Kodkomplexitet - Hur mäts det?

Sara Nilsson
D05, Lunds Tekniska Högskola
dt05sn2@student.lth.se

1 mars 2011

Sammanfattning

Den här rapporten är tänkt för studenter med programmeringsvana och åtminstone viss kunskap om Eclipse och eXtreme Programming (XP). Motivet till rapporten är att ge en ökad förståelse för vad kodkomplexitet är och hur man kan räkna ut hur komplex den kod man arbetar med är. Tanken är också att hitta visa på några verktyg och metoder som kan hjälpa till för att upptäcka och göra koden mindre komplex. Slutligen jämförs även verktygen mmellan vanlig och agil utvecklingsmiljö, för att se om det är samma verktyg som är användbara i bägge fallen.

Innehåll

1 Inledning

De flesta som någon gång har programmerat har också någon gång konstaterat att från att vara fin välstrukturerad kod så går den långsamt över mot något komplext och rörigt. De flesta har nog hört begreppet kodkomplexitet, men hur många vet egentligen vad det innebär riktigt, och hur många vet hur man kan beräkna det och använda det? Det är det här jag vill fokusera på med den här rapporten. Hur man beräknar kodkomplexitet och hur man använder de resultat man får fram. Resten av rapporten kommer att struktureras som följer.
Kapitel 2 ger en inblick i teorin kring Kodkomplexitet och fortsätter sedan vidare till att beskriva några olika sätt att mäta kodkomplexiteten för ett program.
Kapitel 3 ger en utvärdering av två eclipseplugins som kan mäta kodkomplexitet.
Kapitel 4 sammanfattar vad jag kommit fram till och ger en liten slutsats.

2 Teori

Efter hand som våra program växer i både komplexitet och storlek blir kraven större och större på ett sätt att förstå och kommunicera hur stora de egentligen är. [7]

Eftersom människor är olika och det vi försöker mäta är hur komplext en människa tycker det här programmet är så kommer vi aldrig kunna få ett exakt värde eller en exakt mätteknik. [10] Ddäremot finns det en del generella tumregler man kan undersöka efter och få ett ungefärligt värde.

2.1 Mätning av hela program

Funktionspoäng Utifrån designen kan man räkna ut hur många funktionspoäng ett program har, dessa kan sedan användas för att räkna ut hur stort programmet kommer bli, hur mycket arbete som kommer krävas och hur lång tid det kommer att ta.

Vi människor bryter ner problem i små delar och löser delproblem, men om problemet som ska delas ner är ett helt program kan detta bli svårt. [7] Då är funktionspoäng ett sätt att hjälpa till att bryta ner problemet.

Funktionspoäng är ddäremot inte särskilt bra att mäta med om det du vill se är hur mycket arbete det behövs för att underhålla koden (buggfixar, prestandaproblem osv.). En stor del av underhållsarbetet är att leka detektiv och räkna ut vad problemet är.

Antal rader kod En av de vanligaste metoderna att mäta ett programs komplexitet är just dess storlek. [10] Det stämmer ju att ett större program är svårare att sätta sig in i och blir mer komplext, men att bara räkna antalet rader är inte en så exakt metod om man vill jämföra mmellan olika språk. [7] Skriver man samma program i 2 olika språk kommer antalet funktionspoäng vara detsamma, men antalet rader kod kommer säkerligen att vara olika, speciellt om man väljer ett språk som är delvis grafiskt och ett som är rent textbaserat. Här är ett exempel på hur stor skillnad det kan vara mmellan 2 olika språk, i det här fallet Java och Scala:

Java [3]

```
public static void quick_srt(int array[],int low, int n){
    int lo = low;
    int hi = n;
    if (lo >= n) {
      return;
    }
    int mid = array[(lo + hi) / 2];
    while (lo < hi) {
      while (lo<hi && array[lo] < mid) {
        lo++;
      }
      while (lo<hi && array[hi] > mid) {
        hi--;
      }
      if (lo < hi) {
        int T = array[lo];
        array[lo] = array[hi];
        array[hi] = T;
      }
    }
    if (hi < lo) {
      int T = hi;
      hi = lo;
      lo = T;
    }
    quick_srt(array, low, lo);
    quick_srt(array, lo == low ? lo+1 : lo, n);
  }
}
```

Scala [4]

```
<<ListQuicksort1>>=

def qsort[T <% Ordered[T]](list:List[T]):List[T] = {
  list match {
  case Nil => Nil
  case x::xs =>
    val (before,after) = xs partition (_ < x)
    qsort(before) ++ (x :: qsort(after))
  }
}
```

Antalet rader kod beror också på andra faktorer, som programmerarens erfarenhet, metod att räkna på osv. Det absolut sämsta med metoden är att man vet aldrig svaret på hur många rader kod det blir förrän man är klar eller nästan klar.

2.2 Mätning av metoder

Det kan vara intressant att mäta komplexiteten för en viss metod, och för detta finns det många olika sätt att mäta på. [10] Jag väljer här att ta upp några av dem.

