Software Engineering

Assignment 4: Testing

3 Ba INF 2018-2019

Evelien Daems
Lars Van Roy

November 18, 2018

Calculate() function analysis

Control Flow Graph

Een control flow graph is een visuele weergave van een programma waarmee alle mogelijke paden doorheen een programma kunnen gegenereerd worden. Elke binaire node (een node waar twee pijlen uitkomen) komt overheen met een conditie in het programma. Elke mogelijke sequentie van nodes dat eindigt in een eindstaat is een mogelijk pad door het programma, het is echter niet gegarandeerd dat dit pad ook bestaat. Sommige condities kunnen enkel slagen indien andere condities ook slagen en of slagen sowieso x aantal keren. Het is dus niet omdat de sequentie bestaat in de control flow graph dat het ook een mogelijk pad is. Voor de flow graph van de calculate() functie zie appendix A.

Cyclomatic Complexity

De cyclomatic complexity kan berekend worden door het aantal edges - het aantal nodes te beschouwen + 2 of het aantal binaire conditie nodes + 1. Beide geven ons in dit geval 10. Deze waarde symboliseert een bovengrens voor het aantal mogelijke onafhankelijke paden door de flowgraph.
aantal edges - aantal nodes + 2 = 29 - 21 + 2 = 10
aantal binarie condities + 1 = 9 + 1 = 10

independent paths

Wanneer we beginnend van het kortst mogelijke pad verdergaan door steeds een nieuw onafhankelijke pad te nemen krijgen we een mogelijkheid om de verschillende paden voor te stellen. Een nieuw pad wordt onafhankelijk genoemd van de voorgaande paden als er in het nieuwe pad een node voorkomt die nog niet voorkwam in de voorgaande paden. Wanneer we dit doen bekomen we de volgende paden.

index	pad
0	1, 2, 6, 7, 21
1	1, 2, 6, 7, 8, 9, 10, 7, 21
2	1, 2, 3, 2, 6, 7, 8, 9, 10, 7, 21
3	1, 2, 3, 4, 5, 2, 6, 7, 21
4	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 10, 7, 21
5	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 12, 10, 7, 21
6	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 12, 10, 7, 21
7	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 12, 10, 7, 21
8	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 12, 10, 7, 21
9	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 12, 10, 12, 10, 7, 10, 12, 10, 7, 21

Table 1: alle mogelijke paden

test cases

Bij nadere observatie van deze paden zien we dat het merendeel hiervan niet bereikbaar is. sommige condities kunnen niet falen zonder dat andere in het programma ook falen en omgekeerd. Uiteindelijk blijven volgende 4 paden over met bijhorende input en output. Hierbij zijn triviale inputvelden, of m.a.w. inputvelden waarvoor de waarde niet relevant is om het gewenste pad te bekomen, weggelaten. Een volledig overzicht van alle paden en welke paden nu uiteindelijk een bestaand pad door de functie hadden zijn weergegeven in appendix B.

index	input	output
1	m_dbm_tidlist = {}	{{}, {}}
2	m_dbm_tidlist = {0:{0}}; min_sup = 2;	{{}, {}}
4	m_dbm_tidlist = {0:{0}}; min_sup = 1;	{{[0]}, {}}
9	m_dbm_tidlist = {0:{1,2}, 1:{1}}; min_sup = 1;	{{[0], [1]}, {[0, 1]}, {}}

Table 2: test cases

JUnit testing

unit tests

Unit tests zijn testen waarbij kleine stukjes code apart worden getest zonder afhankelijkheden etc. Door dit te doen kunnen we specifiek zoeken naar fouten in de implementatie binnen een unit. Voor elke unit worden er meerdere testcases doorlopen. Voor het core.Clients packet onderscheiden we 6 verschillende units, zijnde: Address, Client, Mailing_Preferences, RegisteredClient, UnregisteredClient en Clients.

Voor elk van deze units schrijven we voor elke methode n of meerdere testen waarbij we de verwachte output vergelijken met de eigenlijke output.

Dit lukt voor 3 van de 6 units maar bij de 3 andere zijn er assertions die foutieve returns opvangen.

In de unit Client faalt de methode getRegisteredClient(string, string) om de gewenste return te geven. Deze zou, indien er bestaande data wordt opgegeven, een referentie moeten geven naar het gerefereerde Client object maar dit doet hij niet, hij returned null onafhankelijk van de gegeven input.

