Aufgabe 4 - Klassendiagramm umsetzen

Ziel

Sie können ein gegebenes Klassendiagramm in den entsprechenden Programmcode übersetzen.

Auftrag

1. Übersetzen Sie die Klassendiagramme (`Bottle` und `BankAccount`) in entsprechenden Python-Code.
   Es wird erwartet, dass Sie vergleichbar der Abbildung 1.8 und dem Beispiel 1.5 im Kapitel 5. Klassendiagramm und Klasse diese Aufgabe umsetzen.
2. Setzen Sie die BZZ-Codeconvention um!
   Die Aufgabe ist nur dann vollständig gelöst, wenn Sie auch die BZZ-Vorgaben bezüglich «guter Programmierung» einhalten.

Vorgehen

Wir möchten, dass das Vorgehen nach TDD (test driven development) im Modul 320 konsequent umgesetzt wird. Die nötigen Testfälle werden durch die Lehrpersonen bereitgestellt und sind pro Übung auf github verfügbar.
Sie finden weiter unten die genauen Anleitung für das jeweilige Vorgehen. Dieses ist bei der Implementation in allen kommenden Übungen anzuwenden.
Es ist wichtig, dass Sie sich dieses Vorgehen aneignen, da auch die Prüfungen so umgesetzt sind und die erreichte Note durch die Anzahl korrekter Tests gegeben ist.

Dauer

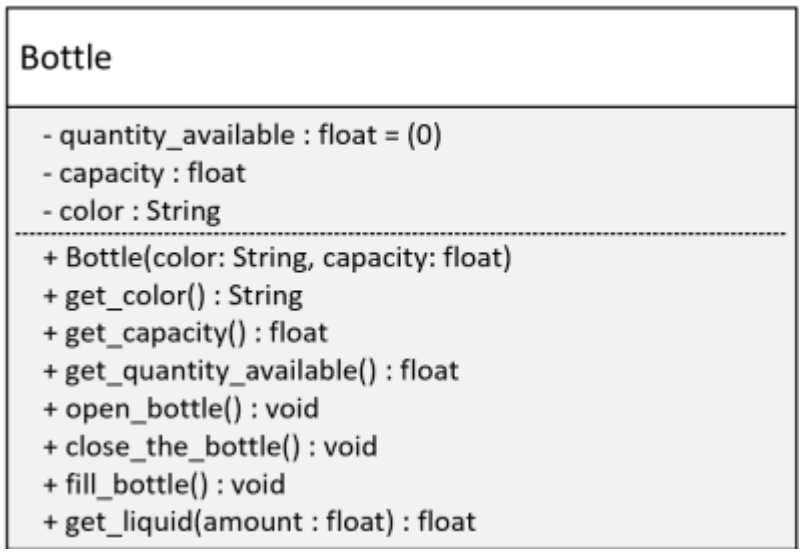
40 Minuten + Hausaufgabenzeit

Abgabe

Commit und Push auf github

Anleitung zu Klasse Bottle

Klassendiagramm:



Beschreibung der Klasse:
Die Klasse Bottle stellt eine einfache Trinkflasche mit 3 Attributen sowie einigen Methoden dar.
Attribute

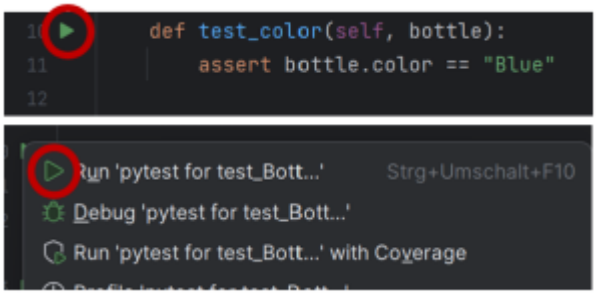
- `quantity_available` hält die aktuelle Menge in der Flasche fest.
- `capacity` ist die maximale Menge, die in der Flasche Platz findet.
- `color` ist die Farbe der Flasche.

