LINKÖPINGS UNIVERSITET

TDP007: Konstruktion av datorspråk

Utvecklarblogg
Seminarieuppgift 4

Författare

Daniel Huber, danhu849@student.liu.se
Theodore Nilsson, theni230@student.liu.se

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2021
Version 1.0

2021-02-19

Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Utvecklarblogg, Seminarie 4 TDP007	190221

Uppgift 1: Att använda constraint-nätverk

Tolkning

Efter översiktliga tester blev det väldigt tydligt att AritmethicConstraint saknade funktionalitet. Även om det inte stod något specifikt i uppgiftsspecifikationen blev det ändå klart att `new_value` behövde uttökas och `celsius2fahrenheit` implementeras för att klara uppgiften.

Metod

Det hittades snabbt att alla variationer av blanka värden i `new_value` inte täcktes och efter att ha kopierat första instansen (rad 167-174) och fyllt i det som saknades i de andra två (rad 176-187) så fungerade `Adder` och `Multi` som specificerat. Detta testades först direkt i irb, sedan i testfunktioner såsom `test_multi` som sedan skrevs över till testfilen. Testfilen skrevs dock först när `celsius2fahrenheit` implementerats.

I testfilen `test_cn.rb` testas både addition och subtraktion med både positiva och negativa tal.

Reflektion

Uppgift 1 löstes så snabbt att vi trodde att vi gjort fel eller missat något. Detta störde utvecklingen i efterkommande uppgift då vi hela tiden tvekade på om vi löst den rätt. Det kändes bra och som vi var på rätt spår efter att ha löst denna då det gick verkligen supersnabbt!

Uppgift 2: Att generera constraint-nätverk

Tolkning

Istället för att manuellt skapa nätverken som i uppgift 1 så skulle en sträng parsas för att skapa ett nätverk som kunde lösa en ekvation. Vi tolkade att uppgiften utgick från primärt kunna parsa strängen `"9*c=5*(f-32)"`.

Metod

Uppgiftens beskrivning lämnade få ledtrådar till vad som skulle kunna vara fel med koden. Det var svårt att förstå hur det var tänkt att koden skulle fungera av att läsa den dokumentation som fanns, detta ledde till att vi fick lov att prova oss fram och jobba utifrån flera hypoteser. Det första vi provade var att försöka följa ledtrådarna som vi fick av de felmeddelanden som visades vid körning för att få ett program som kunde köras på ett enklare uttryck.

get_connector

Det första vi såg var att det saknades en metod som hette `get_connector`. Vi fick av namnet att döma anta att det var tänkt att den skulle returnera en `Connector` som tillhörde det object som metoden användes på. Vi såg att den kunde komma att användas av både object som var `Connector`s, och object av typen `ArithmeticConstraint`. Vi valde att returnera `Connector` oförändrat, och i de fall det var en `ArithmeticConstraint` valde vi att returnera den `Connector` som hette `out` eftersom den särskilde sig från de andra två, `connector_a` och `connector_b`.

replace_conn

När `get_connector` metoden hade skapats kunde vi se att det uppstod problem när två `Connector`s passades in i funktionen `replace_conn`.

Eftersom detta var en funktion som endast anropades när parsern hittat ett ekvivalenstecken förstod vi att det måste ha någonting att göra med att sammanföra två nätverk.

Vi såg också att funktionen orsakade en krasch då två `Connectors` passades in eftersom de fallet inte fångas upp av den existerande styrstrukturen. Detta löstes genom att skapa en fusk-`ConstantConnector` (rad 434, `replace_conn`) med värdet noll som kopplas till en `Adder`:

```
1     nil_connector = ConstantConnector.new("0", 0)
2     exp = Adder.new(nil_connector, lh, rh)
```

Resonemanget var att det motsvarade: a=b <=> a+0=b.

get_connectors

Allting blev mycket lättare när det efter föreläsning nämndes att `get_connectors` även kunde få fel datatyp till sig. Eftersom vi redan skapat en metod, `get_connector` för att göra om `ArithmeticConstraints` till `Connector` kunde vi använda den i `get_connectors` för att lösa problemet. Plötsligt gick det bra att parsa och göra korrekt uträkning av det givna uttrycket.

Reflektion

Då vi inte hade förståelsen för hur programmet borde se ut för att lösa den givna uppgiften tog det mycket längre tid än väntat. Initialt försöktes uppgiften lösas på det svåra sättet utan vår kännedom förrän i slutet. Vi kunde sparat tid om vi börjat tidigare med testdriven utveckling för att säkerställa att tidigare funktionalitet även kvarstod senare. Vilket inte skedde under majoriteten av utvecklingstiden. En komplikation av detta blev att testkod och `puts` skrevs in i alla funktioner så till den grad vi inte visste vad vi letade efter eller varför. På grund av detta blev vi tvungna att ta nya kopior av uppgiftfilerna och helt eller delvis börja om från början flertalet gånger.

Vi kunde också ansträngt oss mer för att förstå hur koden fungerade istället för att bara använda felmeddelanden som vår enda guide. Anledningen till att vi inte gjort detta så bra som vi borde var att koden var visuellt avskräckande, och även delvis därför att kommentarerna var kryptiska. Ytterligare en bidragande faktor var att uppgift 1 var för lätt så vi hade en oro om att vi kanske hade missat något i den som gjorde att det inte funkade i uppgift 2.

Testdriven utveckling

Vid en period i utvecklingen så kunde vi inte skapa ny funktionalitet utan att förstöra tidigare. I och med att vi hade dålig koll på vad vi ändrat under utvecklingens gång bestämde vi oss att det var hög tid att skapa tester för det nuvarande programmet och även tester för framtida planerad funktionalitet, s.k blackboxtestning. Det gjorde det lättare att jobba mer strukturerat och dessutom snabbare kunna kontrollera ifall de ändringar som gjorts genererat oönskad eller önskad effekt. Problematiken att börja med testdriven utveckling mitt i en uppgift är att funktionalitet utöver

Misstagen kan summeras i att vi inte programmerade testdrivet, vi försökte lösa uppgiften på det svåra sättet (utan dedikerade `ArithmeticConstraints` för division och subtraktion) samt att vi bara satte oss in i koden översiktligt. Det tog lång tid att inse hur `get_connector` skulle skrivas in i `replace_conn` och `get_connectors`.