

PR1, Aufgabenblatt 5

Programmieren I – Wintersemester 2021/22

Interfaces, Automatisiertes Testen

Ausgabedatum: 6. Dezember 2021

Lernziele

Schnittstellen mit Interfaces formulieren können; die Unterschiede zwischen Klassen und Interfaces kennen; Dienstleistungen eines Interfaces auf verschiedene Weise implementieren können; den statischen vom dynamischen Typ einer Variablen unterscheiden können; JUnit-Tests lesen können, JUnit-Tests formulieren können.

Kernbegriffe

In Java gibt es einen Mechanismus, mit dem ausschließlich der Umgang mit den Objekten einer Klasse modelliert werden kann, ohne Details ihrer Umsetzung festzulegen: *benannte Schnittstellen* (engl.: *named interfaces*). Mit benannten Schnittstellen kann die Schnittstelle einer Klasse (die durch ihre öffentlichen Methoden definiert ist) explizit im Programm formuliert werden, indem in einem eigenen Konstrukt (mit dem Schlüsselwort `interface` statt `class`) nur die Köpfe der öffentlichen Methoden (ohne ihre Rümpfe) aufgeführt werden. Mit Hilfe benannter Schnittstellen können wir also von konkreten Implementierungsdetails abstrahieren. Zur besseren Unterscheidung werden wir im Folgenden die benannten Schnittstellen von Java als *Interfaces* bezeichnen.

Genau wie eine Klasse definiert ein Interface einen Typ mit Operationen. Da ein Interface nur Methodenköpfe und keinerlei Anweisungen enthält, nennen wir die Methoden in einem Interface von nun an konsequent *Operationen* und sprechen nur noch von Methoden, wenn in Klassen Rümpfe mit Anweisungen vorliegen. Ein Interface kann keine Methodenrümpfe enthalten und keine Konstruktoren definieren, wir können keine Objekte von Interfaces erzeugen. Implementiert eine Klasse ein Interface, so wird dies in der Klassendefinition mit dem Schlüsselwort `implements` deklariert. Verwendet werden die Objekte dieser Klasse dann üblicherweise unter dem Interface-Typ.

Bei einer Referenzvariablen unterscheiden wir zwischen ihrem *statischen Typ* und ihrem *dynamischen Typ*. Der statische Typ wird bei ihrer Deklaration festgelegt und steht bereits zur Übersetzungszeit fest. Der dynamische Typ hängt von der Klasse des Objektes ab, auf das die Referenzvariable zur Laufzeit verweist. Er ist dynamisch in zweierlei Hinsicht: Er kann erst zur Laufzeit ermittelt werden, und er kann sich während der Laufzeit ändern.

Ein Objekt in Java weiß jederzeit, zu welcher Klasse es gehört. Selbst wenn die Variable, die zur Laufzeit auf ein Objekt verweist, statisch mit einem Interface-Typ deklariert wurde, kann mit dem `instanceof`-Operator (engl.: type comparison operator) ein *Typstest* vorgenommen werden, ob das referenzierte Objekt ein Exemplar einer bestimmten implementierenden Klasse ist. Um Operationen aufzurufen, die nicht im statischen Typ definiert sind, muss eine *Typzusicherung* (engl.: reference type casting) vorgenommen werden. Eine solche Typzusicherung verändert weder die Referenz noch das referenzierte Objekt (im Gegensatz zu den Typumwandlungen auf den elementaren Typen, die tatsächlich veränderte Werte liefern).

Testen ist eine Maßnahme zur Qualitätssicherung. Testen kann lediglich Fehler identifizieren, aber in der Regel nicht die Korrektheit einer Software nachweisen. Dennoch ist Testen in der Praxis der Software-Entwicklung unumgänglich, um das Vertrauen in eine Implementation zu erhöhen. Wir betrachten in PR1 primär *Modultests* (engl. unit tests), mit denen softwaretechnische Einheiten (Methoden, Klassen, Teilsysteme) getestet werden. Für jede Klasse einer Anwendung sollte es im Allgemeinen eine eigene Testklasse mit Testfällen geben. Jede Methode der zu testenden Klasse sollte in mindestens einem Testfall der Testklasse vorkommen. Das bedeutet aber *nicht*, dass es eine 1:1 Abbildung zwischen den Testmethoden und den Methoden der zu testenden Klasse gibt. Ein Testfall ruft meist mehrere Methoden der zu testenden Klasse auf, um ihr Zusammenspiel miteinander zu testen.