Methoden

- `__init__(…)` (der Konstruktor der Klasse Bottle) initialisiert die verfügbare Menge (quantitiy_avaible) auf 0, während Kapazität (capacity) und Farbe (color) der Flasche durch Parameter festgelegt werden.
- `color()` liefert die Farbe der Flasche.
- `capacity()` liefert die maximale Menge der Klasse.
- `quantitiy_available()` liefert die in der Flasche vorhandene Menge.
- `open_bottle()` wird leer implementiert (Keyword pass verwenden)
- `close_the_bottle()` wird leer implementiert.
- `fill_bottle()` füllt die Flasche bis zum maximalen Fassungsvermögen.
- `get_liquid(amount)` liefert die angeforderte Menge aus der Flasche, falls diese Menge noch verfügbar ist. Wenn die verfügbare Menge (quantitiy_avaible) kleiner ist, wir diese Menge geliefert und die Flasche ist leer.

Vorgehen:

1. Akzeptieren Sie das Assignment `M320_LU01_A4_CodeAusDiagramm` im GitHub Classroom.
2. Klonen Sie ihr Repository in die Entwicklungsumgebung (vergl. dazu das Vorgene bei der Übung LU01-A4b).
3. Implementieren Sie den Konstruktor (`__init__(…)`) und initialisieren Sie die Attribute gemäss Beschreibung.
4. Implementieren Sie die getter-Methode `color` als `@property` (Sie wissen nicht, was das ist? Dann schauen Sie im Modul 319 nach.) und führen Sie in der Testklasse (`test_Bottle_class.py`) die Testmethode `test_color` aus. Sie muss fehlerfrei ablaufen.



Das Ergebnis des Tests muss wie folgt aussehen:

```
============================= test session starts ==============================
collecting ... collected 1 item

test_Bottle_class.py::TestBottle::test_color <- test_bottle_class.py PASSED [100%]

============================== 1 passed in 0.01s ===============================
```

Wird ein Fehler signalisiert, muss die Methode solange bearbeitet werden, bis der Test «passed» ist.
Beispiel eines fehlerhaften Testlaufs:

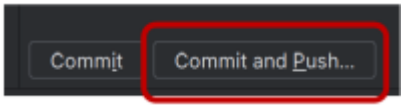
```
============================= test session starts ==============================
collecting ... collected 1 item

test_Bottle_class.py::TestBottle::test_color <- test_bottle_class.py

============================== 1 failed in 0.12s ===============================
FAILED [100%]
test_bottle_class.py:9 (TestBottle.test_color)
'green' != 'Blue'

Expected :'Blue'
Actual   :'green'
<Click to see difference>
```

5. Führen Sie in Ihrer Entwicklungsumgebung das git-Kommando für den commit und push aus. Wichtig: Immer dann, wenn Sie einen Test erfolgreich ausgeführt haben, pushen Sie das Programm. So ist sichergestellt, dass auf git jederzeit lauffähiger (und bewertbarer) Code liegt!

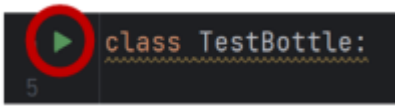


6. Implementieren Sie die getter-Methode `capacity()` und führen Sie in der Testklasse die Testmethode `test_capacity` aus. Sie muss fehlerfrei ablaufen. Pushen Sie ihren Code.
7. Implementieren Sie die getter-Methode `quantitiy_avaible()` und führen Sie in der Testklasse die Testmethode `test_initial_quantity` aus. Sie muss fehlerfrei ablaufen. Pushen Sie ihren Code.
8. Implementieren Sie nun die beiden Methoden `open_bottle()` und `close_the_bottle()` mit der `pass`-Anweisung (leere Methoden ohne Funktion). Führen Sie die Testmethode `test_open_and_close_bottle` aus. Sie muss fehlerfrei ablaufen. Pushen Sie ihren Code.
9. Implementieren Sie jetzt die Methode `fill_bottle()` gemäss der Beschreibung. Überlegen Sie sich, wie Sie sicherstellen können, dass die Flasche ganz gefüllt ist. Führen Sie die Testmethode `test_fill_bottle` aus. Sie muss fehlerfrei ablaufen. Pushen Sie ihren Code.
10. Implementieren Sie nun die Methode `get_liquide(…)` und stellen Sie sicher, dass die gelieferte Menge korrekt ist. Dazu müssen Sie überprüfen, ob die angeforderte Menge (`amount`) in der Flasche verfügbar ist. Weiter müssen Sie sicherstellen, dass der Inhalt der Flasche um den Betrag verringert wird. Reicht der Inhalt nicht, wird einfach der Rest in der Flasche geliefert (und die Flasche ist leer)
11. Testen Sie nun die Methode der Reihe nach mit
- `test_get_liquid_avaible` ob der eingefüllte Wert korrekt ist.

