Københavns Universitet

Softwareudvikling - Deliverable 3

Af
Herluf Baggesen
Mads Buchmann
Philip Girelli

April 2016

1 Implementering af opgaver

Dette afsnit beskriver hvordan planlagte user-stories fra den forrige iteration, såvel som user-stories der er opstået i løbet af denne iteration, er blevet implementeret. Selve organiseringen af hvem der laver hvad, og hvad der mangler at blive lavet, er foregået på samme måde som i tidligere iterationer. For at sikre at alle gruppemedlemmer til hver en tid har en opdateret plan over iterationens indhold, er alle user-stories blevet skrevet ind på gruppens github repository som issues. Det har herefter været muligt for hvert gruppemedlem at blive sat på som ansvarlig for en user-story, hvilket har resulteret i at alle altid kan være opdateret med hvad der sker. Ydermere har det grundet githubs issue system også tilladt at alle gruppemedlemmer kan komme med relevante kommentare og respons til en given user-story på github.

Systemet blev ikke helt anvendt hen mod slutningen af denne iteration, grundet at ansvarsområder endte med at overlappe hinanden, og alt hvad der blev lavet ikke nødvendigvis kunne deles op i user-stories men derimod var generel bugfixing og rettelser af fejl.

For at skrive mest mulig kode med mindst mulige fejl er pair-programming blevet brugt når der skulle skrives kode i denne iteration. Dog blev der hen i mod slutningen afviget fra dette grundet tidspres. Men det har resulteret i at der er blevet gjort overvejelser omkring implementation af designet, og selve designet, som ellers ikke ville have været blevet gjort. Ligeledes har en stor del af fokuset for denne iteration været på at lave et design som lever op til SOLID-principperne, og implementere dette. Tanker omkring designet vil kunne læses i afsnit 2 "programdesign".

Alle de user-stories som blev planglagt til denne iteration er blevet implementeret. Dog er det i retrospekt opdaget at de angivne user-stories til denne iteration ikke var optimalt formuleret. De lød som følger (uden underpunkter):

- Scraperen skal kunne indhente information fra en given webside.
- Scraperen skal kunne tilgå databasen
- Scraperen skal kunne tilpasses til indsamling af information fra andre websider

