# Identification of components

For the identification of components, that could be merged, a number of source code metrics are computed. The overall score of these metrics is evaluated for two components at a time. The overall score can be influenced by weighting each metric with a probability (an integer value between 0-100). How the values from the user dialog (see figure 1) affect the weights of each score is documented in *1.2* .

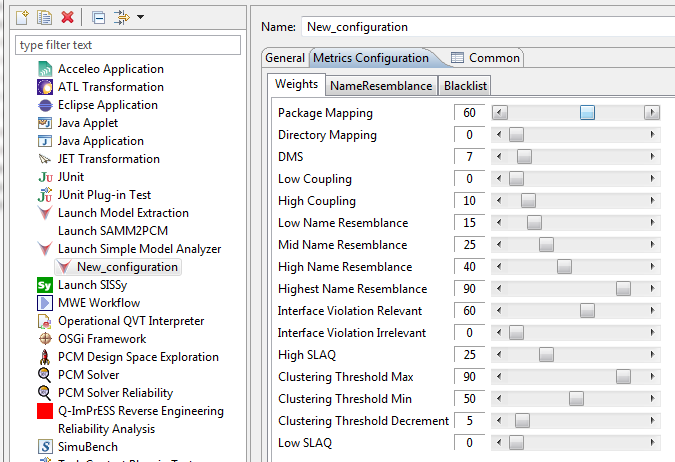


Figure Configuration of source code metrics weights in SoMoX (screenshot from the Run Configuration dialog)

## Overall score

The final score Q is calculated as follows. Figure 2 also gives an overview of the dependencies of the metrics and weights



Figure Calculation of the final score

The functions F1-F6 determine the weights. This is explained in detail in the following section.

## Description of metrics

### Package mapping

Indicator if the components can be mapped to a single package

The weight WP is directly applied from the run dialog by multiplication.

### Directory mapping

Same indicator like package mapping but based on directories instead of packages

The weight WD is also directly applied from the run dialog by multiplication.

### Distance from Main Sequence (DMS)

This metric combines abstractness and instability to rate components.

#### Abstractness

This metric measures the object-oriented abstractness of a component.

|  |  |
| --- | --- |
|  | within a component  within a component |

A Value of “1” characterizes abstract classes; concrete classes will achieve a score of “0”.

#### Instability

Instability characterizes the dependency of a component to others. Components with an instability value of “1” are considered to be instable, because they depend in a very high degree on other components. On the other hand, components with a score of “0” are stable because they depend only on themselves.

|  |  |
| --- | --- |
|  |  |



Figure Example class diagram

The example class diagramm in Figure 3 shows class B that is derived from class A, provides/realizes the interface If1 and is referenced by class C. This leads to the following instability score:

The ideal combination of abstractness and instability for “good” components is described by Figure 4 – the Main Sequence.



Figure Graph of the main sequence

It is shown that abstract components (A=1) should be perfectly stable (I=0) and concrete components (A=0) should be instable (I=1). The connecting line indicates the desired kind of components. Unwanted combinations are i.e. A=0 and I=0. Concrete (no interfaces) that are stable are hard to modify. The other extreme (marked in the Figure 4) A=1 (completely abstract) and I=1 (instable) is also not wanted, because no other class is depending on it.

The DMS metric gives the normalized (to provide a range of values of [0.0; 1.0]) distance from all these ideal combinations.

|  |  |
| --- | --- |
|  |  |

The desired value should be as small as possible. The marked worst case constellation in Figure 4 achieves a score of 1.

### Coupling

Determines the coupling from component A to B. This metric does not influence the overall score directly but is considered for the evaluation of Interface violation and Name resemblance

|  |  |
| --- | --- |
|  |  |

### **Clustering** threshold

To control the merging of components two thresholds can be given. Only those components, whose score lies in the stated range (given by minimum and maximum threshold) will be considered to be merged.

### Interface violation

Measure of interface communication between component A and B.

The example states an interface violation. Class A references class B indirectly by using interface 1 but also directly.

<<class B>>

<<class A>>

<<Interface 1>>

<<realize>>

<<use>>

The interface violation weight (WIV) depends on the coupling (C) and the interface violation score itself:

### Name resemblance

Ratio of same class names of a component and the total number of classes.

Which name resemblance weight (WSN) is used depends of the coupling (C) and name resemblance (SN) score. The dependencies are also illustrated in Figure 5 Computation of Name Resemblance weight.



Figure Computation of Name Resemblance weight

### Sliced Layer Architecture Quality (SLAQ)

The Sliced-Layer-Architecture assumes that the source code is organized in layers (technology orientated) and slices (service orientated). With this structure natural subsystems can be identified.

View

Controller

Domain

Data Access

Contracts

Customers

Users

The used subsystems can be found by examining the structure of packages. Slices are the first sub-packages following an identical prefix of all packages. All deeper following packages are considered to be layers.

com.softhouse.manager.contracts.view

com.softhouse.manager.contracts.administration

com.softhouse.manager.contracts.data

com.softhouse.manager.customers.view.administration

com.softhouse.manager.customers.view

com.softhouse.manager.customers.data

The number of expected subsystems is calculated from the maximum of occurring layers \* slices. This leads to six subsystems in total for the example above.

### Subsystem component metric

Checks if all a components can be mapped to a natural system