ARiSA First Contact Analysis™

Applied Research In System Analysis™ - ARiSA™

"You cannot control what you cannot measure"

Tom DeMarco

Software	Grail
Version	1.3
Programming Language	Java 1.4

Date	2nd of December, 2004
Contact	Applied Research in System Analysis™ – ARiSA™ +46 470 708495 Welf.Lowe@vxu.se
Responsible for Analysis	Welf Löwe

1 The ARISA First Contact Analysis™ Approach

ARiSA First Contact Analysis™ is an assessment method for software systems. It is based on a number of automated analyses. They give a graphical overview of the systems:

- Architecture and structure,
- Design, and
- Complexity.

Such an overview is a valuable documentation of the system allowing to teams members a better communication on system matters. New team members experience a lower learning curve when trying to comprehend an unknown system.

Since all analyses are objective and their results clearly separated from our conclusion, both project managers and developers might experience insights on actual system properties and their divergence from expected and intended system properties.

In addition to the system overview, ARiSA First Contact Analysis™ pinpoints parts of the system that might be critical in further maintenance of the system. ARISA™ recommends reviewing and refactoring specifically these system parts if necessary.

ARiSA First Contact Analysis™ is based a tool called VizzAnalyzer™. It implements state-of-the-art structural program analyses, checkers for best practice designs (design patterns), and metrics calculations. Results are presented on various levels of abstraction with software visualizations and statistics charts. The VizzAnalyzer supports both the identification of weak points or design flaws in software systems and a better understanding of the analyzed system.

An appropriate architecture and good design cannot be formalized and correctly measured. However, research in the field of software architecture and design propose a number of heuristics. Our partner, the Software Technology Group at Växjö University headed by Prof. Welf Löwe, develops such heuristics. These are direct inputs to the ARiSA First Contact Analysis™. Additionally, ARiSA First Contact Analysis™ is strongly influenced by heuristics defined by other leading edge research groups in Europe. Especially, the research group at the Research Center Computer Science in Karlsruhe, Germany, who are pioneers in the field of automatic design support, contributed with many worthwhile heuristics. Moreover, all heuristics applied are confirmed in numerous practical field studies.

Heuristics embody knowledge about good design, which has been proven to be valid in numerous industrial cases studies. However, finally, only system architects, designers, and developers knowing the system are able to properly interpret the analysis results and identify false alarms. Hence, ARiSA First Contact Analysis™ is only an initial step that needs to be followed-up by discussions among the people responsible for the system. The software visualizations and statistics charts that the ARiSA First Contact Analysis™ provides are have proved an excellent basis therefore.

2 Summary

ARiSA™ performed the ARiSA First Contact Analysis™ on the system *Grail*.

1. *Grail* is a small-medium size system. The average of 9 classes per top-level package (subsystem) is rather small. The package hierarchy is flat, just contains `grail` and all 11 subsystem-packages. The averages of 8 methods and 92 code lines per class, respectively, are reasonable values and do not uncover deviations from state-of-the-art design rules.
2. We observe that the declared packages do not contain natural components. If this were the intension major restructuring reducing the interactions over package boundaries would be recommended.
3. The inheritance structure does not show any violations to standard designing guidelines. It should be checked if all of them are planned being extended in the future or if some could be removed as unnecessary abstraction. There are no `MatrixBasedUndirectedGraph`, `MatrixBasedUndirectedNode`, and `MatrixBasedUndirectedEdge` classes, as one would expect when looking at the naming and inheritances of related classes.
4. The classe `grail.properties.GraphProperties` shows in general a high external coupling and low internal cohesion at the same time. This class also shows in general a high change possibility and lack of documentation at the same time. This is a critical class in the system. It should be better documented and, maybe refactored.
5. The class `grail.matrixbased.MatrixBasedDirectedGraph` has a high change possibility but appears to be well documented. Because of its high change possibility, it should be documented better.
6. The following classes are both complex and large:

   a. `grail.matrixbased.MatrixBasedDirectedGraph`

   b. `grail.converters.GML`

   c. `grail.interfaces.AbstractGraph`

   d. `grail.DefaultTreeView`

   These classes might be considered splitting. Special attention should be paid to the class `grail.matrixbased.MatrixBasedDirectedGraph`. This class was recommended to re-engineer because of its high change possibility.