Wir sprechen von *Black-Box-Tests*, wenn die Testklasse sich ausschließlich auf die Schnittstelle der zu testenden Klasse bezieht; *White-Box-Tests* hingegen werden in Kenntnis der Implementation der zu testenden Klasse formuliert und versuchen, möglichst alle Pfade durch den Quelltext zu testen. Bezieht sich eine Testklasse nur auf ein Interface, formuliert sie automatisch einen Black-Box-Test. Tests sollten *reproduzierbar* sein, idealerweise sind sie *automatisiert wiederholbar*.

Aufgabe 5.1 Schnittstelle von Implementation trennen (Termin 1)

Erinnerst du dich noch an Tic Tac Toe aus der Vorlesung? Diesmal wollen wir das Spielfeld genauer untersuchen. Öffne das neue Projekt *TicTacToe* und schau dir die Schnittstellenansicht der Klasse *Spielfeld* an. Die Methoden der Klasse *Spielfeld* spezifizieren zusammen mit den Methodenkommentaren einen Datentyp. („Was ist ein Spielfeld? Was kann man damit machen?“)

Die Klasse *Spielfeld* gibt sowohl die Spezifikation des Spielfelds vor als auch eine konkrete Implementation (drei Exemplarvariablen referenzieren Objekte vom Typ *SpielfeldZeile*). Diese zwei Aspekte des Spielfelds wollen wir im Folgenden mit Hilfe der Interfaces von Java trennen.


5.1.1 Benenne die Klasse *Spielfeld* um, indem du im Quelltext hinter dem Schlüsselwort `class` den Namen *Spielfeld* durch *SpielfeldGeflecht* ersetzt. Speichere den geänderten Quelltext und übersetze die Klasse. Welchen Fehler gibt es und warum? Behebe den Fehler und übersetze die Klasse erneut. Das restliche Projekt lässt sich zu diesem Zeitpunkt nicht übersetzen. Dies ändert sich aber im Verlauf der Aufgabe.

 5.1.2 Erstelle ein Interface namens *Spielfeld*. Wähle dazu im BlueJ-Hauptfenster *Neue Klasse...* → *Interface* und trage den Namen *Spielfeld* ein.

Kopiere anschließend die drei *Methodenköpfe* aus *SpielfeldGeflecht* (ohne die Methodenrumpfe) in das Interface hinein, und zwar inklusive der Kommentare. Sorge dafür, dass sich das Interface übersetzen lässt. Welche Informationen sollten noch in das Interface? **Halte schriftlich fest**, warum ein Interface ohne Kommentare wertlos ist. Übersetze das Interface und wechsle zur Schnittstellenansicht. Überprüfe, ob alle Schnittstellenkommentare darin auftauchen.

5.1.3 Als nächstes wollen wir festlegen, dass die Klasse *SpielfeldGeflecht* eine Implementation der Spezifikation ist, die *Spielfeld* vorgibt. Ergänze dazu *SpielfeldGeflecht* an geeigneter Stelle. Wenn du die Klasse anschließend übersetzt, zieht BlueJ automatisch einen neuartigen Pfeil von *SpielfeldGeflecht* zu *Spielfeld*.

5.1.4 Beim Übersetzen der Klassen *TicTacToe* und *SpielfeldTest* gibt es noch jeweils einen Fehler bei der Exemplarerzeugung `new Spielfeld()`. Warum tritt dieser Fehler auf? Wie kannst du ihn beheben?

 5.1.5 **Erkläre schriftlich** die Unterschiede zwischen den beiden bisher vorgestellten Pfeilformen im BlueJ-Klassendiagramm.

Aufgabe 5.2 Alternative Implementationen (Termin 1)

5.2.1 Implementiere das Spielfeld mit einer weiteren Klasse *SpielfeldString*, die intern Strings der Länge 9 zur Darstellung des aktuellen Spielfeldzustandes verwendet. Lass die Klasse *TicTacToe* diese Implementation von *Spielfeld* verwenden und teste interaktiv, ob alles funktioniert. Das Interface darf dabei nicht verändert werden!

