JHawk 6.1 Documentation-Metrics Guide

Virtual Machinery
February 2020
Version 1.6

Overview

This document is designed to give you an overview of what we are trying to achieve when using code metrics. This is followed by an approach using JHawk to achieve these aims. A complete list of the Metrics available in JHawk, and their definitions, is also provided at the end of the document.

Documentation

The following documents are provided with the distribution (depending on which license you have purchased). They are in PDF format and are located in the 'docs' directory of the distribution –

Document	Personal	Professional	Starter	Demo
JHawk6CommandLineManual – Documentation for the Command line version of JHawk	No	Yes	No	Yes
JHawk6CreateMetric – Documentation explaining how to create new metrics that can be added to JHawk	Yes	Yes	No	No
JHawk6DataViewerManual – Documentation for the JHawk DataViewer product	No	Yes	No	Yes
JHawk6Licensing – Licensing details for JHawk products	Yes	Yes	Yes	Yes
JHawkStarterManual – Documentation for the JHawk Starter edition	No	No	Yes	No
JHawk6UserManual – Documentation for the JHawk standalone application	Yes	Yes	No	Yes
JHawk6UsingMetrics – Documentation outlining how to get the best from JHawk and a list of the metrics implemented by JHawk with details of their calculation.. It also includes an introduction to the area of Java code metrics.	Yes	Yes	Yes	No

An Introduction to Metrics

What is a metric?

A metric is defined as the measurement of something. Out in the physical world we talk about metrics like height, distance, time, volume or area. In this book we will be looking at code quality metrics i.e. measurable aspects of a piece of code that can be used to assess its quality. Having measured some aspect of our code we might compare this value against a defined standard, we might compare it to the same metric calculated for another piece of code or we might compare it to a calculation of the same metric for a previous version of the same code.

What is software quality?

There are a lot of aspects to software quality. The most critical aspect is that the software does what it was designed to do. It should also perform at a speed compatible with its function - Air Traffic Control Software is expected to perform a lot more rapidly than accounting software. When we use JHawk to examine our code our primary interest isn't whether the software works or how well it works – it's what we can learn from the code.

So what can we tell from looking at a bunch of code? Why would we bother doing it – if the software meets the spec and runs like lightning why would the way that it has been written matter?

Well, the thing is, we all know that it does matter. Very seldom do we write clean, brand new code, set it running and leave it untouched forever. Even if there are no bugs, even if the specification was correct from day one software changes - frequently. Almost every software product is obsolete on the day of release. We don't write brand new code when this happens – we modify the existing code. When we have to maintain code then somebody other than the writer probably has to read it. Code is very expensive to write and we therefore expect it to last a long time. I've worked in organisations where there is code that is thirty years old, and it is still critical to the operation of the organisation. The people maintaining that code are not those who wrote it, and in many cases they are not as familiar with the language that the code is written in as the original writers were. The longer that a person takes to understand the code, the more expensive a bug fix becomes – not only in terms of the cost of that persons time but also the damage that the bug will cause to the business of the company while it remains unresolved. In general once a bug is repeatable it is solvable but the quality of the software will have an effect on the time taken from repetition to solution.

As you can see from the example above part of the issue of software quality is readability. Readability has a number of aspects –

- Quantity (or volume)
- Layout
- Complexity

You can think about this in terms of books. If I gave you an academic textbook and a children's book and asked you to find a single typographic error in each of them it's pretty obvious which will prove to be the easier task. A children's book won't be very long, it will have very few words on each page and the number of different words (the vocabulary) will be very small. You will probably know immediately the correct spelling of each of the words in the book. Contrast this with an academic textbook which will generally be fairly long, have very dense type with a large number of words on each page (perhaps with even denser footnotes at the base of the page). There will also be an extensive vocabulary (you may even need to have a dictionary handy to find out how the words used are supposed to be spelt).

You can think about software in the same way – a simple accessor method is as complex as a child's book. A method that sets up a number of parameters prior to the use of a particular communications protocol will be much more complex and reading such a method may require you to understand the API of the protocol. To do this you may have to consult its documentation (the equivalent of having the dictionary to hand).

Presented with ten books we could all pick them up, flick through them and grade them on their complexity. You could pick a large group of people and ask them to do this exercise and they would all come out with pretty much the same order.

The same exercise could be performed by showing a group of programmers a set of Java classes and ask them to grade them on their complexity. Again you would find pretty close agreement on which classes were most complex.

So we could evaluate basic software quality by eye pretty effectively. The problem is that doing that is slow and expensive because we need to pay people to do it. We are, however, dealing with the regular syntax of computer languages – a computer is going to read the code anyway – why don't we get the computer to do the job? And that's where the kinds of metrics that we are interested in come in.

But it's not just those qualities inherent to the code that matter. We also need to think of the relationship between the various pieces of code. For example I might ask you to check an academic textbook for consistency. In this case you would be looking for cases where the text of the book refers to another section e.g.

'In Section 1 we discussed arithmetic and showed that one plus one is equal to one'

When you go to Section 1 you see that it states categorically 'Of course we all know that one plus one equals two.'

Another consistency check might be that I ask you to ensure that any references quoted do indeed back up the claims of the author.

In each case the more of the references there are the more difficult your task will be because you will have to read the sections or papers referred to as well in addition to the original text. Even if the referred sections are in the text, and you have already read them it's quite likely that you will have to read them again.

When we write a large piece of object-oriented code – perhaps an application – we will inevitably write a lot of different classes that will co-operate to achieve the aims of the software. So if I ask a group of programmers to review a number of pieces of code it is likely that they will view code that has external relationships with a lot of other classes as more complex than code that contains references to only a few classes. This is simply because they may well have to understand the referenced classes as well as the class that references them. Here's a simple example. We have a program that makes an external call-

```
List<String> strings = StringListMaker.getStrings(myObject);
if (strings != null) {
        for (String aString:strings) System.out.println("Found "+aString;
}
```

To the getStrings method in the StringListMaker class :

```
 public static List<String>  getStrings(StringContainingObject anObject) {
        List<String> result = new ArrayList<String>();
        if  ((anObject != null) && (anObject.getNames() != null)) {
                for (String aString : anObject.getNames()) result.add(aString);
        }
        return result;
}
```

Reviewing this code you would notice that the check on the 'strings' variable for a null value after the external call is redundant. StringListMaker.getStrings always returns a list – even if it puts nothing in it. Our for loop can cope with an empty list without the check. To make this observation we had to critically examine code in two classes. This was a very simple example but there is still a lot to take in before we see the inefficiency. One way to look at it is that adding an external call or reference brings all the complexity of the external item into the calling code. This is why the relationships between pieces of code are important when judging their complexity.

Some typical metrics

The most commonly used metrics are –

- Lines of Code
- Cyclomatic Complexity
- Halstead Metrics

These are some of the oldest metrics used in software development and they tell us a lot about the quality of software. Lines of Code tells about the quantity (similar to the length of the book) and Cyclomatic Complexity tells you how many different paths there are through each piece of code (how difficult the book is to read). The Halstead Metrics also measure the length and complexity (using different aspects of the code) as well as measuring the vocabulary of the code. You can find more extensive discussions on Lines Of Code (http://www.virtualmachinery.com/sidebar1.htm) and the Halstead Metrics (http://www.virtualmachinery.com/sidebar2.htm).

These three metrics were designed for use in the procedural (as opposed to object-oriented) era. In the procedural code era a program was usually a single chunk of code consisting of a sequential declaration of methods (then known as functions and procedures). Code might be divided into 'modules' which were a convenience to allow different programmers work on different sections of the code simultaneously. When the application was compiled all of these modules were combined into a single executable file. Almost every line was purely functional – exceptions would be the method headers and lines which imported other modules. Variables generally had global scope or none at all.

