Blind mans bluff

“Using automated metrics to help track progress ”

By Dean Margerison dean@dekam.net

www.dekam.net

Introduction

Companies involved in software development traditionally spend a significant amount of time, effort and heartache trying to ascertain how a particular project is progressing. Even with the benefit of having the development team located in the same building our success rate is poor, more often than not the experience has much more in common with a hi tech game of blind mans bluff than a measured and repeatable means of visualising progress. Looking ahead to ever-increasing complexity and the greater use of external development teams the picture can look very bleak indeed.

Thankfully the last few years has seen the development of a number of tools and techniques that can offer a quantum shift improvement in this area. This white paper provides a brief insight into the current problems that many projects face, and what can be done to transform the situation for the better.

Our current challenges

The most common means of tracking progress can be characterised by the following: -

One member of management will be assigned the task of collecting progress data and disseminating this to a wider audience (generally senior management). This person can be the hands on manager of the team or a dedicated project manager who handles a number of projects at the same time.

- They will talk to the development team members individually to gauge what progress has been made. This might be with the aid of a task list generated from the project plan.

- During the conversation the manager will try to find out the developers feeling for 'how well it's going'. Based on experience and knowing the individual developer they themselves try to build up a picture of how well the final delivery is progressing.

- They then write a progress report that includes edited highlights of the conversations with the team, amended to include the manager's educated guess as to how things are actually going.

- Usually some form of project plan or Gantt chart is updated to compliment the progress report, these are then e-mailed out or saved to a shared folder.

The difficulty of achieving the above when the development is being done by a third party can at times make the whole process feel more like playing blind mans bluff and Chinese whispers at the same time.

Using the above methods of visualising how projects are doing has had an extremely patchy success record, one which can be understood if we look at some of the major drawbacks inherent in this process: -

- Progress can appear to be going fine in so far as all the allocated tasks are being completed but the project plan does not reflect the body of work that really needs to be completed.

- Being primarily a subjective process it relies heavily on a complex blend of skills to collect, collate and translate the information gained into an accurate picture.

- The process is time consuming and can constitute a significant overhead on the individual team members.

- Team members do not see any direct benefit to themselves of going through the process and tend to view it as 'getting in the way of their actual work' if you are lucky, or as a 'management sponsored spying mission' if you are not.

- There is no real measure for the quality of what is being produced

- You cannot really see ‘who’ is doing the work
- You cannot measure progress over time, to gauge trends or patterns; ie we have added 3 more developers but are we actually doing more?
- This type of process is almost impossible to replicate from one project to the next, as its success is based on the skills and experience of a very small number of people.

Fact based visualisation

“Taking advantage of the readily available facts that are currently being ignored can transform our understanding of development progress to a level that most senior developers aspire”. Making this level of understanding readily available to the whole team and it’s managers provides us with the timely facts on which to base decisions, and also to see what impact these decision have on progress.

In order to achieve the quantum shift in our capability to visualise progress there are 6 key guiding principles

1. Metrics must be based on facts
2. The facts should have a historical element to track progress over time
3. An automated process should be in place to extract/transform and deliver the metrics
4. When statistics are gathered from several sources they should be delivered to the end user via one mechanism. The current preferred delivery mechanism is via the web in HTML
5. You must be able to re-use metrics in planning new projects and comparing progress between projects.
6. The tools used to achieve the above should be easy to install and maintain

Looking at some of the available tools and techniques we can investigate how the guiding principles can be practically achieved

Source Control Management (SCM) metrics

Linking into a project Source Control Management (SCM) tool is a vital element in unlocking what is actually being achieved by a project. As developers go about their daily tasks they are continually updating the contents with their SCM tool[1]. As this information builds up over time it is generally totally ignored .The view being that 'SCM tools are really only there to protect and manage the source code?'. By taking this view we throw away the biggest source of concrete facts that can be used for tracking purposes, namely: -

- **Lines of code and it's growth over time**
- **No of files and their growth over time**
- **Which developers are active?**
    - What % of the code base they have contributed
    - In what areas they are working

There are a number of tools on the market right now. There are three worth mentioning in the context of this paper; CVSMonitor, StatCVS and Bloof, all are Open Source projects[2] and so are freely available. They provide an insight into what can be achieved and tend to work in a very similar manner. In that they parse the change logs produced by the SCM system in order to produce graphs and text based description as to what has happened over time. Incorporating this into an automated process can deliver pseudo real-time statistics to all interested parties.

