Kvittoprojektet

Introduktion till testing av it-system 2015-10-28

|  |  |
| --- | --- |
| Jacob Ståhl | Jast8350 |
| Albin Fagernes | Alfa9590 |
| Filip Gezelius | Fige7684 |
| Josef Rosen | Joro8913 |

# Introduktion

Vi satt tillsammans och spånade vilket projekt vi ville utföra. Vi kom till den slutsatsen att vi ville bygga kvittohanteringsystemet. Vi kände att de låg oss närmast i vår kunskapskapacitet och att vi kunde ta oss an den utmaningen tillsammans. Efter att vi valt ett projekt så diskuterade vi vilka verktyg vi ville använda oss utav. Vi insåg att de vore mest praktiskt att använda oss utav de verktyg som finns att tillgå i skolan och de vi har lärt oss på föreläsningarna. Så valet blev att vi programmerar med Java i Eclipse och använder oss utav Junit för testsyftet. Versionshantering har vi arbetat med Github för lagringen och med klienten TortoiseGit för själva hanteringen. Se länken nedan för att få åtkomst till vårt repositorium.

<https://github.com/JacobStahl/TestProject>

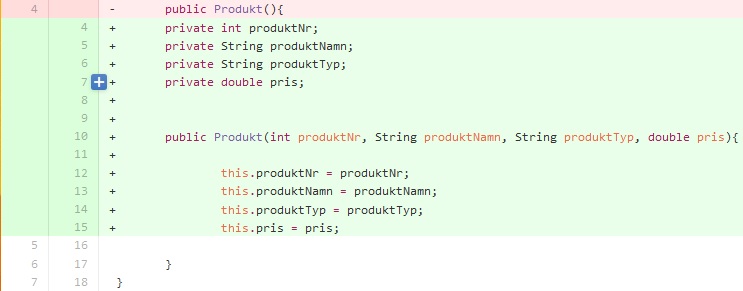
Efter att vi bestämt oss för alla verktyg började vi spåna fram en klasstruktur, dvs. hur vi skulle bygga våra klasser och testklasser. Vi gjorde även en uppdelning av arbetsuppgifter, dvs. vem som ska göra vad. Vi har hela tiden arbetat med olika uppgifter men även granskat varandras arbeten samt hjälpts åt. Mot slutet av projektet när vi granskade helheten så använde vi oss utav verktyg som Emma för att mäta täckningsgraden, Findbugs för att hitta fel i koden och Metrics för att se olika mått. För byggscript så kommer vi använda oss utav Ant.

Kodgranskningen tog vi hjälp av en kompis som är systemutvecklare på ett av de större svenska it-konsultbolagen för att få ett oberoende öga på vår kod. Han hjälpte oss med vad vi borde refaktorera och gav oss lite fingervisningar hur vi skulle tänka.