It's pretty intuitive that a piece of code with 100 lines of code in it is going to be harder to read than one with 10 lines in it – but do you think that it's 10 times as hard? Or 20 times? Or maybe only 5 times? Or maybe it depends on other factors as well as the quantity? – for example how the code is laid out.

Similarly with Cyclomatic Complexity - this is a measure of how many different possible routes there are through a piece of code. For example a piece of code where you just have a number of lines of code in succession has a single path (see the code for the method complexityOfOne below). The only path through this is Line A → Line B → Line C. By adding a single if statement you create a method that has two paths through it (see the method complexityOfTwo below). In this case there are two clear alternate paths : Line A → Line B → Line C (followed if the boolean value addY is true) Line A → Line D (followed if addY is not true)

```
public void complexityOfOne(int x, int y) {
    int xsquared = x*x; //Line A
    int value = xsquared+y; //Line B
    System.out.println("X squared plus Y ="+value); //Line C
}


public void complexityOfTwo(int x, int y, boolean addY) {
    int xsquared = x*x; //Line A
    if (addY) {  /
        int value = xsquared+y; //Line B
        System.out.println("X squared plus Y ="+value); //Line C
    } else {
        System.out.println("X squared ="+xsquared);  //Line D
    }
}
```

In the object-oriented era a line of code can be a wide variety of things. We still have method headers and import statements but we now have 'non-functional' code which contributes to the structure of the application rather than to its functionality. This means that we need to make decisions about how we treat these parts of the code when we are calculating our metrics. An example is the Halstead Vocabulary – in procedural code where all the variables have global scope then no matter how many times myVar appears it will only make a single contribution to the vocabulary. If I have a Java class MYClass with three methods method1, method2 and method3 I might declare myVar as an instance variable and I might declare a local variable myVar in each of methods 1 and 3 and use the instance variable myVar in method 3. How do I calculate the Halstead Vocabulary in a way that makes sense in this class? There is a more extensive discussion on this issue (link-here).

There has been a lot of academic discussion about these metrics and their suitability to object-oriented languages and many suggestions as to how they could be changed to make them useful in this environment.

Other aspects – design and fault prediction

So far our discussion has mainly been about maintainability and while that is important metrics can tell us a lot more.

Design – the way that our classes have been designed, both on their own and in the way that they interact within our application, can have a bearing on how efficiently the application works. It can also impact how much code we have to write- in Object Oriented development is design can replace code. You can think of design acting like a lever that allows the power of each line of code to be multiplied. As an example of this inheritance allows us to reduce the amount of code that we have to write by reusing the lines that we have already written. At a higher level of abstraction design patterns allow us to reuse our design – saving part of the design task. Over the years one fact in software metrics has remained unchanged – less code means fewer errors. If good design means less code – either because you have to write less code or because you can reuse tested and trusted code – then it must surely mean a reduced error count. A number of the metrics that we will discuss later in the book allow us to assess aspects of our design. An example would be the Coupling Between Objects (CBO) metric that we mentioned above.

Fault prediction – In recent years a lot of research work has been directed towards using metrics to predict the weak spots in the coding or design of an application. Again anyone who has programmed for any length of time has seen this. Every application has a 'hot spot' for bugs. Usually it is an area where the specification is complex and fluid and where there is frequent code change. Often the solution in such circumstances is to redesign and rewrite the code in the area involved

Why are there so many metrics?

When you go out there on the Web trying to find information on metrics one of the things that will strike you pretty quickly is the number of different metrics available.

There are a few reasons why this shouldn't really be surprising -

- **History** – in 50 years of programming we've accumulated lots of metrics and haven't really thrown any away. Counting the number of bytes in a program was critical up to 30 years ago not just as a metric but also because of the limited architecture of the machines. We don't really count bytes any more but we still count lines of code. Surprisingly most metrics have survived the transition from procedural to object-oriented programming. They have also survived the myriad of languages that the programming community has invented and discarded. It's reasonable to argue that the fundamentals of programming have not changed – inside every object there are small chunks of procedural programming carrying out the functions of the application.

- **Changes in programming methodologies and disciplines** – Object oriented programming has introduced more measurable aspects. We now need metrics that can handle the concepts of class and package. There have also been a number of software lifecycle processes and quality certifications that have introduced the need for measurement.
- **Continuous process of improvement** – Individual metrics have been improved over the years leading to numerous versions of the same metric. LCOM (Lack of Cohesion between Methods) is an example of this. This frequently leads to confusion when people attempt to compare two different sets of code unwittingly using two different versions of the same metric.
- **Alchemy** – everybody thinks that they have discovered the one true metric. Like the alchemists fruitless struggle I very much doubt there is some magic metric out there that you can apply to code and divide it all neatly into two piles – one of good code and another of bad. But every programmer has the right to try and by using JHawk's mechanism for creating new metrics you can join in this as well!
- **Social factors** – people outside of technology often look it as being a truly scientific endeavour with each part of the technology chosen purely for it's position as the best tool for the job. Like most human activity it is largely driven by social factors – fashion, 'not invented here' or most frighteningly – 'we've always used it – I don't really know why – personally I think it's totally useless'.

In JHawk we have tried to provide you with the widest possible range of available metrics. We also provide you with the ability to create your own metrics and use them in all of the JHawk products. We've made an initial pre-selection for you of those metrics (see the lists at the end of the document) that we consider most useful but you can easily activate any of the other provided metrics as you require.

What makes a good metric?

So given the vast choice of metrics available how do you separate the good ones from the bad and how do you pick the ones that are going to be useful to you. One of the aims of this document is to try and give you enough knowledge to make those decisions. At the very least a useful metric should have the following attributes -

- There should be a direct (or inverse) relationship between the value of the metric and some aspect of the quality of your code e.g. number of arguments to a method, number of methods in a class.
- The measurement should be consistent – it should always give the same answer in the same set of circumstances.
- It should preferably be visible. In other words you should be able to see the reason why the metric is outside of the preferred range. If the metric is not visible it will be difficult to fix. For example the Cyclomatic Complexity metric is based on the number of possible paths through the code. When you are guided to the class by the high value of the metric you will probably see lots of if statements and know that by reducing these, e.g. by dividing the method into a number of sub methods, you will have started to reduce the Cyclomatic Complexity.

It is preferable that a metric has a defined range e.g. between 0 and 1 or 0 and 100. If a metric does not have a defined range then it can be difficult to decide on an acceptable value. For example – the number of arguments in a method has a bearing on the complexity of a piece of code – but what is a reasonable upper value for the number of arguments – is it 3?, is it 5? Or does it depend on particular circumstances?

We can also divide our metrics down into what it is that we want to measure. Lets rank these from least to most abstract –

- Amount of code written
- Effort expended in coding

- Maintainability
- Quality of Code
- Quality of design

The amount of code written is pretty easy – it's just the number of lines of code, or statements or expressions.

The effort probably takes a little more to calculate. It's the amount of code plus the overhead of creating methods, classes and packages. The programmer has to think about how these are partitioned and hopefully they increase the quality of the code by their design choices.

