**Arkitektur - Model View Controller**

Kompleksitet

Allerede på sprint 1 diskuterte gruppa hvordan vi skulle takle komplekstiten i kodebasen. Siden ingen hadde erfaring med større prosjekter fra tidligere var ingen spesielt opplyst på strategier for å takle den slags. Vi hadde fått hint fra student assistent om at Model View Controller kunne være hensiktsmessig, så vi bestemte oss for at alle medlemmer skulle lese seg opp på temaet MVC. Det var dog ikke før sprint 4 at vi endte opp med å implementere MVC – siden behovet for struktur ikke meldte seg før vi skulle implementere db med grensesnitt.

Vi fant ut at MVC kunne implementeres på forskjellige måter i java. Men på grunn av begge eksempler i faglitteraturen virket unødvendig omfattende[[1]](#footnote-1), valgte vi en litt enklere (å kanskje naiv) tilnærming ved å benytte oss av rene java-klasser (uten interfaces). Hensikten med MVC er å systematisk strukturere kodebase i 3 deler, slik at programflyt og logikk er i ’controller’, det grafiske grensesnitt befinner seg i ’view’ og all data/forretnings-regler som programmet bruker ligger i ’model’. Med forretnings-regler mener vi spesielle krav som data har. For eksempel har pris på varer en moms, men den avhenger av om kunden skal spise maten på restaurant eller ta med hjem. Prisen avhenger også om varen overstiger en hvis sum, da frakt vil bli gratis. Dette kalles domain-specific business-rules.

Den største forskjellen fra vår implementasjon og faglitteraturen sin, var at vi valgte bort observer/observable til fordel for getter-metoder når en view lastes inn. Vi syntes dette ble for komplekst å implementere.

MVC har ingen absolutt mal for hvordan mønsteret implementeres, og litteraturen poengterer at vi står fritt til å velge hvordan implementasjonen gjøres avhengig av programmets krav og programmeringsspråkets egenskaper. MVC er kort sagt et arkitektonisk mønster som strategisk takler kompleksiteten når mengden kode øker. Men programmereren står fritt til å velge en taktikk for hvordan dette skal implementeres gitt de forutsetninger det aktuelle programmeringsspråkets har.

Implementasjon

De tre første sprintene ble brukt til å prototype database, grafisk grensesnitt og gjøre oss kjent med verktøyene. Ikke før vi begynte å sette sammen løsningene meldte behovet seg for MVC. Hvordan skulle vi elegant separere GUI-kode fra databasen? Gruppen hadde nå blitt delt inn i 2 fraksjoner – GUI og database. Dette gjorde at behovet for å strukturere kode i henhold til MVC ble enda mer trykkende. Vi valgte derfor å lage mappestrukturen i henholdsvis tre java pakker, model, view og controller. Pakken ’model’ inneholdt all database-spesifikke filer, forretnings-logikk og domene-data ’view’ inneholdt alle GUI filer. Pakken ’controller’ hadde på dette tidspunktet ingen filer.

Vi så oss nødt til å lage controlleren etter hvert som grensesnitt-sidene ble ferdig-lagd.

Database-gruppen som bestod av to personer besluttet at den ene skulle lage kontrolleren til view, mens den andre skulle lage modellen. Siden forståelsen av hvilke krav modellen forandret seg underveis, ble det til at vi formet modellen parallelt med at vi lagde kontrolleren. Ved å benytte oss av å parogrammere for raskt ku. Da kunne den ene oppdatere modellen parallelt med utviklingen controlleren. Dette resulterte i

Den ansvarlige inntok da rollen som model-ekspert og veileder for den som lagde controlleren.

Erfaringer vi gjorde oss var at hvis en person programmerer alene og mangler dokumentasjon på kildekoden man programmerer opp mot, blir programmeringen betydelige mer ineffektiv og frustrerende. Kanskje kunne vi hatt som regel å ikke flette inn kode inn i master før javadoc er skrevet. Altså, skriv javadoc før man deler kode på github. Forsinkelser i sprint 5 fra gui-gruppa førte til at vi måtte avvente med å ferdigstille controller inntil alle hadde lagd ferdig grensesnittene. Vi brukte sprint 5 og 6 til å oppdatere deler av model, og controller etterhvert som vi ble bedre kjent med behovene i grensesnittene.