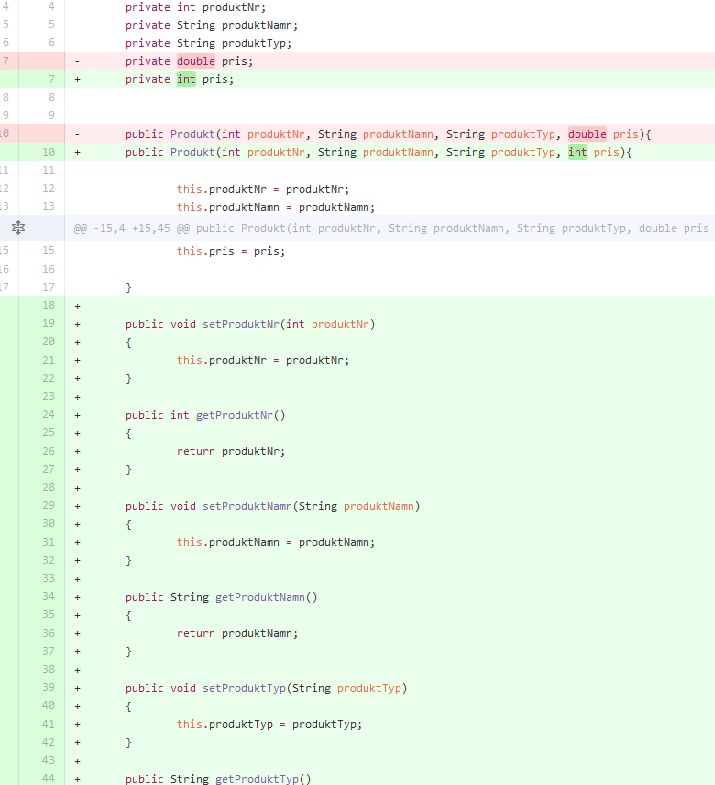
# Slutlig design

En övergripande modell över systemet. Lämpligt format är ett eller flera klassdiagram, plus eventuella andra modeller som behövs för att förstå hur systemet är uppbyggt. Diagrammen ska vara läsbara. Det är dock fullständigt okej att de är detaljerade, bara det går att zooma in ordentligt på dem. Ett tips är att börja med ett översiktligt diagram som inte innehåller mer än paket och klassnamn, och att sedan lägga till mer detaljerade diagram efter det.

# Testdriven utveckling – process

Vi har tillämpat testdriven utveckling till i stort sätt 100 % av de vi har utvecklat. Vi har hela tiden byggt testen innan vi har implementerat metoderna som testerna använder sig av. När vi har testat systemet med vårt teckningsverktyg uppgår det till ca 99 % täckning, vilket är en bra siffra. Vi har valt att inte testa vissa saker då det är för simpla för att göra tester på. Nedan visar vi några kodexempel på hur vi har arbetat enligt TDD. Dessa exempel är hämtade från vårt projekt på GitHub.  
  
Först har vi skapat en produktklass med tillhörande konsturktor för att definiera en början på en klass utan metoder.  


Sedan har vi konstruerat testerna i enlighet med vårt projekt.  


Efter att vi har byggt alla tester så bygger vi sedan metoderna i produktklassen.  


# Testdriven utveckling – erfarenheter

Genom hela kursen har vi fokuserat på den teoretiska biten om testdriven utveckling. Detta projekt gav oss möjligheten att faktiskt tillämpa de teoretiska kunskaperna praktiskt. Till en början var det svårt att göra testerna innan man byggde metoderna. Det kändes som att man gjorde emot allt man gjort innan. Så det tog ett tag innan man kom igång med TDD och att de kändes naturligt att arbeta med.

När vi väl hade kommit igång med ett testdrivet arbetssätt så började allt flyta på och vi byggde alltid testerna innan vi konstruerade metoderna. Fördelen vi har upplevt med TDD är givetvis att vi har mindre fel i koden än vad man kanske hade innan när man utvecklade. Nackdelen vi kände i början var att vi tyckte de tog lång tid att skriva en klass eftersom testerna tog så mycket tid innan man kunde påbörja programmerandet av klassen. Men eftersom man förhoppningsvis tjänar in denna tid i slutänden så känns de värt det.

En annan fördel vi känner är att man inte behöver debugga lika mycket som vanligt. I och med att testet utgör en mall för metoden så blir metoden sällan fel. Det känns även som att man bygger enklare och effektivare kod eftersom man endast gör det minsta möjliga för att få testet att fungera. En fördel är också om man ändrar någonstans i koden så kan man köra alla tester för att se om någon annan programsats har påverkats av den ändringen.

Förhoppningsvis får vi tillfälle att arbeta testdrivet när vi sedan efter utbildningen kommer ut på arbetsmarknaden. Det känns som denna kurs har vart väldigt lärorik och tagit oss till en ny nivå programmeringsmässigt.