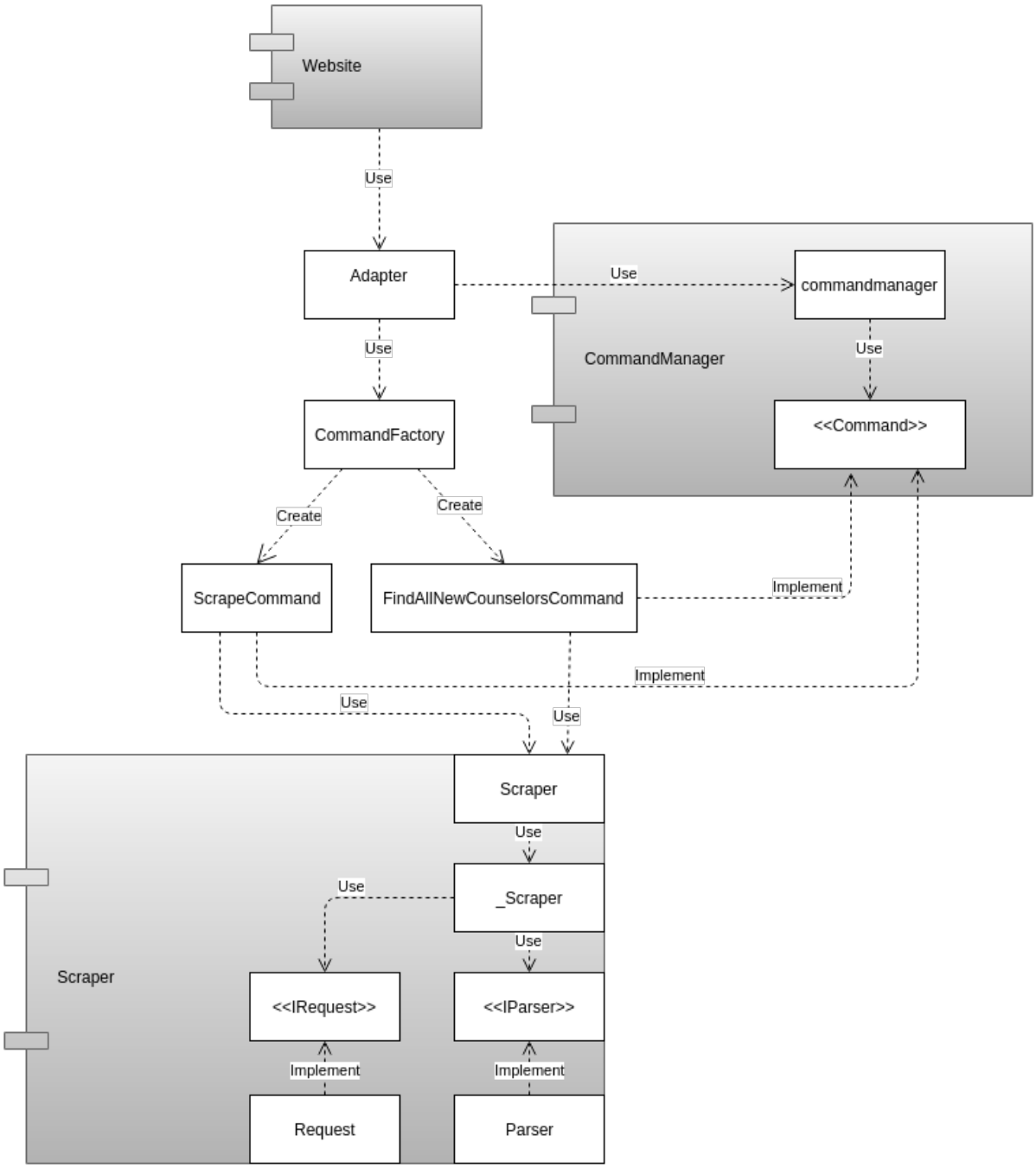
De forgående afsnit omhandler implementations detaljer og design overvejelser. User-stories burde være orienteret omkring brugerne af et system og hvad de kan anvende systemet til. Dette har i mild grad forvoldt problemer i gruppen hvad angår planlægning. Gruppen har desuden arbejdet på funktionalitet som ligger udenfor overstående user-stories. Bla. at en administrator skal være i stand til at planlægge automatiserede opdateringer af vejledere, såvel som at en administrator skal kunne bruge systemet til at oprette nye vejledere i systemet, baseret på KU's liste over ansatte, ved hjælp af et grafisk interface.

En user-story med dette i fokus burde have været omdrejningspunktet for denne iteration, i stedet for specifikke design detaljer for et modul kaldet scraper. Dette mål er ikke blevet nået endnu og vil blive et fremadrettet fokus. Der vil blive gået yderligere i detaljer omkring dette i afsnit 5 "Planlægning af næste iteration".

2 Programdesign

I dette afsnit vil den nuværende struktur for programmet blive beskrevet og vist. Herunder hører der en beskrivelse af programmets moduler samt et diagram der gerne skulle skabe et samlet overblik over afdelinger og funktionalitet. Først vil diagrammerne blive vist.

2.1 Diagram og UML



Figur 1: Diagram over gruppens samlede projekt.

