdruck

Hierbei wird der Wert des Ausdrucks in den angegebenen Typen konvertiert. Diese explizite Typumwandlung wird Cast genannt. Der Cast-Operator ist ein unärer Operator

und hat demzufolge z.B. in Java oder C++ eine höherer Präzedenz als die arithmetischen Operatoren [Pri15], sodass der Cast in der Regel in einem Ausdruck als Erstes ausgewertet wird.

```
1 int i = 1;
2 short s = (short) i + 1;
```

Listing 2.7: Einfacher Cast in Java

In Java findet in 2.7 allerdings keine Prüfung mehr statt, ob die Variable `s` den neuen Wert aufnehmen kann. Weder der Compiler noch die Laufzeitumgebung prüfen, ob der Wertebereich überhaupt passt, was zu schwerwiegenden Fehlern führen kann.

```
1 int i = 1000000000;
2 short s = (short) i;
```

Listing 2.8: Cast mit unpassendem Wertebereich

Durch die Umwandlung von `int` (4 byte) zu `short` (2 byte) werden die höherwertigen 2 Byte des `int`-Wertes verworfen, so dass `s` nach dem Cast den Wert -13.824 annimmt. Wie auch bei der impliziten Typumwandlung in C++ muss der Programmierer sich über den Wertebereich also sehr sicher sein, wenn er negative Seiteneffekte vermeiden will [Gü17].

Die explizite Typumwandlung funktioniert aber nicht unbedingt nur mit primitiven Datentypen, sondern je nach Sprache können auch komplexere zusammengesetzte Typen gecastet werden. Dies ist allerdings nur möglich, wenn z.B das Objekt durch Vererbung oder Implementierung tatsächlich auch zur Klasse des Zieltypen gehört [Str16]. Schauen wir uns dies beispielhaft in Java an:

```
1 Object o1 = new String("test");
2 String s = (String) o1
```

Listing 2.9: Cast von Objekttypen in Java

Diese Zuweisung gelingt, da das von `o1` referenzierte Objekt ein String ist.

3 Typisierung

Nachdem nun die Merkmale zu Typen, Typsicherheit, und Typumwandlung vorgestellt wurden, können wir nun weitere Kriterien behandeln, mit denen die Typsysteme der Programmiersprachen unterschieden werden können. Da wäre zum einen die Unterscheidung zwischen starker und schwacher Typisierung und die Unterscheidung zwischen statischer und dynamischer Typisierung wobei stark/schwach und dynamisch/statisch völlig unabhängig voneinander sind.

3.1 Starke und schwache Typisierung

Wenn zwischen starker und schwacher Typisierung unterschieden werden soll, so kann man nicht generell sagen, dass die eine Sprache stark und die andere schwach typisiert ist. Es ist vielmehr so, dass die einzelnen Sprachen zueinander jeweils unterschiedlich stark typisiert sind. Auch ist die Definition der „Stärke“ in diesem Zusammenhang nicht eindeutig.

Mögliche Merkmale einer Definition wären:

- Nichtvorhandensein von Konvertierungsmethoden und die fehlende Möglichkeit, das Typsystem zu umgehen
- Nur explizite Typumwandlungen möglich
- Typüberprüfung zur Übersetzungs- statt zur Laufzeit
- Implizite Typumwandlung nur zwischen ähnlichen Typen
- Generelle Unterscheidung zwischen Typen [Wik18b]

Im Allgemeinen weist eine stark typisierte Sprache zur Kompilierungszeit strengere Typisierungsregeln auf, was bedeutet, dass Fehler und Ausnahmen wahrscheinlicher während der Kompilierung auftreten. Die meisten dieser Regeln wirken sich auf Variablenzuweisung, Rückgabewerte und Funktionsaufruf aus. Auf der anderen Seite hat eine schwächer typisierte Sprache lockerere Typisierungsregeln und kann unvorhersehbare Ergebnisse erzeugen oder eine implizite Typumwandlung zur Laufzeit durchführen. Eine stark typisierte Sprache überwacht das Erstellen und den Zugriff auf alle Objekte. Dadurch ist dann sichergestellt, dass Variablen immer auf Objekte verweisen, die dem Typ entsprechende

