ARiSA First Contact Analysis™

Applied Research In System Analysis™ - ARiSA™

"You cannot control what you cannot measure"

Tom DeMarco

Software	VizzAnalyzer
Version	1.03
Programming Language	Java 1.4

Date	13th of November, 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 First Contact Analysis™ on the system *VizzAnalyzer*.

1. The *VizzAnalyzer* is a medium size system. The averages of 59 classes per subsystem and 8 methods and 90 code lines per class, respectively, are reasonable values and do not uncover deviations from state-of-the-art design rules.
2. Moving the packages `hlanalyses` and `analyses` into a common package `analysis` and the packages `layout`, `vizz3d`, and `visgraph` into a common package `visualization`, respectively, would lead to a top-level packaging reflecting the top-level architecture of the system. The only class in package `data` should be moved into the `recoder` package. The direct interaction of classes in the `recoder` and the `analyses` packages breaks the intended centralized architectural style. It is highly recommended to refactor it.
3. The inheritance structure does not show any severe violations to standard designing guidelines. However, there are quite view inheritance chains of length two. It should be checked if all of them are planned being extended in the future or if some could be removed as unnecessary abstraction.
4. The following classes show in general a high external coupling and low internal cohesion at the same time:
    a. vizz3d.java3d.visengine.interactions.J3dAdvancedInteraction
    b. vizz3d.common.gui.MainFrameInitiator
    c. recoder.llanalysis.basetree.recoder.RecoderElementVisitor
    d. recoder.llanalysis.LLAInfo
5. The following classes show in general a high change possibility and lack of documentation at the same time:
    a. recoder.llanalysis.llaelements.LLAPackage
    b. analyzer.hlanalysis.HLAnalysis
    c. recoder.llanalysis.basetree.baseelements.BTType
    d. recoder.llanalysis.llaelements.LLAMethod

    The class vizz3d.opengl.visengine.util.GLAppearance has a high change possibility but appears to be well documented. These classes in the packages recoder. llanalysis and vizz3d might be considered worth being reengineered.
6. The following classes are both complex and large:
    a. recoder.llanalysis.basetree.recoder.RecoderElementVisitor
    b. vizz3d.opengl.visengine.Interactions.GLAdvancedInteraction

    These classes might be considered splitting. Special attention should be paid to the class recoder.llanalysis.basetree.recoder.RecoderElementVisitor. This class was

recommended to re-engineer because of its high external coupling and low internal cohesion.

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 *VizzAnalyzer* system as of November 13, 2004.

Top level packages (subsystem)	9
Packages	83
Classes	472
Interfaces	61
Methods	4.478
Public Attributes	327
Lines of code	47.737

Table 1 Global Statistics Measurements

Interpretation: The *VizzAnalyzer* is a medium size system. The averages of 59 classes per subsystem and 8 methods and 90 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 2. shows the architecture of the system. It compares „natural“ components and declared packages. Natural components are sets of classes with high cohesion among each other 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. We observe:

- The „natural“ components and declared components are not contradicting. The declared components are finer grained than the natural components.
- The declared component “data” contains only one single class.

Further analyses split the “green” components depicted in the top ellipsoid of Figure 2. from the others depicted in the two lower ellipsoids of Figure 2.