Design filosofien bag denne opbygning bygger på høj modularitet, i overensstemmelse med Agile designprincipper, som gerne skulle gøre det simpelt og enkelt at opdatere, udskifte eller fjerne elementer af programmet. Hertil er koden skrevet så eksplicit som mulig i håb om, at gøre det nemt at læse og forstå, ved første øjekast. Et specifikt design princip der har været skabelonen for alle klasser er Single-Responsebillity Principle (SRP) fra Agile (Martin & Martin, 2007) som dikterer, at en klasse kun skal have en grund til at skulle ændres. Klassen må altså kun have en afhængighed (dependency) hvilket skal være med til og sikre, at der ingen cykler opstår hvor man ender i et vivar af afhængigheder på afhængigheder.

Den grundæggende struktur for projektet udgøres af tre forholdsvis store moduler som har betegnelserne: Django, CommandManger og Scraper. Django modulet indeholder selve web-frameworket samt en adapter der sørger for kommunikation mellem Django og CommandManager. CommandManageren sørger for håndtering af kommandoer sendt fra Adapteren til Scraperen, som så tilgår en given hjemmeside, og henter den ønskede information.

2.2 Website

Dette modul indeholder gruppens web-framework (Django) samt adapteren.
Adapteren, som ligger mellem Django-frameworket og CommandManageren, har først og fremmest til formål at oversætte information fra Scaperen til en Django forståelig størrelse[1].

Adapteren er en del af det system, som gør det muligt for Django-frameworket og Scraperen, at tale sammen. Når der f.eks. trykkes på "Database updating here!"under /admin/ panelet på hjemmesiden fortages en forespørgsel til klassen `CommandFactory` der så opretter to kommandoer: FindNewCounselorsCommand og Sc. De bliver implementeret i interface som ligger i Commands for så og blive sendt videre det Commandmanageren.

2.3 CommandManager

Dette seperate modul ligger som sagt mellem klassen CommandFactory i Adapteren og Scraper interfacet. CommandManageren er sat ind for at øge alsidigheden af vores program samt overholde den tidligere omtalte modularitet. I Agile kapitel 21 (Martin & Martin, 2007) nævnes begrebet "command pattern"som et af design mønstrene og det er, i store træk, den struktur som gruppens CommandManageren er bygget på. Med dette design mønster kan Adapteren og Scraperen eksisterer uden de behøver at kende til hinanden, og der opnås en smidighed i hele projektet i blandt. Med denne smidighed bliver det nemmere at skifte f.eks. Adaptere eller Scaperen ud.

Selve *Command-pattern*-designmønsteret har til formål og indkapsle en andmodning som et objekt og derved skille eksekverbare opgaver ad. En admodning bliver altså pakket ind i et objekt og kan derefter gives videre til det passende objekt der kan håndtere den ønskede kommando, hvorefter den eksekveres. Derved muliggøres det at sætte rutiner i kø, som f.eks. er blevet gjort i `commandmanager` klassen for at eksekvere dem på angivne tidspunkter.

Dertil bliver det også nemmere at holde styr på hvad der er i eksekverings køen samt hvad der blev eksekveret og hvornår.

2.4 Scraper

Ved udarbejdningen af Scraperen var det i fokus holde at modulariteten og brugervenlighed for øje. Dette blev gjort ved at skabe et interface som er hvad programmer uden for Scaperen for lov til at interagere med. Derved er det udelukkende Scraperen selv der kender til dens indre mekanik og man sikre en modularitet.

Fælles for alle tre moduler er altså en grundlæggende modularitet, som er med til og sikre, forholdsvis, nemme rettelser og updateringer af modulerne. Designmønstrene som gruppens design er bygget på, er pensum fra Code Complete (McConnell) og Agile (Martin & Martin, 2007).

[1] Bemærk dog at selve filen, som indeholder adapterkoden, ligger i Django projektmappen

3 Kodning

De intressante aspekter af kodningen i denne iteration vil i dette afsnit blive gennemgået. Først vil en række kodeeksempler blive gennemgået, med henblik på at give en forståelse af den faktiske implementation af projektet. Derefter vil en række kodekonventioner blive defineret, hvor af nogle er blevet delvist anvendt i denne iteration, men ellers er formålet for disse hovedsageligt at fungere som en retningslinje for fremtidige omformateringer af koden.