# Ekvivalensklassuppdelning – namn på del

En kort presentation av vad ni valt ut för att tillämpa ekvivalensklassuppdelning på. Ni ska kort motivera valet, och ge tillräckligt med information för att det ska gå att bedöma er. Detta avsnitt och de tre föjande (till och med testmatrisen) ska finnas för samtliga delar ni tillämpat ekvivalensklassuppdelning på.

# Ekvivalensklasser – namn på del

Samtliga ekvivalensklasser för denna del presenterade på ett tydligt sätt.

# Testfall – namn på del

Testfallen som ni fått fram från ekvivalensklasserna. Observera att vi inte vill ha någon kod här, utan bara en tydlig presentation av testfallen i någon lämplig tabellform.

# Testmatris – namn på del

En testmatris som visar sambandet mellan ekvivalensklasserna och testfallen för denna del.

# Tillståndsbaserad testning

En kort presentation av vad ni valt ut för att tillämpa tillståndsbaserad testning på och vilket täckningskriterium ni valt att använda er av. Ni ska kort motivera valen, och ge tillräckligt med information för att det ska gå att bedöma er. Glöm inte att ta med själva modellen.

# Testfall för tillståndsbaserad testning

Testfallen som ni fått fram från tillståndsmaskinen. Observera att vi inte vill ha någon kod här, utan bara en tydlig presentation av testfallen i någon lämplig tabellform. Det ska enkelt gå att mappa testfallen till tillståndsmaskinen.

# Granskning

När projektet nu till stor del är klar så har vi genomfört en kodgranskning på projektet. Vi diskuterade hur vi skulle gå tillväga för att få ett gediget och bra resultat. Vi har dels valt att genomföra en granskning själva för att hitta så många fel i koden som möjligt. På nästa sida kan man överskåda en matris som visar alla fel som vi hittade i vårt projekt.

Vi har även valt att ta in en oberoende person för att leta efter överskådliga felaktigheter i koden. Denna person har samma utbildning som oss samt arbetar hen för ett av Sveriges största IT-konsult företag. Vi skickade vårt projekt till hen utan att har gett hen några extra instruktioner utöver vad han skulle leta efter. Felen som han hittade finns även i en ytterligare matris nedan.

Vi valde att granska själva först för att se om vi skulle få samma resultat som den oberoende granskaren. Det visade sig dock att hen hittade andra fel än vad vi hittade.

När vi granskade använde vi oss utav denna checklista, vi använde dock inte alla punkter utav dem utan endast de som va relevanta för vårt projekt och omfång. [Länk till checklista](http://blog.fogcreek.com/increase-defect-detection-with-our-code-review-checklist-example/).

Vi valde att den oberoende granskaren endast skulle kolla efter överskådliga fel för att hen dels var väldigt upptagen med jobbet, men även för att hen inte har de testkunskaper som vi i denna kurs har.

Efter att vi fått alla fel på svart och vitt så refaktorera vi koden tills att alla fel var åtgärdade.

# Granskningsrapport

**Oberoende granskare**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Felaktighet** | **Klass** | **Allvarlighet** |
| O1 | Borde använda stor bokstav på klassnamn t.ex. ”Klassnamn” och inte ”klassNamn”. | Alla | 1 |
| O2 | Blandat svenska och engelska – bör välja ett av språken och hålla sig till det. | Alla | 3 |
| O3 | Inga kommentarer – bör finnas kommentarer så andra kan enklare sätta sig in i systemet. | Alla | 4 |

**Granskning av oss**

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **Felaktighet** | **Klass** | **Allvarlighet** |
| G1 | Teckningsgrad ej 100 % | Kvittoklass, Rabattklass | 3 |
| G2 | Pengarklassens printmetod, fel vid utskrift av ören när ören < 10 | Pengarklassen | 2 |
| G3 | Ingen kontroll för out-of-bound errors | Alla | 1 |

Som man kan se i matriserna ovan så fanns det inte jättemycket allvarliga buggar i vårt system då projektet är tämligen litet. Men de fanns ändå ett par viktiga buggar att åtgärda. Dessa buggar hade vi aldrig hittat om vi inte utförde en formell granskning av systemet. De olika tillvägagångsätten kan man ju alltid diskutera vilket som är bäst. Men vi kände att genom att använda både en oberoende granskare och granska själva gav oss störst chans att hitta så många fel som möjligt.