3 Global Statistics

In most cases, it is a good idea to measure some basic properties of the software system we are dealing with. Knowledge of some basic numbers indicating the size of the system may help to put the more specific measurements in later sections of this report into a better context. **Table 1** summarizes these basic figures for the *Grail* system as of December 2, 2004.

Top level packages (subsystem)	11
Packages	12
Classes	85
Interfaces	15
Methods	785
Public Attributes	77
Lines of code	9.275

Table 1 Global Statistics Measurements

Interpretation: *Grail* is a small-medium size system. The average of 9 classes per top-level package (subsystem) is rather small. The package hierarchy is flat, just contains `grail` and all 11 subsystem packages. The averages of 8 methods and 92 code lines per class, respectively, are reasonable values and do not uncover deviations from state-of-the-art design rules.

4 Analysis of Architecture and Structure

We will analyze both architectural properties and properties of the inheritance structure.

4.1 System Architecture

The architecture of a system consists of components and their interactions. Components are interacting sets of classes and smaller components, i.e. the notion of a component is recursive. Interactions include method calls and field accesses.

An architecture is considered good if its components have a high internal cohesion and low external coupling. The cohesion of a component is defined as the degree interactions among contained classes and components. The coupling of a set of components is defined as the degree interactions among them. The idea is that components do a lot of the work internally and only rarely communicate with other components. This allows, e.g., maintaining, reusing or understanding individual components regardless of other components around. Our analyses will therefore check for high internal cohesion and low external coupling in the components.

On architectural level, different styles have been proven worthwhile for different kinds of systems. For instance, a Tree-Tier-Architecture helps to separate presentation related components, from business logic and data storage. System architects and designers always have an architectural style in mind when designing their system. However, in development and maintenance the style might get deteriorated due to time pressure or misunderstandings. Our analyses check the conformance of the existing architecture in the system with the intended architectural style to uncover deviations.

4.1.1 Evaluation of the Architecture

Figure 1. shows the architecture of the system. It compares „natural“ components and declared packages. Natural components are sets of classes with high cohesion among another and low coupling to the rest of the system. Declared packages of the systems are classes/interfaces of the same top-level package. Note, that packages might be used for structuring of the system that is orthogonal to the component structure, e.g. bringing together classes solving similar problems.

Interpretation: We observe that the declared packages do not contain natural components. This is ok in library systems. However, if it were the intension to declared packages along with components major restructuring reducing the interactions over package boundaries would be recommended.