Lines of code and it's growth over time

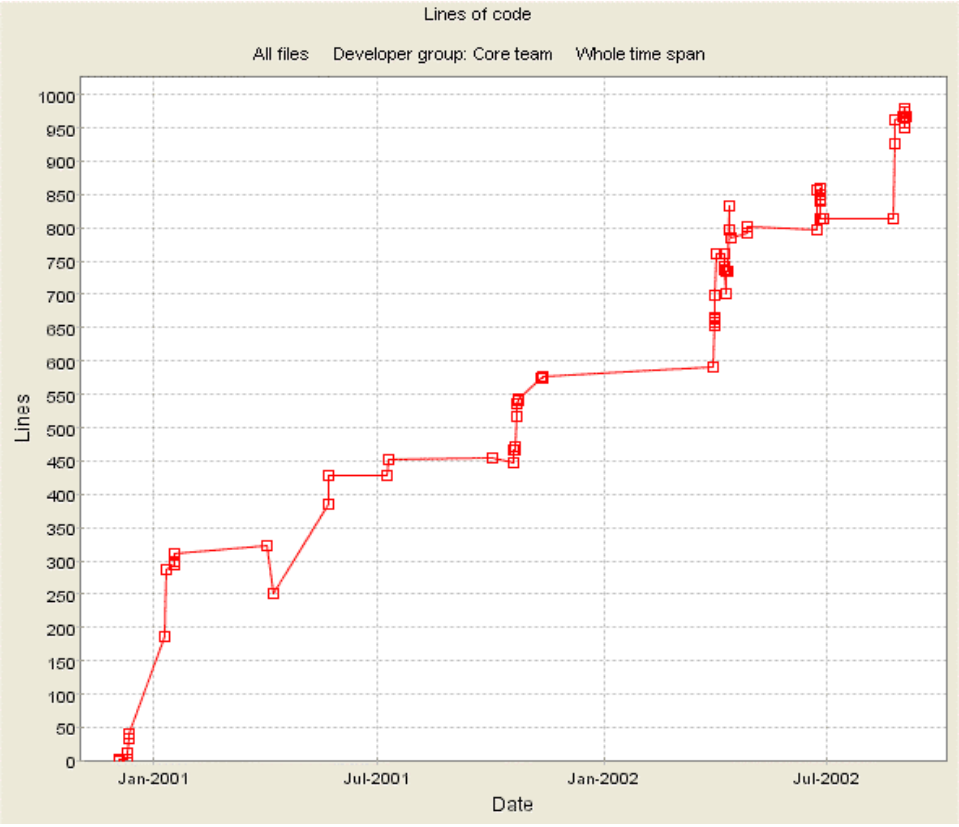


Figure 1 shows a projects code growth over time using Bloof

[1] If your development teams are not currently using a SCM tool you should start worrying now.
[2] Visit the Open Source initiative web-site for more info http://www.opensource.org/