# Granskning – erfarenheter

Efter att vi har granskat vårt projekt ur ett progamkodsperspektiv så har det bidragit till mycket nya kunskaper. Vi har fått en större inblick i hur mycket buggar det egentligen finns som man inte ser med egna ögon när man utvecklar. Vi satt och diskuterade hur vi skulle utföra granskningsprocessen. Vi valde då att använda oss utav våra egna granskningskunskaper, dels för att testa det som vi lärt oss under kursen och dels för att de fanns mycket information om formella granskningar online. Så det var enkelt att komma igång med granskningen. När vi granskade första gången trodde vi att vi skulle hitta mängder med buggar, men vi hittade endast tre nämnvärda så vi kände oss nöjda med vår programmeringsinsats trots allt.

När det kommer till den oberoende personen som granskade så valde vi att ta in en kompis som gått samma utbildning för att få ett par extra ögon på koden. Vi förde sedan en diskussion och gick igenom vad hen hittat.

Flera erfarenheter vi dragit av denna uppgift är hur viktigt det verkligen är med granskning. Man kan spara mycket pengar i slutändan genom att granska kontinuerligt. En annan sak vi har lärt oss är att om man granskar oftare så hjälper det till att hålla en konsekvent och bra kodstandard genom att hela tiden granska och påpeka eventuella standardavvikelser man har kodat.

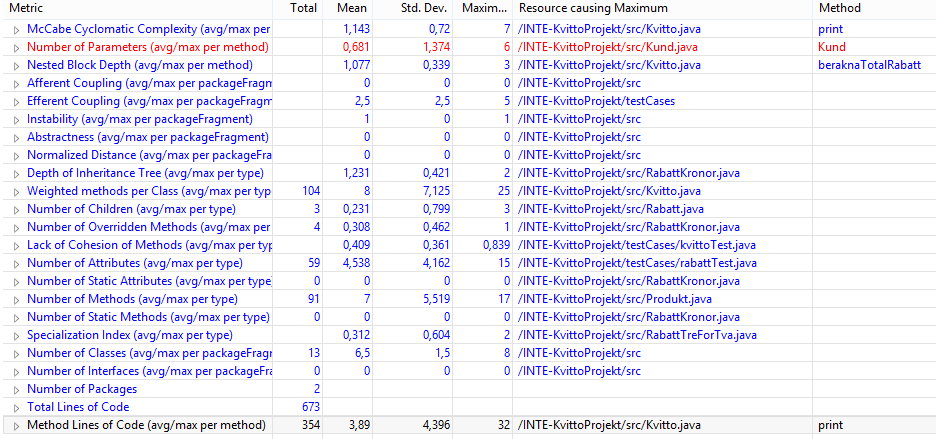
I slutändan är nog den största erfarenheten helheten under granskningen. Det vill säga att de finns olika vägar som leder till samma resultat. Det gäller bara att hitta den vägen man gillar bäst. Oberoende granskningar är lärorika för då har man en person att bolla informationen med. Å andra sidan, granska själv fungerar också men risk för hemmablindhet. Det är helt enkelt en smaksak vilken väg man väljer att gå framöver.