5.2.2 Implementiere einen Aufzählungstyp *Besitzer* für die möglichen Besitzer einer Spielfeldzelle. Schreibe eine neue Klasse *SpielfeldZeileEnum*, die statt `int`-Werten Referenzen auf *Besitzer* in den Zellen ablegt. Auch an der Schnittstelle der neuen Klasse sollen an den richtigen Stellen statt `int`-Werten *Besitzer*-Referenzen übergeben und geliefert werden. Schreib anschließend eine weitere neue Klasse *SpielfeldGeflechtEnum*, die **die gleiche Schnittstelle** wie *SpielfeldGeflecht* anbietet, intern aber Exemplare der Klasse *SpielfeldZeileEnum* verwendet. Lass die Klasse *TicTacToe* diese alternative Implementation von *Spielfeld* verwenden und teste interaktiv, ob alles funktioniert.

5.2.3 *Zusatzaufgabe*: Implementiere das Spielfeld mit einer weiteren Klasse *SpielfeldInteger*, die intern eine `int`-Variable zur Darstellung des aktuellen Spielfeldzustandes verwendet. Verwende die Bit-Operationen von Java, um den Zustand einer Zelle des Spielfeldes in jeweils zwei Bits der `int`-Variablen abzulegen (warum gerade zwei?). Lass die Klasse *TicTacToe* anschließend diese Implementation von *Spielfeld* verwenden und teste interaktiv, ob alles funktioniert.

Aufgabe 5.3 Statischer und dynamischer Typ einer Referenzvariablen (Termin 1)

Öffne das Projekt *Zahlensacke* und schau dir das **Interface** *Zahlensack* **gründlich** an. Die drei implementierenden Klassen setzen die im Interface geforderte Funktionalität mit verschiedenen Mitteln um. Was in den Klassen genau passiert, ist für uns nicht wichtig, wir wollen sie nur benutzen.

5.3.1 Schreibe eine Klasse *Lotto* mit einer Methode *sechsaus49()*, welche sechs verschiedene Zufallszahlen im Bereich 1-49 auf der Konsole (`System.out.println(...);`) ausgibt. **Verwende** dazu innerhalb der Methode einen *Zahlensack*.

5.3.2 Wir wollen die verschiedenen Implementationen auf deren Effizienz überprüfen. Dazu gibt es bereits eine (unvollständige) Klasse *Effizienzvergleich*, die in der Methode *vergleiche* Exemplare der drei Klassen erzeugt und diese an eine Methode *vermesse* weiterleitet. Ergänze den fehlenden Code in der Methode *vermesse*! Greif dazu auf `long System.nanoTime()` zurück, welche die aktuelle Zeit in Nanosekunden liefert. Um vernünftige Messergebnisse zu erhalten ist es notwendig, sehr viele Methodenaufrufe durchzuführen. Verwende dazu eine Zählschleife. Anschließend soll das Messergebnis einfach auf der Konsole ausgegeben werden, sinnvollerweise in der Einheit ms.

5.3.3 Setze einen Haltepunkt in der Methode *vermesse* und erkläre bei der Abnahme anhand des formalen Parameters *sack*, wo man den statischen und den dynamischen Typ sehen kann.

 5.3.4 **Erkläre schriftlich** die Begriffe *statischer Typ* und *dynamischer Typ* am Beispiel der lokalen Variable *zs* aus der Methode *vergleiche*.

Aufgabe 5.4 JUnit-Testklassen benutzen, lesen und verstehen (Termin 2)

JUnit ist eine Software, die automatisierte Modultests in Java unterstützt. Sie erlaubt, mit einfachen Methodenaufrufen in speziellen Testklassen Zusicherungen über eine zu testende Klasse zu überprüfen. Für diese Prüfungen stehen (durch die Zeile `import static org.junit.Assert.*;`) in einer JUnit-Testklasse unter anderem folgende Methoden zur Verfügung:

```
void assertTrue(boolean condition);
void assertFalse(boolean condition);

void assertEquals(boolean expected, boolean actual);
void assertEquals(char expected, char actual);
void assertEquals(int expected, int actual);
void assertEquals(Object expected, Object actual);

void assertSame(Object expected, Object actual);
void assertNotSame(Object expected, Object actual);

void assertNull(Object reference);
void assertNotNull(Object reference);
```

Testklassen können in BlueJ auf Knopfdruck gestartet werden, in einem eigenen Dialogfenster werden dann Fehlermeldungen angezeigt, falls eine Zusicherung nicht erfüllt ist.

