LINKÖPINGS UNIVERSITET

TDP007: Konstruktion av datorspråk

Utvecklarblogg
Seminarieuppgift 3

Författare

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

INSTITUTIONEN FÖR DATAVETENSKAP

Vårterminen 2021
Version 1.0

2021-02-02

Revisionshistorik

Ver.	Revisionsbeskrivning	Datum
1.0	Utvecklarblogg, Seminarie 3 TDP007	020221

Uppgift 1: Domänspecifika språk

Tolkning

Först förstods inte skillnaden mellan att skriva en DSL för datan respektive plocka ut den med regex. Fördelen som stötts på över regex är att DSL än så länge inte nyttjat regex. Det troddes först att datan var tvungen att representeras på ett speciellt sätt. Efter åtskilliga tester visade det sig inte vara fallet. Flertaliga diskussioner har förts om kommaseparerade argument eller ett nyckel-värde.

Andra diskussioner har hållits gällande om allting skulle skötas med hjälp av `missing_method` eller om nya metoder och `accessors` skulle skapas med hjälp av `class_eval` respektive `instance_eval` under runtime. Det tolkades till slut att `missing_method` skulle implementeras på ett sånt generellt sätt att all funktionalitet sköttes med hjälp av den.

Det tolkades att `policy.rb` fick se ut hur som helst och därför har ytterligare data än den datan som tilldelades lagts till. Vi tolkade också att exemplet för inmatningen av data var ett visningsexempel så vår inmatning baseras på att all indata är av klassen `String`.

Metod

Först undersöktes möjligheten att ha värden i policy.rb på enskilda rader men då funktionen av `method_missing` missförstods ändrades `policy.rb` filen till att ha ett funktionsanrop av respektive kategori med tillhörande argument åtskilda av kommatecken. Inmatningsvärdena i klassen `Person` sparades först var för sig i medlemsvariablerna av samma namn:

```
1    @car, @zip, @d_license, @sex, @age = car, zip, d_license, sex, age
```

För att sedan sparas i en array och låta `Person` ärva klassen `Array`, men det ändrades igen då matchningen av användardatan skulle bli tvungen att itereras över vilket lett till onödiga funktionsanrop:

```
1    @categories = [car, zip, d_license, sex, age]
```

För att slutligen sparas i en hash. Med hjälp av hash möjliggörs en väldigt snabb data åtkomst.

```
1    @categories = { "Bilmärke" => car , "Postnummer" => zip , "Körkort" => d_license , "Kön" =>
     sex , "Ålder" => age }
```

I första versionen sparades all data från `policy.rb` varje gång ner i `Person` objektet i en stor hash. Då detta ansågs onödigt ändrades fokus till att på ett enkelt sätt med hjälp av `missing_method`.

Vi ville att method `method_missing` på ett enkelt sätt kunde matchas mot datan i `policy.rb` och att `Person` objektet tog upp så lite redundant data som möjligt. Utöver detta ville vi göra variation i `policy.rb` möjlig så det spelar ingen roll i vilken ordning datan ligger så länge formatet nedan följs.

```
1    Kategori  "sak_i_kategori" => försäkringspoäng
```

Det tolkades att `policy.rb` fick se ut hur som helst och därför har ytterligare data än den datan som tilldelades lagts till. Vi tolkade också att exemplet för inmatningen av data var ett visningsexempel så vår inmatning baseras på att all indata är av klassen `String`.

Nu i efterhand kan såg vi att vi kunde ha skrivit blackboxtester, men då inmatningen förändrades markant över tid valde vi att avvakta med tester tills dess att grundfunktionalitet fanns.

Reflektion

Vi fann det otydligt vad som var skillnaden på DSL och vanlig strängjämförelse då det är lätt att läsa in varje rad från `policy.rb`, köra split på raden och jämföra delarna mot `Persons` initieringslista. Först kändes det inte som scopet eller problemet var tillräckligt stort för att riktigt känna nyttan med DSL, men efter att ha skrivit if-satser för modifiering av `@sum` i `person.rb` så blev vi instruerade att det kanske inte var tanken så logiken flyttades ut till `policy.rb`. Koden i `person.rb` blev lättare att hantera och funktioner för enkel och mer komplicerad jämförelse nu kan skrivas i `policy.rb`.

Uppgift 2: Parsning

Tolkning

Vi tolkade uppgiften som att vi skulle skapa ett språk för att hantera logiska uttryck på det sätt som specificerades av den av seminariet givna BNF:en. Språket definierades så att en variabel kunde vara ett godtyckligt ord som inte var ett av nyckelorden or, and, set, true eller false.

Det var svårt att veta hur mycket som kunde anpassas av BNF:en för att få bra funktionalitet av variabler. Vi landade slutligen i att använda godtyckliga variabelnamn vid tilldelning, och ett $-tecken för att få ut värdet sparat i en variabel.

Metod

För att skapa språket använde vi uppgiftens givna fil som grund och skrev om dice-roller klassens initialize för att matcha den givna BNF:en och bytte variabelnamn till någonting mer passande för uppgiften.

För att tillämpa den givna BRF:en var det hyfsat trivialt att ersätta de existerande `rule` uttrycken med nya regler som motsvarade de som stod i uppgiften. Vi lade sedan till ännu en definition för hur en variabel skulle se ut.

För att kunna testa användningen av DSL:en gjorde vi ändringar i medlemsfunktionen `count` så den kunde anropas med en parameter vars defaultvärde sattes till `standard-in` för att bevara det ursprungliga användningssättet.

Reflektion

Saker som var svåra: tillämpning av variabel tillskrivning, tillämpning av variabel överskrivning. Trots flertaliga försök kunde vi inte få tilldelningen att skriva över värdet för variabler som redan tilldelats värden. Det var också väldigt otydligt hur scopet spelade roll och instansvariabler som skapats utanför `@logicParser` kunde ej kommas åt inuti `@logicParser`.

De delar som inte ingår i klassen `Logic` har vi bortabstraherat och fokus har legat på att förstå hur klassen `Logic` kan användas för att skapa vår DSL.

Testning gjordes genom att dirigera om utskrifter till en variabel som kontrollerades av testet. Testerna gjordes under förutsättning att endast det slutgiltiga svaret skrevs ut. Detta visade sig sedan inte vara fallet vilket framgick av ytterligare test som kontrollerade detta. Lärdomen av detta blir att det är fördelaktigt att testa de antaganden man gjort vid testning. Det ledde också till att testerna kunde skrivas om korrekt.