3.1 Kodeeksempler

I dette afsnit vil et udsnit af projektets implementation af *Command-pattern*-designmønsteret blive gennemgået. Koden kan findes via stien *Product/CommandManager/commandmanager.py* på projektets GitHub side. Designmønsteret er beskrevet i afsnit 2.3 ovenfor.

`CommandManager` klassens opgave er at eksekvere kommandoer på det rigtige tidspunkt. Hver kommando indeholder det tidspunkt som denne skal eksekveres på. `CommandManager`en skal derfor blot vedligeholde en kø af kommandoer, sorteret efter tid indtil eksekveringstidspunkt stigende, og vente indtil den forreste kommando skal eksekveres.

```
class CommandManager():
    def __init__(self):
        self.commandQueue = []
        self.myTimer = Timer(0,self.main_loop)
```

Figur 2: Klasse deklaration og konstruktør

Ovenfor i figur (2) ses `CommandManager` klassens deklaration og konstruktør. Konstruktøren skrives i Python som en metode ved navn `__init__`. Denne metode vil Python herefter eksekvere automatisk når et objekt instanstieres. `CommandManager` klassens konstruktør skaber to variabler: en tom liste der skal fungere som en kommando-kø og en instans af `Timer` klassen fra Threading modulet der kommer som en del af Python 3.X. Mere om denne senere.

```
def main_loop(self):
    if len(self.commandQueue) > 0:
        self.execute_command()
        if len(self.commandQueue) > 0:
            self.create_new_timer()
```

Figur 3: Mainloopet i CommandManager klassen

Figur (3) viser `CommandManager`ens `main_loop` metode. Dette indeholder klassens hovedlogik. Linje for linje fra toppen lyder logikken således: Hvis længden af kommandokøen er større end nul, eksekver den forreste kommando og hvis der er stadig er flere kommandoer skab et ny timer objekt. Eller mere generelt: Eksekver en kommando og vent indtil den næste skal eksekveres.

```
def create_new_timer(self):
    sleepTime = max(0,(self.commandQueue[0].executionTime - datetime.now()).total_seconds())
    self.myTimer = Timer(sleepTime, self.main_loop)
    self.myTimer.start()

def execute_command(self):
    cmd = self.commandQueue[0]
    cmd.execute()
    self.commandQueue.remove(cmd)
```

Figur 4: Metoder til at skabe et nyt Timer objekt og til at eksekvere et Command object

Den første metode vist i figur (4) er `create_new_timer`. Denne metode skaber `Timer` objekter. En `Timer` kræver to argumenter for at blive instanstieret: antal sekunder der skal ventes, og hvilken funktion eller metode der skal kaldes efter endt ventetid. Bemærk at sidstnævnte i dette tilfælde er en højreordens reference til den tidligere beskrevne `main_loop` metode og ventetiden altid er antal sekunder indtil den kommando forrest i kommandokøen skal eksekveres. For at undgå negative ventetider fastholdes `sleepTime`, som udregnes i linje 2 ovenfor, over nul.

Den anden metode vist i figur (4) er `execute_command`. Denne metode er forholdsvis simpel, den tager den første kommando i kommandokøen, eksekvere den og fjerner den fra køen.

```
def enqueue_command(self, newCommand):
    newQueue = self.commandQueue
    for (i,cmd) in enumerate(newQueue):
        if newCommand.executionTime < cmd.executionTime:
            newQueue.insert(i, newCommand)
            #if insert at start
            if i==0:
                self.myTimer.cancel()
                self.create_new_timer()
            break
    else:
        newQueue.append(newCommand)
        if not self.myTimer.is_alive():
            self.create_new_timer()
    self.commandQueue = newQueue
```