In de unit Mailing_preferences faalt de functie third_party() in sommige gevallen. Wanneer de constructor input gelijk is aan (true, true, true) is er geen probleem maar wanneer de constructor input gelijk is aan (true, true, false) geeft de funtie third_party() plots ook true terug. Wanneer de constructor input krijgt gelijk aan (false, false, false) werkt alles wel weer normaal.

Ten slotte heeft de unit RegisteredClient nog een probleem in de functie checkPassword(string). Deze functie zou true moeten return indien het correcte wachtwoord wordt meegegeven als string maar dit is niet het geval.

integration tests

Integration tests zijn testen waarbij meerdere units samen getest worden. Door dit te doen kunnen we zoeken naar fouten in afhankelijkheden in units. Indien alle unittesten slagen weten we dat er niet zozeer problemen zijn in de units op zich maar dat er fouten zijn in het gebruik van de methoden van andere units.

Deze test bestaat in essentie uit hetzelfde als de unit tests. We maken opnieuw gebruik van verschillende methodes maar in plaats van enkel methodes van 1 unit te beschouwen beschouwen we meerdere methoden uit meerdere units en het effect dat ze op andere methoden kunnen hebben.

In de beschouwde integration test worden de calls naar methoden uit core.Clients in het pakket Core onderzocht. Deze methoden worden voor zover de test reikt goed gebruikt en de teruggegeven waarden komen steeds overeen met de verwachte output.

synopsis voor testen

Testen zijn nooit volledig, er zijn steeds methoden die voor sommige situaties wel slagen en voor andere niet. Het is zeer moeilijk om af te wegen wanneer er genoeg testen geschreven zijn. De bedoeling is niet dat het uren duurt om deze testen uit te voeren voor methodes die geen vitale belangen hebben in het verloop van het programma en al evenmin dat er maar een paar testen worden geschreven voor functionaliteiten waarop het hele programma gebaseerd is. Het is dus belangrijk om hier een afweging in te maken.

Ook al zijn niet altijd alle situaties beschouwd, door een percentage van alle mogelijke situaties te beschouwen en hierop extensief te testen beperken we wel het risico op fouten. Er is namelijk een deel van de situaties die zich kunnen voordoen die getest zijn en waarin niets meer zou mogen vastlopen.

Ten slotte is het ook belangrijk om zowel unit tests als integration tests te gebruiken. Unit tests zijn zeer handig om fouten binnen units op te zoeken maar deze zijn niet toereikend over het gehele systeem. Fouten die ontstaan door combinaties van units zijn veel voorkomend en moeilijker te vinden dan fouten in units zelf dus is het ook belangrijk om integration tests te schrijven.

Design by contract

In het algemeen vereisen we bij elke functionaliteit dat er geen operatie kan worden verricht op null objecten en dat de teruggegeven waarden eveneens geen null object kunnen bevatten. Hieronder geven we een kort overzicht van contracten die wat meer context hebben.

Class Catalog
@requires({"item!=null", "$this.findMatch(item) == null"})
@ensures("$old($this.getNumberOfItems())+1==$this.getNumberOfItems()")
public void addItem(Item item) {...}
Voor het toevoegen van een item aan de catalogus wordt er door de klant evreist dat alle items uniek moeten zijn op basis van hun naam, beschrijving en prijs. Dit moet eerst worden gecontroleerd voordat we het item gaan toevoegen aan de catalogus. Nadien moet men verifiëren of het item met succes is toegevoegd aan de catalogus. Anders zou de gebruiker er vanuitgaan dat dit item is toegevoegd en dit item gaan oproepen in een ander deel van het systeem om dan pas te ontdekken dat dit item nooit toegevoegd is geweest.

@requires({"category!=null"})
@ensures({"$result != null"})
public ArrayList<Item> filterCategories(HashSet<String> categories){...}
De klant vereist in de extra specificaties dat het systeem in staat moet zijn om te filteren op de juiste categorie. Hiervoor is een functionaliteit voorzien die als input een category of meerdere category vraagt. Het is van belang dat we controleren dat dit een geldige set is en dus geen leeg object voorstelt. Als teruggegeven waarde moet er altijd een lijst teruggegeven worden met de gevonden items. Maar deze lijst kan nooit een null object zijn want zelfs al zijn er geen items gevonden, er zal dan simpelweg een lege lijst worden verwacht.

@requires({"item!=null"})
public Item findMatch(Item item){...}
Een eigen geschreven functionaliteit om aan de specificatie te voldoen waarin beschreven staat dat een Item uniek moet zijn in de zin van een unieke naam, beschrijving en prijs in de catalogus. Hier is het wel toegestaan om een return waarde te hebben van null aangezien dit betekent dat geen enkel item werd gevonden dat overeenkwam met de input.

