

Projektnamn

Introduktion till testing av it-system HT 2014

Anna Thomasson	Anth1160
Elise Roihuvuo	Tero0337
Karolina Peters	Kape5428
Aframyeos	afro0793
Rohoum	

Introduktion

En kort introduction till projektet där ni också listar de verktyg ni använt. Om ert versionshanteringssystem går att komma åt ska adressen dit finnas med, annars ska det finnas en länk från vilken man kan ladda hem källkoden.

Vi valde att göra uppgiften Kvittan som föreslagits som uppgift. Vi har skapat tre klasser som är tänkta att fungera som en del i ett kassasystem:

- Klassen Receipt som hanterar kvittot och uträkningar av priser rad för rad på kvittot och ger en slutsumma på alla inmatade eventuella varor
- Klassen Product som håller koll på information om varornas namn och originalpris, och om en vara har en eventuell rabatt
- Klassen Discount för att kunna lägga till och ändra rabatter på enskilda varor, och har även en metod för att beräkna det rabatterade priset på en vara om den är rabatterad.

För att skapa och genomföra vårt projekt har vi använt dessa verktyg:

- **Eclipse** i kombination med **JUnit 4** användes för källkodsskrivande och testfallsskrivning
- **EclEmma** användes för att testa hur stor täckningsgrad klasserna hade
- **Findbugs** var verktyget vi använde som kodkritiksystem
- **GitHub** är det versionshanteringssystem vi har valt att använda oss av för versionshantering.
- **Metrics 1.3.6** är det verktyget som vi valde för att samla in statistiska mått.
- **JVM Monitor** heter den profiler som vi använde för att profilera, d.v.s. ta fram olika mått ur programmet som t.ex. minnes användning.

Länk för att ladda ned källkoden till projektet:

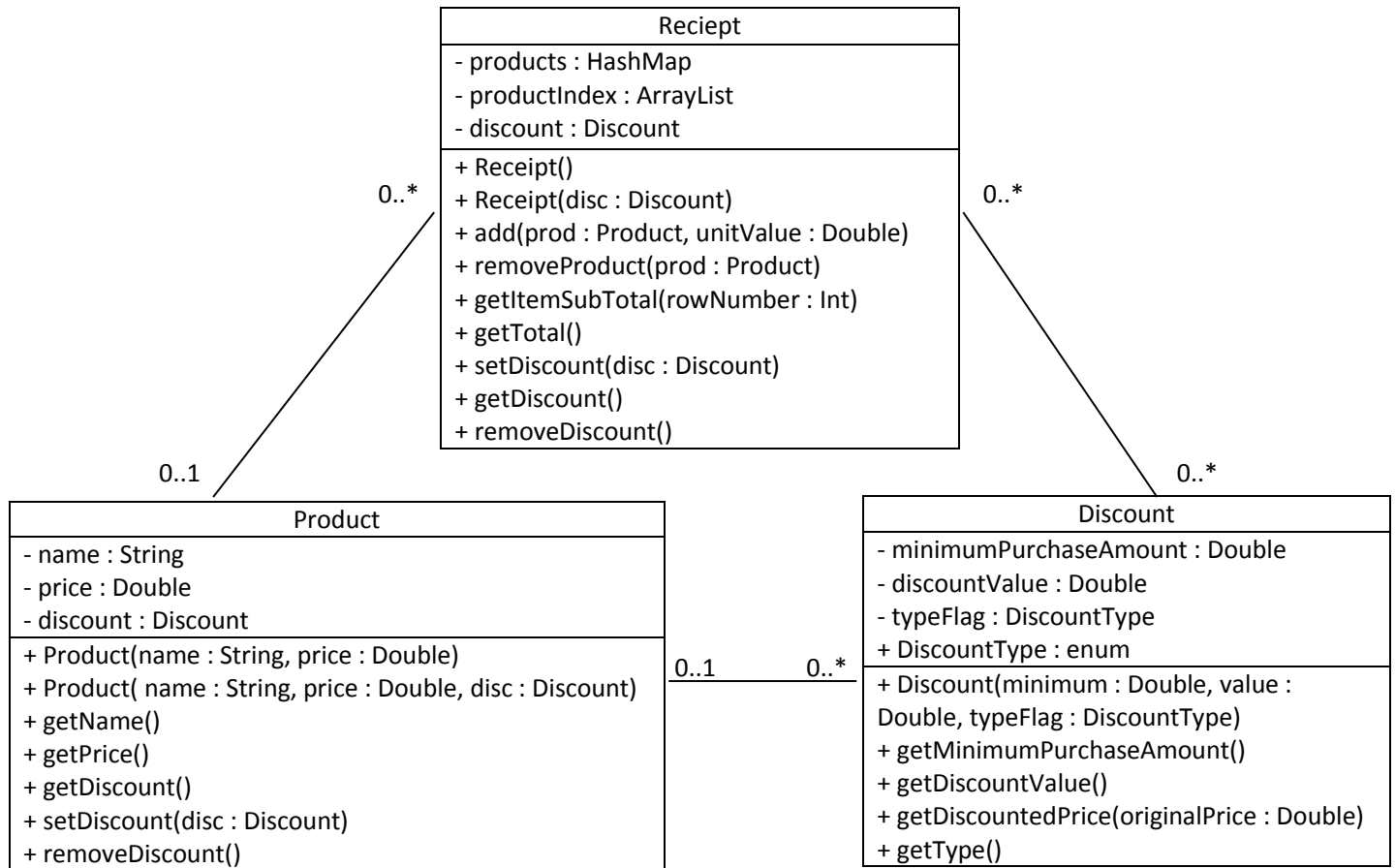
<https://github.com/InteRabatter/inteproj>