Maintainability is the volume and complexity of code so contains the measurements we used for amount and effort. The average size of the methods, classes and packages also affects the maintainability as the maintainer has to look through less code if the code units are smaller. The presence of useful comments also adds to the clarity of the code as does the appropriate naming of classes and methods and a coherent packaging structure. The problem is that while we can measure the number of comment lines it tells us nothing about their quality – the two sets of comments get the same credit –

(a)
```
/**
/* Method to open file, check file length is greater than zero and
/* Less than 10Mb. Roll over if greater than 10 Mb
/**
```

(b)
```
/**
/* Author: John Smith 2008-Jan-01
/* This Method Copyright: The Big Corp 2008
/**
```

So we're already getting into the difficulty of assessing the quality of things that affect our metrics rather than just the quantity. The standard Maintainability Index metric included in the JHawk product comes in two flavours – one of which includes comments in the calculation and one of which does not. If most of your programmers create sensible comments like the first example above then you might consider including comments in your calculations – if they are like the second then it's not really worth it.

It is sometimes possible to include qualitative measures at 'face value' in a quantitative measure if you put in place controls that ensure that value. For example you might review all of your code and as part of that review programmers might be encouraged to only provide worthwhile comments. If all your code is preceded by generated comments like the second example above then you might choose to modify the metric to reduce the number of comment lines to account for the generated ones.

Quality of Code is the next measure and here we have leapt up another level of abstraction. What do we mean by quality of code? We all know what good code looks like – everything is named in a way that suggests its function, methods are short, perform a single function and are appropriately commented. But how can we measure this just by parsing the code? A method can be short and meaningless –

```
public double doIt(double aThing) {
        return 3.14*(aThing*aThing);
}
```

- or short and meaningful –

```
public double calculateCircleArea(double radius) {
        return 3.14*(radius*radius);
}
```

But again we can see that analysing the code will not automatically distinguish between the good and the bad.

There are personal factors in choosing metrics as well - a good metric is one that you understand. For example I am happy with a method with 10 lines of code in it because it's less than a screen full in the IDE that I use; I'm not so happy with one with 40 lines in it because I know it won't fit into a single screen full. This is a good metric for my purposes because I know why I chose it and I can relate the metric result to my requirements.

Relating metrics to quality

Metrics seldom relate directly to quality. Those metrics which measure size tell us nothing about quality. They don't tell us whether we have a lot of good code or a small amount of bad code. Similarly the complexity of code doesn't relate to its quality – for example the implementation of a sort algorithm might be done extremely well but it might also be very complex.

The value that these metrics give us is that we understand that in some circumstances the quantity of something can be an indicator of something bad (or something good). For example a single method with 450 lines of code is almost certainly bad – the code might be fine, but in the context of maintaining the code or understanding the function of the method we know that this is not a good situation. So we relate the metric to quality via its context – a class with 450 lines of code might be perfectly acceptable but a method with 450 lines is not.

Can metrics be counterproductive?

There are three main dangers in using metrics –

- That creating the metrics, and getting the metrics to look 'right' start to consume more effort than writing the code. There are more important things to put in place than metrics measurement – a tight specification process, testing (unit, integration and system) and a defined, repeatable and automated build process for example.
- That the measurement of metrics influences programmer behaviour in a way that is detrimental to the process of producing the software.
- Taking the view that just doing the measurement is the important thing – measurement is useless unless the results are analysed and sensible action taken on foot of these results.

Metrics with JHawk

A sensible approach to using metrics

Like any other powerful tool JHawk produces a lot of information. As you gain experience with JHawk you will learn how to use this information to assess and improve the quality of your code. The biggest problem is how to start on this process. The answer is 'simply'. Choose one or two key metrics to get you started – if you want you can read our extensive section on metrics later in this document and after you have read that make an informed decision on which metrics you feel best suit your purposes. This is fine if you are prepared to take that time right at the beginning but most of us are in a hurry to get started – we've paid for the tool and we want to start getting our moneys worth.

At Virtual Machinery we realise this so that out of the box JHawk has only a few metrics enabled. You can find these in the list at the end of this document. By using the preferences interface you can modify this list to suit your requirements.

We have also enabled the following metrics on the dashboard –

At System Level –

- AVCC (at Package level – the average Cyclomatic complexity of all the methods in that package)
- AVCC (at Class level – the average Cyclomatic complexity of all the methods in that class)
- COMP (at Method level – the cyclomatic complexity of individual methods)

At Package Level -

- AVCC (at Class level – the average Cyclomatic complexity of all the methods in that class)
- COMP (at Method level – the cyclomatic complexity of individual methods)

At Class Level -

- COMP (at Method level – the cyclomatic complexity of individual methods)

As you can see all of these metrics relate to Cyclomatic Complexity. Cyclomatic complexity is a good place to start as it satisfies many of the criteria of a good metric. So what is the best approach to using the information on the dashboard? First try and find the most complex methods in the system. Methods with a complexity greater than 10 should be looked at first. If there are too many of these then look at the fifty worst methods first. Look at the code in these methods and see why they have such high values for complexity – perhaps they have many lines of code – perhaps they are very complex – for example with large switch statements. The next step is to see which of these methods can be divided into a number of smaller methods. It may not always be possible to subdivide a method – for example if it contains a single very large switch statement which cannot be subdivided. Sometimes it may not be obvious how to subdivide a method and sometimes subdivision is made difficult by a poor architectural pattern.

When you have analysed your fifty methods record your decisions on each. Look at those which you have decided are candidates for rewriting. Are there common themes (or anti-patterns) in the way that these methods have been written? Are they all located in one area (e.g. a particular class or package) of the system? Were they perhaps all written by the one person/team? The answers to these questions may well give you clues as to where to look for other instances of poor coding practice that are reducing the quality of your code.

TIP: If you find methods that you feel breach a particular metric limit, but there is a justification for them doing so, then put a comment in the code stating why you think the breach is acceptable. It will save you time the next time this code comes up for review.

TIP: If you know a method needs to be rewritten but realise that other changes will need to be done first (perhaps to the architecture) then put a TODO: in the code so that you will come back to it later.

Automating the process

You can use the command line version of JHawk to automate the production of your metrics. If you look at the documentation for the Command Line version you will find examples of how to do this and how to create batch processes to carry out the analysis of a large number of files at once. There is also a JHawk Ant task which you can use to easily incorporate JHawk metrics into other Ant processes.

Using Filters

Since version 6.1 JHawk has provided filters. These can be used in combination with the warning and danger levels that can be configured on a per metric basis to filter out metrics that breach the warning and /or danger levels. Filters can be created and used 'on –the-fly' in the Standalone version. Filters can also be saved to files in both the Standalone version of JHawk. These filters can then be re-used in the Standalone and Command Line versions of JHawk – see the relevant user documents for more details on how to use filters.

Using the JHawk Data Viewer – how your code quality is changing over time

The JHawk Data Viewer is provided with the professional license. It is an application that allows you to view changes in the quality of your code over time. It uses the basic XML files generated by JHawk. To use Data Viewer you need to create a basic XML file for each snapshot of your code that you want to analyze. You then load these into JHawk Data Viewer and after analysis you can view your data in a number of different ways. You can also export your data to a format that can be viewed using the Google Visualization API. Again you can use the command line jar to streamline this process and there is a description of how best to do this in the documentation for the Data Viewer.

Metrics defined in JHawk

Metrics are defined at four levels System, Package, Class and Method. Only a subset of the metrics are active initially. You can use the Preferences mechanism to activate other metrics as you need them. The first sections here show the metrics enabled ‘out of the box’ at each level. The subsequent sections list all of the metrics available and provide full definitions for each of the metrics.