# Kodkritiksystem

En presentation av de problem som hittats med hjälp av verktyg för statisk analys och en diskussion av dem enligt anvisningarna. Det räcker alltså inte med att bara lista problemen, ni måste förhålla er till dem också. Tänk också på att ni ska göra detta både på koden som den såg ut före granskningen och på koden efter att ni rättat det som kommit fram under granskningen.

|  |  |  |  |
| --- | --- | --- | --- |
| **ID** | **FindBugs varning** | **Klass** | **Allvarlighet 1-5** |
| F1 | Dead store to rabatt in Kvitto.print() | Kvitto | 2 |
| F2 | Integral division result cast to double or float in RabattTreForTva.berakna(Produkt) | RabattTreForTva | 2 |
| F3 | Kvitto.getDate() may expose internal representation by returning datum | Kvitto | 1 |
| F4 | Kvitto.print() concatenates strings using + in a loop | Kvitto | 1 |
| F5 | The class name kundTest doesn't start with an upper case letter | kundTest | 1 |
| F6 | The class name kvittoTest doesn't start with an upper case letter | kvittoTest | 1 |
| F7 | The class name pengaTest doesn't start with an upper case letter | pengaTest | 1 |
| F8 | The class name produktTest doesn't start with an upper case letter | produktTest | 1 |
| F9 | The class name rabattTest doesn't start with an upper case letter | rabattTest | 1 |
| F10 | The import org.junit.Before is never used | kundTest | 1 |

# Statiska mått

Översikt av mått  


Lack of cohesion of methods:

Representerar mängden par metoder som inte är kopplade på något sätt i en klass. Hög LCOM tyder på att man bör refaktorera då man har många metoder i klasser som inte är relaterade. Våra klasser visar relativt låg, genomsnitt cirka 0,4. Detta medelvärde höjs något eftersom att testklasser räknas med och de har lägre sammankoppling. Eftersom vi har lågt LCOM värde tyder det på att metrics inte kan se något tydligt fall där vi antagligen behöver refaktorera.

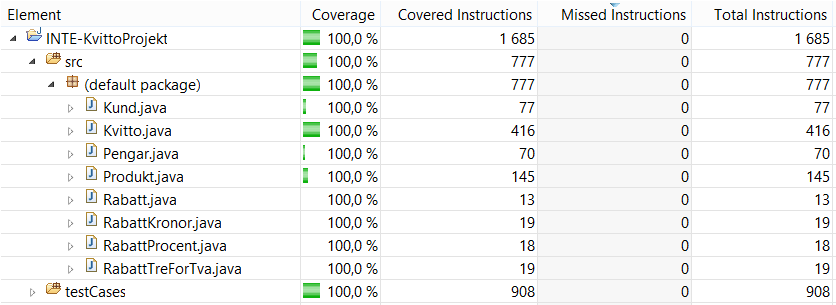
Nested Block Depth:

Visar metoder och konstruktorer och räknar hur många kapslade block dessa har. Om en metod har väldigt många kapslade block kan den bli väldigt svår att förstå. Vårt genomsnitt är på 1,077 och maxvärdet är 3. Det är alltså antagligen inget problem med för många kapslade metoder då det inte borde utgöra något problem med att förstå koden.

McCabe Cyclomatic Complexity:

Visar på komplexiteten av programmet genom att mäta mängden oberoende vägar genom koden. Rekommendationen är att generellt sett hålla sig under ett värde på 10 i alla moduler och försöka dela upp dessa om värdet blir för högt. Vårt medelvärde 1,132 och maxvärdet är 7. Medelvärdet tyder på att programmet är ganska simpelt och inte har så många olika vägar genom koden. Maxvärdet är relativt högt men det kommer enbart från en metod: print. Denna har fler "if"-satser för utskrift vilket gör att värdet höjer sig något. Värdet är dock fortfarande under 10. Metrics visar alltså inga metoder som antagligen behöver brytas upp på grund av för hög komplexitet, vilket inte är förvånande i ett så litet program.

