SE1, Aufgabenblatt 8

Softwareentwicklung I – Wintersemester 2011/12

Interfaces, Systematisches Testen

MIN-CommSy-URL: ..... https://www.mincommsy.uni-hamburg.de/
CommSy-Projektraum ..... SE1 CommSy WiSe 11/12
Ausgabewoche ..... 08. Dezember 2011

Kernbegriffe

In Java gibt es ein Sprachkonzept, 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 programmiersprachlich 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 Interfaces wird somit ermöglicht, von konkreten Implementierungsdetails zu abstrahieren. Zur besseren Unterscheidung nennen wir im Folgenden die benannten Schnittstellen von Java *Interfaces*.

Genau wie eine Klasse definiert ein Interface einen Typ mit seinen 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 enthält keine Methodenrümpfe und kann auch keine Konstruktoren definieren, wir können also mit Interfaces keine Objekte 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 und ihrem dynamischen Typ. Der deklarierte Typ einer Variablen definiert ihren *statischen Typ* (statisch, weil er zur Übersetzungszeit feststeht). 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 verliert niemals die Information, 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 ein *Typtest* vorgenommen werden, ob das referenzierte Objekt ein Exemplar einer implementierenden Klasse ist. Um eine Operation an einem solchen Objekt aufzurufen, die nicht im statischen Typ definiert ist, kann eine *Typzusicherung* durch einen Typ-Cast 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 unser Vertrauen in eine Implementation zu erhöhen. Wir betrachten in SE1 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 geben, in der alle Methoden der Klasse getestet werden.

Tests können in *Positiv-* und *Negativtests* unterschieden werden. Positivtests testen die *Korrektheit*, indem sie die geforderte Funktionalität durch korrekte Benutzung überprüfen. *Negativtests* testen zusätzlich die *Robustheit*, indem sie das Verhalten der Software bei fehlerhafter Benutzung überprüfen. Tests sollten *reproduzierbar* sein, idealerweise sind sie *automatisiert wiederholbar*.

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 internen Struktur der zu testenden Klasse formuliert und versuchen, möglichst alle Teile der Implementierung zu testen. Bezieht sich eine Testklasse nur auf ein Interface, formuliert sie automatisch einen Black-Box-Test.

Lernziele

Schnittstellen mit Interfaces formulieren können; die Unterschiede zwischen Klassen und Interfaces kennen; den statischen vom dynamischen Typ einer Variablen unterscheiden können; Typtests und Typzusicherungen verstehen und programmieren können; Positiv- und Negativtests unterscheiden können; Black-Box-Tests formulieren können.

Aufgabe 8.1 Schnittstelle von Implementation trennen

Erinnert ihr euch noch an Tic Tac Toe? Diesmal wollen wir das Spielfeld genauer untersuchen. Öffnet das neue Projekt *TicTacToe* und schaut euch die Schnittstellenansicht der Klasse `Spielfeld` an. Die aufgelisteten Methoden spezifizieren zusammen mit den beschreibenden Kommentaren 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 beiden unterschiedlichen Aspekte des Spielfelds wollen wir im Folgenden mit Hilfe der Interfaces von Java trennen.

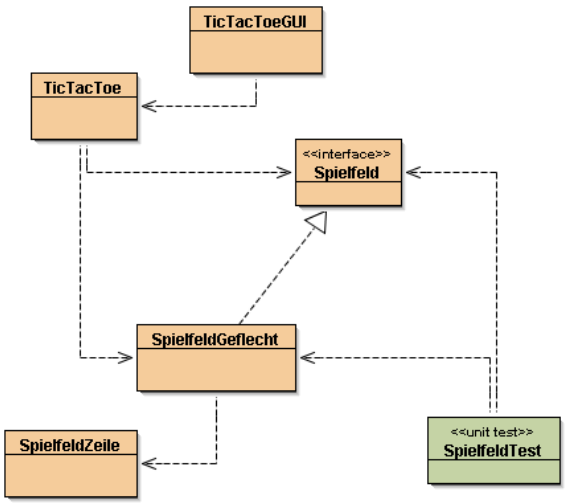
8.1.1 Benennt die Klasse `Spielfeld` um, indem ihr im Quelltext hinter dem Schlüsselwort `class` den Bezeichner `Spielfeld` durch `SpielfeldGeflecht` ersetzt. Speichert den geänderten Quelltext und übersetzt die Klasse. Welchen Fehler gibt es und warum? Behebt den Fehler und übersetzt diese Klasse erneut. Das restliche Projekt lässt sich zu diesem Zeitpunkt nicht übersetzen. Dies ändert sich aber im Verlauf der Aufgabe.

8.1.2 Erstellt ein Interface namens `Spielfeld`. Wählt dazu im BlueJ-Hauptfenster *New Class...* → *Interface* und tragt den Namen `Spielfeld` ein.

