HSLU Hochschule Luzern

Objektorientierte Programmierung (OOP)

Übung SW06: Polymorphie / Unit-Testing

Themen: Nutzung der Polymorphie (O08), Unit-Testing (E02)
Zeitbedarf: ca. 240min.

Roland Gisler, Version 1.5.6 (HS24)

1 Polymorphie

1.1 Lernziele

- Sie können Methoden (und Konstruktoren) überladen.
- Wie wissen wie und wo man Methoden überschreiben kann.
- Sie können zwischen statischem und dynamischen Datentyp unterscheiden.
- Sie verstehen das Konzept des Subtyping.

1.2 Grundlagen

Für diese Übung greifen wir auf verschiedene Übungen der Vorwochen zurück und überarbeiten bzw. ergänzen diese. Damit Sie diese Weiterentwicklung später auch nachvollziehen können, empfehlen wir Ihnen die entsprechenden Klassen und Interfaces in die aktuelle Woche (package) zu **kopieren** und dort zu erweitern.

1.3 Aufgaben

Überladen (overload) von Methoden:

a.) Verwenden Sie die `Point`-Klasse aus SW04. Implementieren Sie eine Methode `moveRelative(int x, int y)`, mit welcher Sie den Punkt mittels eines Vektors (`x`- und `y`-Abschnitte) relativ zur aktuellen Position verschieben können.

b.) Ein Point kann gleichzeitig auch einen Vektor darstellen. Somit können wir die Methode `moveRelative(…)` auch mit einem `Point` als Parameter überladen und entsprechend implementieren! Probieren Sie es aus!

c.) Als dritte Variante könnten wir den Vektor auch als Polarkoordinaten (Winkel und Betrag) angeben. Damit es etwas genauer wird, verwenden wir für den Winkel den Datentyp `double`. Wie würde somit die Signatur aussehen?

Diese dritte Variante ist eine gute Programmier- (und auch Geometrie-) Übung. Wichtig ist insbesondere die Frage: Ist diese (technisch mögliche) Überladung auch gut? Oder gäbe es bessere Alternativen? Überlegen und diskutieren Sie!

d.) Was mit normalen Methoden funktioniert, ist auch mit Konstruktoren möglich. Vermutlich haben Sie für die `Point`-Klasse einen Konstruktor `Point(int x, int y)` implementiert (wenn nein: sofort ergänzen!). Nun möchten wir einen Punkt auf Basis eines bereits bestehenden Punktes erstellen (sprich: diesen kopieren). Ergänzen Sie einen zweiten Konstruktor mit der Signatur `Point(Point point)` und implementieren Sie ihn, aber **ohne** Coderedundanzen zu erzeugen! Solche Konstruktoren bezeichnet man als **Copy-Constructors**.

Überschreiben (override) von Methoden:

e.) In SW05 haben Sie eine abstrakte **`Element`**-Klasse und drei konkrete Elemente (**`Stickstoff`**, **`Quecksilber`** und **`Blei`**) implementiert. Die in der Basisklasse abstrakt definierten Methoden haben Sie in den konkreten Elementen überschrieben! Studieren Sie den Code den Sie dort geschrieben haben nochmal!

f.) Implementieren bzw. überschreiben Sie in der **`Element`**-Klasse die bereits auf **`Object`** definierte **`String toString()`**-Methode. Konsultieren Sie dafür ggf. die API-Dokumentation der Klasse **`Object`** und studieren Sie die Intention dieser Methode.

g.) Überschreiben Sie die **`toString()`**-Methode auch auf der Klasse **`Quecksilber`** und ergänzen Sie den String mit dem Warnhinweis «GIFTIG». Achtung, auch hier sollen Sie keine Coderedundanzen produzieren, stattdessen verwenden Sie die Implementation der **`Element`**-Klasse wieder. Sie erinnern sich noch an das Schlüsselwort? Wenn ja: **`super`**! ☺

Subtyping, dynamischer und statischer Datentyp:

h.) Nun verwenden wird die Klassen **`Shape`**, **`Circle`** und **`Rectangle`** wieder.
Probieren Sie folgendes aus: Definieren Sie zwei Variablen vom Typ **`Shape`** (z.b. **`shape1`** und **`shape2`**). Und weisen Sie diesen je ein (neu erzeugtes) **`Circle`**- bzw. **`Rectangle`**-Objekt zu. Warum funktioniert das?