Spezifikationen vorhalten. Schwach typisierte Sprachen haben, je schwächer die Typisierung wird, eben weniger Einschränkungen bei der Zuordnung. In schwach typisierten Sprachen kann es daher möglich sein, ein Objekt einer Variablen zuzuordnen, ohne dass das Objekt zwingenderweise die Spezifikationen der aufnehmenden Variable erfüllt [Str16].

```
1 Object o1 = new String("test");
2 Object o2 = new Integer(1);
3 String s1 =  o1; // Type mismatch
4 String s2 = (String) o1;
5 s1 = (String) o2; // Laufzeitfehler
```

Listing 3.1: Fehler bei der Typumwandlung in Java

Java ist zum Beispiel eher stark typisiert, da es nicht so ohne Weiteres möglich ist, einen beliebigen Speicherbereich als Repräsentation eines Objektes zu interpretieren. Damit `o1` einer Variablen vom Typ `String` zugewiesen werden kann, muss zwingend eine explizite Typumwandlung vorgenommen werden. Ob die Umwandlung zur Laufzeit klappt, hängt davon ab, ob das Objekt tatsächlich die Zieltypen der Umwandlung implementiert, was in Zeile 5 zu einem Laufzeitfehler führt. Ebenso besitzt z.B. Python eine starke Typisierung, da beim Umwandeln in der Regel explizite Funktionen verlangt werden.

Dahingegen wird C++ als eher schwach typisierte Sprache angesehen, da explizite Typumwandlungen vorgenommen werden können, welche die Typsicherheit verletzen. So ist in C++ folgender Aufruf möglich:

```
1 MeineKlasse* meinObjekt = meineKlasse();
2 String* text = new String("test");
3 text = (String*) meinObjekt;
```

Listing 3.2: Zuweisung in C++

Obwohl in Zeile 3 die Variable `text` das nicht kompatible Objekt `meinObjekt` vom Typ `MeineKlasse*` zugewiesen bekommt, wird zur Übersetzungszeit kein Fehler angezeigt. Soweit ähnelt C++ in seinem Verhalten Java. Aber während in Java der Fehler dann spätestens zur Ausführung entdeckt wird, bleibt dieser in C++ im Zweifel weiter unentdeckt. Wahrscheinlich wird erst der erste Programmzugriff auf die Variable zu einem Programmabbruch führen. Um die zu verhindern bietet C++ den Operator `dynamic_cast` [Str10].

Syntax: `dynamic_cast`<type-id>(Ausdruck)

Der `dynamic_cast`-Operator wird zur Laufzeit und nicht zur Kompilierzeit ausgeführt, da der Typ einer Variabel zur Kompilierzeit mitunter noch nicht bekannt ist. Hat das Objekt dann zur Laufzeit wirklich den angegebenen Typen, wird die Adresse mit dem

angegeben Datentyp zurückgeliefert. Hat das Objekt allerdings einen anderen Datentypen als den angegebenen, dann wird ein Nullzeiger zurückgeliefert.

In C++ sind unsichere Typumwandlungen, nicht nur explizit, sondern auch implizit möglich. So lassen sich z.B. `double` zu `int` (siehe Listing 2.6) oder `double` zu `char` implizit umwandeln. Diese Umwandlungen werden von Compiler akzeptiert, obwohl der gespeicherte Wert vom zugewiesenen Wert abweichen kann. Dies kann beim Programmieren zu Fehlern führen, da diese Abweichung zunächst einmal nicht ersichtlich ist. Da die unsichere Typumwandlung allgemein aber möglich ist, wird C++ daher den schwach typisierten Sprachen zugeordnet [Str16].

3.2 Statische und dynamische Typisierung

Die Betrachtung statischer und dynamischer Typisierung fällt jetzt wesentlich klarer aus, als dies bei starker bzw. schwacher Typisierung der Fall ist. Hier kann klar unterschieden werden, da Programmiersprachen entweder statisch oder dynamisch sind.