Antal rader kod Antalet rader kod kan ge ett mått på när man borde se sig om i sin metod och börja fundera på att bryta ut den i flera små i stället. Ett vanligt råd är att man aldrig ska ha mer än en sida för en metod [10]. Att det är just en sida är för att man ska kunna se hela metoden, man ska kunna se allt relevant i en metod utan att behöva scrolla. Därför är också måttet en sida varierande med upplösning, teckensnitt osv osv.

Nästling Ytterligare en bra riktlinje för programmering är att inte nästla mer än nödvändigt. Ju fler nivåer man har på koden desto svårare är det att läsa den.

Flöde En väldigt viktig del av den totala komplexiteten för en metod är flödeskomplexiteten, det finns 2 vanliga sätt för att mäta den. Cyclomatic Complexity 2.3 där man räknar hur många branches man gör och NPATH Complexity 2.4 som räknar alla möjliga exekveringsvägar genom en metod. Det innebär att man bland annat använder nästlingsdjupet och komplexiteten för de booleska variablerna.

2.3 Cyclomatic Complexity

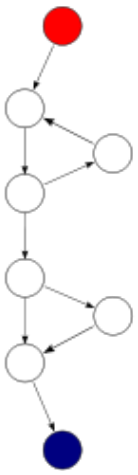
Cyclomatic Complexity är en metod för att räkna ut hur svårt ett program är att testa och underhålla. [8] [11] Det går i praktiken ut på att räkna ut hur många olika vägar man kan ta genom ett program, även om det i vissa program är totalt omöjligt för att det finns ett oändligt antal varianter. Därför har också beräkningen av komplexiteten ändrats till att räkna grundläggande vägar i stället för alla vägar. Det finns 2 olika formler för att beräkna cyclomatic complexity, vilken man ska använda beror på om slutnoden knyter an till startnoden Figur 2 [1] eller inte Figur 1 [2]. Om grafen ser ut som i Figur 2 så ska man använda formeln $v(G) = e - n + p$ där v(G) är cyklomatic complexity, e antalet bågar i grafen, n antalet noder och p antalet anslutna komponenter. Om vi tar Figur 2 som exempel så gäller följande värden $e = 9$, $n = 8$ och $p = 1$. Om vi sätter in värdena i formeln får vi svaret 3, vilket innebär att det räcker med 3 sätt att gå igenom grafen för att kunna uttrycka alla sätt att gå igenom grafen.

Ser ddäremot grafen ut som i Figur 1 gäller en annan formel, nämligen $v(G) = e - n + 2p$ Om vi tar Figur 1 som exempel så gäller följande värden $e = 10$, $n = 8$ och $p = 1$. Om vi sätter in värdena i formeln får vi svaret 3, vilket innebär att det räcker med 3 sätt att gå igenom grafen för att kunna uttrycka alla sätt att gå igenom grafen.

Vill man inte skriva en graf kan man räkna antalet vilkorssatser och lägga till 1, dock får man inte glömma att t.ex.

[1] http://en.wikipedia.org/wiki/File:Control_flow_graph_of_function_with_loop_and_an_if_statement.svg

[2] http://en.wikipedia.org/wiki/File:Control_flow_graph_of_function_with_loop_and_an_if_statement_without_loop_back.svg



Figur 1: Det här är en kontrollgraf med ingångsnoden a och utgångsnoden f [1]

```
if(a && b){
    ...
}
```

räknas som 2 eftersom det är samma sak som att skriva

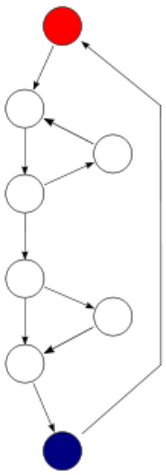
```
if(a){
    if(b){
        ...
    }
}
```

Om man vill kunna testa sina metoder på ett bra sätt bör man inte ha en cyclomatic complexity högre än 10, kommer man upp i 10 bör man refaktorisera eller skriva om sin metod. Undantaget är om man har en stor case-sats eller liknande.

Structured testing Structured testing är ett sätt att mäta täckningen på dina tester, om man testar ett basset av vägar så är alla andra vägar täckta av tester eftersom de består av olika varainter av just de tester du redan gjort.

2.4 NPATH Complexity

NPath Complexity mäter hur många icke cykliska vägar det finns i ett program, och underlättar för att räkna ut hur många testfall man behöver [9]. Genom att skriva kod med låg NPath-komplexitet får man en kod som är mycket lättare att testa. Precis som för Cyklomatic Complexity 2.3 finns problemet at det ubland



Figur 2: Det här är en kontrollgraf med ingångsnoden a och utgångsnoden f [2]

finns oändligt antal möjliga vägar att gå eftersom man kan ta olika antal varv i looparna.

