

Labosessie week 3 en 4

Methoden met returntype

Theorie

Theorie die je vóór deze labosessie moet verwerkt hebben, heb je gezien in het hoorcollege van 6 oktober.

Doel

In dit labo leren we **methodes toepassen met een returntype (verschillend van void)**. We maken oefeningen volgens het concept van **unit testing**. Ook maken we gebruik van de **debugger** van BlueJ. Uiteraard verdiepen we verder hoe zelf klassen te ontwerpen.

Formaat

Elke labosessie is onderverdeeld in drie verschillende delen: startoefeningen, labo-oefeningen en thuisoefeningen. **Startoefeningen** zijn oefeningen waarvan we verwachten dat je ze thuis oplost, vóór de labosessie. Op die manier geraak je vertrouwd met de materie en check je of je de theorie van de hoorcolleges goed begrepen en verwerkt hebt. **Labo-oefeningen** zijn oefeningen die tijdens de labosessie worden behandeld. **Thuisoefeningen** zijn oefeningen die dieper ingaan op de materie. Ze zijn optioneel voor wie sneller werkt, of als extra oefeningmateriaal bij het studeren/oefenen thuis. Neem contact op met jouw labo-assistent voor feedback op deze oefeningen.

Leerdoelstellingen

Na deze sessie moet je in staat zijn om de volgende concepten te begrijpen en te implementeren:

- Begrijpen en zelf schrijven van methoden met een returntype verschillend van void.
- Unit testing begrijpen en oefeningen kunnen maken volgens dit formaat.
- De debugger van BlueJ kunnen gebruiken.
- Klassen met de adequate instantievariabelen en methoden ontwerpen op basis van een gegeven probleembeschrijving.

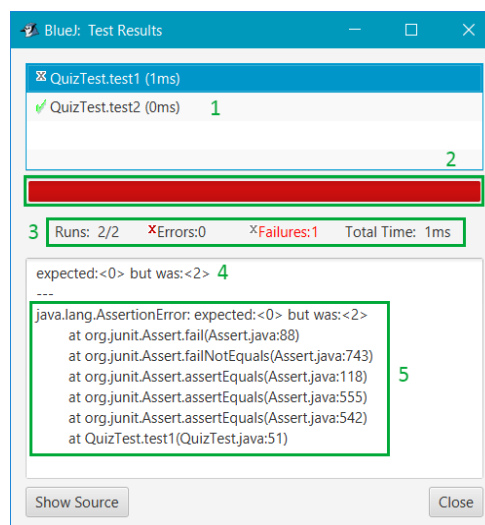
Startoefeningen

Oefening 1: Intro tot unit testing

Download, unzip en open het project Quiz vanop Toledo. Open de klasse Quiz en tracht de code te begrijpen. Deze klasse stelt een quiz voor met één multiplechoicevraag. Laten we dit eens uittesten.

1. Maak een nieuw Quiz-object door te rechtsklikken op de klasse en de constructor aan te roepen. Geef eender welke vraag in met bijhorende antwoorden (denk eraan, strings staan in Java tussen “dubbele aanhalingstekens”). Geef ook het nummer van het correcte antwoord mee (een integer).
2. Je ziet nu een rood quiz-object verschijnen onderaan het scherm. Wanneer je hierop rechtsklikt zie je alle methoden die gedefinieerd zijn voor dit object. Voer de methode `printQuestion` uit. Er opent nu een venster waarin de vraag en bijhorende antwoorden geprint worden.
3. Voer de methode `answerQuestion` uit, en geef het nummer van het juiste antwoord in. Check wat er geprint is. Werkt dit zoals verwacht?