**Parprogrammering**

På oppfordring av student-assistent og senere vit.ass. ble vi oppfordret til praktisere disiplinen parprogrammering. Parprogrammering går ut på at to programmere sitter ved samme maskin og programmerer på ett tastatur. Den ene som programmerer inntar fører-rollen mens den andre inntar navigatør-rollen. Motivasjonen for å gjøre dette ligger i å redusere antallet feil i programmet. Dette skjer ved at programmerer (fører) fokuserer på de små taktiske løsninger, for eksempel hvordan man lager en for-løkke, hvor den andre (navigatøren) ser programmet fra et design-perspektiv – altså de strategiske valgene. Psykologien spiller her en viktig rolle. Følelsen av å ikke ville svikte den man parprogrammerer med gjør at du alltid er skjerpet. Og hvor den ene kan stille spørsmål til den andre – som da inntar en ekspertrolle. Vi opplevde parprogrammering som en effektiv måte for å kunne lære seg den koden man skulle programmere opp mot (controller vs model), spesielt siden store deler av javadoc manglet på daværende tidspunkt.

Siden gruppen jobbet mye hver for seg i starten ble vi nødt til å parprogrammere for å implementere controlleren med view. Samtidig som at vi par-programmerte controlleren bedret vi forståelsen av modelen, og som konsekvens ble den umiddelbart oppdatert. I sprint 5 ble parprogrammering et viktig verktøy for å ta igjen det tapte fra sprint 4. Selv om motivasjonen for par-programmering er å reduserer antallet feil i programmet, opplevde vi at parprogrammering ga oss en høyere produktivitet, enn om vi satt hver for oss.

**Git**

Vi valgte allerede på møte 1 å bruke Git til å holde kontroll på kildekode.

Opplæring i bruk av Git bød på utfordringer siden ingen, utenom ett medlem, hadde tidligere erfaring med bruk av verktøyet. Selv han hadde ikke brukt Git i et større prosjekt før. Derfor måtte vi planlegge hvordan arbeidsflyt skulle være og hvem som skulle styre hva. Etter litt kursing – fra vår selvutnevnte gitmaster – kastet vi oss ut i det. Det tok 2-3 sprinter før alle hadde hentet og delt kode over github. Github fungerte som en sentralisert kode-brønn. Lærings-kurven kan sies å ha vært bratt – men alle har vært enige om dette har vært nyttig kunnskap til en senere anledning.

Arbeidsflyt

Git er til forskjell fra SVN desentralisert noe vi mente ville skalere bedre for den enkelte utvikler. Med desentralisert mener vi at hver enkelt gruppemedlem kunne ha sin egen lokale ’topic’-branch å kode opp mot. Slik håpet vi å kunne kode hver for oss uten å komme i konflikt med hverandres kode. For eksempel hadde vi en egen database-branch som database ansvarlig hadde ansvar for. Så hvis noe i database skulle utbedres, lagde utvikleren en <topic>-<subtopic>, for eksempel database-vare. Når vare-tabellen da var implementert, flettet utvikler database-vare inn i database-branchen. For å kunne dele denne koden, ble den nylig oppdaterte database-branchen pushet til et sentralt repo på Github slik at vi kunne dele koden med andre. Da kunne han som gitmaster hente ned alle aktuelle oppdateringer fra alle kollegaer, flette det inn i master-treet. For så legge det ut på github igjen. Da kunne medlemmene oppdatere seg ved å hente siste kode fra gitmasters master-tre.

Eksempel (fra kommando-linja):

Gruppemedlemmet starter med å oppdatere sin master-branch fra gitmaster:

git pull

Lager en ny branch:

git branch database-vare master

Sjekker ut ny branch:

git checkout database-vare

Legger til forandringer, men en beskrivende log.

git commit -a -m 'vi kan nå hente ut varer fra database'

Del den nye branchen med gitmaster:

git push origin master

Gitmaster henter ned ny kode og deler den med andre:

git fetch --all

git merge medlemx/database-vare

git push origin master

1. <http://java.sun.com/products/jfc/tsc/articles/architecture/>

   <http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller> [↑](#footnote-ref-1)