System level metrics enabled initially

Metric Code	Description
NAME	Name of System
NOPK	Number of Packages in the system
NOS	Number of Statements in the System
AVCC	Average Complexity of all the methods in the System
HBUG	Cumulative Halstead Bugs of all the components of the system
HEFF	Cumulative Halstead Effort of all the components of the system
MI	Maintainability Index (including comments)
CCML	Total number of comment lines in the system
NLOC	Total number of lines of code in the system

Package level metrics enabled initially

Metric Code	Description
NAME	Name of Package
NOCL	Number of Classes in Package
NOS	Number of statements in Package
AVCC	Average Cyclomatic Complexity
HBUG	Cumulative Halstead Bugs of all the components in the package
HEFF	Cumulative Halstead Effort of all the components in the package
HLTH	Cumulative Halstead Length of all the components in the package
HVOL	Cumulative Halstead Volume of all the components in the package
MI	Maintainability Index (including comments)
CCML	Total number of comment lines in the package
NLOC	Total number of lines of code in the package

Class level metrics enabled initially

Metric Code	Description
NAME	Name of class
NOMT	Number of methods in class
LCOM	Lack of Cohesion of Methods
AVCC	Average Cyclomatic Complexity of all the methods in the class (TCC/NOMT)
NOS	Total Number of Java Statements in class
HBUG	Cumulative Halstead Bugs of all the components in the class
HEFF	Cumulative Halstead Effort of all the components in the class
UWCS	Unweighted class size
INST	Number of instance variables (or attributes) defined in this class
PACK	Number of packages imported by this class
RFC	Response for class
CBO	Coupling Between Objects
MI	Maintainability Index (including comments)
CCML	Total number of comment lines in the class
NLOC	Total number of Lines of Code in the class

Method level metrics enabled initially

Metric Code	Description
NAME	Name of method
COMP	Cyclomatic Complexity
NOCL	Number of comment Lines
NOS	Number of Java Statements
HLTH	Halstead Length of the method
HVOC	Halstead Vocabulary of the method
HEFF	Halstead Effort for the method
HBUG	Estimated Halstead Bugs in the method
CREF	Number of classes referenced in the method
XMET	Number of calls to methods that are not defined in the class of the method.
LMET	Number of calls to local methods i.e. methods that are defined in the class of the method.
NLOC	Number of Lines of Code in the method

System level metrics (all)

Metric Code	Description
NAME	Name of System
NOPK	Number of Packages in the system
NOCL	Total Number of Classes in the System
NOMT	Total Number of Methods in the System
NOS	Number of Statements in the System. This is a total of the number of Java statements for each of the packages defined in the system. See definition of Java statements in 'Method level Metrics (all)' table.
TCC	Total Cyclomatic Complexity of all the methods in the System
AVCC	Average Complexity of all the methods in the System TCC/NOMT In the standard jhawk properties file the warning level is set to 10 and the danger level to 15.
HBUG	Cumulative Halstead Bugs of all the methods in the System (see definition of Halstead Bugs in 'Method level Metrics (all)' table)
HEFF	Cumulative Halstead Effort of all the components in the System (see definition of Halstead Effort in 'Method level Metrics (all)' table)
HLTH	Cumulative Halstead Length of all the components in the System (see definition of Halstead Length in 'Method level Metrics (all)' table)
HVOL	Cumulative Halstead Volume of all the components in the System (see definition of Halstead Volume in 'Method level Metrics (all)' table)
MI	Maintainability Index (including comments). See definition of MI in 'Class level Metrics (all)' table. In the standard jhawk properties file the warning level is set to 85 and the danger level to 65.
MINC	Maintainability index (No comments). See definition of MINC in 'Class level Metrics (all)' table.
CCOM	Total Number of Comments in the system. This is a total of the number of comments for each of the packages defined in the system.
CCML	Total number of comment lines in the system. This is a total of the number of comment lines for each of the packages defined in the system.
NLOC	Total number of lines of code in the system. This is a total of the number of lines of code for each of the packages defined in the system.

Package level metrics (all)

Metric Code	Description
ABST	The Abstractness metric is calculated as follows - (numAbst + numInt) / NOCL Where numAbst is the number of abstract classes defined in the package., numInt is the number of interfaces defined in the package and NOCL is the total number of classes defined in the package. Abstractness is the ratio between the number of abstract classes and the total number of classes in a package. A value of 0 means the package is fully concrete, 1 means that it is fully abstract. In some cases the level of abstraction may be intentional so the value is not an indication of quality but where the result conflicts with the designers intention (e.g. a package called myco.myapp.interfaces with a low abstraction value might ring alarm bells).
AVCC	Average Cyclomatic Complexity of all the methods in the package (see definition of Cyclomatic Complexity in 'Method level Metrics (all)' table). Calculated using the Total Cyclomatic complexity and the number of methods – TCC/NOMT In the standard JHawk properties file the warning level is set to 10 and the danger level to 15.
CCOM	Total Number of Comments in the package. This is a total of the number of comments at package level plus a total of the number of comments for each of the classes defined in the package.
CCML	Total number of comment lines in the package. This is a total of the number of comment lines at package level plus a total of the number of comment lines for each of the classes defined in the package.
DIST	The Distance metric is calculated from the Abstractness (ABST) and Instability (INST) metrics – *abs*(1-(ABST+INST)) A package should be balanced between abstractness and instability, i.e., somewhere between abstract and stable or concrete and unstable. Stable packages should also be abstract packages (A = 1 and I = 0) while unstable packages should be concrete (A = 0 and I = 1).
FIN	Fan In (or Afferent Coupling). This is calculated as the number of packages in the code analyzed by JHawk that reference this package.
FOUT	Fan Out (or Efferent Coupling) This is calculated as the number of packages in the code analyzed by JHawk that this package references.
HBUG	Cumulative Halstead Bugs of all the methods in the Package (see definition of Halstead Bugs in 'Method level Metrics (all)' table)
HEFF	Cumulative Halstead Effort of all the components in the Package (see definition of Halstead Effort in 'Method level Metrics (all)' table)
HLTH	Cumulative Halstead Length of all the components in the Package (see definition of Halstead Length in 'Method level Metrics (all)' table)
HVOL	Cumulative Halstead Volume of all the components in the Package (see definition of Halstead Volume in 'Method level Metrics (all)' table)

INST	The Instability metric is calculated from the Fan In and Fan Out metric – FOUT/(FOUT + FIN) Instability is a measure of the potential impact of a change related to the package. It's value can be between 0 and 1. A completely stable package (value of INST is zero) can still change internally but should not as it will have an impact on dependent packages. An completely unstable package (INST value of 1) can change internally without consequences on other packages.
MI	Maintainability Index (including comments). See definition of MI in 'Class level Metrics (all)' table. In the standard jhawk properties file the warning level is set to 85 and the danger level to 65.
MINC	Maintainability index (No comments). See definition of MINC in 'Class level Metrics (all)' table.
MAXCC	Maximum Cyclomatic Complexity – the highest value for Cyclomatic Complexity displayed by any method in the Package
NAME	Name of Package
NOCL	Number of Classes in Package
NOMT	Number of Methods in Package
NOS	Number of Java statements in Package. This is a total of the number of Java statements in the package an the number of Java Statements for each of the classes defined in the package. See definition of Java statements in 'Method level Metrics (all)' table.
NLOC	Total number of lines of code in the package. This is a total of the number of lines of code at package level plus a total of the number of lines of code for each of the classes defined in the package.
RVF	Review Factor (see section on Review Factor in this document) A value of 100 or more indicates that this package should be reviewed.
TCC	Total Cyclomatic Complexity – this is the total cyclomatic complexity of all the methods in the Package. See definition of Cyclomatic Complexity in 'Method level Metrics (all)' table.