Vad NPath försöker göra är att hitta en förbättrad lösning av Cyklomatic Complexity, där formlerna kan variera allt från en linjär lösning till en exponentiell funktion. Förutom det så kan antalet vägar genom grafen som inte kan testas varierar mmellan 0 och 2^N, där N är antalet bågar i grafen. Ett annat problem är att Cyclomatic Coplexity behandlar alla typer av satser likadant, oavstt om det är en while, if for eller annan. Även så ses det ingen skillnad på att ha 3 for-loopar i rad eller at ah 3 nästlade for-loopar, men en nästlad for-loop kan kännas mycket mer psykologiskt komplex än två loopar i rad. Många forskare är överens om att psykologisk komplexitet har större inflytande på kodkvalitén [9]. Genom att utveckla NPath Complexity har man försökt lösa dessa problem.

NPath gjordes från början med tanke på C, men kan även användas på andra programmeringsspråk.

if Formeln för att räkna ut komplexiteten för en if-sats är $NP_{if} = NP_{if-range} + NP_{else-range} + NP_{expr} + 1$, där $NP_{if-range}$ är komplexiteten för den kod som körs om if-satsen är true, NP_{expr} är komplexiteten för uttrycket somstyr if-satsen. $NP_{else-range}$ är endast med i uttrycket om det finns en else-sats, i så fall är det komplexiteten för den kod osm körs om man går in i else-satsen.

while Formeln för att räkna ut komplexiteten för en while-sats är $NP_{while} = NP_{while-range} + NP_{expr} + 1$, där $NP_{while-range}$ är komplexiteten för den kod som är inom while-satsen och NP_{expr} är komplexiteten för utrrycket som styr while-satsen. Om man har en do-while-sats ändrar man bara formlen så att det som står i do-satsen utgör värdet för $NP_{while-range}$.

for Formeln för att räkna ut komplexiteten för en for-sats är $NP_{for} = NP_{for-range} + NP_{expr1} + NP_{expr2} + NP_{expr3} + 1$, där $NP_{for-range}$ är komplexiteten för den kod som är inom for-satsen, NP_{expr1} är komplexiteten för det första deluttrycket, NP_{expr2} det andra deluttrycket och NP_{expr3} det sista deluttrycket.

switch Formeln för att räkna ut komplexiteten för en switch-sats är $NP_{switch} = NP_{expr} + NP_{default} + \sum_{i-1}^{i=n} NP_i$, där NP_{expr} är komplexiteten för uttrycket som styr switch-satsen, $NP_{default}$ är den kod som körs om man går in i default och NP_i är komplexiteten för det i:te caset i switch-satsen och n är antalet cases. Om ett case är tomt eller faller vidare till ett annat så räknas det som att det har komplexiteten 1.

break Ett break beräknas alltid ha komplexiteten 1.

return Eftersom return kan komma tillsammans med ett uttryck så är formeln för att beräkna komplexiteten $NP_{return} = NP_{expr}$, där NP_{expr} är uttrycket som står tillsammans med return.

expr Ett expression (&&, ||) måste ju då också kunna beräknas, annars kommer vi aldrig få något resultat. Dock är detta ganska enkelt, komplexiteten är antalet operatorer.

Totalt ger detta att NPath Complexity beräknas genom $NPATH = \sum_{i=1}^{i=n} NP_i$, där$NP_i$ är komplexiteten för det i:te statsmentet i koden. Tänk på att den tatala komplexiteten alltid är produkten av komplexiteterna för alla statements. Nedan kommer ett exempel:

```
if ( ch == ‘a’ ){
actrff;
Func-a( ) ;
}
if ( ch == ‘b’ ){
bctr++;
Func-b( ) ;
}
if ( ch == ‘c’ ){
cctr++;
Func-c ( ) ;
}
if ( ch == Id’ ){
dctr++;
Func-d( );
}
```

Eftersom alla if-satser består av kontinuerlig kod, är if-range = 1 för dem alla. Eftersom expr inte innehåller varken && eller || så är den lika med 0. För att få ut komplexiteten för if-satserna tar vi således $1 + 0 + 1 = 2$. För att få ut den tatala komplexiteten tar vi sedan $2 * 2 * 2 * 2 = 16$

3 Tester

För att göra det lättare att kunna testa programmen oavsett plattform valde jag att testa 2 plugins till Eclipse. Versionen av Eclipse jag har använt är Helios.

3.1 Metrics

Den version av Metrics [3] jag provat är 1.3.6

3.1.1 Installation

Metrics installeras genom att man går in på Help → Install new Software. Välj att installera från `http://metrics.sourceforge.net/update`. Följ instruktionerna på skärmen och starta om Eclipse.

3.1.2 Körning

Innan du kan börja använda Metrics måste du se till att du står i Java-perspektivet, högerklicka på det projekt du vill köra metrics på och välj Properties. Välj metrics och se till att rutan med enable är ibockad.

För att börja använda Metrics välj Window → Show View → Other → Metrics → Metrics View.