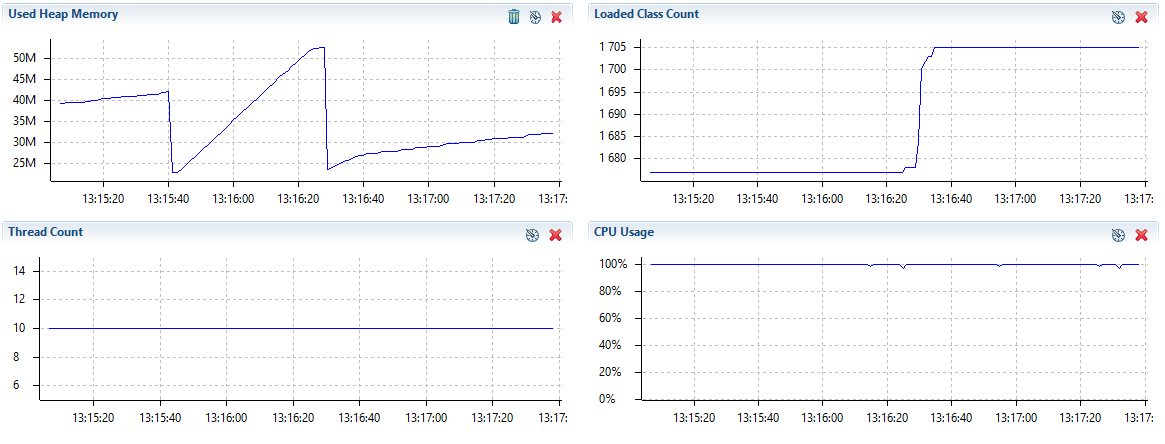
# Täckningsgrad



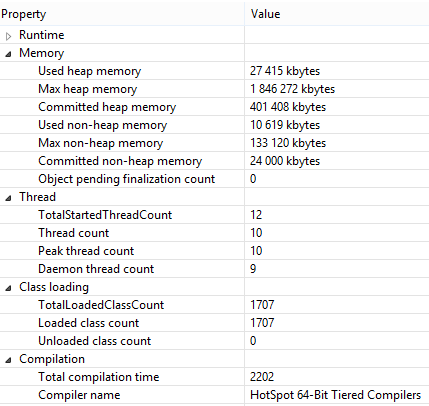
# Profiler

En kort presentation av hur ni gått tillväga för att testa koden med en profiler och vilka resultat ni fick fram.

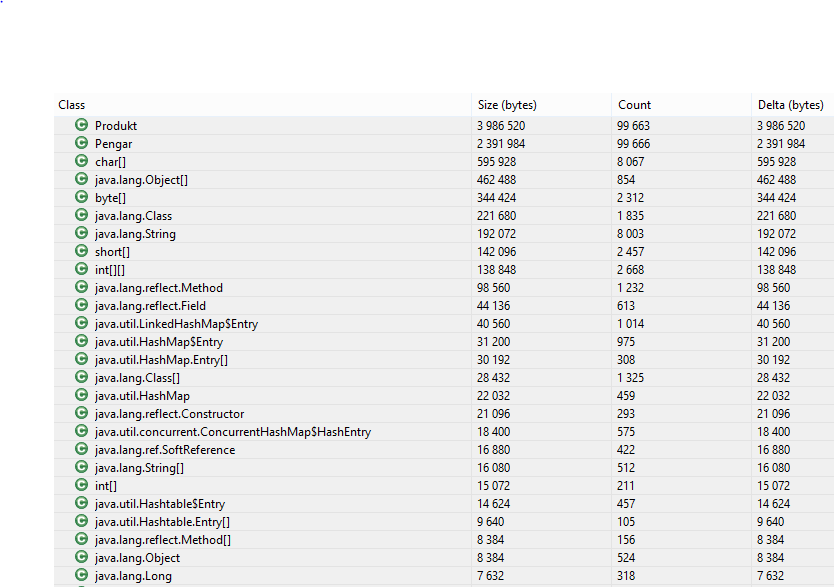
**Tidslinje**

****

**Översikt**

****

**Minne**

****

# Byggscript

Byggscriptets första (seriösa) version, och den slutliga.

# Övrigt

Här kan ni ta upp övrigt av relevans för bedömningen av ert arbete. Om avsnittet inte behövs kan det plockas bort.