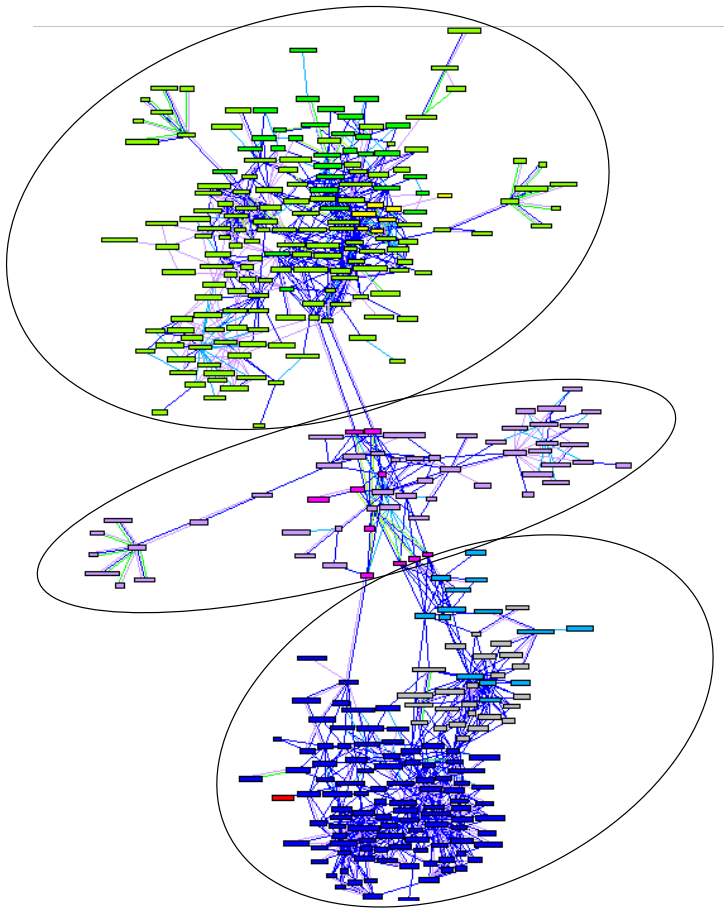

Figure 2. The class interaction graph shows classes and interface nodes and their interaction, i.e. calls and field accesses. It uncovers the architecture of the *VizzAnalyzer*. Ellipsoids characterize „natural“ components, i.e. sets of classes with high cohesion among each other and low coupling to the rest of the system. The colors indicate declared packages of the systems where classes/interfaces of the same top-level package are depicted with the same color.

Figure 3. shows the declared components, i.e. packages, of one of the „natural“ components of the *VizzAnalyzer*. For the declared packages there is no distinctively high cohesion and low coupling observable.

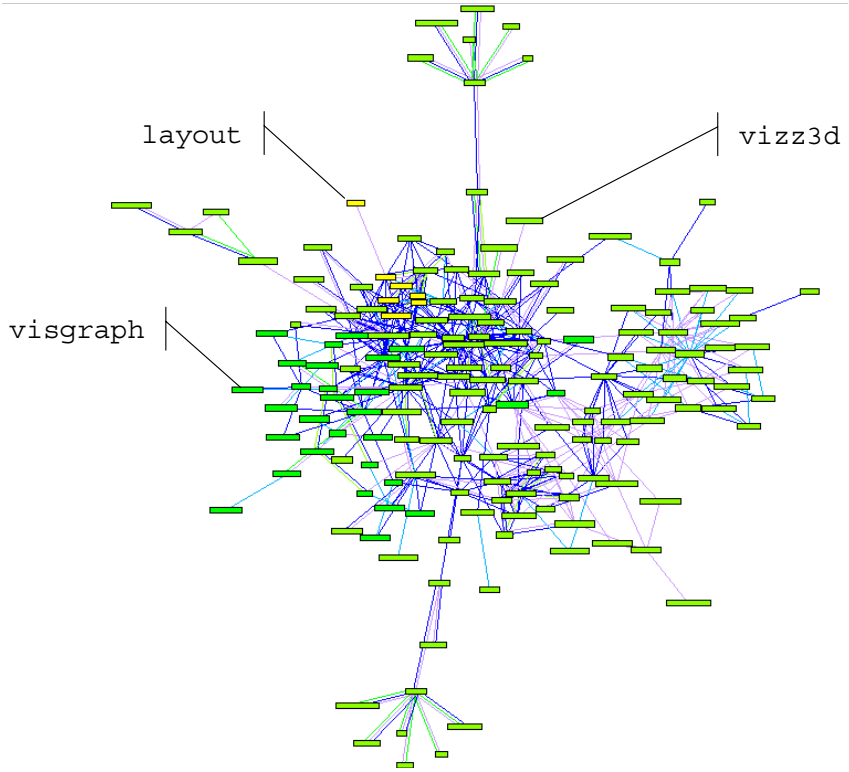

Figure 3. The class interaction graph of the top level packages `layout` (yellow), `visgraph` (dark green), and `vizz3d` (light green).

Figure 4. shows the declared components, i.e. packages, of the other two of the „natural“ components of the *VizzAnalyzer*. For the declared packages `plugIn` there is no distinctively high cohesion and low coupling observable. The packages `hlanalyses` and `analyses` are highly interwoven. Together, however, they show high cohesion and low coupling. So do the packages `core` and `recoder`.