No of files and their growth over time

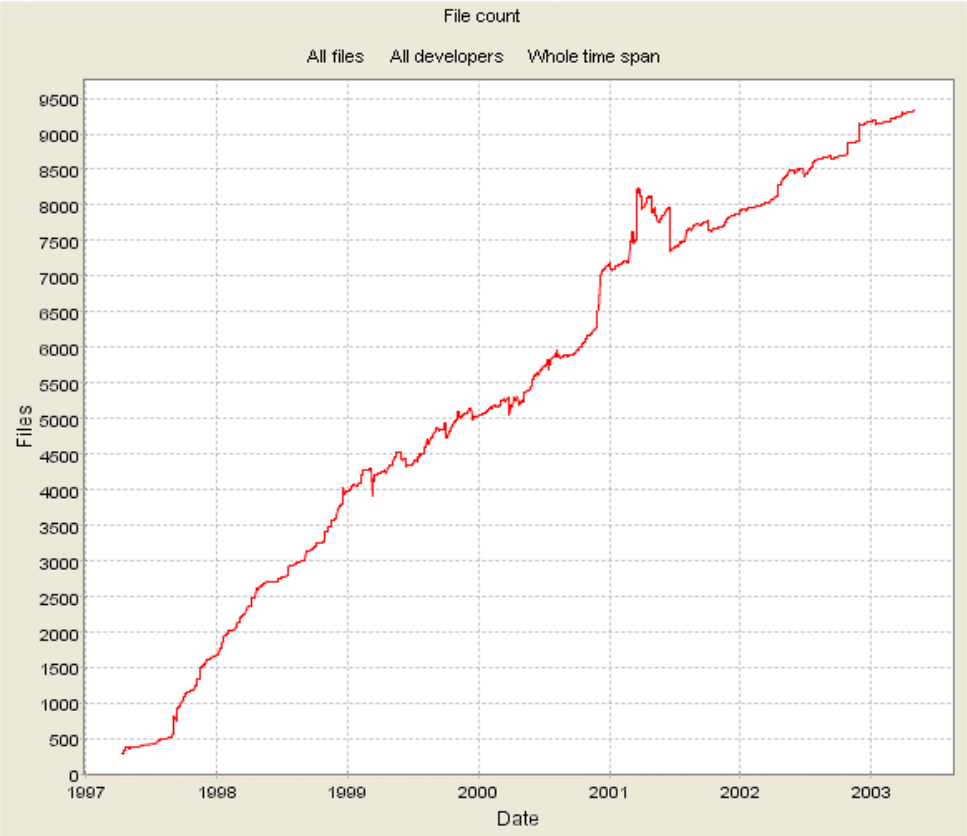


Figure 2 shows the kdebase projects file count growth over time using Bloof

Seeing the code base grow over time can be an invaluable means of understanding the day to day progress being achieved. Some of the patterns you might see include :-

- A vertical rise in the number of lines of code/files could be an indication that an awaited piece of development has been introduced into a project. It could also point to code being copied from a previous project for re-use (this is particularly interesting for third party developed projects in that it shows that they are not necessarily developing all of the delivered code-base for your sole use).

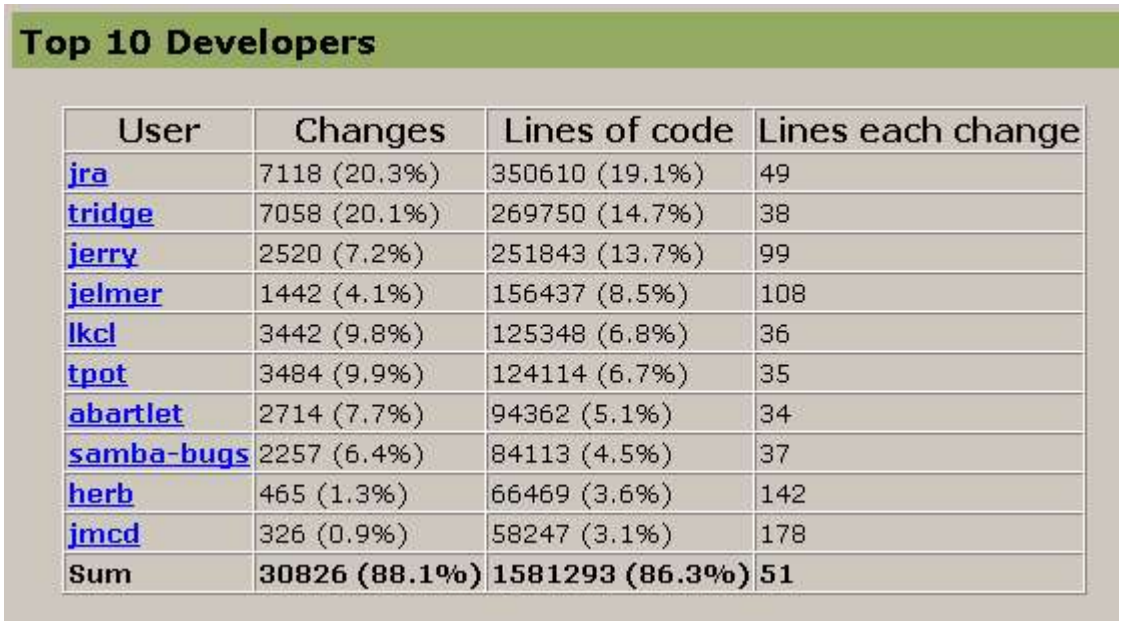
- A drop in the number of lines of code/files could indicate a refactoring process going on to optimise the code, or that functionality is being removed.