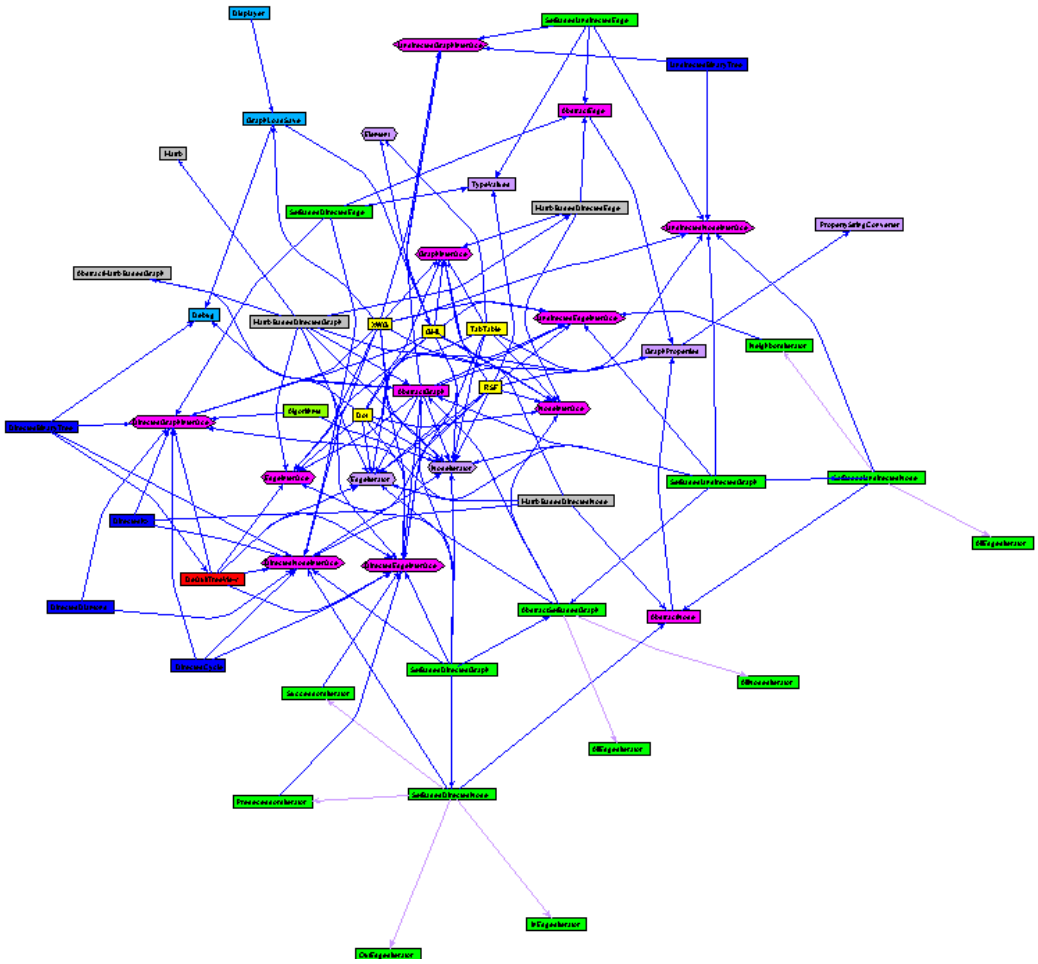

Figure 1. The class interaction graph shows classes and interface nodes and their interaction, i.e. calls and field accesses. The colors indicate declared packages of the systems where classes/interfaces of the same top-level package are depicted with the same color. This shows the architecture of *Grail*. No „natural“ components become obvious, i.e. sets of classes with high cohesion among each other and low coupling to the rest of the system.