3.2.1 Statische Typisierung

In Programmiersprachen mit statischen Typsystemen wird der Typ von Variablen und Parametern im Quelltext deklariert. Variablen sind somit einem festen Typen zugeordnet, sodass dadurch eingeschränkt wird, welche Objekte den Variablen zugewiesen werden können.

```
1 String foo = "pokipsi";
2 char[] charArray = foo.toCharArray();
3 foo = new int[3];       //Fehler: Typen nicht kompatibel
```

Listing 3.3: Statisch in Java

In Zeile 1 bekommt die Variable `foo` den Typ `String` zugewiesen und wird mit dem Wert `pokipsi` initialisiert. In Zeile 2 braucht zur Laufzeit nicht mehr überprüft werden, ob `foo` die Methode `toCharArray()` überhaupt unterstützt, da dies bereits zur Übersetzungszeit bekannt ist. Das bedeutet dann natürlich auch, dass der Fehler in Zeile 3 direkt zur Übersetzungszeit entdeckt wird, da der Compiler weiß, dass `foo` vom Typen `String` ist, und aufgrund der statischen Typisierung jetzt nicht ohne Weiteres ein Objekt eines anderen Typen zugewiesen bekommen kann.

In statisch typisierten Programmiersprachen stellt der Compiler also schon beim Kompilieren fest, wenn eine Variable ein Objekt eines unpassenden Typen zugewiesen bekommt. Und das wiederum ermöglicht dem Compiler anhand des deklarierten Typen zu überprüfen, ob an der Variable aufgerufene Operationen oder Methoden zur Spezifikation

der deklarierten Klasse gehören.

Gegenüber dem dynamischen Typsystem hat das Statische einige Vorteile:

- Die Überprüfung, ob eine Operation zulässig ist oder nicht, muss nicht zur Laufzeit gemacht werden. Dies wurde bereits vom Compiler übernommen, sodass kompilierte Anwendung besser optimiert werden können.
- Dadurch, dass Variablen einen deklarierten Typ haben, ist der Programmcode lesbarer, da schneller klar wird, welche Aufgabe eine Variable hat.
- Fehler im Programm können frühzeitig durch den Compiler erkannt werden [Str16, S. 96].

3.2.2 Dynamische Typisierung

Programmiersprachen mit einem dynamischen Typsystem, zeichnen sich dadurch aus, dass Variablen im Quellcode keine deklarierenden Typen zugeordnet sind. Eine Variable kann dann Referenzen auf beliebige Objekte aufnehmen. Die Prüfung, ob eine Methode oder Operation auf, oder mit dem referenzierten Objekt durchgeführt werden kann, wird dann dynamisch zur Laufzeit und eben nicht statisch zur Übersetzungszeit durchgeführt. Beispiele für dynamisch typisierte Sprachen sind z.B. JavaScript, Python und Ruby.

Schauen wir uns nun folgendes Beispiel in Python an:

```
1  a = 1       # a enthaelt durch Zuweisung eine ganze Zahl
2  a += 1.0    # addiert die Gleitkommazahl 1.0 und legt
3              # neuen Wert (mit anderem Typ) in a ab
4  a.upper()   # Scheitert: a ist keine Zeichenkette
5  a           # gibt den Wert von a aus
6  -> 2.0
7  a = "jetzt ist a ein String"
8  a += 1      # Scheitert: Inhalt von a ist jetzt ein String
9  a.upper()   # Gibt den neuen String aus
10 -> 'JETZT IST A EIN STRING'
```

Listing 3.4: Dynamisch in Python (1)

In Zeile 1 nimmt `a` den Typ `int` an. In Zeile 2 kann `a` wegen der dynamischen Typisierung dann aber mühelos einen neuen Typen, in diesem Fall einen `float` zugewiesen bekommen. In Zeile 4 wird überprüft, ob das von `a` referenzierte Objekt auch tatsächlich die `upper()` Methode unterstützt. Da `a` nun vom Typ `float` ist, kann diese aber nicht darauf angewendet werden. Nachdem `a` in Zeile 7 durch die Zuweisung den Typen `str` annimmt, ist danach dann die `upper()` Methode anwendbar und der String wird anschließend ausgegeben.