<https://github.com/InteRabatter/inteproj.git>

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.

Klassdiagram:



Testdriven utveckling – process

En översikt över hur ni tillämpat TDD med exempel från olika personer och olika faser i projektet. Om ni har använt versionshanteringssystemet ordentligt bör all information som efterfrågas här finnas i det. Tänk på att kodexemplen ska vara läsbara.

Vi valde att arbeta som en enhet. All kod och alla testfall har vi tagit fram tillsammans som grupp. Efter att vi tagit fram ett lämpligt kravspec började vi tänka ut hur vi skulle utforma testfallen för uppfylla de kraven vi bestämde oss för. Sedan skrev vi testfallen tills vi var nöjda med dom. Därefter skrev vi källkoden till metoderna som testfallen täckte.

I början när vi först började skriva testfall så gjorde vi en commit efter varje gång vi skrivit ett färdigt och fungerande testfall, men efter hand när vi fått mer erfarenhet av processen kände vi oss säkra nog att skriva längre testfall och kodblock – eller ett par små testfall – mellan varje commit. Vi tyckte att det var lite roligt att se tillbaka och vi märkte hur bara en liten grej som en commit har utvecklats under projektets gång.

Testdriven utveckling – erfarenheter

En diskussion om vilka era erfarenheter ni dragit av att tillämpa TDD. Det finns inget rätt eller fel här. Enda sättet att bli underkända är att bara fuska över punkten och säga något pliktskyldigt.

Jag tyckte att TDD krävde mycket energi eftersom man var tvungen att tänka ”dubbelt så mycket” när man ville göra något. För det minsta lilla var man tvungen att stanna till, tänka ut ett testfall som verkar vettigt och sedan skriva kod. Det tar död på det roliga med koda; att testa sig fram och leka. Jag kan tänka mig dock hur det kan hjälpa till att hålla reda på ett stort projekt, och att motverka regression buggar.

– Elise

Till att börja med måste jag säga att TDD helt tog bort det roliga med att skriva källkod. Att behöva skriva testfall först ströp kreativiteten och det kändes väldigt oflexibelt. Det kändes också lite som att famla i blindo eftersom jag hela tiden försökte hålla både testfallens utformning och den tilltänkta metodens utformning i huvudet innan vi ens började skriva ned testfallen.

