Kvittoprojektet

Introduktion till testing av it-system 2015-10-29

|  |  |
| --- | --- |
| 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.

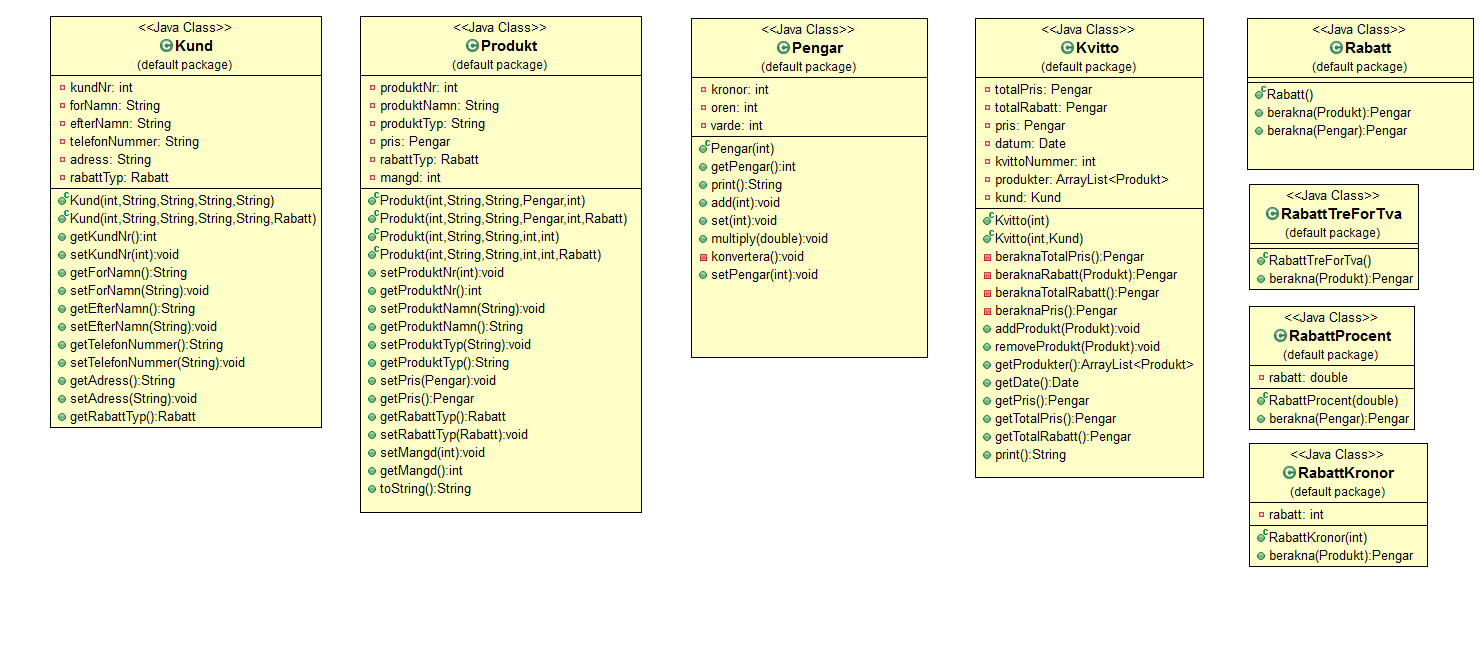
<https://github.com/JacobStahl/TestProject>

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

**Initiala klasser**



De klasser som skapades och användes under utvecklingsprocessen var Kund, Produkt, Pengar, Kvitto, Rabatt, RabattTreForTva, Rabattprocent och Rabattkronor.

**Kund**

Klassen kund innehåller variabler som exempelvis kundNr, forNamn, och efterNamn som har get metoder för att returnera information om en viss kund och set för att sätta värdet och tilldela en kund information. I kundklassen finns också variablen rabattTyp av klassen rabatt som håller koll på olika typer av rabatter.

**Produkt**

I produktklassen skapades variabler som produktNr, produktNamn, produktTyp och mangd som blivit tilldelade get och set metoder för att kunna returnera och sätta värden på de produkter som finns i klassen. I produktklassen finns även variablen pris som av klassen Pengar som håller reda på vilket pris produkten ska ha och variablen rabattTyp som indikerar på vilken rabatt det är som gäller för en viss produkt.