Class level metrics (all)

Metric Code	Description
AVCC	Average Cyclomatic Complexity of all the methods in the class (see definition of Cyclomatic Complexity in ‘Method level Metrics (all)’ table). Calculated using the Total Cyclomatic complexity and the number of methods – TCC/NOMT In the standard jhawk properties file the warning level is set to 10 and the danger level to 15.
CCML	Total number of comment lines in the class. This is a total of the number of comment lines at class level plus a total of the number of comment lines for each of the methods defined in the class.
CCOM	Total Number of Comments in the class. This is a total of the number of comments at class level plus a total of the number of comments for each of the methods defined in the class.
CBO	Coupling Between Objects. Two calculate this metric two sets of classes are created. (1) Those classes in the code analyzed by JHawk that reference this class. (2) those classes in the code analyzed by JHawk that this class references. From the intersection of these two sets a third set is created of the classes that this Class references AND that reference this class. The size of this set is the value of the CBO metric. A high CBO value for a class suggests that it will be difficult to reuse as it indicates that the class is too dependent on other class.
COH	Cohesion. The Cohesion metric is calculated as follows – NumRefs/(NOMT * INST) where numRefs is the sum of the number of attribute references in each of the methods in the class. Higher values are better.
DIT	Depth of inheritance tree for this class. This is calculated in the same way as the number of superclasses (NSUP).
EXT	Number of external method calls made from the class i.e. the number of calls made to methods in other classes (including classes that are not in the group of classes under analysis e.g. classes in the JDK, classes in third party packages). A higher number of external method calls increases that classes dependence on other classes making it more difficult to maintain.
FIN	Fan In (or Afferent Coupling). This is calculated as the number of classes in the code analyzed by JHawk that reference this class. A high value can indicate that a class is doing too much and may be a candidate for refactoring.
FOUT	Fan Out (or Efferent Coupling) This is calculated as the number of classes in the code analyzed by JHawk that this class references. A high value can indicate that a class is doing too much and may be a candidate for refactoring. Values over 50 are generally viewed to be bad.
HBUG	Cumulative Halstead Bugs of all the methods in the class (see definition of Halstead Bugs in ‘Method level Metrics (all)’ table)
HEFF	Cumulative Halstead Effort of all the components in the class (see definition of Halstead Effort in ‘Method level Metrics (all)’ table)
HIER	Number of methods called that are defined in the hierarchy of the class.
HLTH	Cumulative Halstead Length of all the components in the class (see definition of Halstead Length in ‘Method level Metrics (all)’ table)

HVOL	Cumulative Halstead Volume of all the components in the class (see definition of Halstead Volume in ‘Method level Metrics (all)’ table)
INST	Number of instance variables (or attributes) defined in this class. A high number of instance variables can indicate a class that has too many responsibilities. Mitigating factors would include the definition of constants (providing they constitute a coherent group).
INTR	Number of interfaces implemented by this class
MOD	Number of modifiers (public, protected etc defined for this class)
LCOM	Lack of Cohesion of Methods. This is calculated according to the formula defined by Henderson-Sellars. This calculation is also known as LCOM5 by the Fraunhofer institute (see L. C. Briand, J. W. Daly, and J. Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. Empirical Software Engineering: An International Journal, 3(1):65–117, 1998) The calculation is – ((1/INST) * (numRefs - NOMT))/ (1-NOMT)) Where numRefs is the sum of the number of attribute references in each of the methods in the class, INST is the number of instance variables defined in the class and NOMT is the number of methods in the class. This version of LCOM has values in the range 0 to 2. Lower values are better – any value over 1 should be viewed as an indicator of poor code.
LCOM2	The LCOM2 metric is calculated by keeping a count of the number of method pairs that share instance variable references and a separate count of those that do not share instance variable references. The number of those that have instance variable references in common is subtracted from those that do have none in common. If the value returned is negative it is set to zero. Values of LCOM2 closer to zero are considered to be better. (see L. C. Briand, J. W. Daly, and J. Wüst. A Unified Framework for Cohesion Measurement in Object-Oriented Systems. Empirical Software Engineering: An International Journal, 3(1):65–117, 1998)
LMC	Number of Local method calls i.e. calls to methods that are defined in this class.
MAXCC	Maximum Cyclomatic Complexity of any method in the class (see definition of Cyclomatic Complexity in ‘Method level Metrics (all)’ table)
MI	Maintainability Index (including comments). This is a complex calculation involving a number of different metrics – EffortPart = 3.42 * *log*(HEFF/NOMT) CyclomaticPart = 0.23 * (TCC/NOMT) LinesPart = 16.2 * *log*(NOS/NOMT) CommentPart = 50 * *sin*(*sqrt*(2.46*(CCOM/NOMT))) Note here that 1. the log value used is the natural log 2. the sin value used is measured in radians We then combine the parts to create the MI value. – 171 – effortPart – cyclomaticPart – linesPart + commentPart Classes with a MI less than 65 are difficult to maintain, modules between 65 and 85 have reasonable maintainability and those with MI above 85 have excellent maintainability. In the standard jhawk properties file the warning level is set to 85 and the danger level to 65.
MINC	Maintainability index (No comments). This is calculated in the same way as MI above but the calculation does not include the commentPart.
MPC	Message Passing Coupling – the total number of external methods called by all the methods in the class.

NAME	Name of class
NCO	Number of commands (number of methods in the class that do not return a value)
NLOC	Total number of Lines of Code in the class. This is a total of the number of lines of code at class level plus a total of the number of lines of code for each of the methods defined in the class.
NOMT	Number of methods in class. A high number of methods in a class can be an indicator that the class is doing too much.
NOS	Total Number of Java Statements in class. This is a total of the statements at class level plus a total of the number of Java statements for each of the methods defined in the class. See definition of Java statements in 'Method level Metrics (all)' table
NQU	Number of queries (number of methods in the class that return a value)
NSUP	Number of superclasses to this class. If this is a class (rather than an interface) then the number of superclasses is all the classes above this class in the hierarchy (including the Object class). If this is an interface then it is a count of all the interfaces that this interface extends plus, recursively, all the interfaces that these extend.
NSUB	Number of subclasses of this class. All the classes that have this class in their hierarchy. This is calculated recursively down to the lowest classes in the hierarchy.
PACK	Number of packages imported by this class. A class that imports a large number of packages becomes more difficult to maintain due to these interdependencies.
R-R	Reuse ratio. This is calculated from the number of superclasses excluding Object divided by the total of the classes in the inheritance tree (including this class) – (NSUP-1)/((NSUP-1)+NSUB+1)
RVF	Review Factor (see section on Review Factor in this document) A value of 100 or more indicates that this class should be reviewed.
S-R	Specialization Ratio. This is calculated from the number of subclasses divided by the number of superclasses excluding Object - NSUB/(NSUP-1)
RFC	Response for class is calculated by totalling the number of methods declared in the class and the number of methods external to the class called from code within the class. i.e – NOMT +EXT A high value for RFC indicates a class that is more complex and therefore more difficult to test and maintain.
SIX	The SIX (Specialization Index) metric is calculated as follows - (nsomt*DIT)/(nsomt+NOMT+ inhmt) Where nsomt is the number of non-static overridden methods, DIT is the depth of inheritance tree (defined elsewhere in this table), NOMT is the number of methods defined in the class and inhmt is the number of methods that this class inherits from other classes. The Specialization Index metric measures the extent to which subclasses override their ancestors classes. A higher value of the SIX metric suggests that a class is more difficult to maintain. It is suggested as an indicator of classes that may require further investigation.
SUPER	Name of Superclass
TCC	Total Cyclomatic Complexity of all the methods in the class (see definition of Cyclomatic Complexity in 'Method level Metrics (all)' table)
UWCS	Unweighted class size. This is calculated by totalling the number of instance variables

	defined in the class and the number of methods defined in the class i.e. NOMT + INST A value for UWCS over 100 is viewed as indicator of poor code.