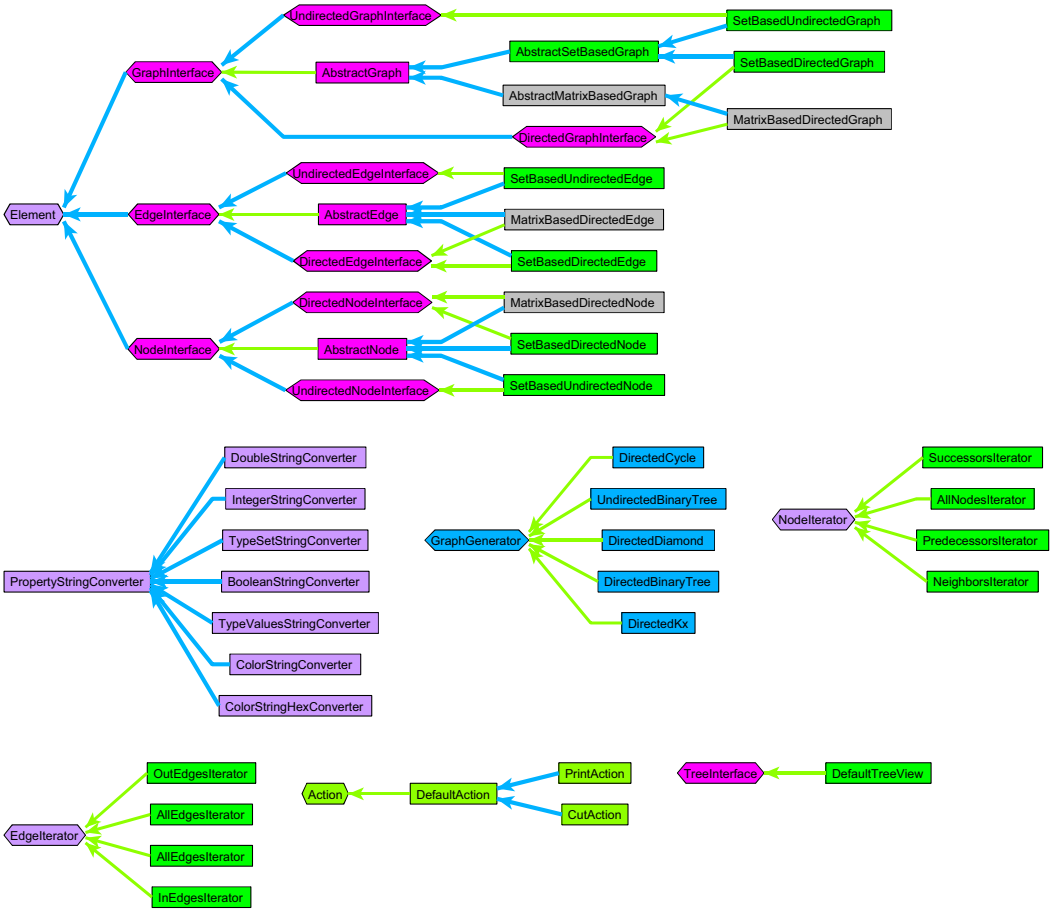

Figure 2. The inheritance structure of *Grail*. Nodes are classes (boxes) and interfaces (diamonds). The color scheme of nodes encodes the top-level packages of classes and interfaces. The color scheme of edges distinguishes implements (green) and extends (blue) relations.

4.2 Inheritance Structure

The inheritance structure of a system is defined by implements and extends relations of classes and interfaces. Such a structure is expected to follow a few general design rules.

Chains in the inheritance structure, i.e. substructures where super-classes/-interfaces only have one single sub-class/-interface and are themselves the only super-classes/-interfaces are considered a sign of unnecessary abstraction. In libraries and frameworks however, further subclasses might come form the application using them. Hence, chains do not necessarily indicate a design flaw.

Inheritance structures that contain direct and indirect, i.e. transitive, relations between two classes are to avoid since the transitive relation does not contributed to the system semantics.

Inheritance structures should not cross too many package boundaries. Moreover, they should neither be too deep nor too wide. These are rather fuzzy recommendation. Even though, one should observe the average in a system and have a closer look at escapes.

Figure 2. depicts the inheritance structure of *Grail*. Nodes represent classes/interfaces; edges represent extends/inherits relations. Again, the color scheme encodes the top-level packages of classes and interfaces. It only depicts classes/interfaces that are in an inheritance relation.