- A sustained period where the number of lines of code is not changing significantly could indicate a period of bug fixing.

The projects 'velocity' (code growth) is a very key metric. Having access to several of these graphs for similar projects coupled to graphs from your own projects can make estimating the final delivery date[3] much more accurate.

[3] Relies on establishing the final code size. This can be estimated or derived by benchmarking your project to others that you have statistics for. You can also use www.sourceforge.net, which has 70,000 projects for you to benchmark against.

Which developers are active/Who's doing the work?



Top 10 Developers

User	Changes	Lines of code	Lines each change
jra	7118 (20.3%)	350610 (19.1%)	49
tridge	7058 (20.1%)	269750 (14.7%)	38
jerry	2520 (7.2%)	251843 (13.7%)	99
jelmer	1442 (4.1%)	156437 (8.5%)	108
lkcl	3442 (9.8%)	125348 (6.8%)	36
tpot	3484 (9.9%)	124114 (6.7%)	35
abartlet	2714 (7.7%)	94362 (5.1%)	34
samba-bugs	2257 (6.4%)	84113 (4.5%)	37
herb	465 (1.3%)	66469 (3.6%)	142
jmcd	326 (0.9%)	58247 (3.1%)	178
Sum	**30826 (88.1%)**	**1581293 (86.3%)**	**51**

Figure 3 shows who is contributing what to the project - StatCVS

The power of the above chart is quite obvious, and before I use one example to prove the point I would like to say that **this metric needs to be handled with extreme care**.

Whilst there is a strong correlation between the no of lines coded, and the overall contribution to a project; there will be instances where the creation of a relatively small piece of code can take a disproportionate amount of time to produce; and deliver an equally disproportionate benefit to your project.

If developers feel that this metric is being abused they can easily change their code to skew the results and make them meaningless: quantity is not necessarily an indication of quality!.

Lets say that you commissioned a piece of work from a third party developer, and as part of that work requested that they use the Concurrent Version System (CVS) SCM tool. CVS is Opens Source and so would not cost them money to buy and it means that you can use tools like StatCVS, CVSMonitor or Bloof. From this you could then see who is doing the work for you. If for example you noticed that the users jra and tridge from the chart above suddenly stopped contributing code to the project, alarm bells would start ringing straight away and you would probably want to meet with your third party developer for an explanation.

Without these types of tools you really would be playing blind mans bluff. Being blissfully unaware that the two people who had contributed 40% of your code had suddenly stopped is not a situation you would want to be in. I wonder how often this has happened in the past?.

Code Complexity

Using other metrics tools can also add to the body of knowledge you build up over time to increase your level of understanding of the progress being made. Tools like JavaNCSS work directly from the actual code base and as such are independent of any SCM tool, but they are specific to a particular language or set of languages. These tools can generally be integrated into an automated build process to generate complexity metrics such as McCabes Cyclomatic Complexity Number (CNN).

The Cyclomatic Complexity Number is one of the most popular methods of representing complexity/functionality using a single ordinal number.
Project teams can use this information to decide if areas of code are potentially too complicated and so can aid code review. You can also collect this information over time to represent the total complexity/functionality of your code base and plot this along with your other code-related metrics.