**Pengar**

Klassen pengar innehåller variablerna kronor, oren och varde. Pengaklassen används för att undvika beräkning med decimaltal. Metoderna multiply och add används för att kunna addera och multiplicera med pengaklassen.

**Kvitto**

Kvittoklassen håller reda variablerna datum och kvittoNummer samt använder metoder som beraknaTotalPris och beraknaTotalRabatt som beräknar det totala priset och den totala rabatten på kvittot. Kvittoklassen har också en print metod som skriver ut det aktuella kvittot. Kvittoklassen kan också använda sig av en "list" av produkter.

**Rabatt**

Rabattklassen innehåller metoderna berakna(Produkt):pengar som tar emot en instans av klassen produkt och berakna(Pengar):pengar tar emot en instans av klassen pengar.

**RabattProcent**

RabattProcent innehåller fältet rabatt av datatypen double och ärver från klassen Rabatt. Rabattprocent beräknar hur stor rabatt man får i procent.

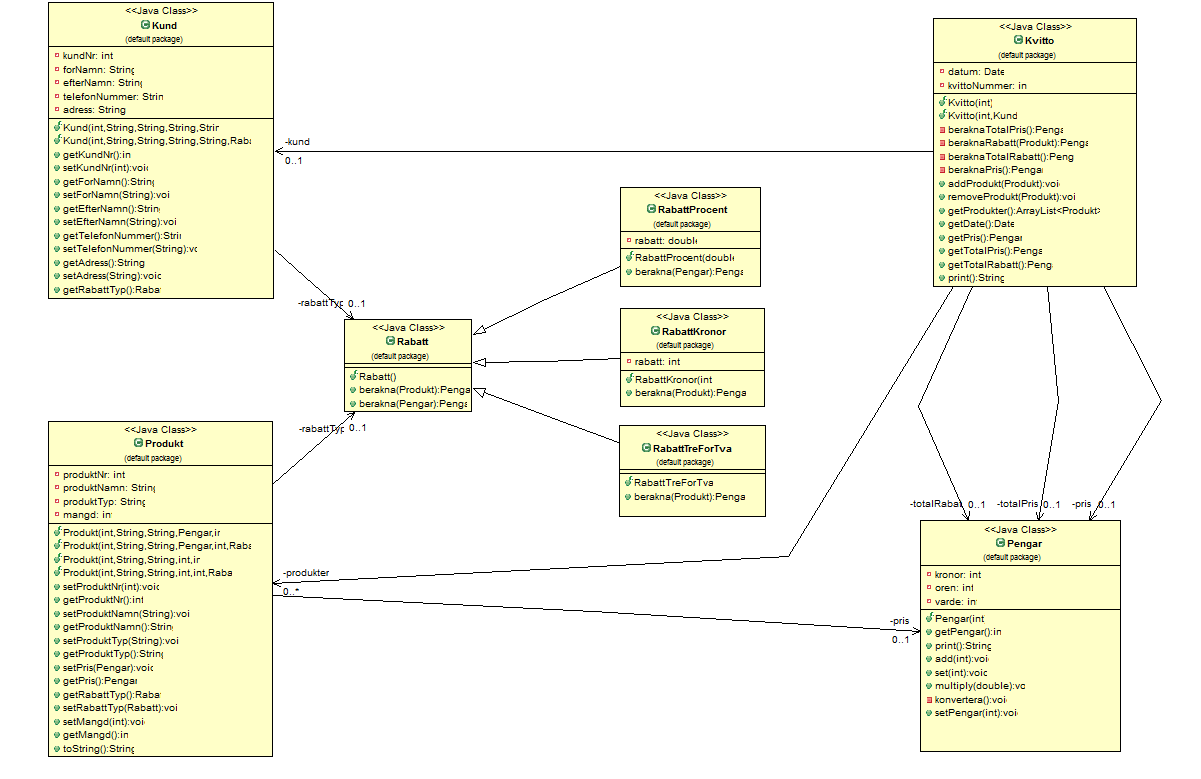
**RabattKronor**

RabattKronor innehåller fältet rabatt av datatypen int och ärver från klassen Rabatt. Rabattprocent beräknar hur stor rabatt man får i kronor.