Figur 5: Metoden til at indsætte et nyt Command objekt i kommandokøen

Figur (5) viser `enqueue_command` metoden som placerer en ny kommando i køen, det skal dog vedligeholdes at køen er sorteret efter stigende ventetid. Som input tager metoden kun den nye kommando der skal placeres i køen. I første linje kopieres den nuværende kommandokø så der kan arbejdes på den uden at den er i brug. Herefter starter en `for - else` konstruktion. `else` delen af denne konstruktion vil blive kørt hvis for-loopet terminerer *normalt*. En unormal termination af for-loopet sker med et `break` kodeord. For-loops i python kan iterere over elementer i en liste samtidigt med at en såkaldt *index tæller* vedligeholdes. Dette opnås ved den specielle *(tæller, element reference) in enumerate(liste)* syntaks i starten af for-loopet.

Loopet iterere over kommandokøen. For hver iteration tjekkes der om den nye kommando har en mindre ventetid en den kommando fra køen som er i fokus. Hvis dette er tilfældet indsættes den nye kommando foran denne. Når den nye kommando er blevet placeret i køen bliver der spurgt om `i` er lig 0 eller med andre ord om kommandoen er blevet placeret forrest i køen. Hvis dette er tilfældet skal den `Timer` som `CommandManager`en stoppes og en ny `Timer` med en ventetid lig den nye kommandos ventetid skal laves. Når kommandoen placeres i køen foran en anden kommando brydes loopet. `else` delen eksekveres ikke i dette tilfælde.

Hvis den nye kommandos ventetid er større end alle andre kommandoer i køen vil for-loopet dog terminere normalt. I dette tilfælde appendes kommandoen blot på køen. Et specielt tilfælde som også fanges her er når den nye kommando indsættes i en tom kø. Dette fanges ved at spørge om `CommandManager`ens `Timer` har en igangværende nedtælling. Man siger at `Timer`en er *i live* i dette tilfælde. Hvis `Timer`en ikke er i live betyder det at der også skal skabes et nyt `Timer` objekt. Til sidst sættes `CommandManager`ens interne reference til kommandokøen lig den liste der blev modificeret i metoden.

3.2 Kodekonvention

Gruppen er blevet enige om at følge en kodekonvention baseret på en blanding af metoden fra afsnit 11 i *Code Complete* (McConnell, 2004) og Python standard formatering (Denne kan findes på Pythons hjemmeside under PEP 8). Følgende illustrerer kort denne konvention.

Variabel navne Småt forbogstav, stort bogstav for at separere ord.
Eks. `variabelName`

Metode/funktions navne Små bogstaver, underscore for at separere ord.
Denne konvention er en python standard
Eks. `function_or_method_name`

Klasse navne Stort forbogstav, stort bogstav for at separere ord.
Eks. `ClassName`

Selvdokumenterende kode Navngivningen af entiteter i koden bør beskrive dennes formål.
Dette mindsker behovet for kommentarer og gør kodens formål læseligt.

Nævneværdigt er det at denne konvention er blevet formuleret af gruppen. som en respons på projektets nuværende tilstand hvad angår formatering af koden. Kodebasen følger derfor ikke på nuværende tidspunkt disse konventioner men står i stedet overfor en omformatering i næste iteration af projektet.

4 Formelle inspektioner

Der er i opgave formuleringen til denne deliverable blevet skrevet at der skal udføres formelle inspektioner til denne iteration. Dette er ikke blevet gjort. Delvist grundet at gruppen ikke vidste det var et krav til denne deliverable inden at rapportskrivningen begyndte. Men ligeledes har naturen af arbejdsprocessen ført til at alle har kendskab til al kode, og det har ikke indfundet sig som en naturlig del af processen at udføre formelle inspektioner.
Alle har set alt kode og haft mulighed for at diskutere den. Dette har været et biprodukt af at arbejde med pair-programming og være en gruppestørrelse på 3. Der har været stort fokus på at delingen af viden inden for gruppen skulle være højk.