Nee dus. Er zit blijkbaar een fout in onze code. De groene klasse QuizTest had je dat ook kunnen vertellen. Dit is een voorbeeld van een testklasse, een klasse die één of meer methodes bevat die als doel hebben om te checken of de code die je in een andere klasse hebt geschreven correct is. Als je rechtsklikt op deze klasse en ‘Test All’ selecteert, worden al deze testmethodes uitgevoerd en wordt de output getoond in een resultaatvenster:



We kunnen veel nuttige informatie afleiden uit dit venster:

1. Er zijn twee testmethodes uitgevoerd. Eén is gefaald (grijze X), de andere geslaagd (groene V).
2. Niet alle tests slaagden. Als dat wel zo was geweest, zou deze balk groen zijn gekleurd.
3. Informatie over de uitgevoerde tests. Merk op dat er een verschil is tussen tests die een error (crash) in je code veroorzaken (rode X), en tests die niet het verwachte resultaat produceren (grijze X).

4. Als je een gefaalde test selecteert kan je hier de reden zien waarom hij gefaald is. In dit geval verwacht de testklasse dat het correcte antwoord 2 is, terwijl het in realiteit 0 was¹.
5. De callstack – m.a.w., de plaats in de code waar de test faalt. Je zal hier typisch eerst een aantal interne Java-methoden zien die je niet veel wijzer maken, maar onderaan zie je wel relevante informatie: de test is gefaald in de klasse QuizTest, methode test1, lijn 51.

Laten we eens analyseren wat er gebeurt op die lijn. Je kan op 'Show Source' klikken onderaan het venster, of je kan gewoon de testklasse openen en zelf naar de betreffende lijn navigeren.

```

43  /**
44   * Tests the constructor and getters for question and correct answer
45   */
46  @Test
47  public void test1()
48  {
49      Quiz quiz = new Quiz( "Who plays agent Coulson in the Marvel Cinematic Universe?", "Simon Pegg", "Clark Gregg", "Nick Clegg", 2 ); //Creates a new quiz object with given values
50      assertEquals( quiz.getQuestion(), "Who plays agent Coulson in the Marvel Cinematic Universe?" ); //checks whether the method getQuestion returns the given string
51      assertEquals( quiz.getCorrectAnswer(), 2 ); //Checks whether the correct answer is set to 2
52  }

```

Dit is een voorbeeld van een testmethode. Elke testmethode bevat minstens één aanroep van een assert-methode (meestal `assertEquals`). Een assert call checkt of twee waarden gelijk zijn aan elkaar, en normaal gezien vergelijk je hier het resultaat van een methode-aanroep met een waarde – de verwachte returnwaarde van de methode. In deze testmethode hebben we twee asserts, de tweede staat op lijn 51, waar onze test faalt. Deze assert verwacht dat de waarde die gereturd wordt door `getCorrectAnswer` gelijk is aan 2, maar zoals we hierboven zagen retournt de methode 0.

Je kan controleren dat dit klopt: door op je quiz-object te dubbelklikken (of te rechtsklikken en 'Inspect' te selecteren) opent er een venster dat de waarde van alle instantievariabelen van dit object toont. Hier kan je zien dat de waarde van `correctAnswer` inderdaad 0 is.

Maar hoe komt dit nu, wat gaat er precies fout? Probeer dat eerst zelf te ontdekken door de code in de klasse Quiz te inspecteren en te vergelijken met wat er gebeurt in de testklasse. Als je denkt dat je het probleem hebt gevonden (en opgelost), of als je het niet kan vinden, het antwoord staat hieronder.

Het probleem ligt in de constructor van de Quiz-klasse. Hier krijgen we vijf parameters met waarden voor al onze instantievariabelen, maar we gebruiken de vijfde parameter niet. Als je niet expliciet een waarde toekent aan een instantievariabele in de constructor, zal de variabele een standaardwaarde krijgen. Wat die waarde is hangt af van het type van de variabele: voor getallen is het 0, voor booleans is het false, voor objecten is het null, enzovoort. Het is een goed idee om in de constructor altijd expliciet een waarde te geven aan alle instantievariabelen.

In de loop van het semester zal je nog dikwijls in contact komen met unit testing, al vanaf deze labosessie waar we zullen leren om een klasse te bouwen op basis van een gegeven testklasse. Dit is ook de aanpak van het examen, dus zorg ervoor dat je begrijpt hoe dit principe werkt.