Class Cart

@requires({"quantity>=0", "item!=null"})
@ensures("$old($this.contents().get(item))+quantity==$this.contents().get(item)")
public void addItem(Item item, int quantity){...}

Bij het toevoegen van een item aan een Cart object is het belangrijk dat het item een bestaand en geïnstantieerd object is. Want voor de gebruiker heeft het geen zin dat er een ijdel object wordt toegevoegd aan het winkelwagentje. Dit zou eveneens problemen kunnen geven bij verdere bewerkingen op Cart en dus onderweg voor errors kunnen zorgen. De vereiste dat een posititef aantal moet worden toegevoegd, is natuurlijk triviaal want het is onrealistisch dat een gebruiker een negatief aantal items in zijn of haar winkelwagen wil.
Na het toevoegen van een item moet er gegarandeerd worden dat er niets is mis gegaan tijdens het toevoegen. We zullen dus eisen dat de oude winkelwagen is geupdate met het gevraagde item en bijhorende hoeveelheid.

@requires({"$old($this.contents().containsKey(item)==true", "item!=null", "quantity>=0"})
@ensures("$this.contents().get(item)==$old($this.contents().get(item))-quantity")
public void removeItem(Item item, int quantity) {...}

Voor een item kan verwijderd worden is het belangrijk te controleren of dit ook effectief aanwezig is in de content van het Cart object. Quantity moet een positief getal zijn om een correct verloop van de removeItem() functionaliteit te garanderen. Indien een gebruiker toevallig een negatief getal zou toekennen aan de parameter dan voldoet het resultaat van de functie niet meer aan de verwachtingen die men heeft bij het uitvoeren. Dan zou het verwijderen resulteren in het toevoegen van items. Idem aan de functie addItem() eisen we de garantie dat een item effecief verminderd is in hoeveelheid met aantal quantity.

@requires({"$this.contents()!=null"})
@ensures({"$result>=0", "$result!=null"})
private float getCostFloat(){...}

Deze functie hebben we zelf toegevoegd om de totale waarde van de winkelwagen in float notatie te berekenen. De contents van het object moet geïnitialiseerd zijn om de kost te kunnen berekenen en het resultaat kan niet negatief zijn. Indien dit wel het geval is zou de webshop de klant gaan betalen om producten te kopen, wat helemaal niet de bedoeling is. Het resultaat moet ook bestaan indien alles goed is verlopen.

@ensures({"$result.length()!=0"})
public String getCost() {...}

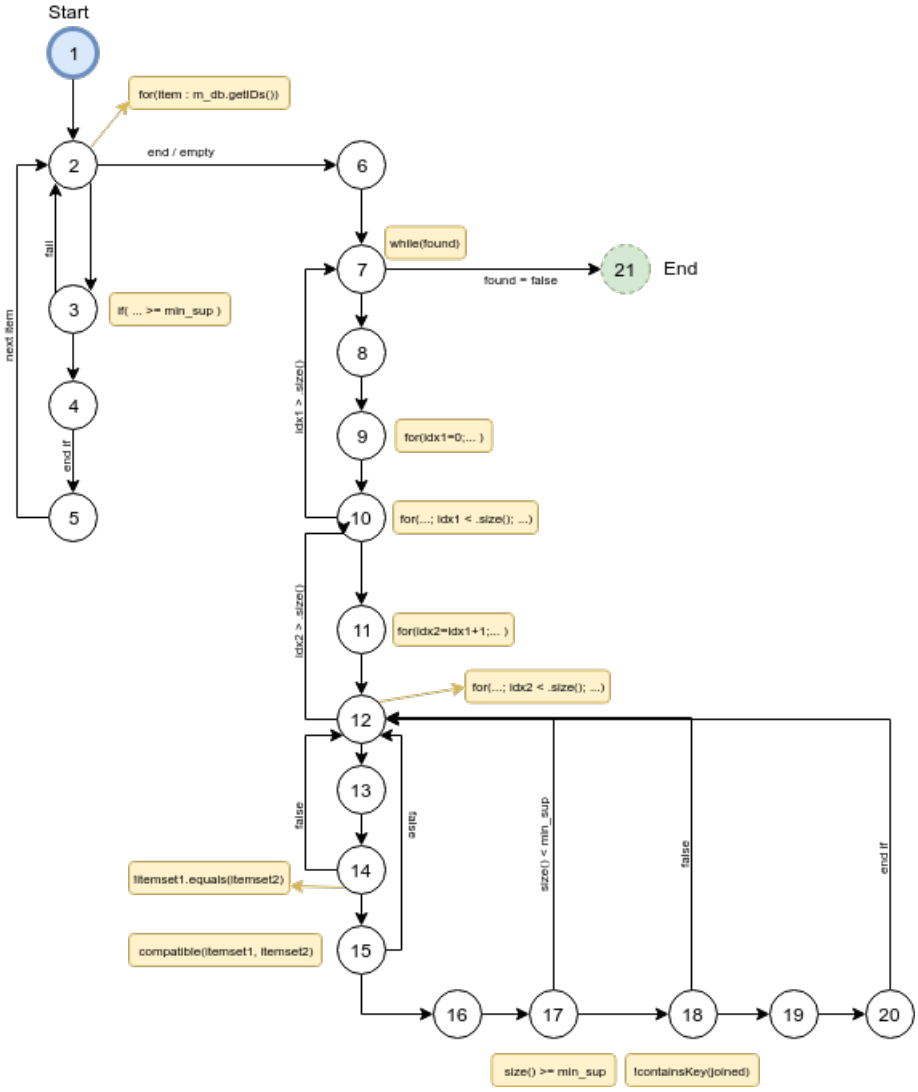
Dit is de functionalieit die de totale prijs van de winkelwagen zal teruggeven in tekst formaat. Als gebruiker verwachten we een geldige return waarde die nooit leeg kan zijn als alles goed is verlopen en met leeg bedoelen we een string van lengte nul. Een string zal nooit null kunnen zijn.