Först kommer du se ett kort felmedelande eftersom projektet inte har körts ännu, kör projektet en gång så kommer data att samlas och du får en tabell som ser ut som Figur 3. Alla rader som är blå kan dubbelklickas på så kommer man direkt till den kod som har det maximala värdet för den rad du klickat på. Sedan version 1.2.0 finns en trädstruktur i listan så du kan själv välja hur långt ner du vill gå. I trädet sorteras alla barn i fallande ordning med högst värde först.

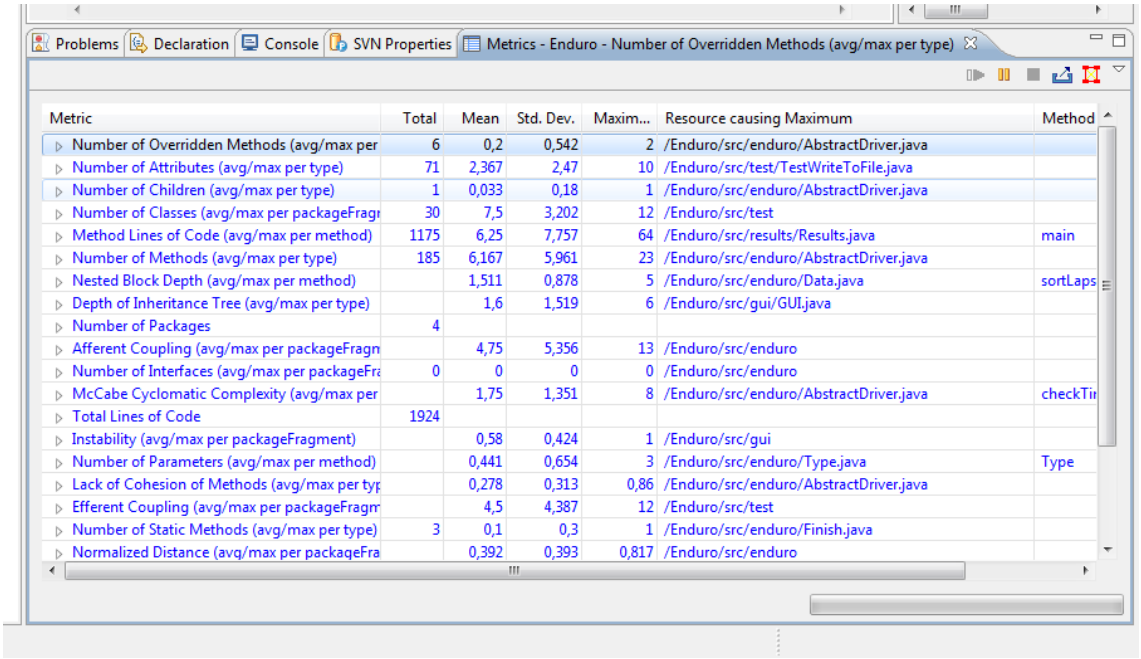
Man kan också exportera informationen till en XML-fil, detta gör man genom att trycka på knappen Export XML Figur 4. Välj ett passande filnman och tryck spara.

En annan funktionalitete det finns i programmet är att man kan göra en Dependency Graph, detta gör man genom att Välja det paket man vill göra den för och trycka på knappen Open the Dependency Graph View Figur 5. Ett exempel på hur Dependency Graph kan se ut finns i Figur 6. Alla paket som har ett cykliskt beroende färgas röda så man kan se vilka det är, övriga paket färgas blå. Den lilla gula cirkeln berättar hur många paket som ingår i cykeln och hur lång den längsta vägen runt är. Om man högerklickar på den gula cirkeln kan man välja Analyze Details som då använder Eclipse Search Engine för att räkna ut vilka underpaket eller klasser som ingår i cykeln. Genom att klicka på olika filer eller paket kan man få dem i centrum och se just deras beroenden.

3.1.3 Fördelar

Verktyget är enkelt att använda, och grafer och liknande är väldigt tydliga. Sorteringen efter mest komplex först gör det lätt att följa trädet ner och se var man först ska försöka förbättra sin kod.Att verktyget är helt gratis och inte gör allt för stora förändringar av eclipse i grunden är också väldigt positivt.

[3] http://metrics.sourceforge.net/



Figur 3: Ett exempel på hur Metrics View kan se ut

Figur 4: Knappen för att skapa en XML-fil

3.1.4 Nackdelar

Den massiva mängden information i grundfönstret kan vara ganska förvirrande, man vet inte alltid vad man ska titta på först eller vad som är ett allvarligt fel och vad som bara är normalt. Pluginen är inte alltid så intuitiv heller, vissa saker är väldigt svåra att komma på hur man gör om man inte läst manualen noga först.