- `test_get_liquid_not_avaible` ob bei einem zu grossen Wert für `amount` das Ergebnis korrekt ist.
- `test_get_liquid_partial_avaible` ob bei einem Wert kleiner dem Fassungsvermögen die Werte korrekt sind.

Führen Sie nach jedem der Tests eine commit und push aus!

12. Führen Sie nun die Testklasse als Ganzes aus. Klicken Sie auf den grünen Pfeil neben dem Klassennamen.



Das Testergebnis muss nun wie folgt aussehen:

```
============================= test session starts ==============================
collecting ... collected 8 items

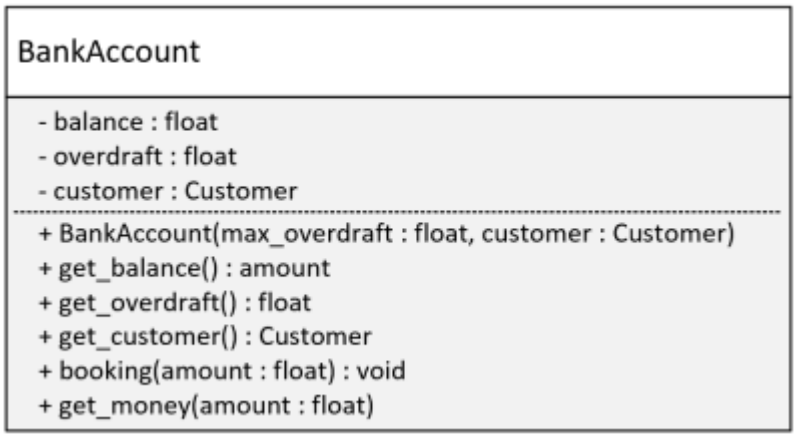
test_bottle_class.py::TestBottle::test_color PASSED                      [ 12%]
test_bottle_class.py::TestBottle::test_capacity PASSED                   [ 25%]
test_bottle_class.py::TestBottle::test_initial_quantity PASSED           [ 37%]
test_bottle_class.py::TestBottle::test_fill_bottle PASSED                [ 50%]
test_bottle_class.py::TestBottle::test_get_liquid_available PASSED       [ 62%]
test_bottle_class.py::TestBottle::test_get_liquid_not_available PASSED   [ 75%]
test_bottle_class.py::TestBottle::test_get_liquid_partial_available PASSED [ 87%]
test_bottle_class.py::TestBottle::test_open_and_close_bottle PASSED      [100%]

============================== 8 passed in 0.02s ===============================
```

—-

Anleitung zu Klasse BankAccount

Klassendiagramm:



Beschreibung

Die Klasse BankAccount beschreibt ein Bankkonto für einen Kunden (`Customer`).
Das Konto kann innerhalb eines bestimmten Wertes überzogen werden, d.h. dass auch ein negativer Saldo möglich ist.
Attribute

- `balance` gibt den aktuellen Kontostand (Saldo) wieder.
- `overdraft` legt fest, um welchen Betrag das Konto überzogen werden darf, d.h. welcher

Minusbetrag möglich ist. (typisch für ein Kreditkonto)
- `customer` ist die Referenz auf ein Objekt der Klasse `Customer`.