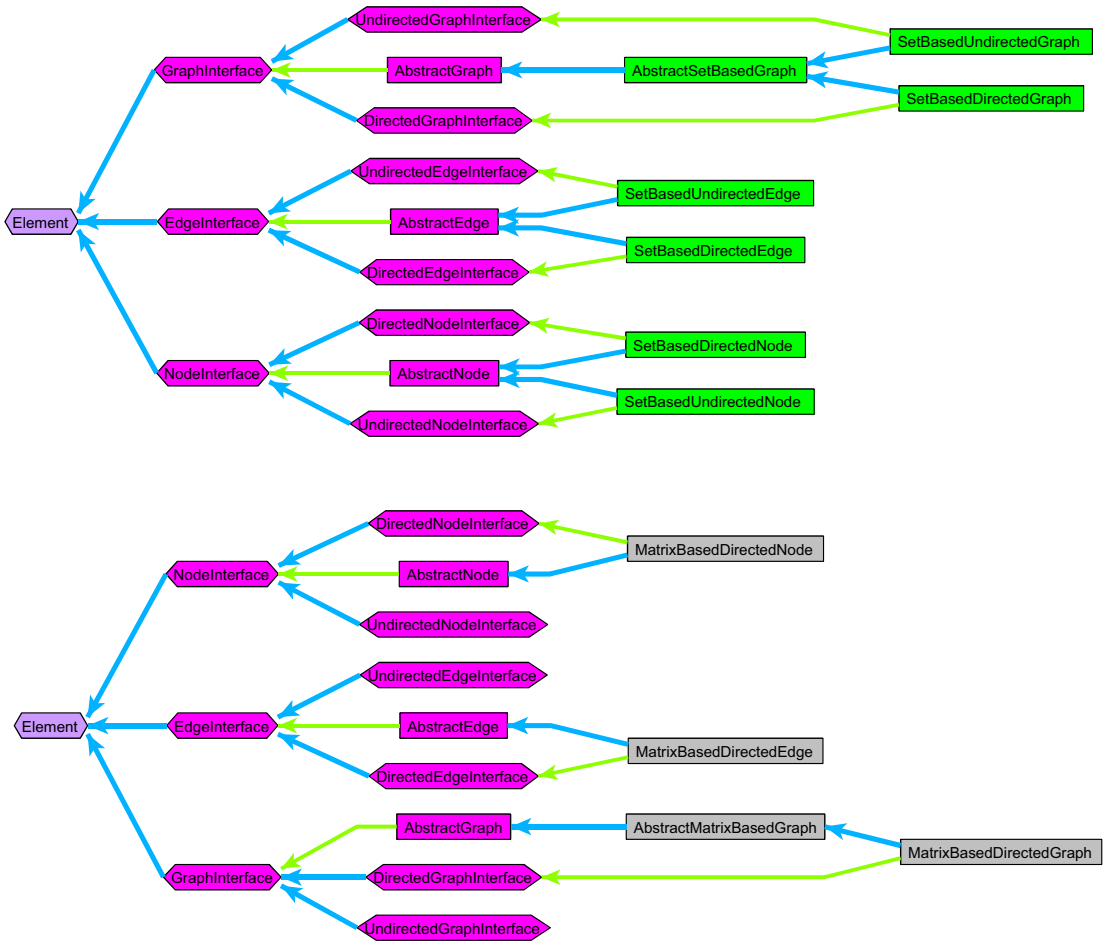

Figure 3. Copies a part of inheritance structure of *Grail*. Depicted are inheritance structures related to `SetBased*` (top, green) and `MatrixBased*` (bottom, gray) classes.

In the major inheritance structure, there is only on chain from the interface `TreeInterface` to the class `DefaultTreeView`..

We do not observe any two classes with both direct and indirect inheritance relations.

There are at most three packages involved in any of the inheritance hierarchies; the maximum inheritance depth is 4, maximum width is 7. Only the latter could be considered an escape from the average.

Figure 3. depicts the part of the inheritance structure of *Grail* that is related to `SetBased*` and `MatrixBased*` classes. The name schema of these classes and their dependency induce an intended similarity of these two sets of classes. The respective inheritance structures are diverging.

Interpretation: The inheritance structure does not show any violations to standard designing guidelines. It should be checked if all of them are planned being extended in the future or if some could be removed as unnecessary abstraction. There are no `MatrixBasedUndirectedGraph`, `MatrixBasedUndirectedNode`, and `MatrixBasedUndirectedEdge` classes, as one would expect when looking at the naming and inheritances of related classes.

5 Design Metrics

In the same way cohesion and classes indicate good design of components, these measures are applicable to assess the design of individual classes. Here, we expect high cohesion of methods and fields within a class and a low coupling between classes.

Tight Class Cohesion (TCC) is the relative number of directly connected methods in a class. TCC indicates the degree of connectivity between visible methods in a class. Given the number *n* of local methods (excluding inherited methods), TCC is defined as

$$\text{TCC} = \frac{ndp}{np}$$

with $np = \frac{n \times (n-1)}{2}$ the possible pairs of these methods and *ndp* the number of method pairs actually calling another.