Functions

Nr.	NCSS	CCN	Javadoc	Function
1	1	1	1	junit.samples.money.IMoney.add(IMoney)
2	1	1	1	junit.samples.money.IMoney.addMoney(Money)
3	1	1	1	junit.samples.money.IMoney.addMoneyBag(MoneyBag)
4	1	1	1	junit.samples.money.IMoney.isZero()
5	1	1	1	junit.samples.money.IMoney.multiply(int)
6	1	1	1	junit.samples.money.IMoney.negate()
7	1	1	1	junit.samples.money.IMoney.subtract(IMoney)
8	3	1	1	junit.samples.money.Money.Money(int,String)
9	2	1	1	junit.samples.money.Money.add(IMoney)
10	4	3	1	junit.samples.money.Money.addMoney(Money)
11	2	1	1	junit.samples.money.Money.addMoneyBag(MoneyBag)
12	2	1	1	junit.samples.money.Money.amount()
13	2	1	1	junit.samples.money.Money.currency()
14	8	6	1	junit.samples.money.Money.equals(Object)
15	2	1	1	junit.samples.money.Money.hashCode()
16	2	1	1	junit.samples.money.Money.isZero()
17	2	1	1	junit.samples.money.Money.multiply(int)
18	2	1	1	junit.samples.money.Money.negate()
19	2	1	1	junit.samples.money.Money.subtract(IMoney)
20	4	1	1	junit.samples.money.Money.toString()
21	1	1	1	junit.samples.money.MoneyBag.MoneyBag()
22	4	3	1	junit.samples.money.MoneyBag.MoneyBag(Money[])
23	3	1	1	junit.samples.money.MoneyBag.MoneyBag(Money,Money)
24	3	1	1	junit.samples.money.MoneyBag.MoneyBag(Money,MoneyBag)
25	3	1	1	junit.samples.money.MoneyBag.MoneyBag(MoneyBag,MoneyBag)
26	2	1	1	junit.samples.money.MoneyBag.add(IMoney)
27	2	1	1	junit.samples.money.MoneyBag.addMoney(Money)
28	2	1	1	junit.samples.money.MoneyBag.addMoneyBag(MoneyBag)
29	3	2	1	junit.samples.money.MoneyBag.appendBag(MoneyBag)
30	10	5	1	junit.samples.money.MoneyBag.appendMoney(Money)
31	3	1	1	junit.samples.money.MoneyBag.contains(Money)
32	14	11	1	junit.samples.money.MoneyBag.equals(Object)
33	6	4	1	junit.samples.money.MoneyBag.findMoney(String)
34	6	2	1	junit.samples.money.MoneyBag.hashCode()
35	2	1	1	junit.samples.money.MoneyBag.isZero()
36	7	3	1	junit.samples.money.MoneyBag.multiply(int)

Figure 4 shows the complexity of some of the Junit samples code-base using JavaNCSS

NCSS – Non commented source statements
Javadoc – No of comment line in the code

Unit/Regression test results

There are numerous testing frameworks available for you to utilise. Taking JUnit as an example, you can integrate this into an automated system to generate daily statistics on the quality of the code being produced and the overall health of the project. Collecting these statistics over time and plotting them graphically can also provide a much-needed confidence that projects have a good level of quality and greatly reduces integration issues that could dramatically effect project delivery.

Home

Packages

junit.samples.money

Classes

MoneyTest

Unit Test Results

Summary

Tests	Failures	Errors	Success rate	Time
21	15	0	28.57%	0.301

Note: *failures* are anticipated and checked for with assertions while *errors* are unanticipated.