Det var väldigt frustrerande i början att skriva testfall tyckte jag eftersom det kändes väldigt baklänges med att först skriva en kod för att testa en metod som inte ens fanns än, istället för att skriva ett kodstycke och sedan köra programmet för att se om det gav ett önskat resultat. Däremot kan jag tänka mig att det kan vara tidssparande och i många fall nödvändigt att skriva testfall på det här sättet beroende på vad det är för typ av program. Väldigt stora program kan det

vara nödvändigt att kunna testa bara delar av programmet på ett mer ordnat sätt, eftersom det skulle bli väldigt svårt och tidskrävande att köra ett stort program bara för att se om en liten hjälpmetod fungerar eller inte.

– Karolina

Arbetet vi hade inom de tre veckorna var krävande och den erfarenheten inom programmering som jag hade bakom mig var till stort hjälp att utveckla med till TDD. I början kändes det onaturligt för mig att skriva ett testfall och sedan skapar metoden som skulle testas men när slutresultatet var nått förstod jag hur viktigt och bra TDD var.

– Aframyeos

Det har verkligen känts bakvänt att använda sig av testdriven utveckling. Man är så van att bara skriva på sina kodrader, kompilera och göra lite formella granskningar emellanåt. Vet inte riktigt om jag skulle tillämpa det i kommande projekt men det är definitivt något man har med sig. Även om de var väldigt bakvänt och irriterande att bara: "nej den här raden kod får jag inte skriva för jag inte har något test ännu". Den stora fördelen av att göra på det här sättet är att man inte skriver mer kod än nödvändigt och gör halvfärdiga saker utan ser till att den bit man skrivit fungerar fristående och sen lägger till bit för bit.

– Anna

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 följande (till och med testmatrisen) ska finnas för samtliga delar ni tillämpat ekvivalensklassuppdelning på.

Vi har valt att arbeta med klassen Discount i vårt "Kvitton & Rabatter" program. Discount innehåller värdet på rabatten, information om hur den hanteras och en metod som räknar ut rabatterad pris. Metoden arbetar med 3 attribut; "minimum": minsta antalet mängd man måste köpa för att få rabatt, "value": värdet på rabatten och "typeFlag": en flagga som säger om rabatten är i procent eller kronor, d.v.s. hur "value" skall hanteras. Vi valde att arbeta med den här klassen för att den har ett bekvämt antal attribut. Vi anser att det mest intressanta i vårt program är tillämpningen av rabatter och därför valde vi att kolla på vاران typeFlag, gränsvärden samt eventuella undantag. Dessutom kände vi att vi ville välja en mer central del av programmet och få mer erfarenhet av just ekvivalensklassuppdelning. Därför tog vi en av de mer komplexa metoderna i våra klasser.

Ekvivalensklasser – namn på del

Samtliga ekvivalensklasser för denna del presenterade på ett tydligt sätt.

Ekvivalensklassindelning av indata för Konstruktorn för klass "Discount".				
Variabel	Valida ekvivalensklasser	Invalida ekvivalensklasser	Gränsfall	Anteckningar
minimum	(e1) >0	(e2) <0	(e3) =0	Rabatt ges oavsätt köpt mängd om minimum=0
value[%]	(e4) >0 && <100	(e5) <0 (e6) >100	(e7) =0 (e8) =100	
value[abs]	(e9) >0 && <y	(e10) <0 (e11) >y	(e12) =y (e13) =0	"y" är priset på varan innan någon rabatt.

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.

ID	minimum	value[%]	value[abs]	Förväntat resultat
t1	8	4		Ok
t2	5		3<y	Ok
t3	-1	4		fail
t4	8	-1		fail
t5	8		-1	fail
t6	8	120		Fail
t7	8		>y	fail
t8	0	4		ok
t9	8	0		fail
t10	8	100		fail
t11	8		=y	fail
t12	8		=0	fail

Testmatris – namn på del

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

x-led = testfall, y-led = ekvivalensklassindelning

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12
e1	x											
e2			x									
e3								x				
e4	x											
e5				x								
e6						x						
e7									x			
e8										x		
e9		x										
e10					x							
e11							x					
e12											x	
e13												x

Tillståndsbaserad testning

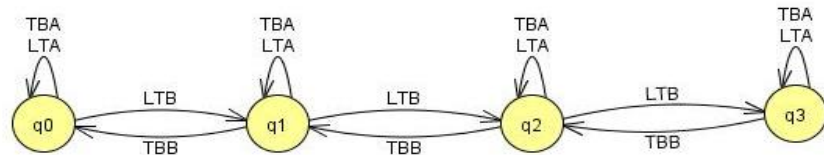
En kort presentation av vad ni valt ut för att tillämpa tillståndsbaserad testning på och vilket täckningskriterium ni valt att använda er av. Ni ska kort motivera valen, och ge tillräckligt med information för att det ska gå att bedöma er. Glöm inte att ta med själva modellen.