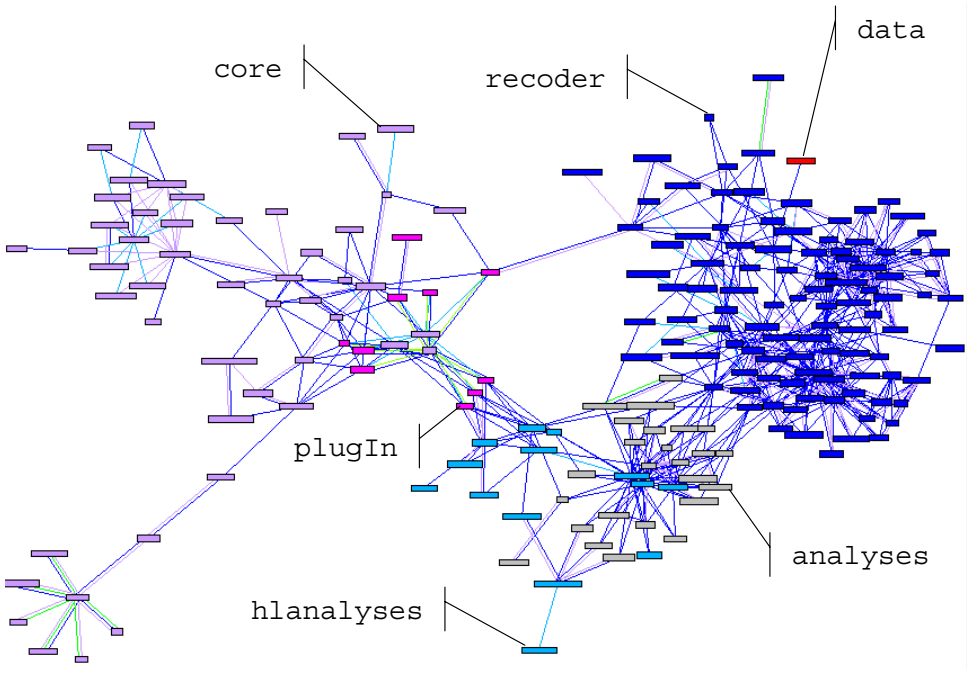

Figure 4. The class interaction graph of the top level packages `core` (lila), `plugIn` (purple), `hlanalyses` (light blue), `analyzer` (grey), and `vizz3d` (dark blue).

Interpretation: The packages `core` and `recoder` contain "natural" components. A merger of the packages `hlanalyses` and `analyses` would also be such a "natural" components, likewise a merger of packages `layout`, `vizz3d` and `visgraph`. One might consider moving these packages in into two `analysis` and `visualization`, respectively. This would lead to a more balanced top-level packaging representing the top-level architecture of the system. This would increase the understandability of the system.

The only class in package `data` should be moved into the `recoder` package.

Package `plugIn` does not define a component. Again there are other structuring principles that are expressed using the package construct and, hence, this is not a design flaw.

4.1.2 Intended vs. actual Architecture

A discussion with the designers of the *VizzAnalyzer* uncovered the actually intended architectural style: it is a centralized architecture. The central component is located in package `core`. All other components should only interact with the core component via classes in the `plugIn` package (which, by the way, showed that this package is not intended to contain a component).

Figure 4. uncovers a direct interaction of classes in the `recoder` and the `analyses` packages.

Interpretation: The direct interaction of classes in the `recoder` and the `analyses` packages breaks the intended architectural style. It is therefore a severe design flaw; it is highly recommended to refactor it.