Class Category
@ensures("$this.getCategories().length()==10")
public Category(){...}
De Catalog moet verplicht standaard 10 categoriën bevatten die staan opgesomd in de specificaties van de klant. Dus zorgen voor een postconditie die kijkt of bij het aanmaken van de Catalog de lijst ook effectief bestaat uit deze categoriën. Het is niet nodig om per categorie te gaan kijken of er wel de juiste in staan want dit zit hardcoded in het programma. Het is enkel nodig om te kijken of de lijst niet leeg is en wat er dus op zou kunnen duiden dat er iets is misgegaan bij het aanmaken van het object.

@requires({"category.length()!=0", "!$this.getCategories().contains(category)"})
@ensures({"$this.getCategories().contains(category)==true", "$old($this.getCategories().size())+1==$this.ge
public void addCategory(String category){...}
Voor het toevoegen van een categorie toe te laten, willen we eerst kijken of categorie geen lege string is, want het is onmogelijk om een lege categorie te hebben. Ten tweede moeten alle categoriën uniek zijn en controleren we of de in te voegen string als geen deel uitmaakt van de categoriën.

Class Item
@requires({"price > 0"})
public Item(String name, String desc, float price) {...}
De prijs van een item in de catalogus moet een niet-nul positief getal zijn volgens de specificaties.

appendix A


Start
1
for(Item : m_db.getIDs())
2
end / empty
6
fall
7
while(found)
21
End
found = false
3
if(... >= min_sup)
next item
4
8
idx1 > .size()
9
for(idx1=0;...)
end if
5
10
for(...; idx1 < .size(); ...)
idx2 > .size()
11
for(idx2=idx1+1;...)
for(...; idx2 < .size(); ...)
12
13
false
false
size() < min_sup
false
end if
14
itemset1.equals(itemset2)
compatible(itemset1, itemset2)
15
16
17
18
19
20
size() >= min_sup
!containsKey(joined)

appendix B

index	pad	input	output
0	1, 2, 6, 7, 21	bestaat niet	bestaat niet
1	1, 2, 6, 7, 8, 9, 10, 7, 21	m_dbm_tidlist = {}	{{}, {}}
2	1, 2, 3, 2, 6, 7, 8, 9, 10, 7, 21	m_dbm_tidlist = {0:{0}}; min_sup = 2;	{{[0]}, {}}
3	1, 2, 3, 4, 5, 2, 6, 7, 21	bestaat niet	bestaat niet
4	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 10, 7, 21	m_dbm_tidlist = {0:{0}}; min_sup = 1;	{{[0]}, {}}
5	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 12, 10, 7, 21	bestaat niet	bestaat niet
6	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 12, 10, 7, 21	bestaat niet	bestaat niet
7	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 12, 10, 7, 21	bestaat niet	bestaat niet
8	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 12, 10, 7, 21	bestaat niet	bestaat niet
9	1, 2, 3, 4, 5, 2, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 12, 10, 12, 10, 7, 10, 12, 10, 7, 21	bestaat niet	bestaat niet

Table 3: paden + test cases