The TCC for a class is 0 if $np = 0$. The resulting values range from 0.0 to 1.0 on a rational scale; we scale them between 0 and 100. Higher values indicate better cohesion of the classes. Low values indicate that a class could be split.

Data Abstraction Coupling (DAC) represents the number of abstract data types (ADTs) defined in a class. Counted are the fields defined in a class referencing a user defined type, not a primitive, language or library defined type. Inherited fields are not counted. The DAC is calculated for each class and interface. The values are integer values ranging from 0, indicating no other ADT is referenced, to a maximum number on an absolute scale; we scaled them between 0 and 100.

The higher the DAC is, the more complex is the coupling of a class with other classes. It is recommended to keep DAC low, or merge some classes, otherwise.

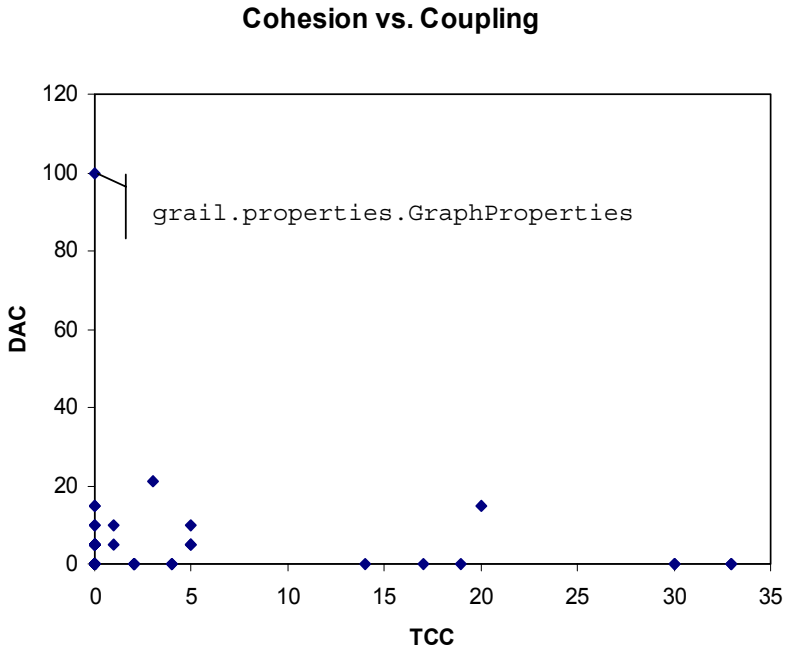

Chart 1. Relative intra-class-cohesion (TCC) and relative inter-class-coupling (DAC).

Chart 1. depicts the values of TCC and DAC for classes and interfaces of *Grail*. Each class/interface is a point in the diagram; its *x*-position indicates the class'/interface's TCC value, its *y*-position indicates the class'/interface's DAC value. Both values are linearly scaled between 0-100.

Since, high cohesion and low coupling are desired, classes in the top-left corner are to review.

Change Dependency Between Classes (CDBC) determines the potential amount of follow-up work to be done in a client class when a server class is modified. It also indicates the strength of a coupling. The goal is to keep changes in the system local by reducing the system coupling. Lower values indicate lower coupling and, thus, a better stability in the system.

The CDBC value is defined between a client and a server class as the number of methods, which need to be (potentially) changed in the client if a server changes. The CDBC value is between 0 and the count of methods in the class. We compute the average CDBC value of each (client) class over all (server) classes it is directly connected with. The scale is rational.

Lack of Documentation (LOD) measures the amount of undocumented declarations per class (counted are the class declaration itself and the method declarations, but not field declarations). Only "JavaDoc" style documentation is taken into account. Documentation within methods is ignored. Only the syntax of the comments is parsed, not the semantics.