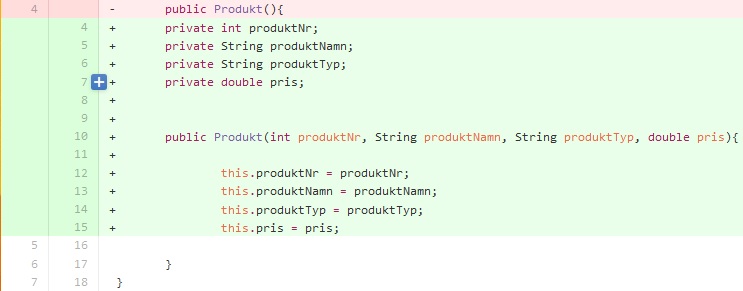
**RabattTreForTva**

RabattTreForTva har metoden berakna(Produkt):pengar som tar emot en produkt och returnerar ett pengarobjekt med rabatten som värde.

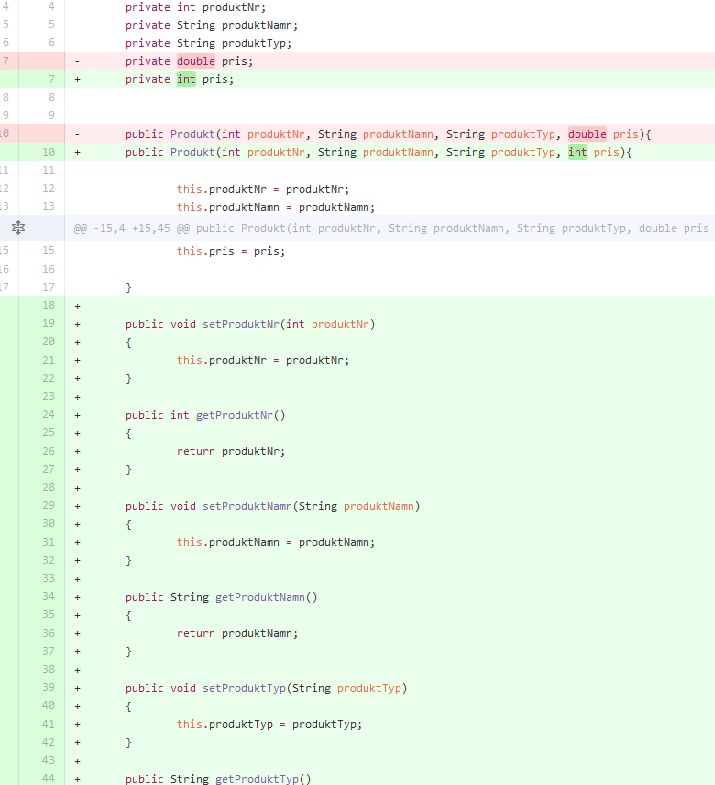
**Slutgiltig design med beroenden**



# 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 – 3 för 2

Från början hade vi bara ett test som prövade om man fick 3 för 2. Fler tester behövdes, men det verkade onödigt att testa alla värden mellan 0 och oändligheten. Med hjälp av ekvivalensklassuppdelning delade vi upp värdena i partitioner efter förväntat resultat. För att kunna se vilka värden vi borde testa.

# Ekvivalensklasser – 3 för 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Antal gratis | 0 | 1 | 2 | 3 | n |
| Antal produkter | |  |  | | --- | --- | | 0 | 2 | | |  |  | | --- | --- | | 3 | 5 | | |  |  | | --- | --- | | 6 | 8 | | |  |  | | --- | --- | | 9 | 11 | | |  |  | | --- | --- | | 3n | 3n+2 | |

# Testfall – 3 för 2

**treForTvaLoop()** - Här kunde vi göra en loop som testar att rabatten beräknas korrekt för varje partition. Eftersom det teoretiskt finns ett oändligt antal partitioner testade vi ett antal som var helt otroligt att någon skulle nå i verkligheten, men som gick snabbt att köra. Loopen börjar på noll och ökar antalet produkter med 3 för varje färdigt varv. Den kontrollerar sedan att den uträknade rabatten stämmer överens med det förväntade resultatet.