¹ Als je de error leest lijkt het alsof het omgekeerd is, maar dat is niet zo. De informatie wordt gewoon op een onintuïtieve manier weergegeven.

Oefening 2: Intro tot debugging

Als het je niet is gelukt om in de vorige oefening de fout te vinden, was er nog een ander hulpmiddel dat je hierbij had kunnen helpen: de debugger. Dit is een manier om lijn per lijn je code te doorlopen, en te inspecteren wat er achter de schermen gebeurt terwijl je code uitgevoerd wordt.

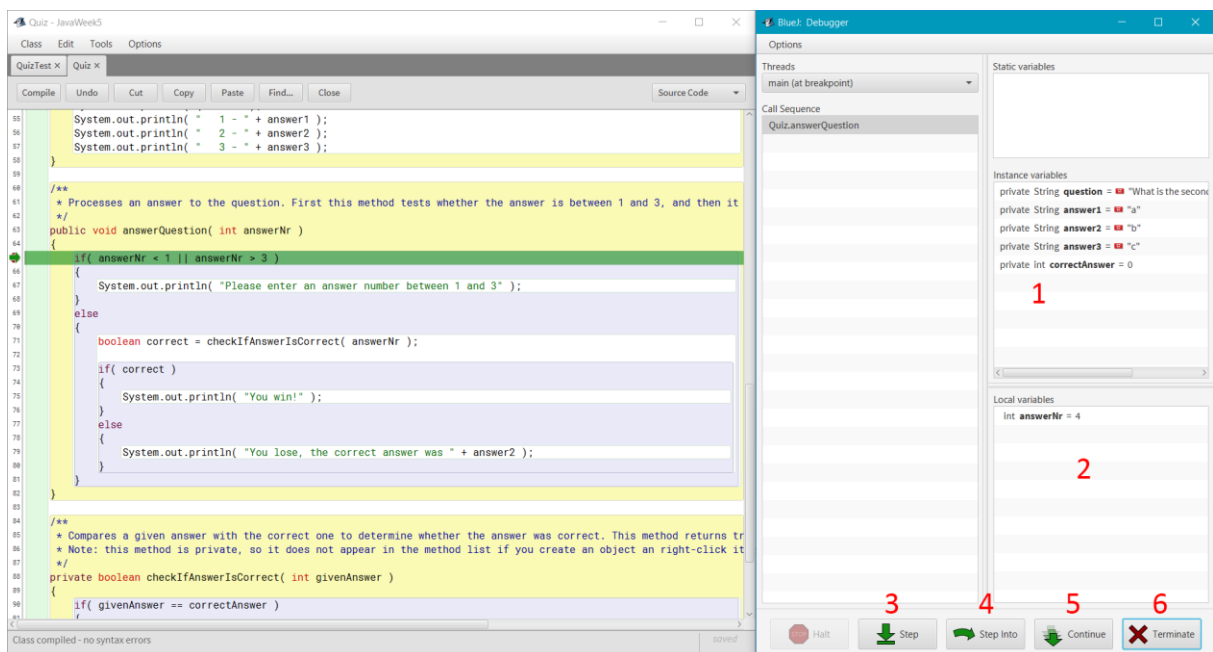
1. Om de debugger te activeren moet je minstens één breakpoint toevoegen op een bepaalde lijn. Dit onderbreekt de uitvoering van je code op die lijn. Voeg een breakpoint toe door te klikken op het eerste lijnnummer van de methode `answerQuestion` (compileer eerst!):

```


60 /**
61  * Processes an answer to the question. First this method tests whether the answer is between 1 and 3, and then it prints an output based on whether the given answer was correct.
62  */
63 public void answerQuestion( int answerNr )
64 {
65     if( answerNr < 1 || answerNr > 3 )
66     {
67         System.out.println( "Please enter an answer number between 1 and 3" );
68     }
69     else
70     {
71         boolean correct = checkIfAnswerIsCorrect( answerNr );
72
73         if( correct )
74         {
75             System.out.println( "You win!" );
76         }
77         else
78         {
79             System.out.println( "You lose, the correct answer was " + answer2 );
80         }
81     }
82 }