Vi har skapat en scenario där varor placeras till och från ett kvitto.

Tillstånden baseras på antalet bananer

som finns i kvittot, 0 till 3 (q0 till q3). Som täckningskriterie har vi valt tillståndstäckning (state coverage) och från det har vi skapat ett testfall. Vi upptäckte snabbt att ett testfall enkelt täckte hela scenariot så vi körde på det.

Vi tyckte att det inte spelade så stor roll vilken del vi valde att applicera tillståndsbaserad testning på. Programmet är ganska litet vilket gör det svårt att få ut någon konkret substans. Vi såg det mer som en möjlighet att testa tillståndsbaserad testning för att få erfarenhet av detta.



LTA = Lägg till annan (Produkt)

TBA = Ta bort annan (Produkt)

LTB = Lägg till banan

TBB = Ta bort banan

Scenario

Först börjar Kalle med tomt kvitto q_0 . Kalle köper ett kaffe paket (LTA), sedan köper han en banan (LTB) och hamnar på q_1 , Kalle ångrar sig och lämnar tillbaka kaffepaketet (TBA) och då är han fortfarande på q_1 .

Kalle köper en tomat (LTA) och då står han fortfarande på q_1 , Kalle köper en till banan (LTB) och då hamnar han på q_2 , Kalle köper en melon (LTA) och är fortfarande på q_2 , Kalle ångrar sig och lämnar tillbaka en banan (TBB) och hamnar han på q_1 , Kalle köper en appelsin (LTA) och är han fortfarande på q_1 , Kalle köper en till banan (LTB) och då hamnar han på q_2 , Kalle ångrar sig och lämnar tillbaka en banan (TBB) och hamnar då på q_1 , Kalle ångrar sig och lämnar tillbaka en banan (TBB) och hamnar på q_0 , Kalle köper en banan (LTB) och hamnar på q_1 , Kalle köper en till banan (LTB) och hamnar på q_2 , Kalle köper en till banan (LTB) och hamnar på q_3 och då är alla tillstånd är täckta.

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.

ID	Beskrivning	Täckta tillstånd
1	<ol style="list-style-type: none">1. Påbörja inhandlandet av varor2. Köp kaffe3. Köp banan4. Ångra kaffe5. Köp tomat6. Köp banan7. Köp melon8. Ångra banan9. Köp apelsin10. Köp banan11. Ångra banan12. Ångra banan13. Köp banan14. Köp banan15. Köp banan	<ol style="list-style-type: none">1. q02. q03. q14. q15. q16. q27. q28. q19. q110. q211. q112. q013. q114. q215. q3

Granskning

En kort presentation av den del av koden ni valt ut för att göra en formell granskning av och processen ni använt er av inklusive eventuella checklistor, scenarier, edyl. Ni ska kort motivera valen, och ge tillräckligt med information för att det ska gå att bedöma er.

Vi började med att sätta oss ned och diskutera hur vi skulle genomföra granskningen, och vi valde att använda en checklista för att identifiera möjliga fel och brister i programmet. Vi tittade runt lite på iLearn2 efter olika checklistor, och i slutändan valde vi denna: https://ilearn2.dsv.su.se/pluginfile.php/26931/mod_resource/content/0/checklist.pdf

Vi valde att använda en checklista mycket för att det verkade vara ett effektivt och enkelt sätt att granska vår kod. Vi beslutade oss också för att granska *hela* programmet - inte bara en del av det -, mest av två anledningar:

Dels består vårt program bara av tre mycket enkla klasser; Receipt, Product, och Discount. Även tillsammans utgör klasserna inte särskilt mycket kod att granska, vilket vi ansåg inte skulle vara mycket till granskning i slutändan.