# Testmatris – 3 för 2

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Antal gratis | 0 | 1 | 2 | 3 | n |
| Antal i korgen | |  |  | | --- | --- | | 0 | 2 | | |  |  | | --- | --- | | 3 | 5 | | |  |  | | --- | --- | | 6 | 8 | | |  |  | | --- | --- | | 9 | 11 | | |  |  | | --- | --- | | 3n | 3n+2 | |
| Testfall | testTreforTvaLoop() |  |  |  |  |

# Beslutstabeller - Rabatter

För att testa olika rabatter och hur dem samspelar gjorde vi en beslutstabell för att se vilka möjliga kombinationer som fanns och hur dem skulle fungera. När vi skapade testfall för beslutstabellerna märkte vi att mycket kod i kvitto-klassen behövde refaktoreras för att kunna hantera flera rabatter på samma gång. Kombinationen av en tre för två-rabatt och 10 % på köpet-rabatt gjorde det också viktigt att testa hur kundrabatten beräknades. Att få 10 % på summan innan 3 för 2-rabatten dragits av skulle ju bli märkligt. Testerna kollar att totalpriset och totalrabatten blev det vi förväntade oss.

# Beslutstabell – Rabatter

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Conditions** |  |  |  |  |
| Kundrabatt 10 % | T | T | F | F |
| 3 för 2 | T | F | T | F |
| **Actions** |  |  |  |  |
| Rabatt | 1 gratis och -10 % | -10 % | 1 gratis | Ingen rabatt |

# Testfall - Rabatter

**testTvaRabatter**() – Skapar ett kvitto med två produkter och en kund. Den ena produkten har en 3 för 2-rabatt och kunden har 10 % rabatt på köpet.

**testIngenRabatt**() – Skapar ett kvitto med två produkter och en kund. Inga rabatter.

**testKundRabatt**() – Skapar ett kvitto med två produkter och en kund. Kunden har 10 % rabatt på köpet.

**testProduktRabatt**() - Skapar ett kvitto med två produkter och en kund. Den ena produkten har en 3 för 2-rabatt.

# Testmatris – Rabatter

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Conditions** |  |  |  |  |
| Kundrabatt 10 % | T | T | F | F |
| 3 för 2 | T | F | T | F |
| **Actions** |  |  |  |  |
| Rabatt | 1 gratis och -10 % | -10 % | 1 gratis | Ingen rabatt |
|  |  |  |  |  |
| **Testfall** | testTvaRabatter() | testKundRabatt() | testProduktRabatt() | testIngenRabatt() |

# 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 | Täckningsgrad 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

**Feltabell**

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

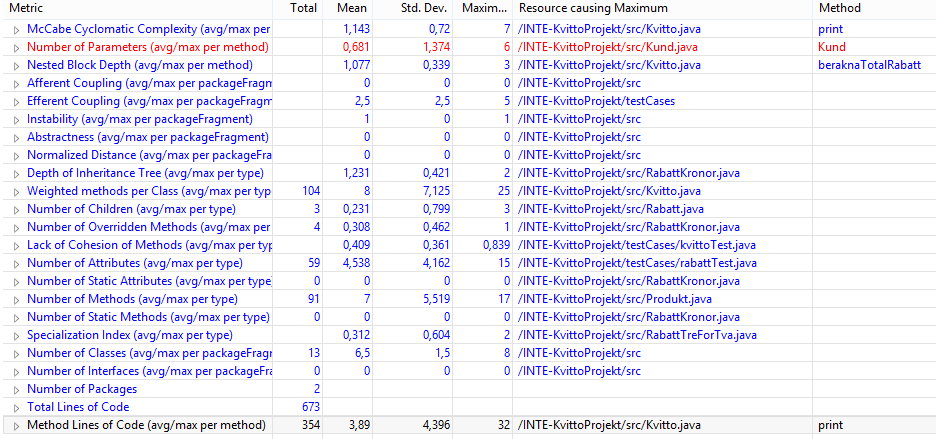
Initialt så hittade FindBugs inga fel, vi var då tvungna att skruva upp hur strikt verktyget skulle reagera. När vi satte den på max så kom dessa fel upp. Som man tydligt kan se så finns det inte så jättemycket allvarliga fel som FindBugs hittade vid körningen. De som hittades var mest att namnkonventioner som var felaktiga samt lite annat som rör hur metoder kan skrivas effektivare. Den hittade även en importsats som inte används.