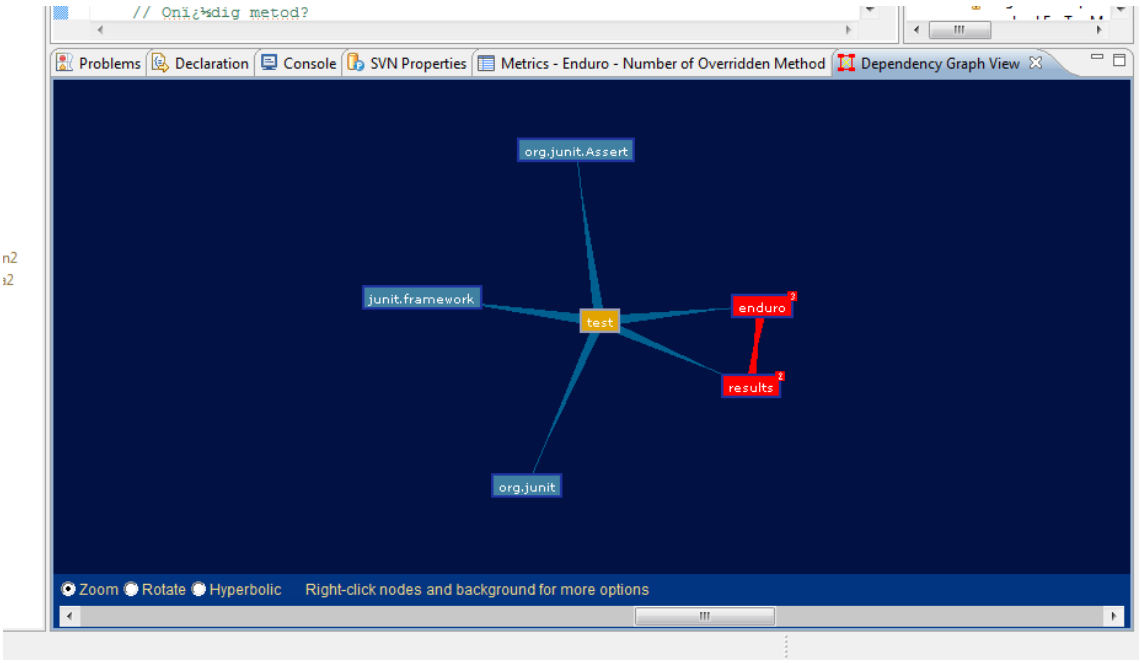
3.2 Clover

Den version av Clover [4] jag provat är 3.0.2.v20100413110000.

Eftersom Clover så pass stor funktionalitet som den har väljer jag att bara berätta om vissa delar, för en fullständig beskrivning av all funktionalitet referear jag till pluginens egen hemsida, se fotnoten ovan.

[4] `http://confluence.atlassian.com/display/CLOVER/Clover-for-Eclipse`

Figur 5: Knappen för att visa Dependency Graph



Figur 6: Ett exempel på hur Dependency Graph View kan se ut

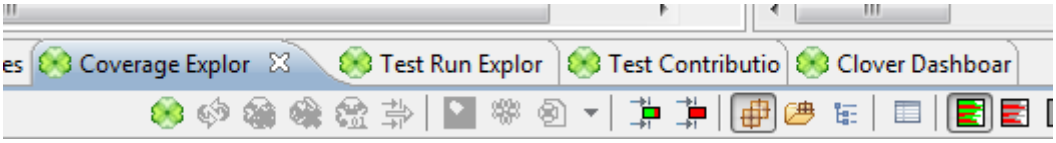
3.2.1 Installation

Metrics installeras genom att man går in på Help och väljer "Install new Software"i Eclipse. Välj att installera från `http://update.atlassian.com/eclipse/clover`. Följ instruktionerna på skärmen och starta om Eclipse.

3.2.2 Körning

För att aktivera Clover gör du så här: Högerklicka på projektet och välj *rightarrow* Clover → Enable / Disable on. Välj de projekt du vill aktivera Clover för. När du aktiverat Clover kommer det automatiskt upp 4 nya Flikar Figur7.

Coverage Explorer visar hur bra dina tester täcker ditt projekt, här kan du även välja att ändra inställningarna för hur Clover ska hantera dina olika projekt Figur 8. Du kan även se en färgkodning i koden, denna visar hur pass täckt din kod är, siffran ute till vänster visar hur många gånger just den kodraden körts.



Figur 7: De 4 flikarna som utgör den stora delen av Clover

Test Run Explorer visar precis som JUnit vilka tester som gått igenom och inte, men tar även ett steg längre och låter en se hur stor del av koden som täcks av dessa tester Figur 9. Om man väljer ett test på vänster sida ändras vyn på höger sida så man kan se hur stor del av de olika klasserna just det testet täcker.

3.2.3 Inställningar

I Clover finns det stora möjlighheter att anpassa vad man vill se i sin Coverage Explorer. För att anpass trycker du på knappen Choose Columns to Display Figur 10.

Om inte den metod för komplixiteteberäkning man vill ha finns så finns det även möjlighet att göra egna formler. Tryck bara på ny i Column Chooser så hoppat ett fönster du kan skriva in din egen formel i upp Figur 11.