Method level metrics (all)

Metric Code	Description
CAST	Number of class casts in the method
COMP	Cyclomatic Complexity. This is calculated from the number of logical branch points in the method. The method itself is counted as 1 logical branch point. The ?, if, switch, for, while and catch operators count as logical branch points. Within a switch statement each occurrence of the break keyword is treated as a logical branch point. A Cyclomatic Complexity over 10 is viewed as an indicator of poor code. In the initial jhawk.properties file the ‘Warning’ level is set to 10 and the ‘Danger’ level to 15.
CREF	Number of different classes referenced in the method. This will include classes that are referenced in variable declarations, argument types, casts, exceptions thrown and caught, instantiations of variables through the new operator and direct references to class methods and variables. The CREF value includes both class and interface references. A method with a high number of class references can suggest that a method is doing too much.
EXCR	Number of exceptions referenced by this method. This figure does not include the exceptions referenced.
EXCT	Number of exceptions thrown by this method. These are the thrown exceptions identified in the method signature
HBUG	Estimated Halstead Bugs in the method. This is calculated by dividing the Halstead Volume (HVOL) by 3000.
HDIF	The Halstead Difficulty of a method is an indicator of method complexity. It is calculated from the number of unique operators (UOR), number of operands (NAND) and the number of unique operands (UAND) using the formula – (UOR/2)* (NAND/UAND)
HEFF	The Halstead Effort for the method is an indicator of the amount of time that it will take a programmer to implement the method. It is calculated from the Halstead volume (HVOL) and the Halstead Difficulty (HDIF) using the formula – HVOL*HDIF
HLTH	The Halstead Length of the method. This is the sum of the number of operators plus the number of operands. It is an indicator of method size.
HVOC	The Halstead Vocabulary of the method. This is the sum of the number of unique operators plus the number of unique operands. It is an indicator of method complexity.
HVOL	The Halstead Volume of a method is an indicator of method size. It is calculated from the Halstead vocabulary (HVOC) and the Halstead Length (HLTH) using the formula – HLTH * log2(HVOC)
LMET	Number of calls to local methods i.e. methods that are defined in the class of the method. A high number of method calls can be an indicator that the method is doing too much.
LOOP	Number of loops in the method. Loops are indicated by thefor and while operators.
MDN	Maximum Depth of Nesting. This is the depth of the deepest nested loop in a method. A Maximum Depth of Nesting of 4 or greater is viewed as an indicator of a method that is overly complex, difficult to test and that should be split into sub-methods. In the initial jhawk.properties file the ‘Warning’ level is set to 4 and the ‘Danger’ level to 6.
MOD	Number of modifiers (public, static, protected) etc in the method declaration
NAME	Name of method
NAND	Number of operands in the method. An operand is a Java token that can have operations carried out on it by other Java tokens (called operators – see above). Operands include variables, numeric and string literals, special variables such as true, false, null, void, super and this, classes and primitive types and methods.

NEXP	Number of Java Expressions in the method. A Java expression is a code fragment that evaluates to a single value e.g. var1 * var2 or “FIRST “+” STRING”.
NLOC	Number of Lines of Code in the method. A line of code is any non-blank line in a code file that is not a comment. A line of code with an end of line comment will be counted as a line of code and a comment.
NOA	Number of arguments in method signature. A very large number of arguments can suggest that a method is doing too much.
NOC	Number of comments. This is the number of discrete comments in the code. I.e. a multi-line comment will be counted as one comment.
NOCL	Number of comment Lines. This is the number of lines of comments in the code including the start and end tokens for the comments if these are on separate lines. An end of line comment on a line that also includes code will be counted as a comment line (it will also be counted as a line of code).
NOPR	Number of operators in the method. An operator is a Java token that is used to carry out an operation on another Java token (called an operand – see below). Examples of operators would be the arithmetic operators (+, -, *, / etc) and logical operators (&&, || , ! etc). Java keywords such as for and while are also operators.
NOS	Number of Java statements in the method. A Java statement is defined as a series of Java tokens terminated by a semi-colon.
TDN	Total Depth of Nesting. This is a total of the depth of nesting of all the loops in a method.
VDEC	Number of variables declared in the method. A very large number of variables declared can suggest that a method is doing too much.
VREF	Number of variable references in the method. This is the total number of references i.e. if var1 is referred to 3 times and var2 is referred to 4 times the value of VREF is 7. A very large number of variable references can suggest that a method is doing too much.
XMET	Number of calls to methods that are not defined in the class of the method. A high number of external calls will increase the dependence of the method (and hence its class) on other classes, making it more difficult to maintain the class. It can also be an indicator that the method is doing too much.

Using the Review Factor Metric

We have introduced the Review Factor metric as a means to allow our customers to develop metrics which can help them decide which code might need to be peer reviewed before being accepted into the main body of code. This can be useful when assessing the quality of legacy code or of code components created by an external source.

It is an implementation of the philosophy outlined in our discussion document – 'How can I find out which code needs to be reviewed?'- http://www.virtualmachinery.com/sidebar7.htm.

The metric is based on how assessing easy a piece of code is to read or understand. Code that is trivial (such as accessor methods) or that has little complexity is assumed to be less likely to be prone to error.

How the Review Factor Metric works

At each level a maximum review factor is defined. A number of criteria are defined with penalty values for each violation and a maximum value for the total of all violations. If

Package Level

At the package level the review factor for each class in the package is calculated. If one or more classes in the package exceed the maximum review factor for the class then the package is marked for review by returning a value of 100 plus the number of classes that have exceeded the class level maximum review factor

Class Level

At the class level the maximum review factor is set to 100.

Factor	Value	Notes
class.review.factor.max	100	
class.iv.max	30	Maximum value for instance variables
class.iv.mult	1	Multiple per instance variable
class.ivconstant.mult	0	Multiple where instance variable is a constant
class.countablemethods.max	10	Maximum value for countable methods
class.countablemethods.mult	0.5	Multiple for countable methods

Each instance variable is counted by adding the appropriate multiple (class.iv.mult for an ordinary instance variable, class.ivconstant.mult for a constant) to a running total. If the maximum value (class.iv.max) is exceeded then the maximum review factor (class.review.factor.max) is returned and no further processing of this class takes place – i.e. it is viewed as being a candidate for review.

Each method in the class is then processed as described in the method section below. If a method violates any of the following conditions then the class will be viewed as being a candidate for review. The score returned will be the class.review.factor.max plus a value of 1 for each countable method (JHawk treats countable methods as those that are not possible accessors. It defines accessor methods as those that start with "get", "set" or "is" followed by an instance variable name) that violates one of these constraints. So, in the case of the default values, if you receive a return value of 103 this will mean that three methods have violated at least one of the conditions and rendered the class a candidate for review. :

- Maximum cyclomatic complexity (method.cc.max)
- Maximum number of lines of code (method.loc.max) exceeded
- Maximum method review factor(method.review.factor.max) returned

For methods that do not violate any of these conditions a review value is created using the metrics defined at method level. The review values for each method are totalled and this is divided by the number of countable methods to give an average review value for the class.
In the final step we the number of countable methods and multiply this by the appropriate multiple (class.countablemethods.mult) – if this value exceeds the maximum (class.countablemethods.max) then the maximum value is used. This value is then added to the average review value for the class.