**Lösningstabell**

|  |  |  |
| --- | --- | --- |
| **ID** | **Lösning** | **FelId** |
| L1 | Omvandlade parametern som tidigare var integer till en double | F1 |
| L2 | Ändrade så att datum returnerade en klon av sig själv istället för att exponera sitt eget värde. | F2 |
| L3 | Ändrade konkatenerandet till en stringbuilder istället | F3 |
| L4 | Ändra klassnamn till korrekt formatering ”Klassnamn” | F4,F5,F6,F7,F8 |
| L5 | Ta bort import som aldrig används | F9 |

Så här såg lösningarna ut på alla fel som FindBugs kunde hitta. Felen var hyfsat enkla att rätta till då felbeskrivningarna var väldigt bra och de fanns gott om information på internet att tillgå. Vi anser att kodkritiksystem är väldigt behjälpliga på det sättet att verktyget presenterar felen på ett ypperligt sätt för att man som utvecklare skall kunna ta sig an det på bästa sätt. Efter att alla fel är rättade så påvisar Findbugs att de inte finns några ytterligare problem.

# 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.

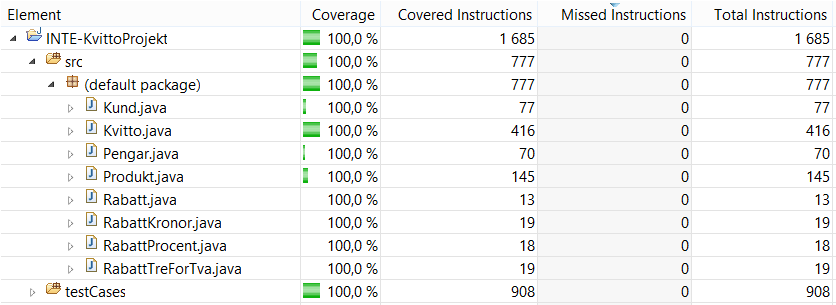
Nested Block Depth:

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



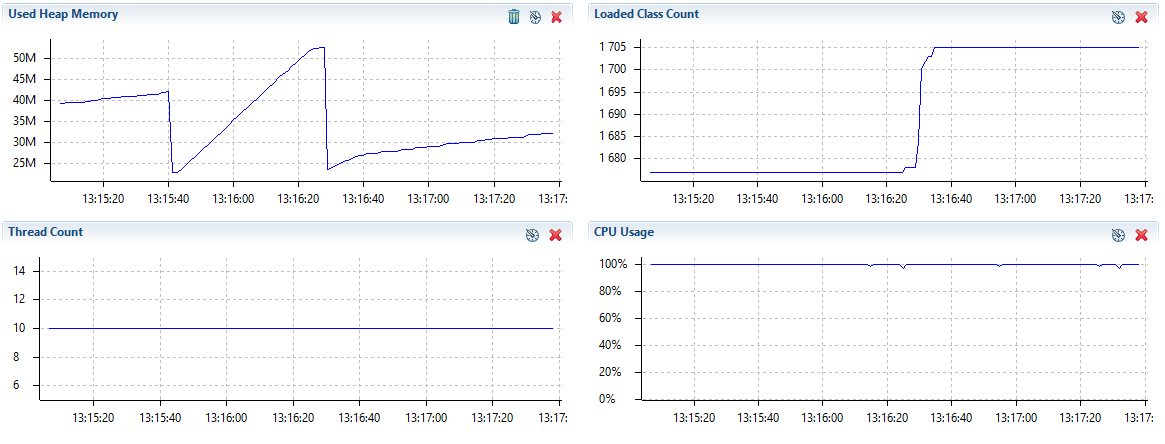
# Profiler

Profilern vi valt att använda är JVM Monitor. För att ha något att köra profilern på var vi tvungna att skriva ett nytt test eftersom att de tester vi redan skapat blir klara alldeles för fort.