Kopiert die drei *Methodenköpfe* aus `SpielfeldGeflecht` (ohne die Methodenrümpfe) in das neue Interface hinein, und zwar inklusiv der Kommentare. Hinter den Methodenköpfen müsst ihr jeweils ein Semikolon einfügen. Denkt auch daran, den ausführlichen Klassenkommentar zu kopieren. **Haltet schriftlich fest**, warum ein Interface ohne Kommentare wertlos ist. Übersetzt das Interface und wechselt zur Schnittstellenansicht. Überprüft, ob alle Schnittstellenkommentare darin auftauchen.

8.1.3 Als nächstes wollen wir festlegen, dass `SpielfeldGeflecht` eine Implementation der Spezifikation ist, die durch `Spielfeld` vorgegeben ist. Ändert dazu in der Klasse `SpielfeldGeflecht` die Zeile `class SpielfeldGeflecht` in
`class SpielfeldGeflecht implements Spielfeld`
Wenn ihr die Klasse anschließend kompiliert, wird in BlueJ automatisch ein Pfeil von `SpielfeldGeflecht` zu `Spielfeld` gezogen, der unsere letzte Änderung grafisch darstellt.

8.1.4 Beim Übersetzen der Klassen `TicTacToe` und `SpielfeldTest` gibt es noch jeweils einen Fehler beim Konstruktoraufruf `new Spielfeld()`. Warum tritt dieser Fehler auf? Wie könnt ihr ihn beheben? Wenn sich das Projekt erfolgreich kompilieren lässt, sollte es in etwa wie folgt aussehen:



Zielvorstellung für Aufgabe 8.1

8.1.5 Sicher ist euch bereits aufgefallen, dass die Klasse `SpielfeldTest` grün ist. BlueJ kennzeichnet auf diese Weise spezielle *JUnit*-Testklassen. Unsere enthält bereits Testfälle, die ihr mit einem Rechtsklick auf das Klassensymbol und dem Menüeintrag *Alles testen* ausführen lassen könnt. Es öffnet sich ein neues Fenster. Erklärt eurer/m Betreuer/in bei der Abnahme, was ihr dort seht.

8.1.6 Öffnet den Quelltext der Klasse `SpielfeldTest`. Dort seht ihr drei Test-Methoden, von denen die ersten beiden bereits kommentiert sind. In der dritten Test-Methode wird die Methode `assertEquals` mit zwei Parametern aufgerufen, 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.

Ändert den ersten Parameter eines der `assertEquals`-Aufrufe und lasst die Tests erneut durchlaufen. Was passiert nun? **Macht diese Änderung wieder rückgängig.** Viele weitere Möglichkeiten von JUnit werden in Aufgabe 8.3 eingeführt.

Was genau testet die dritte Test-Methode? Schreibt einen Kommentar und benennt sie sinnvoll um.

Aufgabe 8.2 Statischen und dynamischen Typ von Referenzvariablen unterscheiden

Öffnet das Projekt *Zahlensaecke* und schaut euch das Interface `Zahlensack` 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.

8.2.1 Schreibt eine Klasse `Lotto` mit einer Methode `sechsAus49()`, welche sechs verschiedene Zufallszahlen im Bereich 1-49 auf der Konsole ausgibt. Verwendet dazu innerhalb der Methode einen Zahlensack.

8.2.2 Wir wollen die verschiedenen Implementationen auf deren Effizienz überprüfen. Dazu gibt es bereits eine (unvollständige) Klasse `Effizienzvergleicher`, die in der Methode `vergleiche` Exemplare der drei Klassen erzeugt und diese an eine Methode `vermesse` weiterleitet. Ergänzt den fehlenden Code in der Methode `vermesse`! Greift 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. Verwendet dazu eine Zählschleife. Anschließend soll das Messergebnis einfach auf der Konsole ausgegeben werden, sinnvollerweise in der Einheit ms. Was beobachtet ihr, wenn ihr `vergleiche` mit verschiedenen Werten für `groesse` aufruft, z.B. 100/1000/10000?

8.2.3 Setzt einen Haltepunkt in der Methode `vermesse` und erklärt eurem Betreuer anhand des formalen Parameters `sack`, wo man den statischen und den dynamischen Typ sehen kann.

8.2.4 **Erklärt schriftlich** die Begriffe *statischer Typ* und *dynamischer Typ* am Beispiel der lokalen Variable `zs` aus der Methode `vergleiche`.

Aufgabe 8.3 Testunterstützung durch JUnit

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 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.

Um die JUnit-Unterstützung in BlueJ zu aktivieren, müsst ihr unter *Werkzeuge -> Einstellungen -> Diverses* den Punkt „Testwerkzeuge anzeigen“ auswählen.

Beim Ausführen einer Testklasse durch JUnit werden alle Methoden dieser Klasse aufgerufen, deren Name mit „`test`“ beginnt, also z.B. mit **`test`**`DruckeDokument` oder **`test`**`eDruckeDokument`.