Methoden

- `__init__(…)` (der Konstruktor der Klasse BankAccount) initialisiert den Saldo (`balance`) auf 0.0 und legt den Überzug (`overdraft`) sowie den referenzierten Kunden (`customer`) fest.
- `balance()` liefert den aktuellen Saldo des Kontos (kann auch negativ sein).
- `overdraft()` liefert den max. Betrag, um den das Konto überzogen werden darf.
- `customer()` liefert die Referenz zu einem Customer-Objekt.
- `booking(…)` bucht einen Betrag (`amount`) ins Konto ein und erhöht somit den Saldo (`balance`).
- `get_money(…)` bucht einen Betrag (`amount`) vom Konto ab. Dabei darf der Betrag max. so gross sein, dass Saldo + Überzug nicht überschritten werden. Ist der Betrag zu gross, liefert die Methode den Wert 0.0 zurück (= kein Bezug möglich).

**Vorgehen: **

1. Implementieren Sie den Konstruktor (`__init__(…)`) und initialisieren Sie die Attribute gemäss Beschreibung.
2. Erstellen Sie die getter-Methdode (als `@property`) für das Attribut `balance` und testen Sie dies mit der Testmethode `test_initial_balance` in der Datei `test_BankAccount_class.py`.
   Der Test muss fehlerfrei ausgeführt werden. Pushen Sie ihren Code.
3. Erstellen Sie die getter_Methode für das Attribut `overdraft` und testen Sie diese mit `test_initial_overdraft`.
   Der Test muss fehlerfrei ausgeführt werden. Pushen Sie ihren Code.
4. Erstellen Sie die getter-Methode für das Attribut `customer` und testen Sie diese mit `test_customer`.
   Der Test muss fehlerfrei ausgeführt werden. Pushen Sie ihren Code.
5. Erstellen Sie die Methode `booking(…)` gemäss der Beschreibung. Testen Sie die Methode mit `test_booking`.
   Der Test muss fehlerfrei ausgeführt werden. Pushen Sie ihren Code.
6. Erstellen Sie die Methode `get_money(…)` gemäss der Beschreibung. Achten Sie darauf, wie der angeforderte Betrag bezüglich Saldo (`balance`) und Überzug (`overdraft`) geprüft werden muss.
7. Testen Sie nun die Methode der Reihe nach mit
   - `test_get_money_available` für einen korrekten Bezug.
   - `test_get_money_not_avaiable` für einen nicht gültigen Bezug.
   - `test_get_money_overdraft` für einen Bezug innerhalb der Kredit-Limite.
   - `test_balance_after_transaction` für die Kontrolle des Saldos.
   Führen Sie nach jedem der Tests einen commit und push aus!
8. Führen Sie nun die Testklasse als Ganzes aus.
   Das Testergebnis muss wie folgt aussehen:

```
============================= test session starts ==============================
collecting ... collected 8 items

test_BankAccount_class.py::TestBankAccount::test_initial_balance PASSED  [ 12%]
test_BankAccount_class.py::TestBankAccount::test_initial_overdraft PASSED [ 25%]
test_BankAccount_class.py::TestBankAccount::test_customer PASSED         [ 37%]
test_BankAccount_class.py::TestBankAccount::test_booking PASSED          [ 50%]
test_BankAccount_class.py::TestBankAccount::test_get_money_available PASSED [ 62%]
test_BankAccount_class.py::TestBankAccount::test_get_money_not_available PASSED [ 75%]
test_BankAccount_class.py::TestBankAccount::test_get_money_overdraft PASSED [ 87%]
test_BankAccount_class.py::TestBankAccount::test_balance_after_transactions PASSED [100%]

============================== 8 passed in 0.02s ===============================
```

© René Probst

From:
https://wiki.bzz.ch/ - **BZZ - Modulwiki**

Permanent link:
https://wiki.bzz.ch/modul/archiv/m320/learningunits/lu01/aufgaben/lu1-aufgabe_3

Last update: **2026/08/30 14:06**