Method Level

At method level the following factors are used:

Factor	Value	Notes
method.review.factor.max	100	Maximum value for method review
method.cc.max	15	Maximum value for cyclomatic complexity. Exceeding this will cause the class to be reviewed
Method.cc.penaltylevel	5	Exceeding this cyclomatic complexity value will cause method.cc.penalty to be added to the method review factor
Method.cc.penalty	40	Penalty for exceeding method.cc.penaltylevel
method.numargs.penaltylevel	5	Exceeding this number of arguments will cause method.numargs.penalty to be added to the method review factor
method.numargs.penalty	40	Penalty for exceeding method.numargs.penaltylevel
method.vardecl.penaltylevel	5	Exceeding this number of variable declarations will cause method.vardecl.penalty to be added to the method review factor
method.vardecl.penalty	40	Penalty for exceeding method.vardecl.penaltylevel
method.casts.multiple	1	Multiple for casts
method.casts.penalty.max	10	Maximum penalty for exceeding number of casts times method.casts.multiple
method.loc.max	100	Maximum lines of code in a method. Exceeding this value means that the class containing the method will be marked for review
method.loc.multiple	0.5	Multiple per line of code
method.loc.penalty.level	50	Level at which lines of code start to be counted
method.loc.penalty.max	24	Maximum penalty which can be applied
method.extmethods.multiple	0	Multiple for each reference to an external method
method.extmethods.penalty.level	5	Level at which references to external methods start to be counted
method.extmethods.penalty.max	5	Maximum penalty which can be applied
method.expressiondensity.multiple	1	Multiple to be used when calculating if method.expressiondensity.penalty is to be applied.

method.expressiondensity.penalty.level	2	Level above which the expression density multiplied by method.expressiondensity.multiple will incur the method.expressiondensity.penalty
method.expressiondensity.penalty	40	Penalty applied if the expression density times method.expressiondensity.multiple exceeds the method.expressiondensity.penalty.level
method.samepackageref.multiple	1	Multiple to be applied to references to classes in the same package
method.samepackageref.penalty.max	5	Maximum penalty for numbers of references to classes in the same package
method.otherpackageref.multiple	1	Multiple to be applied to references to classes in other packages
method.otherpackageref.penalty.max	10	Maximum penalty for numbers of references to classes in other packages
method.interfaceref.multiple	1	Multiple to be applied to references to interfaces
method.interfaceref.penalty.max	5	Maximum penalty for numbers of references to interfaces
method.javalibref.multiple	1	Multiple to be applied to references to classes in the java libraries
method.javalibref.penalty.max	5	Maximum penalty for numbers of references to classes java libraries
method.thirdpartypackageref.multiple	2	Multiple to be applied to references to classes in third party libraries
method.thirdpartypackageref.penalty.max	10	Maximum penalty for numbers of references to classes in third party libraries

There is a common theme in these values where a multiple and a max value are defined. We can call this the standard penalty calculation – this is the minimum of the number of items that meet the criteria times the multiple, and the maximum. So if there are 20 items, the multiple is 1 and the maximum is 10 the value 10 will be returned. If there are 9 items, the multiple is 1 and the maximum is 10 the value 9 will be returned.

In some cases a level is also defined – this is a level after which we apply the multiple. i.e. number of items – penalty level times multiple. So if there are 20 items, the penalty level is 5, the multiple is 2 and the maximum is 50 the value 30 will be returned ((20-5)*2). As before if the maximum is exceeded the maximum will be returned. If no multiple is defined the multiple is assumed to be 1.

Cyclomatic complexity – if the cyclomatic complexity exceeds the penalty level then the penalty is applied
Number of Arguments– if the number of arguments exceeds the penalty level then the penalty is applied
Variable declarations – if the number of variable declarations exceeds the penalty level then the penalty is applied
Casts - The penalty applied is the maximum of the number of casts times the multiple or the maximum penalty
Lines of code – the penalty is calculated from the number of lines of code less the penalty level times the multiple. If the value exceeds the maximum penalty then the maximum penalty is used.
Expression density – The expression density is calculated by dividing the number of expressions by the number of statements. This is then multiplied by the expression density multiple. If the value exceeds the maximum penalty then the maximum penalty is used.

External method references - the penalty is calculated from the number of external method references less the penalty level times the multiple. If the value exceeds the maximum penalty then the maximum penalty is used.
External class references – The external class references are divided into different categories :

- Classes in the same package
- Classes in another package included in the analysis set
- References to interfaces
- Classes in one of the Java libraries (package name starts with "java." or "javax.")
- Classes in another third party library

References can only be in one of these categories. In each case the review factor is updated using the standard penalty calculation described above.

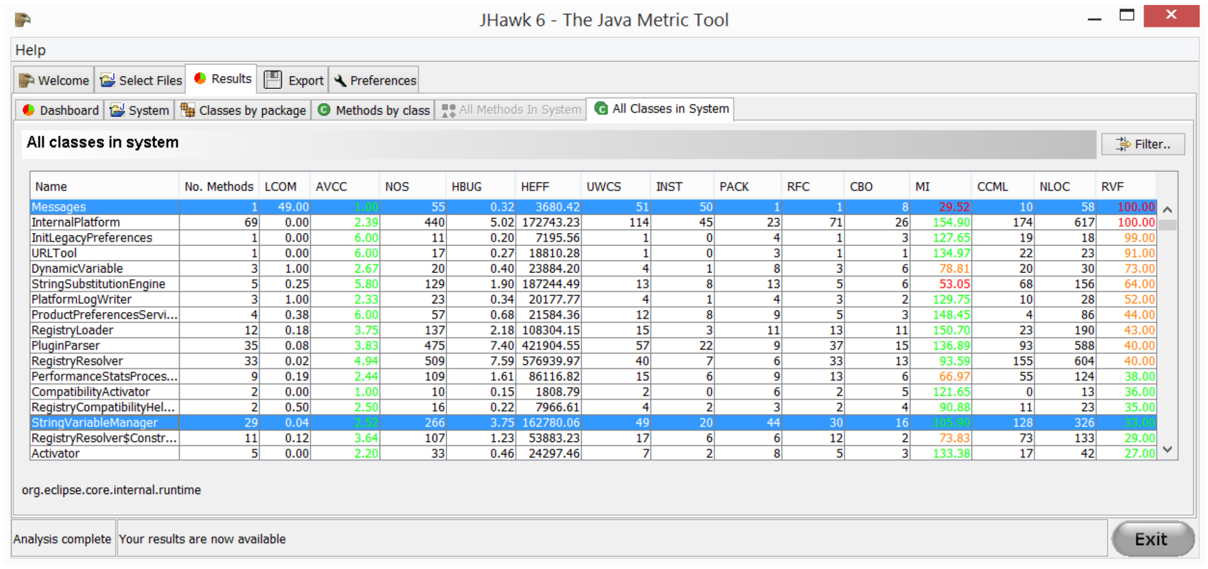
The method review factor is created by adding up all the penalties applied to the method. If the penalties exceed the maximum method review factor value then the maximum method review factor will be returned.

Modifying values to be used in the Review Factor metrics

Any of the values can be modified by changing the corresponding values in the reviewfactor.properties file. The initial file contains the default values. You can adjust the values to suit your own requirements.
If you wish to remove a particular aspect from the calculations then the simplest approach is reduce the multiple to zero (if there is a multiple involved) or to set the penalty to zero.