The LOD value is calculated for each class or interface as the number of undocumented declarations (local not inherited methods). A LOD of 0 indicates that all possible entities are documented; a higher value indicates the lack of documentation.

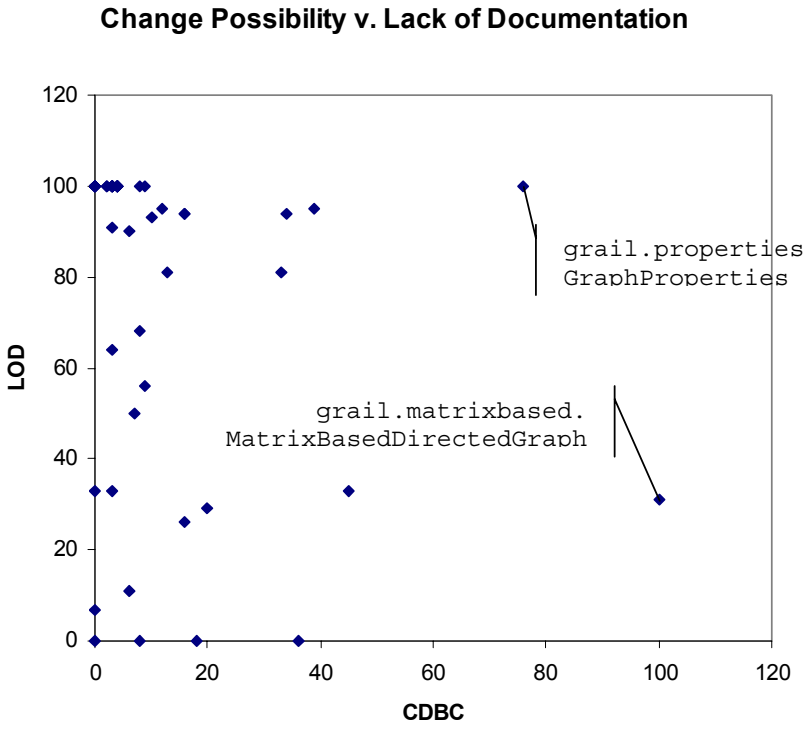

Chart 2. Relative possibility of changes of a class (CDBC) and relative lack of documentation of classes and methods (LOD).

Chart 2. depicts the values of CDBC and LOD for classes and interfaces of the *VizzAnalyzer*. Each class/interface is a point in the diagram; its *x*-position indicates the class'/interface's CDBC value, its *y*-position indicates the class'/interface's LOD value. Both values are linearly scaled between 0-100, where the highest absolute values measured in each dimension are mapped to 100.

Since, low possibility of changing a class/interface and low lack of documentation are desired, class/interface in the top-right corner are to review. These classes are likely to be changed and are poorly documented at the same time. Class/interface in the bottom-right corner might be critical, too. These classes are likely to be changed, too, but at least well documented.

Interpretation: The classe `grail.properties.GraphProperties` shows in general a high external coupling and low internal cohesion at the same time. This class also shows in general a high change possibility and lack of documentation at the same time. This is a critical class in the system. It should be better documented and, maybe refactored.

The class `grail.matrixbased.MatrixBasedDirectedGraph` has a high change possibility but appears to be well documented. Because of its high change possibility, it should be documented better.