```

2. Om het breakpoint te activeren moeten we deze functie aanroepen. Creëer een quiz-object met door jou gekozen parameters, maar zorg ervoor dat het juiste antwoord nummer 2 is.
3. Rechtsklik en voer de methode `answerQuestion` uit met als antwoordwaarde 4. Er opent nu een nieuw debuggerscherm. Je bekijkt dit best samen met je code, als volgt dus:



4. Inspecteer de instantie- en lokale variabelen in panelen (1) en (2). Is dit wat je verwacht?
5. Nu kan je vrij door je code lopen met knoppen 3 tot 6. De Step-knop (3) beweegt de groene cursor lijn per lijn. Step Into (4) springt naar binnen een functieaanroep. Continue (5) laat het programma uitvoeren zoals normaal. Terminate (6) stopt de debugger.
6. Klik een aantal keer op de Step-knop (3), en kijk welke takken van de if-statement er worden gevolgd. Check panelen (1) en (2) om te zien of dit werkt zoals verwacht.

7. Maak nu een nieuw, identiek object (zelfde parameterwaarden), maar voer deze keer `answerQuestion` uit met waarde 2.
8. Gebruik Step tot lijn 71. Hier roepen we een andere methode van de Quiz-klasse aan. Als je hier nog eens op Step klikt, wordt deze call uitgevoerd en gaat de debugger naar de volgende lijn. Klik in plaats daarvan op Step Into. Nu springt de debugger in de methode `checkIfAnswerIsCorrect`, en gaat daarin lijn per lijn verder.
9. Blijf nu met Step door je code lopen. Denk bij elke stap na welke lokale of instantievariabele er zal veranderen en controleer je idee. Verwijder ten slotte het breakpoint door op het rode stopteken te klikken  .

Labo-oefeningen

Oefening 1: Een Barrel aanmaken

Download het project `barrel_start` van Toledo, unzip en open het. In dit labo zullen we stap voor stap een `Barrel`-klasse (vat) bouwen, gespreid over meerdere oefeningen. Momenteel bevat het project enkel een testklasse.



a. Klasse en constructor

1. Open de testklasse. Je zal zien dat al de testmethodes momenteel uitgecommentarieerd zijn. We doen dit omdat de code anders niet zou compileren, aangezien ze afhangt van een klasse en methodes die nog niet bestaan.
2. Verwijder de eerste testmethode uit commentaar. De simpelste manier om dit te doen is om een `/` te zetten na de `*` van de laatste lijn commentaar (lijn 47), voor de `@Test` annotatie, en dan de laatste `*/` te verwijderen na de methode (lijn 56). Zorg dat je zeker NIET de `@Test` annotatie verwijdert, deze annotatie zorgt er namelijk voor dat de compiler de methode herkent als testmethode.
3. Momenteel zal je code niet compileren, en je kan zien waarom: de klasse `Barrel` en haar **constructor** bestaan nog niet. Maak ze aan. De constructor heeft drie parameters nodig: het soort vloeistof in het vat, de maximale capaciteit en de inhoud. Sla deze informatie op als (private!) instantievariabelen. Welke types² ga je gebruiken, op basis van de testklasse?

b. Getters

4. Je code zal nu nog steeds niet compileren. Als je de testmethode nakijkt zal je zien dat er nog drie methodes ontbreken in de `Barrel`-klasse: `getLiquidType()`, `getContent()` en `getCapacity()`. Dit zijn voorbeelden van **getters** of **accessors**: methodes die de waarde van één van de instantievariabelen van de klasse returnen. Implementeer deze methodes.
5. Hierna zou je code wél moeten compileren³. Voer je eerste test uit door te rechtsklikken op de groene testklasse en 'Test All' te selecteren.

c. Een meer robuuste constructor

6. Haal nu test 2 uit commentaar. Analyseer de testcode om te zien welke veranderingen er nodig zijn aan je constructor (vergeet ook het commentaar niet te lezen!). Implementeer de nodige veranderingen en check of de test slaagt.