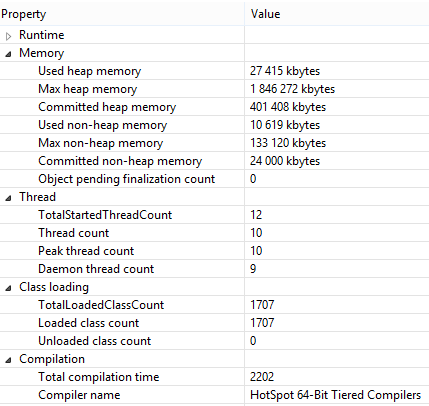
Testet vi gjorde är en simpel while-loop där vi för varje iteration lägger till en produkt i kvittot och sedan testar om priset på kvittot blir korrekt. Vi kör denna while-loop cirka 50000 gånger för att det ska ta tillräckligt lång tid så att vi kan använda profilern men inte så långt att man inte orkar vänta tills det är klart.

Resultat från profilern finns nedan. Det som verkar mest intressant i och med att det är test vi kör och inte ett körbart program är den information vi får ut angående minne då programmet kräver ungefär lika mycket minne i test som det gör annars. Där skulle man kunna se vilka klasser som använder mest minne och eventuellt hitta ställen där man ska försöka ändra klasser för att spara minne.

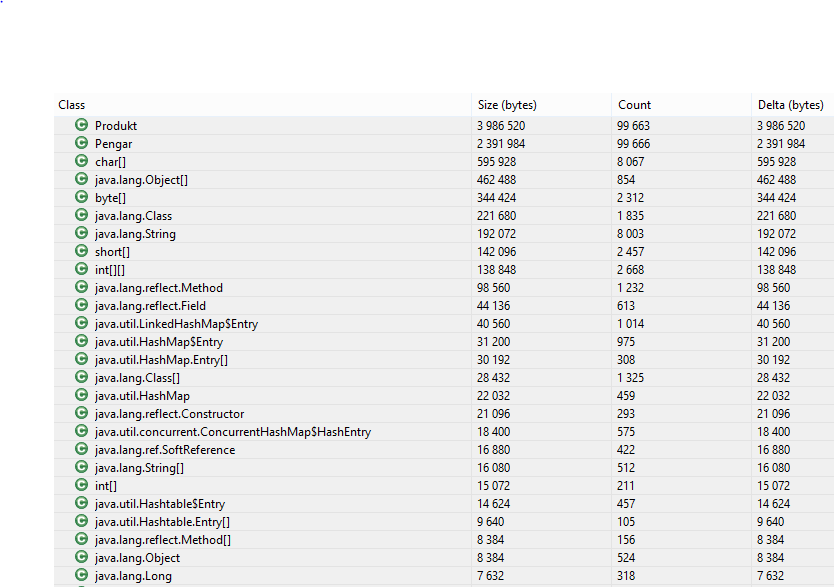
**Tidslinje**

****

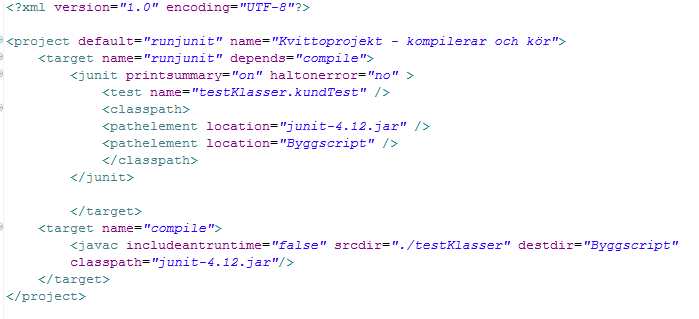
**Översikt**

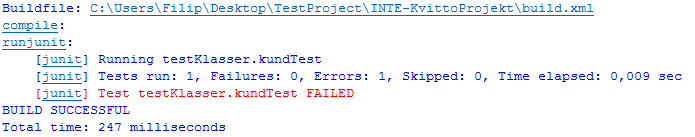
****

**Minne**

****

# Byggscript **Byggscript med Ant (första versionen)** Den första versionen av byggscript fungerade, men exekverade inte koden på så sätt att vi kunde ta del av den information som beskriver huruvida testen lyckas eller inte. Den första versionen byggscript körde igenom våra testklasser och i vår "source directory" kunde vi se att klasserna hade testas, men utan något resultat.