3.2.4 Fördelar

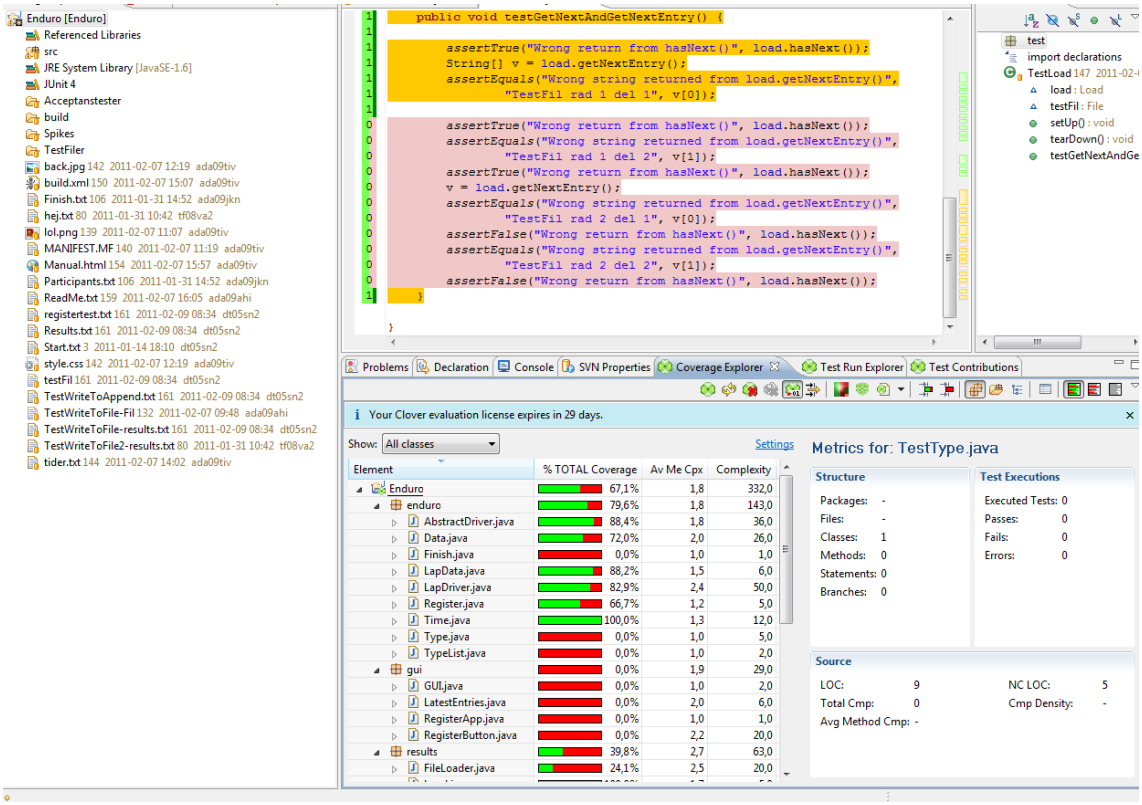
Fördelarna med Clover är att det går att anpassa, är väldigt snyggt och lättanvänt och ger både JUnits funktioner och komplexitetsberäkningarna i samma fönster. Färgkodningen och numreringen av rader i det vanliga fönstret gör det väldigt lätt och överskådligt att se vad man lyckats testa och inte.

3.2.5 Nackdelar

En nackdel med Clover är att om man vill ha mer än 30 dagar trial så måste man köpa en licens för verktyget. Det är svårt att hitta inställningsfönstret och när man väl kommer dit finns inte så mycket information.

4 Slutsats

Det finns massor av olika sätt att mäta kodkomplexitet, vilket som är bäst beror mycket på av vilken anledning man vill mäta komplexiteten. Är det för underhåll kan Cyclomatic Complexity vara bra, så man kan se hur lättestad koden är, och hur mycket arbete som krävs för att få testningen heltäckande. Är det däremot före ett projekt kanske Funktionspoäng kan vara en bra metod att använda eftersom den är densamma oavsett språk och kodningsstil. Då kan man lättare planera tidsåtgång osv. Det kommer ständigt nya sätt att mäta på