Oefening 2: Overloading

Overloading laat je toe om meer dan één methode te hebben met dezelfde naam in een klasse. De enige vereiste is dat het aantal of type van de parameters verschillen. Zo zijn `add(int a, int b)` en `add(int a, int b, int c)` mogelijk; alsook `print(String a, int b)` en `print(String a, String b)`. Wat bv. niet kan is `add(int a, int b)` en `add(int c, int d)`, want beide

² Merk op dat er twee types zijn voor decimale getallen: `float` en `double`. Zoals de naam suggereert is `double` dubbel zo precies als `float` (of meer accuraat: het gebruikt dubbel zoveel bytes). Voor waarden kan je aangeven dat je `float` precisie wil door een `f` aan het einde te zetten (bv. `10.5` is `double` precisie, `10.5f` is `float` precisie).

³ Gebruik altijd de compile-knop in je projectview, nooit die in de code editor. Deze laatste compileert niet alle klassen, waardoor je soms compileerfouten krijgt bij code die je al hebt aangepast in een andere klasse.

methodes hebben twee integers als parameters, zodat het voor de compiler onmogelijk uit te maken valt wat het verschil is.

Overloading werkt ook met constructors. Haal test 3 uit commentaar en gebruik overloading om je Barrel klasse uit te breiden zodat de test slaagt.

Oefening 3: De inhoud van een barrel veranderen

a. Setters

1. **Setters** of **mutators** zijn de tegenhanger van getters. Het zijn methoden die de waarde van één van de instantievariabelen veranderen. Een typische setter retournt niets (void). Haal test 4 uit commentaar en implementeer de gevraagde setter.
2. Het is geen goed idee om blindelings setters aan te maken voor alle instantievariabelen. Probeer bijvoorbeeld eens een setter aan te maken voor de inhoud van de barrel:

```
public void setContent( double newContent )
```
3. Nu zullen we deze setter testen door manueel een Barrel-object aan te maken (rechtsklik op de klasse en run de constructor) met een capaciteit van 20 en een content van 0. Inspecteer het object om te zien of deze waarden correct werden opgeslagen. Voer nu de setter uit, met een waarde van 40. Inspecteer je object nogmaals. Je hebt nu een barrel met tweemaal zoveel inhoud als capaciteit, wat onmogelijk zou moeten zijn!

b. De barrel vullen

4. Het is duidelijk dat het veranderen van de inhoud van een barrel wat meer denkwerk (en code) vraagt dan een setter kan bieden. Verwijder `setContent` zodat we niet langer willekeurige waardes kunnen toewijzen aan onze inhoud. We zullen nu twee aparte methodes schrijven: `fill`, die vloeistof toevoegt aan de barrel; en `consume`, die vloeistof verwijdert uit de barrel. Beide methodes zullen ervoor zorgen dat we geen illegale operaties kunnen uitvoeren.
5. Haal test 5, 6 en 7 uit commentaar. Deze hebben te maken met het vullen van een barrel. Analyseer de code in deze testmethodes en de bijhorende commentaar om te kijken hoe je code moet omgaan met speciale gevallen. Implementeer de `fill`-methode zodat alle drie de testen slagen. Merk op dat de `fill`-methode niets retournt.

c. De barrel uitgieten

6. Haal tests 8 en 9 uit commentaar, deze hebben te maken met het uitgieten van vloeistof uit de barrel. Check opnieuw je testcode om te zien wat je methode moet kunnen doen.
Hint 1 : de methode `consume` retournt altijd de hoeveelheid vloeistof die effectief uit de barrel is verwijderd (dus als je probeert om meer te consumeren dan de barrel bevatte, zal de hoeveelheid die er in de barrel zat worden teruggegeven).
Hint 2 : voor test 9 heb je een lokale variabele nodig⁴.