An diesem Beispiel kann man sehr schön das Verhalten einer dynamisch typisierten Sprache bei der Zuweisung von Werten und Variablen sehen. Ein weiteres Merkmal dynamisch typisierter Programmiersprachen lässt sich bei der Definition von Funktionen sehen:

```
1 def add(x,y):
2     return x + y
```

Listing 3.5: Dynamisch in Python (2)

In diesem kurzen Code-Schnipsel lässt sich weder der Typ der Parameter ablesen die von der Funktion im Aufruf erwartetet werden, noch ist erkennbar von welchem Typ der Rückgabewert dieser Funktion ist. Das bedeutet für die dynamisch typisierten Sprachen einen Vorteil gegenüber statisch typisierten Sprachen. Diese einfache Funktion ist nun mit allen Typen zu verwenden, die eine Addition erlauben, wobei der Rückgabewert dann das Ergebnis dieser Addition ist. In Java würden wir für jeden passenden Datentypen eine eigene `add`-Funktion benötigen.

Auch wenn das statische Typsystem, gerade für Anfänger, einige Vorteile bietet, so hat das dynamische Typsystem durchaus Vorteile:

- Das dynamische Typsystem ist wesentlich flexibler. Da Variablen, Parameter und Ergebnisse von Funktionen keine deklarierenden Typen haben, können die Funktionen mit einer größeren Vielzahl von Objekten verwendet werden.
- Es entfällt die Notwendigkeit der expliziten Typumwandlung.

Auf der anderen Seite schafft die dynamische Typisierung aber auch Nachteile:

- Je nach Zustand des Programms können Funktionen Objekte unterschiedlichen Types zurückgeben. Das kann natürlich zu Problemen beim Programmieren führen. In statisch typisierten Sprachen liefert eine Methode immer einen festen Typen zurück.
- Da den Funktionen nicht immer anzusehen ist, welchen Datentyp sie als Parameter erwarten, oder welchen Typen sie als Rückgabewert haben, erschwert dies die Lesbarkeit des Codes.
- Zum besseren Verständnis des Codes ist in dynamischen Sprachen in der Regel ein erhöhter Dokumentationsaufwand notwendig.
- Da Fehler häufig erst zur Laufzeit entdeckt werden, erschwert dies die Fehlersuche.

4 Zusammenfassung

In der vorliegenden Arbeit wurden die einführenden Themen der Typisierung behandelt. Diese Arbeit umfasst aber bei Weitem nicht alle Aspekte des Programmierens, die mit der Typisierung verbunden sind. Sie kann aber als Grundlage für den Einstieg in das Thema dienen und sollte jetzt dabei behilflich sein, zu erkennen, welche Art der Typisierung eine bestimmte Programmiersprache hat, bzw. wie stark oder schwach ihr Typsystem ist. Darauf aufbauend sollte man sich mit der Umsetzung der jeweiligen Konzepte einzelner Programmiersprachen beschäftigen können, um diese sicher anzuwenden. Je komplexer die Software wird, die wir entwickeln, desto eher kommen wir in die Situation, dass Typumwandlungen notwendig sind. Auch hier sollte die Arbeit ein grundlegendes Verständnis geschaffen haben, um zwischen impliziten und expliziten Typumwandlungen zu unterscheiden und sich auf dieser Basis mit den weitergehenden Eigenschaften der Typumwandlung auseinandersetzen zu können.

Mit dem Wissen um die unterschiedlichen Typisierungskonzepte und den Umgang mit Typen in Abhängigkeit der gewählten Sprache, ist es einfacher, die passende Sprache zu wählen und sich mit den tiefergehenden Details auseinanderzusetzen, um auf dieser Basis schlussendlich typsichere Software zu entwickeln.

Literaturverzeichnis

[Gü17] Kai Günster. *Einführung in Java.* Rheinwerk Computing, Bonn, 2017.

[Pri15] Ulla Kirch; Peter Prinz. *C++ lernen und professionell anwenden.* mitp, Frechen, 2015.

[Str16] Bernhard Lahres; Gregor Raýman; Stefan Strich. *Objektorientierte Programmierung.* Rheinwerk Computing, Bonn, 2016.

[Wik18a] Wikipedia. Type System, 2018. [Online; accessed 13-April-2018].

[Wik18b] Wikipedia. Starke Typisierung, 2018. [Online; accessed 16-April-2018].

[Ull18] Christian Ullenboom. *Java ist auch eine Insel.* Rheinwerk Computing, Bonn, 2018.

[Str10] Bjarne Stroustrup. *Einführung in die Programmierung mit C++.* Pearson Studium, München, 2010.