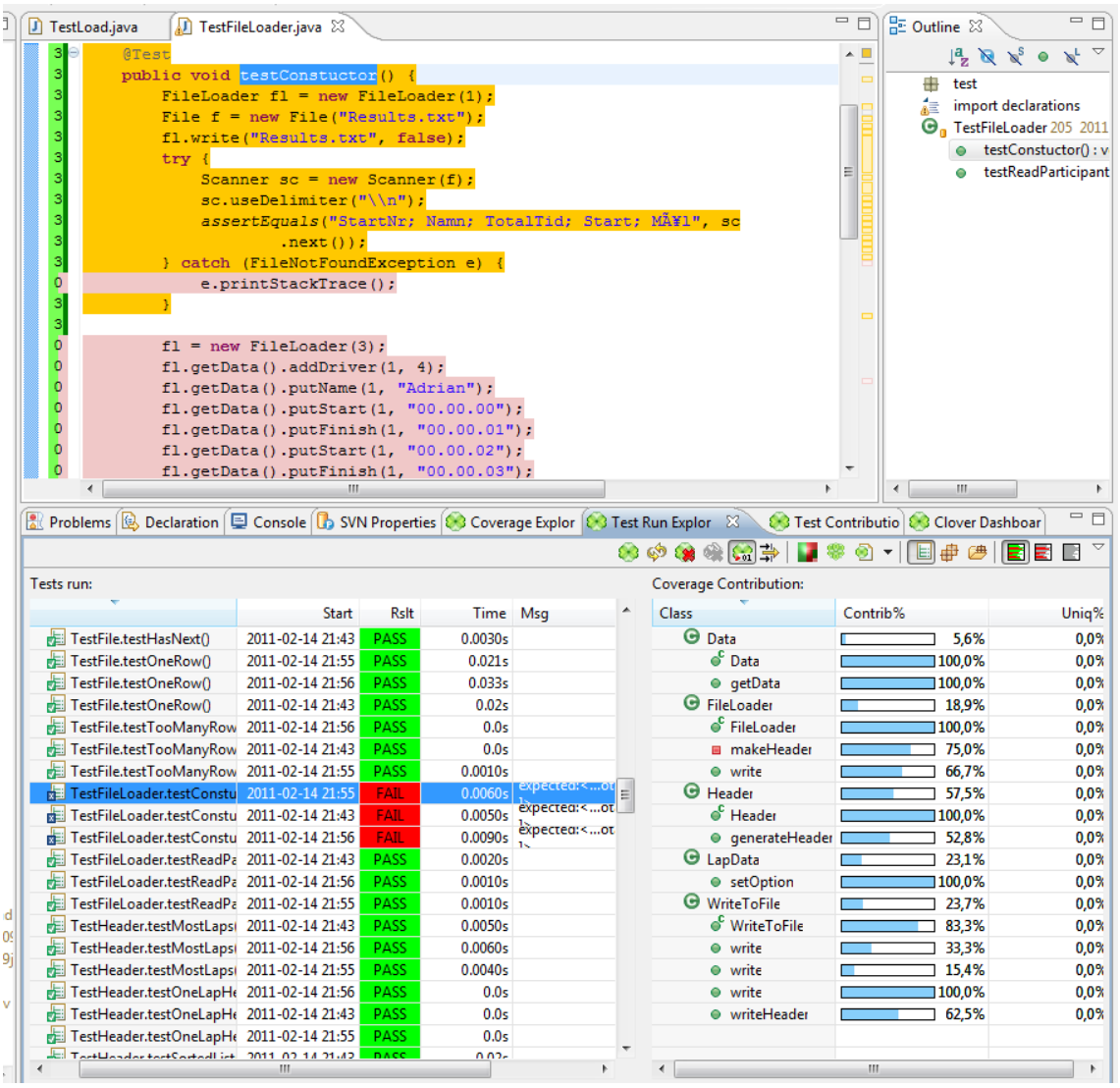
Figur 8: Ett exempel på hur Coverage Explorer kan se ut

så i framtiden finns det säkerligen ett som kan fungera bra både för underhåll, under utvecklingen och före utvecklingen.

Programmen jag testade var bägge av bra kvalitet, Metrics verkade välanvänt när jag läste om det på nätet. Tyckte dock själv bättre om Clover eftersom det var mycket lättare att använda och gav mycket bättre feedback till användaren. Dock tycker jag inte det är värt priset 300$ för en licens, så mycket bättre än Merics är det inte. Metrics nackdelar var att det inte hade lika bra feedback till användaren och att det kräver en djupläsning av manualen för att använda. Metrics lär fortsätta bli bättre, det har kommit stora förbättringar för varje version innan och det lär fortsätta så.