i.) Können Sie über diese **`shape`**-Variablen die Methode **`move(x,y)`** aufrufen? Überlegen Sie zuerst und probieren Sie es erst danach aus.

j.) Nun möchten wird über die **`shape`**-Variable welche den Kreis enthält mit **`getDiameter()`** den Durchmesser abfragen. Was müssen Sie machen, damit das auch tatsächlich möglich ist? Tipp: Die gleiche Technik haben wir schon zur Umwandlung von elementaren Datentypen verwendet!

k.) Erklären Sie sich selber anhand der letzten Aufgabe die Begriffe statischer und dynamischer Datentyp!

l.) Subtyping funktioniert nicht nur mit vererbten Klassen, sondern auch mit allen von Klassen implementierten Interfaces! Wenn Sie also aus SW04 eine Klasse haben die **`Switchable`** ist (oder auch aus SW05: **`Named`**), dann können Sie auch «nur» diese Typen verwenden, um die entsprechenden Funktionen aufzurufen. Probieren Sie es aus. **Das ist Polymorphie**!

2 Unit Testing mit JUnit-Framwork

2.1 Ziele

- Gute und sinnvolle Unit-Tests schreiben und nutzen können.
- Funktionsweise des JUnit Testing Frameworks verstehen.
- JUnit in der IDE Ihrer Wahl anwenden können.

2.2 Grundlagen

JUnit, welches von Kent Beck entwickelt wurde, gehört zu den mit Abstand populärsten (Testing-)Frameworks. Die meisten gängigen IDE's integrieren dieses Framework und vereinfachen dessen Verwendung sehr. Auch im zur Verfügung gestellten Template-Projekt ist JUnit 5 bereits integriert. Sie finden es in ihren Projekten in der Liste der (Test-)Dependencies. Im Rahmen dieses Tutorials sollen Sie sich mit JUnit vertraut machen.

Wichtig: Wir arbeiten mit der neusten Generation 5.x (Jupiter) von JUnit.

2.3 Aufgaben

a.) Verwenden Sie eine Klasse mit einer einfachen Methode mit Parametern und Rückgabewert wieder, z.B. die **`max(x,y[,z])`**-Funktion aus der SW03. Wir wollen uns in diesem Beispiel absichtlich nur mit einer trivialen Funktion auseinandersetzen, damit wir uns vollständig auf das Testen konzentrieren können.

b.) Lassen Sie sich von Ihrer IDE einen JUnit-Testcase zu Klasse/Methode von a) erstellen, und studieren Sie das generierte Codegerüst. Machen Sie sich mit den Namenskonventionen (der Testklasse und der Testmethoden) vertraut. Lesen Sie in der JUnit-Dokumentation bzw. API nach, welchen Effekt die Annotationen **`@Test`**, **`@Disable`**, **`@BeforeAll`**, **`@BeforeEach`**, **`@AfterEach`** und **`@AfterAll`** haben.

c.) Überlegen Sie sich nun die Testfälle für die in a) vorgeschlagene **`max(x,y)`**-Funktion. Es sind «nur» drei wesentliche Fälle. Implementieren Sie die Testfälle einzeln und machen Sie sich bei dieser Gelegenheit mit den zahlreich verfügbaren **`assert*()`**-Methoden der Klasse **`org.junit.jupiter.api.Assertions`** vertraut.

Hinweis: Widerstehen Sie unbedingt der Versuchung, alle Testfälle bzw. Assert-Statements in eine **einzige** Testmethode zu packen. Überlegen Sie sich aber warum! Was sind die Vor- und Nachteile? Tipp: Wir schreiben Testfälle, um im Fehlerfall den Fehler möglichst schnell (selektiv) zu finden!

d.) Führen Sie die Testcases in Ihrer IDE aus. Lassen Sie einen Testcase (durch einen absichtlich eingebauten Fehler in der Implementation) auch bewusst einmal failen, damit Sie die verschiedenen Ausgaben bei einem Failure kennenlernen.

e.) **Ab sofort gilt für all Ihre zukünftigen Übungen und Projekte**:
Identifizieren Sie fortlaufend alle Klassen und Methoden welche sich gut und einfach mit (J)Unit-Tests testen lassen (was sind eigentlich die Kriterien?) und Implementieren Sie für diese ab sofort konsequent (und noch besser: **im Voraus ➔ Test First)** Testfälle!