Packages

Name	Tests	Errors	Failures	Time(s)
junit.samples.money	21	0	15	0.301

Figure 5 Results of running JUnit tests

Collecting Unit test results over time can give a powerful indication of progress and quality

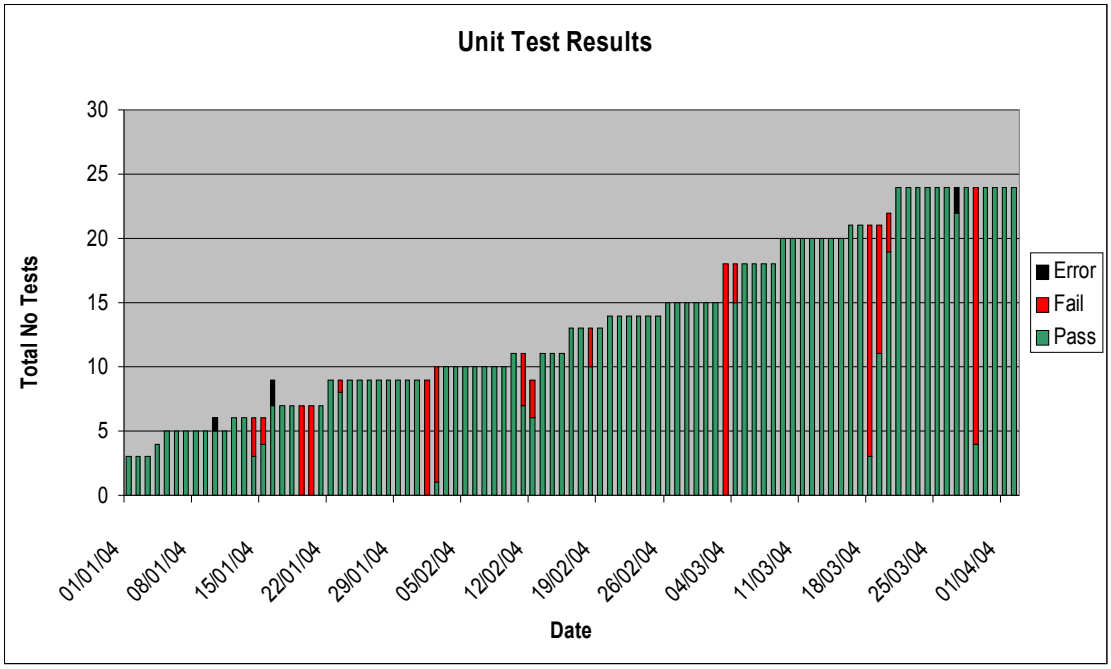


Figure 6 plotting test results over time

Code Coverage

Having automated unit tests in place is an important element of establishing a quality based metric. But to really appreciate how good your tests are you really need the ability to see just how much of the code base is actually being exercised.

For example you might have 200 unit tests running every night and feel good that they all pass on a regular basis, BUT your tests might only touch 5% of your code-base.

This is where code coverage tools really come into their own.

With them you can not only see how much of a particular class or module is being tested but also what the figure is overall . This particular class of tools can be illustrated by the two screenshots on this page. They come from Clover, which is a commercial product that can be incorporated into an automated build process in order to report of the coverage of Java based tests

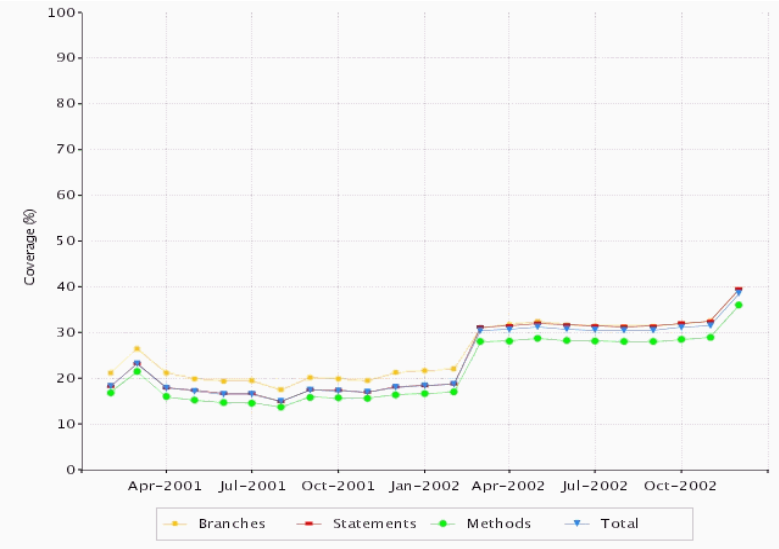


Figure 7 Percentage of code tested over time using Clover

Using Clover and tools like it allow you to greatly improve the quality of your projects in such a way as to make a very positive contribution to a successful delivery. They can also be used as a key enabling strategy for implementing ISO 9000 or improving your companies Capability Maturity Model (CMM) status.