Oefening 4: Testcode om printmethodes te testen

Als (voorlopig) laatste onderdeel van onze barrel-oefening gaan we een methode schrijven die informatie over de barrel print op het scherm. Haal test 10 uit commentaar. Je zal zien dat de code voor deze test een stuk ingewikkelder is dan die van eerdere testen. Dit komt omdat er een hoop

⁴ Een lokale variabele is een variabele die gedefinieerd is binnenin de body van een methode. Dit soort variabele kan enkel gebruikt worden binnen het codeblok (methode, if, lus) waarin ze werden gedefinieerd.

“boilerplate code” nodig is zodat je testcode kan lezen wat je hebt geprint vanuit een methode. Je kan het merendeel van deze code negeren, het enige dat relevant is zijn de strings in het `try`-blok.

Implementeer nu de methode `showInfo` in je `Barrel`-klasse en zorg ervoor dat ze exact de output print die je testcode verwacht, tot op hoofdletters en spaties toe, over drie afzonderlijke lijnen. Het is natuurlijk wel de bedoeling dat je de instantievariabelen gebruikt om de output te genereren.

Oefening 5: Vloeistof overgieten

Haal nu de laatste drie testen uit commentaar. Deze hebben te maken met het overgieten van vloeistof van één vat naar een ander. Analyseer de testen en kijk of je deze code kan implementeren. Enkele tips:

- Je hebt nu twee vaten nodig. Je hebt er al één (de barrel waar je de methode op opriep), maar waar komt de tweede vandaan? Hoe kan je informatie doorgeven aan een methode? Hoe heb je bijvoorbeeld hij het vullen of leeggieten van een barrel de hoeveelheid meegegeven? Een extra barrel doorgeven gebeurt op exact dezelfde manier.
- Wanneer we vloeistof willen overgieten van barrel A naar barrel B kunnen we niet meer vloeistof overgieten dan A bevat, en niet meer toevoegen dan waarvoor B ruimte heeft. Je hebt net twee methodes geschreven die exact dit doen. Gebruik deze methodes, en vermijd van je code gewoon te copy-pasten.
- Deze hele methode kan geschreven worden in één enkele lijn code! (maar als je het kan doen in een paar lijnen extra is het ook goed).

Als je deze methode verwarrend vindt, geen paniek! De komende twee weken zullen we focussen op communicatie tussen objecten, en dat is nu net wat hier gebeurt.

Oefening 6: Heater

In deze oefening maken we opnieuw een klasse gebaseerd op een gegeven testklasse (met UML), maar dit keer minder geleid. Je kan vertrekken van het `heater_start` project op Toledo. We geven hier een minimale probleemomschrijving, tracht ook de testcode te analyseren om te zien wat er verwacht wordt van je code en implementeer de nodige methoden om alles tests te laten slagen.

Probleemomschrijving



Elke heater (thermostaat) heft een bereik van temperaturen waarin hij opereert, gedefinieerd door een minimale en maximale temperatuur. Wanneer je de huidige temperatuur wil veranderen wordt de waarde telkens aangepast met een zekere stapgrootte, gedefinieerd door het attribuut `increment`. Definieer een constructor die je toelaat om al deze waarden te initialiseren. Zorg ervoor dat de minimale temperatuur hier altijd lager is dan de maximale, anders wissel je hun waarden. Als de initiële waarde van de temperatuur buiten het bereik `[minTemp, maxTemp]` ligt, vervang ze dan door het gemiddelde van de extremen.

Definieer een accessor die de waarde van temperatuur weergeeft. Definieer de mutators `warmer` en `cooler`, die als effect hebben dat de huidige temperatuur verhoogd of verlaagd wordt met de waarde van `increment`. Zorg ervoor dat deze twee methodes de temperatuur niet buiten het toegelaten bereik kunnen laten veranderen. Ten slotte moet het ook mogelijk zijn om de waarde van `increment`

te veranderen tijdens de levensduur van de heater, met de methode `setIncrement`. Deze methode checkt of `newIncr` een positieve waarde heeft, anders print ze de verwachte foutboodschap op het scherm (zie testklasse).