Den andra anledningen var att vi ville på så många sätt vi kunde förbättra vår kod genom granskningen. Vi ville inte missa eventuella fel som vi kanske annars inte hade upptäckt om vi inte granskat all vår kod.

Efter vi tagit alla nödvändiga beslut för att kunna starta granskningen så diskuterade vi hur vi skulle dela upp oss. Vi är fyra stycken i gruppen och valde till slut att granska koden i par istället för att separera och granska en och en, och ha ett möte efteråt där vi diskuterade vad vi hittade.

När vi alla var klara så träffades vi igen för att diskutera vad vi kommit fram till.

Båda paren märkte ganska omedelbart att listan vi valt var väldigt noggrann och heltäckande - vilket naturligtvis är bra, men på grund av det var det en del av punkterna på checklistan som inte kunde appliceras på vårt program. Detta på grund av att programmet är väldigt litet och enkelt, och därmed saknade saker som arv och dylikt. Under arbetens gång så hade vi ett par diskussioner angående användandet av arv för att implementera vår rabatt-klass, men löste det på ett sätt vi blev nöjda med - så det var därför som programmet blev så litet.

Ingen av paren hittade några fel på koden i sig under granskningen, men alla var överrens till en viss grad om att våra metoder och variabler inte var tillräckligt tydliga för en eventuella användare eller programmerare. Det tog vi en närmare till på och beslutade oss för att ändra några metodnamn och variabelnamn.

Vi upptäckte också att vi behövde ha bättre koll på att få rätt returvärdet, så löste vi det genom att använda oss av assert.

Vi såg också att vi helt hade glömt att använda oss av kommentarer som förklarade vad varje metod och variabel fyller för funktion i programmet, men som nu har lagts till.

Granskningsrapport

En lista över de påträffade felen och hur pass allvarliga ni bedömer dem.

Punkt 7: metoden `getLineSubTotal` i klassen `Receipt` är otydlig, exempel på alternativ: `getItemSubtotal`

Metoden `getMinimum` i klassen `Discount` är otydlig, exempel på alternativ: `getDiscountMinimum`, `getMinimumPurchaseAmount`.

`getValue` : `getDiscountValue`

punkt 8: inga metoder har någon koll på variabel värden, en lösning är att använda assert satser.

Punkt 9: inga metoder har någon koll på returvärden, en lösning är att använda assert.

Punkt 13:

Variabel namnet `"p"` för en `Product`-objekt i metoden `add` i klassen `Receipt` är inte beskrivande.

Variabel namnet `"lineIndex"` för i metoden `getItemSubTotal` i klassen `Receipt` är inte beskrivande.

Metoden `getTotal` i klassen `Receipt` har variabel namn som `"e"` och `"p"`, dessa är inte beskrivande.

Instance variabel namnet `"minimum"` är inte beskrivande.

value : discountValue

type : typeFlag

punkt 43 till 51:

kommentarer existerar ej.

Granskning – erfarenheter

En diskussion om vilka era erfarenheter ni dragit av att tillämpa granskning. Det finns inget rätt eller fel här. Enda sättet att bli underkända är att bara fuska över punkten och bara säga något pliktskyldigt. Ni förväntas förhålla er till såväl kursboken som utdelat material och IEEE Std 1028.

Vi tycker att granskning är bra generellt sett, en formel granskning känns dock väldigt ”tungt”; man måste förbereda sig och det finns protokoll som man ”måste” följa. Andra former av granskning tycker vi att är mycket trevligare, som t.ex. en walkthrough, man kan utföra den nästan när som helst vilket ger upphov till spontana granskningar. Vi tycker dock att listan som vi använde i våran formella granskning var bra, den fungerade som en lathund med saker som är bra att tänka på som man kanske missar annars.