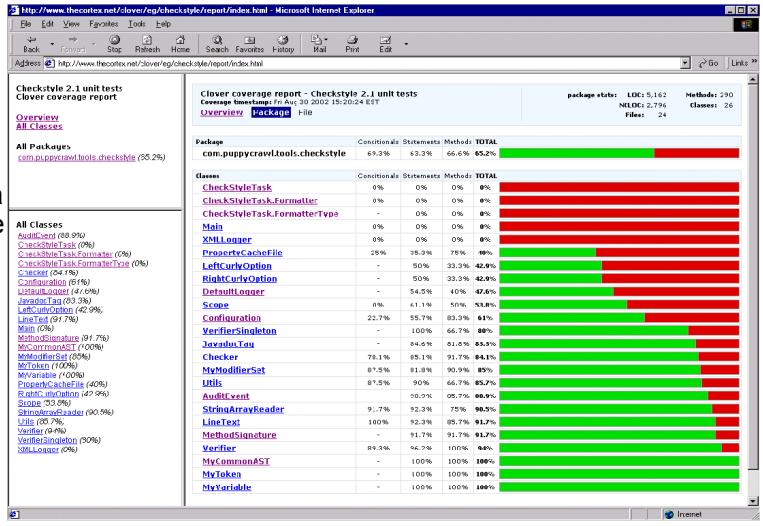


Figure 8 Detail code coverage report by class using Clover

Automating the process

The glue that brings together all of the tools and technologies that appear in this paper is a robust build tool. The likes of ANT[4] provide for award winning cross platform capabilities with unrivalled third party integration. This means that you can bring the generation of all of your key metrics into one build system. Doing this reduces the complexity of the generation/transformation and deployment of the metrics and means that the minimum manual intervention is required.

Benefits

Hopefully you can see how some of the elements that have been covered in this paper are applicable to your environment. In summary I would say that the major benefits of adopting fact based visualisation can be categorised as :-

- Establishes a repeatable process
- Removes a major part of the guesswork from the equation
- Requires minimal effort to produce the statistics
- Allows for comparison to other internal and externally developed projects in a consistent manner
- Taking advantage of Open Source Technologies provides you with an ultra low cost of adoption
- Aids in project estimation
- Makes the impact of decisions on project delivery much more transparent
- Enables the project team to gain valuable insight into how they are progressing, allowing them to take corrective measures early in the lifecycle.

A word of warning

Having access to such a body of information is a very powerful tool for your company, but there are a couple of points to bear in mind.

- Make the information freely available to the project team.
- Don't make it appear that you are using the data to beat the team up with.
  This is one sure-fire way of destroying all of the benefits that can be gained, and alienating the people who you rely on to deliver projects.

Ultimately, the team members should gain the most by being able to more accurately judge how they are doing,so that they can be the first to take corrective action.

[4] Ant is a widely used cross platform build tool http://ant.apache.org/

Useful Links

SCM

Concurrent Version System – The premier Open Source tool
http://www.cvshome.org/

SCM Metrics

CVSMonitor
http://ali.as/devel/cvsmonitor/

StatCVS
http://statcvs.sourceforge.net/

Bloof
http://bloof.sourceforge.net

Code Parsing Mertics

JavaNCSS
http://www.kclee.com/clemens/java/javancss/

Code Coverage

Clover
http://www.thecortex.net/clover/

Build systems

Ant
http://ant.apache.org/

Maven
http://maven.apache.org/

Unit Testing Frameworks

Junit – Java Unit Testing
http://www.junit.org/index.htm

Nunit - .Net unit Testing
http://www.nunit.org/

General links to testing Frameworks
http://www.xprogramming.com/software.htm