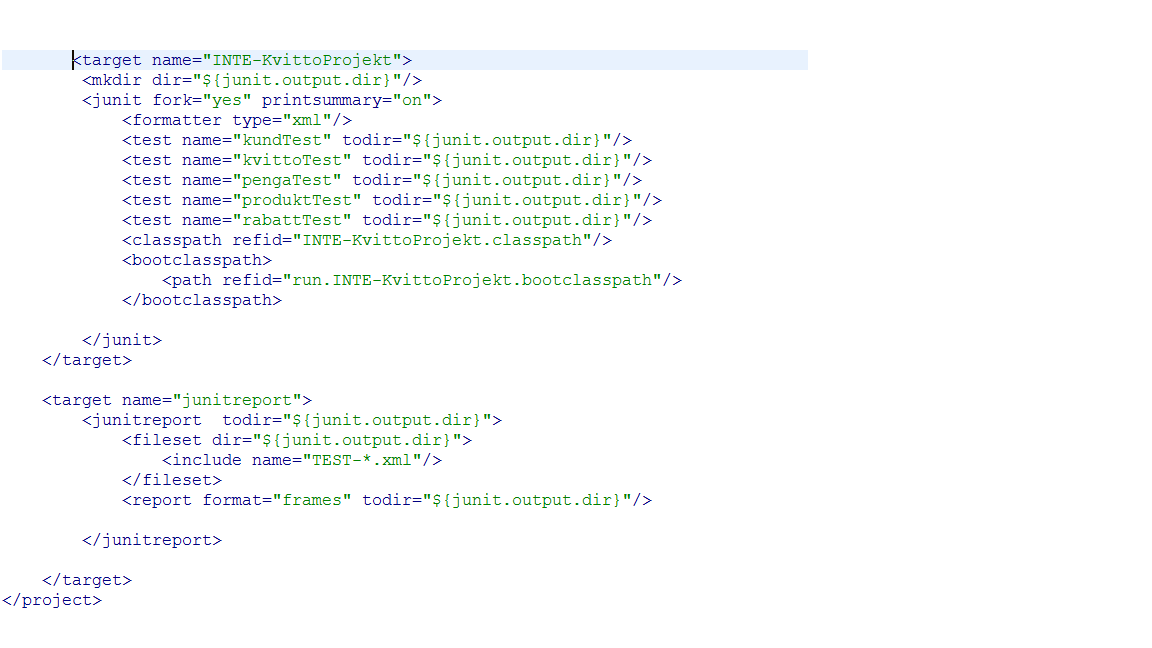


Koden ovan visar den första versionen byggscript som skapades. Problemet med byggscriptet i xml-filen var att vi inte kunde specificera vilken klass vi ville testa samtidigt som testen vi utförde genererade "errors".

Ovan visar vad som skrevs ut i konsolen när man exekverat byggscriptet. Scriptet ville inte hitta de specificerade klasser vi ville testa mot och genererade "failed" var gång vi försökte köra testen fast byggscriptet ändå skrev ut "build successful".

**Den slutgiltiga versionen**

Vi fick till slut testen och byggscripten att fungera när vi insåg att Eclipse kunde autogenerera och skapa egna byggscript. Det fanns en funktion som automatiskt skapade byggscript på vårt "package" innehållandes våra testklasser vi ville köra mot.



Ovan visar det slutgiltiga byggscriptet som användes för att testa våra testklasser. I xml-filen som scriptet skapats i kunde vi specificera vilka klasser vi ville testa och i vilket "output directory" testerna och resultaten skulle sparas. I projektmappen skapades en mapp, "junit" som sparade ner testerna och konverterade testerna till html-filer som vi sedan kunde öppna i webbläsaren.

**Buildfile: C:\Users\Filip1\Desktop\TestProject\INTE-KvittoProjekt\build.xml**

**junitreport:**

**[junitreport] Processing C:\Users\Filip1\Desktop\TestProject\INTE-KvittoProjekt\junit\TESTS-TestSuites.xml to C:\Users\Filip1\AppData\Local\Temp\null512991225**

**[junitreport] Loading stylesheet jar:file:/C:/Users/Filip1/Downloads/eclipse-java-luna-SR2-win32-x86\_64/eclipse/plugins/org.apache.ant\_1.9.2.v201404171502/lib/ant-junit.jar!/org/apache/tools/ant/taskdefs/optional/junit/xsl/junit-frames.xs**