If you want to modify the values used in the Review Factor metrics a good approach is to take a known body of code and separate it into code that you would feel happy leaving out of a review and code that you think should be reviewed. Run the analysis of your code using the default values of the Review Factor and look at which pieces of code fall on either side of the line and see if that matches your original opinion. Using the values for other metrics produced by JHawk you can start to adjust the factors contributing to the Review Factor accordingly. For example in the data below we can see two classes (highlighted on the screenshot below) - 'Messages' and 'StringVariableManager'. 'Messages' has a review factor of 100 and yet doesn't seem very 'busy' compared to 'StringVariableManager' – the main thing separating them is the number of instance variables, and when we look at the criteria in 'reviewfactor.properties' we see that the maximum value for the instance variables calculation is 30 and the multiple is 1 (but it is zero for constants). We have a couple of potential strategies here if we want to take this class out of the review category –

- If we believe that the code is correct we can either increase the maximum value or we can decrease the multiple
- If we believe that the code should be improved and that we should create the instance variables as constants then we could change the code to make them constants - this would mean that they would not count towards the instance variable penalty.



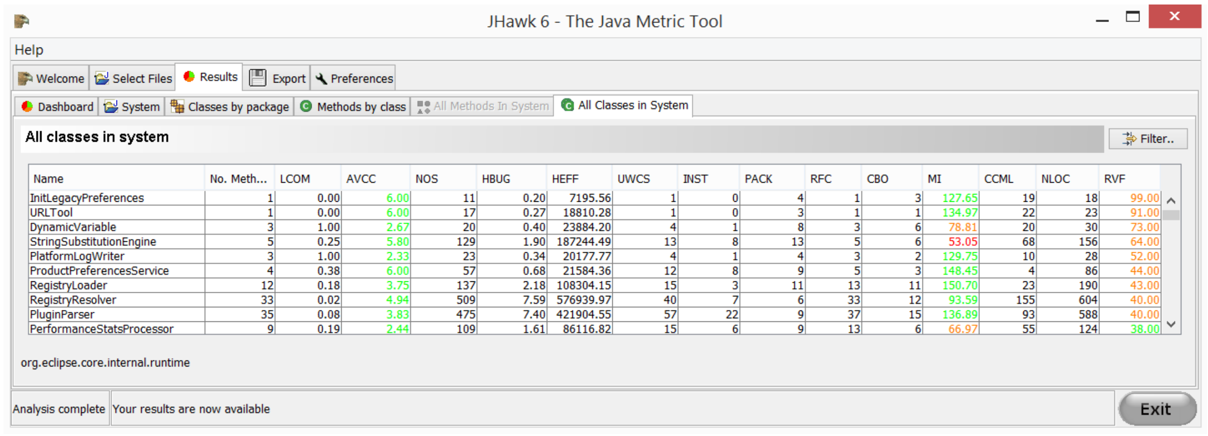
All classes in system

Name	No. Methods	LCOM	AVCC	NOS	HBUG	HEFF	UWCS	INST	PACK	RFC	CBO	MI	CCML	NLOC	RVF
Messages	1	49.00	1.00	55	0.32	3680.42	51	50	1	1	8	29.52	10	58	100.00
InternalPlatform	69	0.00	2.39	440	5.02	172743.23	114	45	23	71	26	154.90	174	617	100.00
InitLegacyPreferences	1	0.00	6.00	11	0.20	7195.56	1	0	4	1	3	127.65	19	18	99.00
URLTool	1	0.00	6.00	17	0.27	18810.28	1	0	3	1	1	134.97	22	23	91.00
DynamicVariable	3	1.00	2.67	20	0.40	23884.20	4	1	8	3	6	78.81	20	30	73.00
StringSubstitutionEngine	5	0.25	5.80	129	1.90	187244.49	13	8	13	5	6	53.05	68	156	64.00
PlatformLogWriter	3	1.00	2.33	23	0.34	20177.77	4	1	4	3	2	129.75	10	28	52.00
ProductPreferencesServi...	4	0.38	6.00	57	0.68	21584.36	12	8	9	5	3	148.45	4	86	44.00
RegistryLoader	12	0.18	3.75	137	2.18	108304.15	15	3	11	13	11	150.70	23	190	43.00
PluginParser	35	0.08	3.83	475	7.40	421904.55	57	22	9	37	15	136.89	93	588	40.00
RegistryResolver	33	0.02	4.94	509	7.59	576939.97	40	7	6	33	13	93.59	155	604	40.00
PerformanceStatsProces...	9	0.19	2.44	109	1.61	86116.82	15	6	9	13	6	66.97	55	124	38.00
CompatibilityActivator	2	0.00	1.00	10	0.15	1808.79	2	0	6	2	5	121.65	0	13	36.00
RegistryCompatibilityHel...	2	0.50	2.50	16	0.22	7966.61	4	2	3	2	4	90.88	11	23	35.00
StringVariableManager	29	0.04	2.52	266	3.75	162780.06	49	20	44	30	16	105.90	128	326	33.00
RegistryResolver$Constr...	11	0.12	3.64	107	1.23	53883.23	17	6	6	12	2	73.83	73	133	29.00
Activator	5	0.00	2.20	33	0.46	24297.46	7	2	8	5	3	133.38	17	42	27.00

org.eclipse.core.internal.runtime

Analysis complete Your results are now available

If we take the first approach and increase the allowable penalty to 60 (rather than 30) then we can see that Messages is no longer over the review threshold (neither is 'Internal Platform' – which also exceeded the instance variable limit) :-



All classes in system

Name	No. Meth...	LCOM	AVCC	NOS	HBUG	HEFF	UWCS	INST	PACK	RFC	CBO	MI	CCML	NLOC	RVF
InitLegacyPreferences	1	0.00	6.00	11	0.20	7195.56	1	0	4	1	3	127.65	19	18	99.00
URLTool	1	0.00	6.00	17	0.27	18810.28	1	0	3	1	1	134.97	22	23	91.00
DynamicVariable	3	1.00	2.67	20	0.40	23884.20	4	1	8	3	6	78.81	20	30	73.00
StringSubstitutionEngine	5	0.25	5.80	129	1.90	187244.49	13	8	13	5	6	53.05	68	156	64.00
PlatformLogWriter	3	1.00	2.33	23	0.34	20177.77	4	1	4	3	2	129.75	10	28	52.00
ProductPreferencesService	4	0.38	6.00	57	0.68	21584.36	12	8	9	5	3	148.45	4	86	44.00
RegistryLoader	12	0.18	3.75	137	2.18	108304.15	15	3	11	13	11	150.70	23	190	43.00
RegistryResolver	33	0.02	4.94	509	7.59	576939.97	40	7	6	33	12	93.59	155	604	40.00
PluginParser	35	0.08	3.83	475	7.40	421904.55	57	22	9	37	15	136.89	93	588	40.00
PerformanceStatsProcessor	9	0.19	2.44	109	1.61	86116.82	15	6	9	13	6	66.97	55	124	38.00

org.eclipse.core.internal.runtime

Analysis complete Your results are now available

Looking at the class Messages it can be seen that the instance variables, although static, are in fact unpopulated and so the class could not be changed without affecting its functionality.

The Review Factor metric is an experimental metric – we have provided the code for the metric at both package and class level in the appendices to this document. This will show you how we have implemented the metric and will help you if you want to modify this metric or to use it as a basis for your own metric.

You will also find the document 'JHawk6CreateMetric' (included in the JHawk documentation) useful as it contains a complete list of the interface calls available for use in customised metrics.