Wichtiger Hinweis: Auch wenn die Erstellung der Testfälle etwas Zeit benötigt, gewinnen Sie damit mittelfristig sehr viel Zeit, weil die Tests danach beliebig oft und vollautomatisch ausgeführt und validiert werden können (➔Regression). Sie müssen nichts mehr von Hand machen, und auch die Resultate nicht auf Basis von mühsamen Konsolenausgaben nachvollziehen!

Das Schreiben von Unit-Tests lohnt sich somit immer und auch für kleinste Projekte!

3 Entwickeln nach dem Test-First-Prinzip

3.1 Ziele

- Erleben des Test-First-Ansatzes aus "extrem programming" (XP).
- Erstellen von auf JUnit basierenden Testfällen.

3.2 Grundlagen

Im Rahmen dieser Übung soll ein fachlich sehr einfaches Problem gelöst werden: Eine **`Calculator`**-Klasse soll implementiert werden, welche die **`addition(int, int)`** von zwei Summanden anbieten soll. Im Unterschied zum vorherigen Beispiel wollen wir nun konsequent den Test-First-Ansatz anwenden. Folgen Sie dazu **bitte exakt** den folgenden Teilschritten!

3.3 Aufgaben

a.) Überlegen Sie sich als erstes die Schnittstelle (also nur den Methodenkopf) der **`addition(…)`**-Methode für die **`Calculator`**-Klasse.

b.) Codieren Sie diese Schnittstelle als Java-Interface. Dokumentieren Sie diese als weitere kleine Übung wieder sauber mit JavaDoc. Die IDE sollte Sie dabei unterstützen. Denken Sie an die einfache, aber wichtige ‚*Ein-Satz-mit-Punkt.*'-Regel.

c.) Erstellen Sie nun eine noch leere Implementation des Interfaces von b), also eine Klasse, welche Sie in der IDE auf Basis des Interfaces generieren lassen können, und die noch **keine** Implementation enthält. Implementieren Sie also die eigentliche Addition **noch nicht**!

d.) Schreiben Sie nun **die Testfälle** als JUnit-Testcases. Lassen Sie sich auch hier ein Grundgerüst (auf Basis des Interfaces) von Ihrer IDE erstellen und programmieren Sie die Testfälle dann aus. Instanziieren Sie dafür ein Objekt der (noch leeren) **`Calculator`**-Klasse in eine Referenz vom Typ des Interfaces ➔ angewandte Polymorphie!
Überlegen Sie sich wirklich **gute** und **sinnvolle** Testparameter! Es kann sehr helfen sich eine kleine Tabelle mit Werten und den erwarteten Resultaten zu notieren.

e.) Sobald Sie die Testcases programmiert haben, können Sie diese ausführen. Natürlich werden Sie nun feststellen, dass die Testcases konsequent failen, weil die **`Calculator`**-Klasse ja noch gar nicht implementiert ist!

f.) Implementieren Sie nun schrittweise die **`Calculator`**-Klasse und beobachten Sie, wie Ihre Testfälle immer 'grüner' werden, also immer weniger Failures auftreten.
Eine erste Implementation der Addition könnte z.B. einfach den Wert '**`0`**' zurückgeben. Wenn Sie sich gute Testfälle überlegt haben, existiert vielleicht ein '**`0 + 0 = 0`**'-Testfall der dann bereits «grün» wird, obschon Sie noch gar nicht wirklich addieren!

Wichtig: Wenn Sie während der Implementation plötzlich Ausnahmen oder Sonderfälle entdecken, dann schreiben (oder korrigieren) Sie konsequent **zuerst** den dazu passenden **Testfall** (der diese Ausnahme prüft), **bevor** Sie die eigentliche Implementation vornehmen bzw. korrigieren!

g.) Wenn alle Testfälle «grün» sind, vergleichen Sie Ihr Interface und Ihre Testfälle mit anderen Studierenden. Haben Sie, obschon es sich um ein wirklich sehr einfaches Beispiel handelt, wirklich an alles gedacht? Haben Sie sinnvolle Testdaten verwendet?
Hinterfragen Sie sehr kritisch, die Chance ist tatsächlich gross, dass Sie eine Überraschung[1] erleben. Haben Sie schon einmal die Resultate Ihrer Additionen auf die Konsole ausgegeben (oder alternativ: Schritt für Schritt mit dem Debugger ausgeführt)?

[1] Es soll eigentlich nicht zur Regel werden, dass Sie erst bei den Übungen überrascht werden! ☺