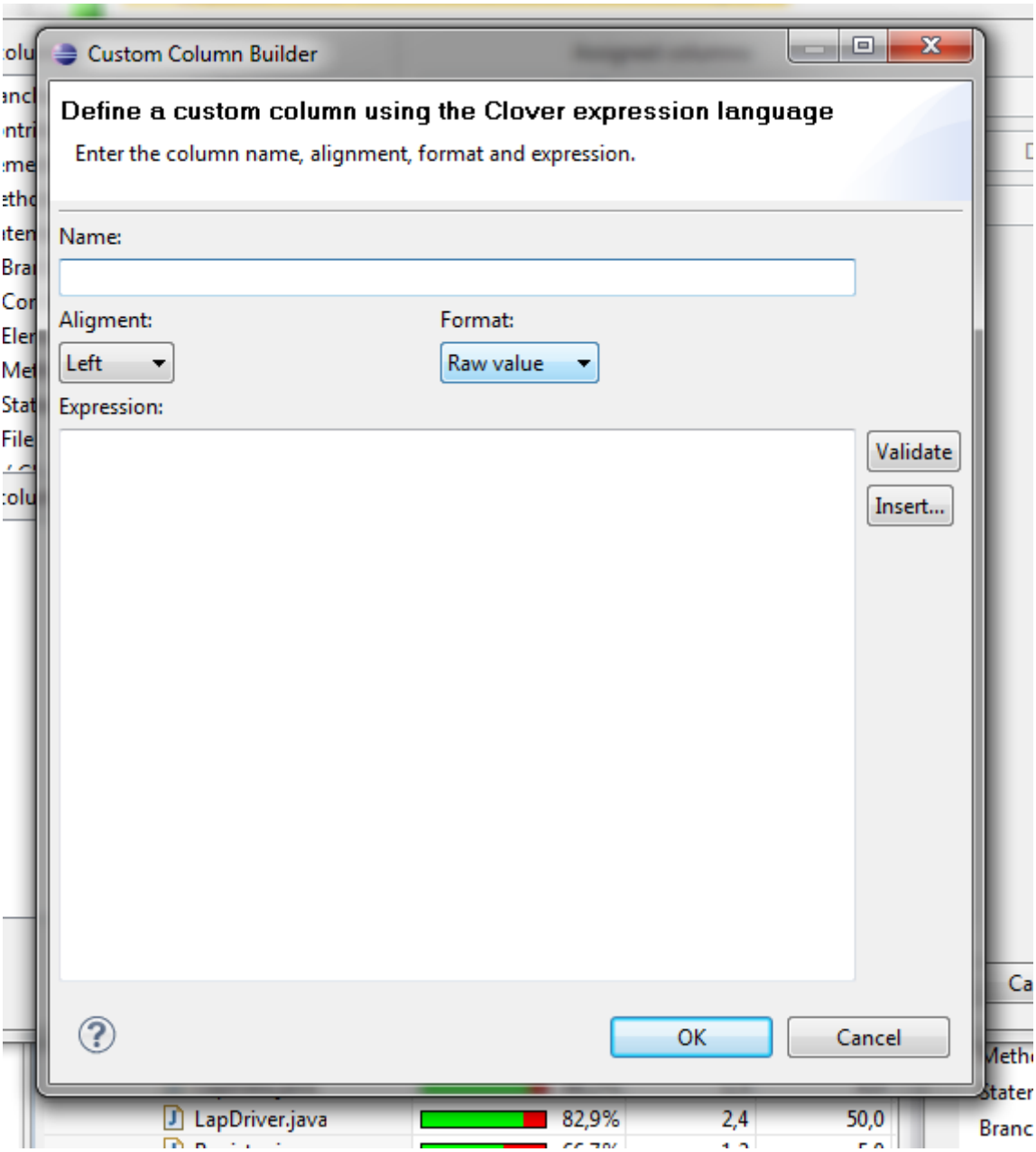
Om det hade funnits mer tid hade jag utökat antalet verktyg och tittat även utanför Eclipsepluginer, det finns säkerligen en värld av bra verktyg där ute det gäller bara att hitta dem. Jag hade också försökt fördjupa mig mer i vad som är riktigt nytt i det teoretiska området av kodkomplexiteten, finns det nyare algoritmer, är det något nytt på väg som kanske kan användas under hela processen? Finns det några enkla metodiker som inte inkluderar någon extra

programvara? Jag hade också gjort större undersökningar av de verktyg jag valde ut, kanske bett några andra personer testa dem för att se hur de upplevde dem, och testat dem på fler kodbaser, nu blev det bara på en enda. Jämförelsen mellan agil och vanlig utveckling råkade också hmana mmellan stolarna, det hanns inte riktigt med.



Figur 9: Ett exempel på hurTest Run Explorer kan se ut

Figur 10: Knappen för att anpassa Coverage Explorer



Figur 11: Fönster för att skriva in egna formler och liknande för komplexitetsmätning

Referenser

[1] http://en.wikipedia.org/wiki/File:Control_flow_graph_of_function_with_loop_and_an_if_statement_without_loop_back.svg,.

[2] http://en.wikipedia.org/wiki/File:Control_flow_graph_of_function_with_loop_and_an_if_statement.svg.

[3] Quick sort in java. /urlhttp://www.roseindia.net/java/beginners/arrayexamples/QuickSort.shtml.

[4] Quicksort (scala). http://en.literateprograms.org/Quicksort_(Scala),.

[5] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1994.

[6] David L. Lanning and Taghi M. Khoshgoftaar. Modeling the relationship between source code complexity and maintenance difficulty. *Computer*, 27:35–40, September 1994.

[7] David Longstreet. *Function Points Analysis Training Course*. Longstreet Consulting Inc.

[8] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[9] Brian A. Nejmeh. Npath: a measure of execution path complexity and its applications. *Commun. ACM*, 31:188–200, February 1988.

[10] Fred Swartz. Java: Complexity measurement, 2006. http://leepoint.net/notes-java/principles_and_practices/complexity/complexity_measurement.html.

[11] Arthur H. Watson and Thomas J. McCabe. Structured testing: A testing methodology using the cyclomatic complexity metric. NIST Special Publication 500-235, National Institute of Standards and Technology, Computer Systems Laboratory, NIST, Gaithersburg, MD 20899-0001, 1996.