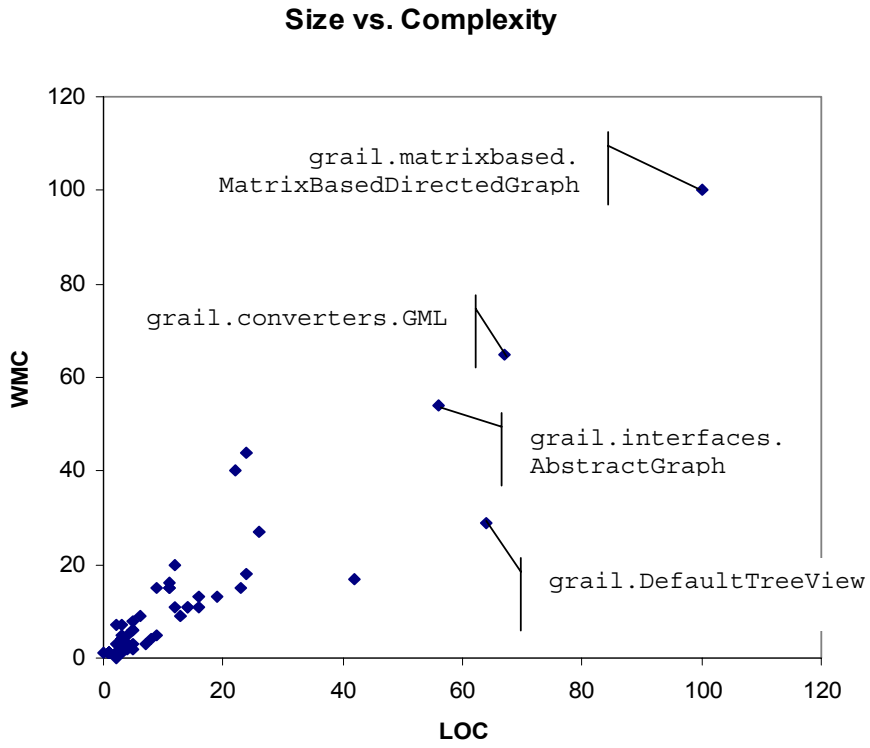

Chart 3. Relative size in lines of code (LOC) and relative complexity in weighted method complexity (WMC).

6 Complexity Metrics

The complexity of a class is often considered interesting since it points at classes that are hard to understand and to maintain.

Weighted Method Count (WMC) computes the complexity of the methods of a class and is also a measure for the complexity of a class. It gives a good idea about how much effort is required to develop and maintain a class. The methods are weighted according to McCabe's Cyclomatic Complexity Metric. It counts the possible execution branches in a method for the branching statements: `if`, `for`, `while`, `do`. It is assumed that each branch has the same complexity/weight. We scaled the values between 0 and 100.

Lines of Code (LOC) count the lines of code in a class and interface, respectively. It's an absolute metric that we scaled between 0 and 100.

Classes that are both highly complex and large are critical when it comes to understanding and maintaining a system. It is especially alerting if they are recommended to be restructured because of structural and/or design issues (see previous sections).

Chart 3. depicts the values of LOC and WMC for classes and interfaces of the *VizzAnalyzer*. Each class/interface is a point in the diagram; its *x*-position indicates the class'/interface's LOC value, its *y*-position indicates the class'/interface's WMC value. Both values are linearly scaled between 0-100, where the highest absolute values measured in each dimension are mapped to 100.

Since, high complexity of class/interface and large classes are critical, class/interface in the top-right corner are to review. These classes are hard to be changed.

Interpretation: The following classes are both complex and large:

1. grail.matrixbased.MatrixBasedDirectedGraph
2. grail.converters.GML
3. grail.interfaces.AbstractGraph
4. grail.DefaultTreeView

These classes might be considered splitting. Special attention should be paid to the class `grail.matrixbased.MatrixBasedDirectedGraph`. This class was recommended to re-engineer because of its high change possibility (CDBC).

7 Conclusions

This report documented the results of the ARiSA First Contact Assessment of *Grail*. Our assessment included and architecture analysis in general, a comparison between intended and actual architecture, an analysis of the inheritance structure, analyses of coupling and cohesion on class level, and analyses of the understandability and maintainability of the software.

We separated analyses from interpretation. All analyses results are factual. However, we need to emphasize that their interpretation should be taken with care. It can only point to suspicious spots of the system. Developers and designers experienced with the system should crosscheck each interpretation.

None of the results revealed a severe problem. We conclude that the current design and implementation of the software provides a solid foundation for future development cycles.

It is recommended to integrate a quality assessment in this future development. The ARiSA group would be like to give support in that venture with both tools and expert knowledge.