8.3.1 Findet Dokumentation im Internet zu den oben genannten Prüfmethoden von JUnit und **erklärt schriftlich**, was diese Methoden machen und wie sie verwendet werden sollen.

8.3.2 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.

8.3.3 Öffnet das Projekt *UhrenanzeigeTest*. Startet den Test durch einen Klick auf *Tests starten*. Versucht die Ausgaben von JUnit zu interpretieren. Was ist der Unterschied zwischen *Fehler* (engl.: error) und *Nicht bestanden* (engl.: failure)? Schaut euch dazu insbesondere den Quelltext der Klasse `UhrenanzeigeTest` gründlich an. Korrigiert danach den Programmierfehler in der Klasse `Uhrenanzeige`, so dass der Test fehlerfrei durchläuft.

8.3.4 *Zusatzaufgabe:* Testet eine Implementation des Interfaces `Zahlensack` mit JUnit. An welchen Stellen müsst ihr wissen, um welche Implementation es sich handelt? Beziehen sich eure Testfälle auf das Interface oder auf die Implementation? Diskutiert dies mit euren Betreuern.

Aufgabe 8.4 Alternative Version des Spielfelds implementieren und testen

8.4.1 Das nächste Ziel ist eine alternative Implementation des Spielfelds. Erstellt eine neue Klasse namens `SpielfeldZeichenkette` und fügt wieder `implements Spielfeld` ein. Versucht, die Klasse zu übersetzen. Warum schlägt dies fehl? **Begründet schriftlich**.

8.4.2 Benutzt die Klasse `StringBuilder`, um `SpielfeldZeichenkette` zu implementieren (d.h. die Klasse `SpielfeldZeichenkette` sollte eine Exemplarvariable vom Typ `StringBuilder` haben). Exemplare der Klasse `StringBuilder` sind veränderbare Zeichenketten (im Gegensatz zu Exemplaren der Klasse `String`, welche unveränderlich sind). Verwendet eine Zeichenkette der Länge 9, um das Spielfeld zu repräsentieren. Der Konstruktoraufruf `new StringBuilder("\0\0\0\0\0\0\0\0\0")` liefert eine veränderliche Zeichenkette, die aus neun (unsichtbaren) Zeichen mit dem Unicode 0 besteht.

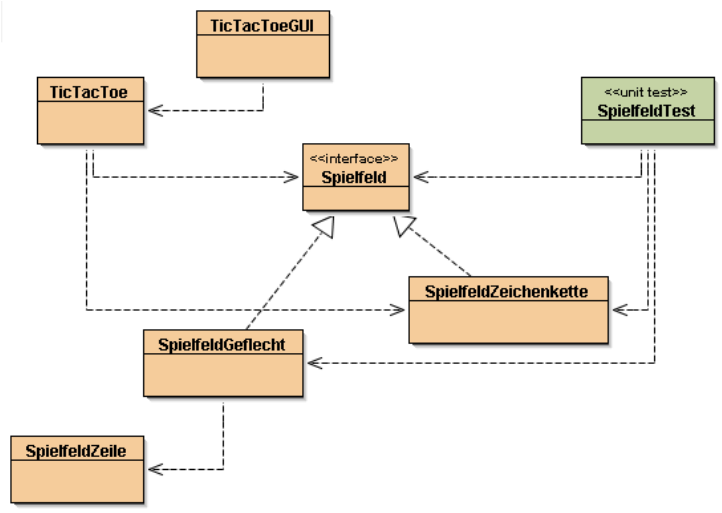
8.4.3 Implementiert nun alle Methoden, die das Interface `Spielfeld` vorgibt. Dafür sind folgende Operationen der Klasse `StringBuilder` interessant: `charAt`, `setCharAt` und `indexOf`.

Macht euch Gedanken darüber, wie ihr eine zweidimensionale Positionsangabe (0-2, 0-2) eineindeutig in einen eindimensionalen Index 0-8 umrechnen könnt.

Um einen `int i` in einem `StringBuilder` abzulegen, muss dieser erst per `(char)i` gecastet werden. (Dies funktioniert nur für `int`-Werte zwischen 0 und 65535 verlustfrei, was hier kein Problem darstellt.)

8.4.4 Testet `SpielfeldZeichenkette`, indem ihr den Konstruktor `SpielfeldTest` entsprechend anpasst. Sobald die Tests erfolgreich durchlaufen, könnt ihr die neue Klasse auch in der Spiellogik verwenden. Passt dazu den Konstruktor in der Klasse `TicTacToe` analog an und spielt ein paar Spiele.

Das fertige Projekt sollte jetzt wie folgt aussehen:
(Hinweis: BlueJ aktualisiert die Darstellung der Beziehungen nicht immer korrekt.)



Zielvorstellung für Aufgabe 8.4

8.4.5 **Erklärt schriftlich** die Unterschiede zwischen den verschiedenen Pfeilformen im BlueJ-Klassendiagramm.