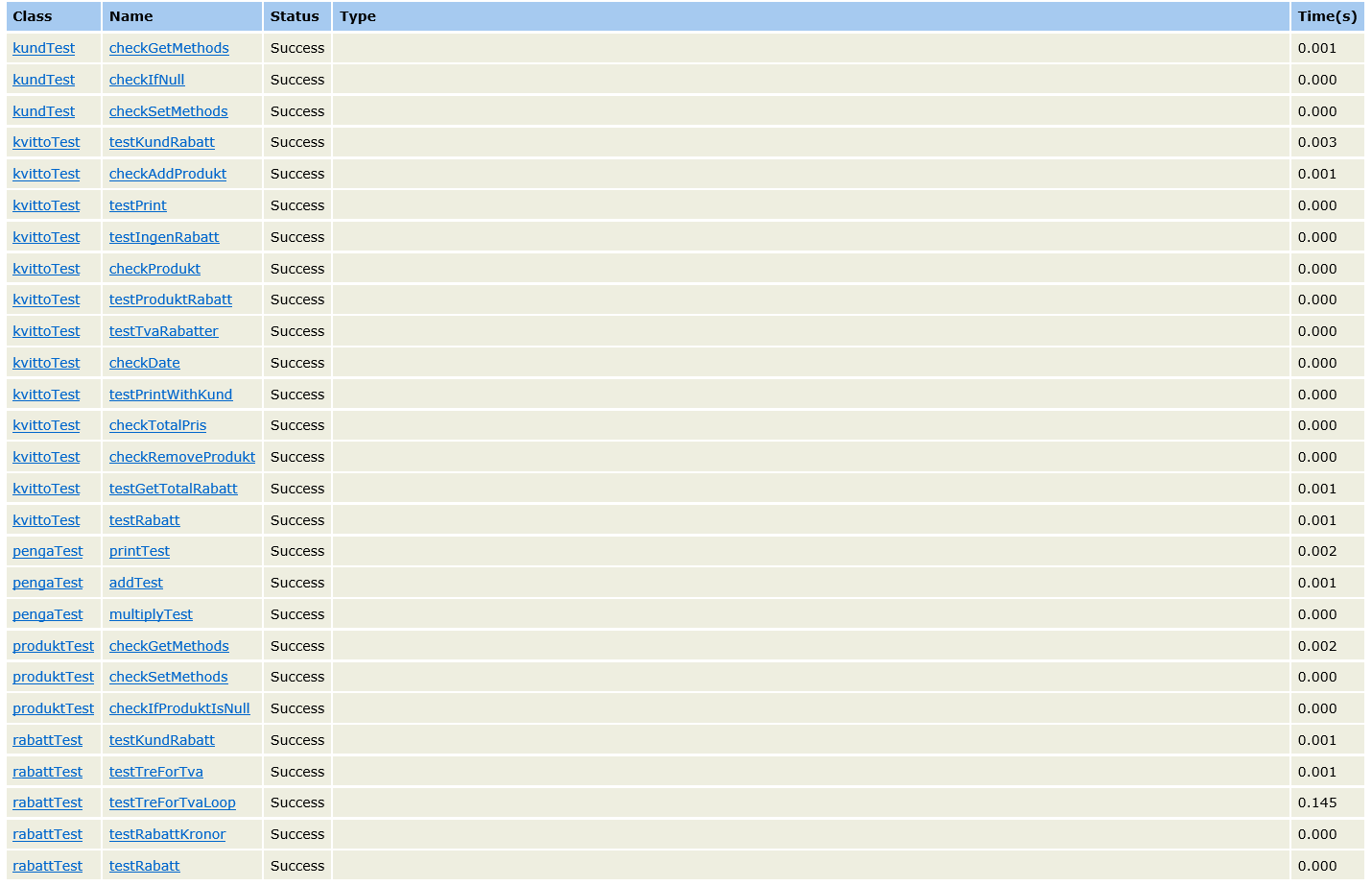
**l[junitreport] Transform time: 973ms**

**[junitreport] Deleting: C:\Users\Filip1\AppData\Local\Temp\null512991225**

**BUILD SUCCESSFUL**

**Total time: 1 second**

Ovanstående är följande som skrevs ut i konsolen i samband med exekvering av testerna.



Ovan ser man en som bild visar testerna i en webbläsare och beskriver vilka klasser och metoder som körts i de olika klasserna. Ovan kan man se att alla test har status "success" och då inte genererat några fel samt att man kan se hur fort det gick att köra testen.