Da projektet bliver lavet i en læringssammenhæng vil der i den næste iteration skulle være fokus på at få udført en række formelle inspektioner, med henblik på at forbedre programmets kode på en struktureret måde. Yderligere vil blive gennemgået i det næste afsnit 5: "Planlægning af næste iteration".

5 Planlægning af næste iteration

I slutningen af den forrige iteration måtte det erkendes, at der ikke var tid til at færdiggøre den implementation af user-stories, som var blevet sat for øje. Gruppen vurderer, at dette ikke skyldes størrelsesordnen af de user-stories, som blev planlagt men snare gruppens fortolkning af disse, hvilket ledte til en overambitiøs plan for implementationen. Nogle områder af projektet følger derfor på nuværende tidspunkt ikke de standarder som bestræbes i resten af projektet. Udbedring af disse mangler vil være i fokus for næste iteration, delvist for at følge forrige iteration til døren og delvist for at hæve kodekvaliteten af hele projektet.

5.1 Brugerflade til opdatering af databasen

Formålet er at en admin, mens serveren kører, kan tilgå en udvidelse af Django-frameworkets adminpanel, der tilbyder en grænseflade til at planlægge automatiske opdateringer af serverens database. Følgende user-stories synes at dække dette formål:

1. En admin kan opdatere en til flere vejlederes informationer automatisk
2. En admin kan planlægge en opdatering af en til flere vejlederes informationer, som udføres senere af serveren
3. En admin kan se en oversigt over planlagte opdateringer
4. En admin kan slette en planlagt opdatering
5. En admin kan bruge serveren til at opdage, og oprette datafelter til, nye vejledere automatisk

Bemærk at der i stedet for det typiske *bruger* er skrevet *admin* i overstående Dette skyldes at kun administratorer bør har adgang til denne funktionalitet og almindelige brugere skal derfor ikke have adgang til denne brugerflade.
I første øjekast synes disse mål at være ret omfattende, dog eksisterer meget af funktionaliteten allerede i projektet. Opgaven består derfor i at give administratorer adgang til disse værktøjer.
Det forventes at alle disse user-stories implementeres i næste iteration.

5.2 Forbedringer og mangler

Formålet er her at forsøge at adressere nogle af de mangler i kodekvaliteten som blev nævnt tidligere. Disse krav handler derfor ikke om ny funktionalitet eller lign. men derimod formatering, optimering og generelle forbedringer.

Formatering af hele projektet i henhold til en kodestandard

Der ligges en værdi i at projektets kode er både let at læse og at forstå derfor skal hele projektet formateres efter kodestandarden beskrevet i afsnit 3.2 i den kommende iteration.
Grundet at der ikke er blevet udført formelle inspektioner og projektet bliver lavet i en læringssammenhæng (beskrevet i afsnit 4), vil de ændringer der ville skulle laves blive opdaget ved hjælp af en formelinspektion for at opdage effekterne af sådan en.

Unittesting

Især i et servermiljø er det vigtigt at den involverede kode er stabil og er i stand til at håndtere alle slags input som kan blive præsenteret for den. For at sikre stabilitet bør modulerne *Scraper* og *Command Manager* og deres underklasser derfor unittestes. Andre dele af projektet mangler også unittests heriblandt enkelte *views* samt adapteren og dens underklasser.

6 Bibliografi og kilder

- Martin, R. C., Martin, M. (2007). *Agile principles, patterns, and practices in C#*. Upper Saddle River, NJ: Prentice Hall.
- Cockburn, Alistair (1997, September/October). *Structuring Use Cases With Goals*.
  Tilgået Febuar 23, 2016, på `http://alistair.cockburn.us/Structuring+use+cases+with+goals`
- CRUD-modellen, Mullender 2004.
  Tilgået Febuar 23, 2016, på `https://msdn.microsoft.com/en-us/library/ms978509.aspx`
- Steve McConnell, S. (n.d.), 2004 *Code Complete*.