Beim Ausführen einer Testklasse durch JUnit 4 werden alle public Methoden dieser Klasse aufgerufen, die die Annotation `@Test` besitzen. In JUnit 3 und früheren Versionen war es noch notwendig, Methodennamen mit „test“ beginnen zu lassen, also z.B. `testDruckeDokument` oder `testeDruckeDokument`. Dies entfällt seit JUnit 4, ist jedoch aus Tradition weiterhin üblich.

5.4.1 In BlueJ sind *JUnit*-Testklassen grün. Die Klasse *SpielfeldTest* im Projekt *TicTacToe* enthält bereits Testfälle, die du mit einem Rechtsklick auf das Klassensymbol und dem Menüeintrag *Alles testen* ausführen lassen kannst. Es öffnet sich ein neues Fenster. Erkläre bei der Abnahme, was du dort siehst.


5.4.2 Öffne den Quelltext der Klasse *SpielfeldTest*. Dort siehst du vier Test-Methoden, von denen die ersten drei bereits kommentiert sind. In der vierten Test-Methode wird die

Methode `assertEquals` mit zwei Parametern aufrufen, die einen Vergleich durchführt. Der erste Parameter gibt einen Erwartungswert an, und der zweite Parameter ist ein Ausdruck, dessen Ergebnis mit dem Erwartungswert verglichen wird.

Ändere den ersten Parameter eines der `assertEquals`-Aufrufe und lasse die Tests erneut durchlaufen. Was passiert nun? **Mach diese Änderung wieder rückgängig.**

Was genau testet die vierte Test-Methode? **Schreib einen möglichst exakten Javadoc-Kommentar** und benenne die Methode anschließend passend dazu sinnvoll um.

- 5.4.3 Öffne das Projekt *UhrenanzeigeTest*. Starte den Test durch einen Klick auf *Tests starten*. Versuch die Ausgaben von JUnit zu interpretieren. Schau dir dazu den Quelltext der Klasse *UhrenanzeigeTest* gründlich an. Wo kannst du ablesen, in welcher Quelltextzeile der Fehler aufgefallen ist? Korrigiere danach den Programmierfehler in der Klasse *Uhrenanzeige*, so dass der Test fehlerfrei durchläuft.

-  5.4.4 **Erkläre schriftlich:** Was ist der Unterschied zwischen `assertEquals` und `assertSame`?

Warum gibt es für `assertSame`, `assertNull` und `assertNotNull` keine Überladungen für die primitiven Datentypen?

Hinweis: Jedes Java-Objekt bietet die Operation `boolean equals(Object other)` an.

Aufgabe 5.5 JUnit-Testklassen schreiben (Termin 2)

- 5.5.1 Sorge dafür, dass im Projekt *TicTacToe* jede Implementierung von *Spielfeld* nicht nur interaktiv, sondern auch von der vorgegebenen JUnit-Testklasse getestet wurde, indem du mehrfach die Exemplarerzeugung anpasst und die Testklasse jeweils ausführst (falls du das nicht schon in Aufgabe 5.2 getan hast). Werden noch Fehler aufgedeckt?
- 5.5.2 Ein Exemplar deiner Klasse *KartenTripel* aus dem Projekt *MauMauSimulation* von Aufgabenblatt 4 soll sinnvollerweise drei *unterschiedliche* Karten (ungleich null!) enthalten und diese bei Nachfrage auch liefern. Teste die Klasse *KartenTripel* möglichst vollständig mit einer JUnit-Klasse. Auch wenn deine Klasse *KartenTripel* die hier gestellten Anforderungen bisher nicht erfüllt: Schreibe *zuerst* die Testklasse und lasse die Tests fehlschlagen; korrigiere erst dann deine Klasse *KartenTripel*.
- 5.5.3 Teste eine Implementation des Interfaces *Zahlensack* mit JUnit. An welchen Stellen musst du wissen, um welche Implementation es sich handelt? Beziehen sich deine Testfälle auf das Interface oder auf die Implementation? Diskutiere dies bei der Abnahme.