UML

Heater
-temperature: int -maxTemp: int -minTemp: int -increment: int
+Heater(int temp, int min, int max, int delta) +warmer(): void +cooler(): void +getTemperature(): int +getMinTemperature(): int +setMinTemperature(int newMin): void +getMaxTemperature(): int +setMaxTemperature(int newMax): void +getIncrement(): int +setIncrement(int newIncr): void

Thuisoefeningen

Oefening 1: Ticket Machine

Voor deze oefening moet je een nieuwe klasse schrijven *zonder* de hulp van een testklasse. Analyseer de probleemomschrijving en voer volgende stappen uit:

1. Ontwerp een UML voor je klasse, inclusief alle nodige instantievariabelen en methoden. Wees grondig: voorzie ook returntypes en parameters voor je methoden.
2. Vertrekkend van dit UML-diagram implementeer je je klasse in Java.

Een oefening op deze manier aanpakken voelt heel verschillend aan dan wanneer je een UML en/of testklasse gegeven hebt om je te gidsen. Het kan zijn dat bepaalde onderdelen van de opgave dubbelzinnig of niet geheel duidelijk zijn. Dit is normaal: wanneer een programmeur een stuk software moet schrijven vanaf nul, gebaseerd op de specificaties van een klant, dan moet deze programmeur zelf bepaalde beslissingen maken over de structuur en het gedrag van de code. Als je twijfelt over bepaalde van je beslissingen kan je altijd contact opnemen met je klant (in dit geval je labobegeleider).

Probleemomschrijving



Deze parkeerautomaat gebruikt een eenheidsprijs (eurocent) voor een bepaalde tijdseenheid (minute). Als je bv. een eenheidsprijs hebt van 20 eurocent voor een tijdseenheid van 15 minuten, dan betekent dit dat 1 uur parkeren 80 eurocent kost.

Wanneer je geld in de automaat stopt (voldoende, d.w.z. minstens zoveel als de eenheidsprijs), dan print de machine een ticket voor een periode die overeenkomt met de maximale tijdsduur die mogelijk is voor dat bedrag, namelijk een geheel aantal keer de tijdseenheid, met het teveel als wisselgeld. Een voorbeeld: als je met de gegevens van de vorige paragraaf, 90 eurocent in de automaat stopt, krijg je een ticket voor 1u (4x 15 minuten) en geeft de machine 10 eurocent wisselgeld terug.

Het moet mogelijk zijn om de eenheidsprijs en de tijdseenheid te veranderen. We willen ook de totale inkomsten van de automaat kunnen opvragen, en we willen de mogelijkheid hebben om de automaat leeg te halen (al het geld eruit te halen).

Oefening 2: MoneyBox

Voor deze oefening krijg je enkel een probleemomschrijving. Probeer om je eigen testklasse te schrijven om de MoneyBox-klasse te testen. Bij het schrijven van testklassen zijn programmeurs altijd geïnteresseerd in zogenaamde “edge cases”. Bijvoorbeeld: als je wil testen of een functie true returnt voor waarden 50 en hoger, en anders false, dan hoeft je niet elke integer te testen. Wat je bijvoorbeeld zou kunnen doen is waarden 5, 49, 50, 51 en 100 testen. Je kan een testklasse maken door te rechtklikken op een ‘gewone’ klasse en op ‘Create Test Class’ te klikken.

Merk op dat het zelf schrijven van testklassen niet zal worden geëvalueerd op het examen, je hoeft dit dus niet per se te kunnen. Het kan echter wel een goede oefening zijn, want je moet testklassen wel kunnen lezen, en als je ze kan schrijven zal dat geen probleem vormen.

Probleemomschrijving

Je kan geld deponeren in een kluis, en je kan zien hoeveel geld de kluis bevat. Om geld in te kluis te stoppen heb je geen toegangscode nodig. Maar als je wil opvragen hoeveel geld de kluis bevat, of je geld uit de kluis wil halen is dat wel het geval. Als je daar de verkeerde toegangscode ingeeft, krijg je een foutboodschap. Het moet ook mogelijk zijn om de code te veranderen. Implementeer de nodige controles zodat je geen onrealistische operaties kan uitvoeren met je kluis. Definieer een goed testscenario en implementeer dit in een aparte testklasse.