När vi arbetat med våran kod har vi använt oss av en slags ickeofficiell granskning, en variant på parprogrammering: ”grupprogrammering”. Varje gång när vi skulle skapa testfall eller skriva själva metoderna diskuterade vi om hur man skulle kunna göra, vilka lösningar som var lämpligast och mest funktionsdugligt i förhållande till vårt program. I stort sätt allt vi gjort i hela projektet har diskuterats väldigt flitigt fram och tillbaka och vi har byggt väldigt mycket på varandras idéer. Vi har dragit nytta av varandras styrkor och svagheter för att till exempel förhindra eventuella misstag som kunde gjorts och skapa en bättre källkod med bättre struktur och funktionalitet.

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.

Vi använder oss av **FindBugs** som kodkritiksystem. Vi ställde in den på lägsta nivå (striktaste) så att den klagar på det lilla minsta. Systemet rapporterade inga fel. Vi har kört FindBugs på koden innan granskningen och på den senaste versionen.

Statiska mått

En presentation och diskussion kring ett antal lämpliga statiska mått på koden. Att vi inte specificerar exakt vilka mått som ska tas upp beror på att olika verktyg har olika uppsättningar, men vi förväntar oss fler och mer intressanta mått än bara rena storleksmått som LOC, #klasser, #metoder, etc. Även här är det viktigt att förhållas sig till måtten, inte bara lista dem.







Vi använde oss av Metrics för eclipse för att ta fram statiska mått för vårt system. Ett intressant mått är "Lack of Cohesion of Methods". Vi fick ett medelvärde på 0.4. Ett lågt värde innebär att vi har hög cohesion, vilket är önskvärt. Hög cohesion innebär att vi har bra koppling mellan metoderna och dess attribut.

"Efferent Coupling" är ett mått på hur mycket/många saker något har kunskap om. Vi fick ett värde på 3 för att vi har 3 klasser i projektet.

Täckningsgrad

En översikt över vilken täckningsgrad era testfall uppnått. Denna kan antagligen tas rakt av från verktyget ni använt för att mäta den. Om ni inte uppnått fullständig täckning så ska detta förklaras och motiveras.

Vi valde statement coverage som täckningsgrad. Vi har uppnått 100% täckning. Vi använde oss av EcEmma för att se vår coverage.

▷  Discount.java	 100,0 %	23	0	23
▷  Product.java	 100,0 %	22	0	22
▷  Receipt.java	 100,0 %	48	0	48

Profiler

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

Vår projekt är inte så lämplig att profilera eftersom programmet körs och avslutas omedelbart; den sitter inte och väntar eller arbetar på något. Vi profilerade programmet genom att skapa ett litet `main()` metod där vi skapar några produkter & rabatter och lägger dom i ett kvitto. För att profilera programmet lägger vi en brytpunkt vid sista kod satsen, kör programmet i debug mode och låter den sitta där medan vi utforskar resultaten från profilern. Vi använder oss av profilern "JVM Monitor".

Det finns inte så mycket intressant att titta på; vi har en tråd, som i vårt fall är pausad och använder i princip ingen CPU, det mest intressanta är väll minnet. Här är några highligts: Högst upp på listan har vi `char[]` som använder 456KB minne, vi har 1521 stycken `Class` pekare som använder 183KB minne, `String` pekare har vi 6575 stycken av och de använder 157KB minne.

Vi använder oss av `double` till mycket: värdet för pengar, värdet för kvantitet av produkt osv. I listan hittar vi 4 st `double[]` och det tar upp 336 bytes!

Byggsript

Byggsriptets första (seriösa) version, och den slutliga.

Här följer vår första fungerande byggsript (blev även vår slutgiltiga).

```
<?xml version="1.0" encoding="UTF-8"?>
<project name="InteProj" default="runjunit">
  <target name="runjunit" depends="compile" description="Run Forest! Run!">
    <junit printsummary="on">
      <test name="inteproj.AllTests"></test>
      <classpath>
        <pathelement location="./resources/junit.jar"></pathelement>
        <pathelement location="./resources/hamcrest.jar"/>
        <pathelement location="./compiled"></pathelement>
      </classpath>
      <formatter type="plain" usefile="false"/>
      <formatter type="plain"/>
    </junit>
  </target>
  <target name="compile" description="Do the macarena!">
    <javac srcdir="./src;./test_src" destdir="./compiled" includeantruntime="false"
      classpath="./resources/junit.jar"></javac>
  </target>
</project>
```