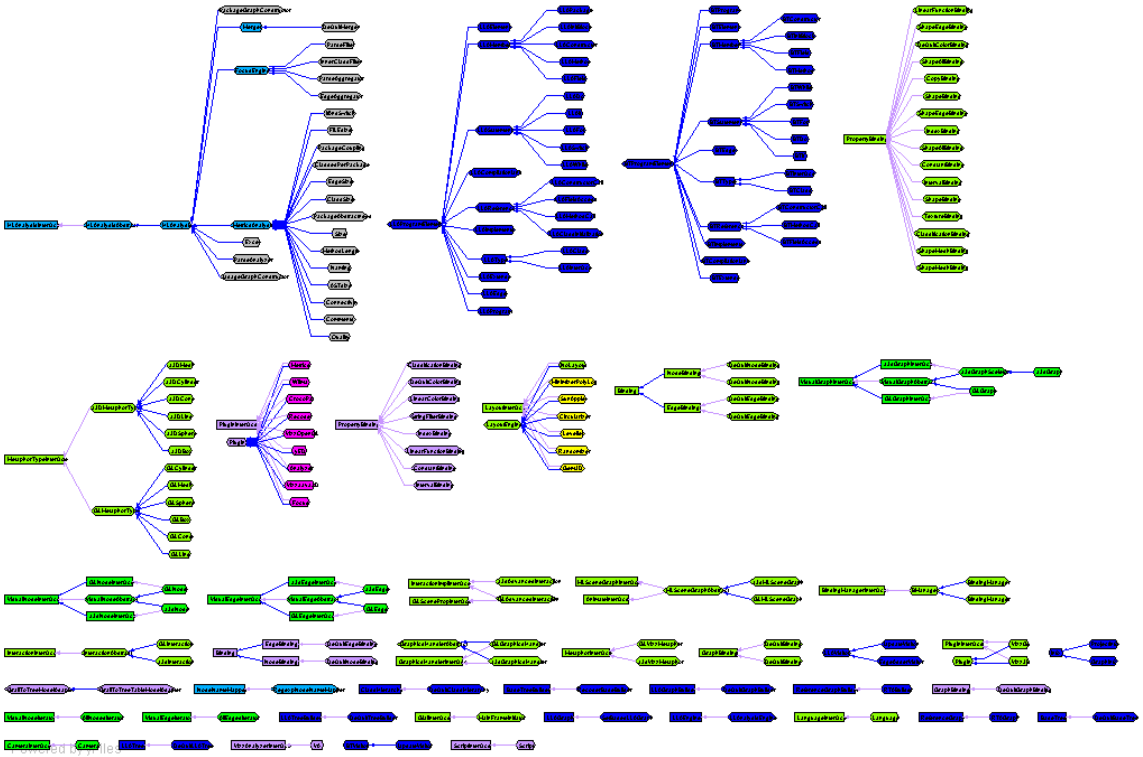

Figure 5. The inheritance structure of the *VizzAalyzer*. The color scheme encodes the top-level packages of classes and interfaces.

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.

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 5. depicts the inheritance structure of the *VizzAalyzer*. 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.

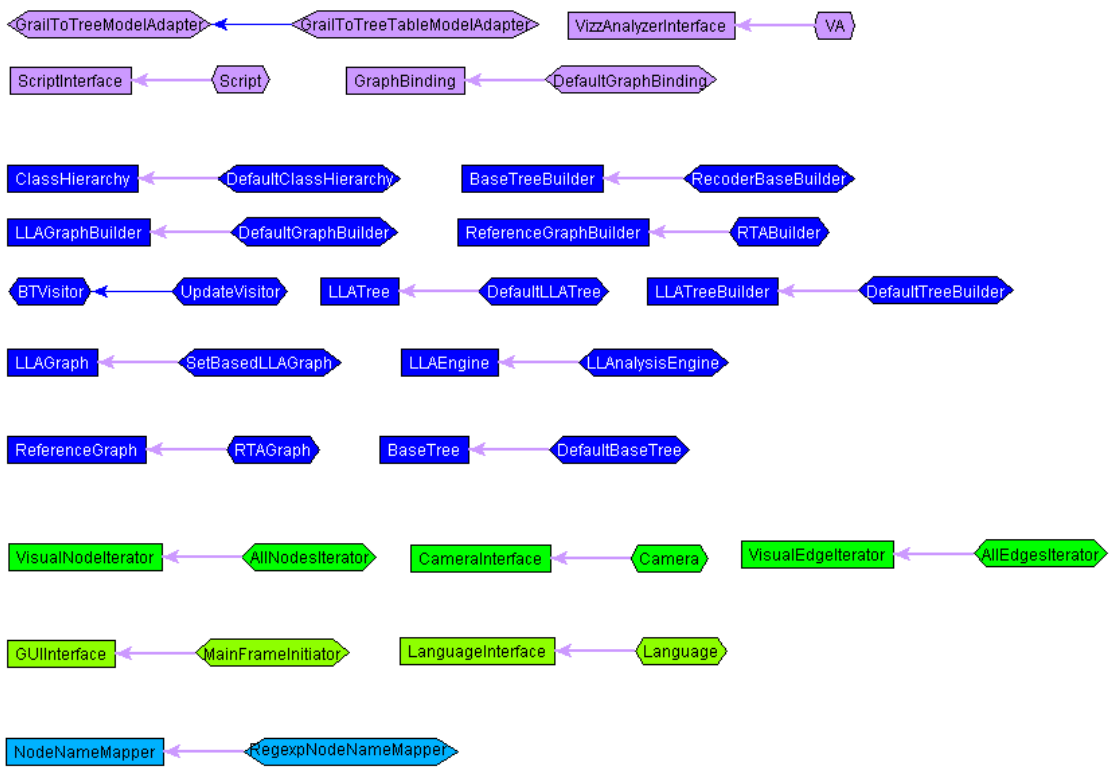

Figure 6. Reordering of the classes and zoom into the inheritance structure of the *VizzAalyzer*.

In the major inheritance structure, there is only on chain from the interface `HLAnalysisInterface` to the abstract class `HLAnalysisAbstract`. On the other hand there are quite a few interfaces with just one implementation. These are depicted in the lower part of Figure 5. Figure 6. reorders this part according to the packages the structures are contained in and zooms into this part of the structure. From top to bottom, colors encode the packages `core`, `recoder`, `visgraph`, `vizz3d`, and `hlanalyses`.

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

There are at most two packages involved in any of the inheritance hierarchies; the maximum inheritance depth is 4, maximum width is 16. Only the latter could be considered an escape from the average.

Interpretation: The inheritance structure does not show any severe violations to standard designing guidelines. There are quite view inheritance chains of length two. 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 16 classes implementing the interface `PropertyBinding`. It is the root of a rather flat hierarchy. One might consider to further structure this hierarchy.

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

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

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. depicts the values of TCC and DAC for classes and interfaces of the *VizzAnalyzer*. 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, where the highest absolute values measured in each dimension are mapped to 100.

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

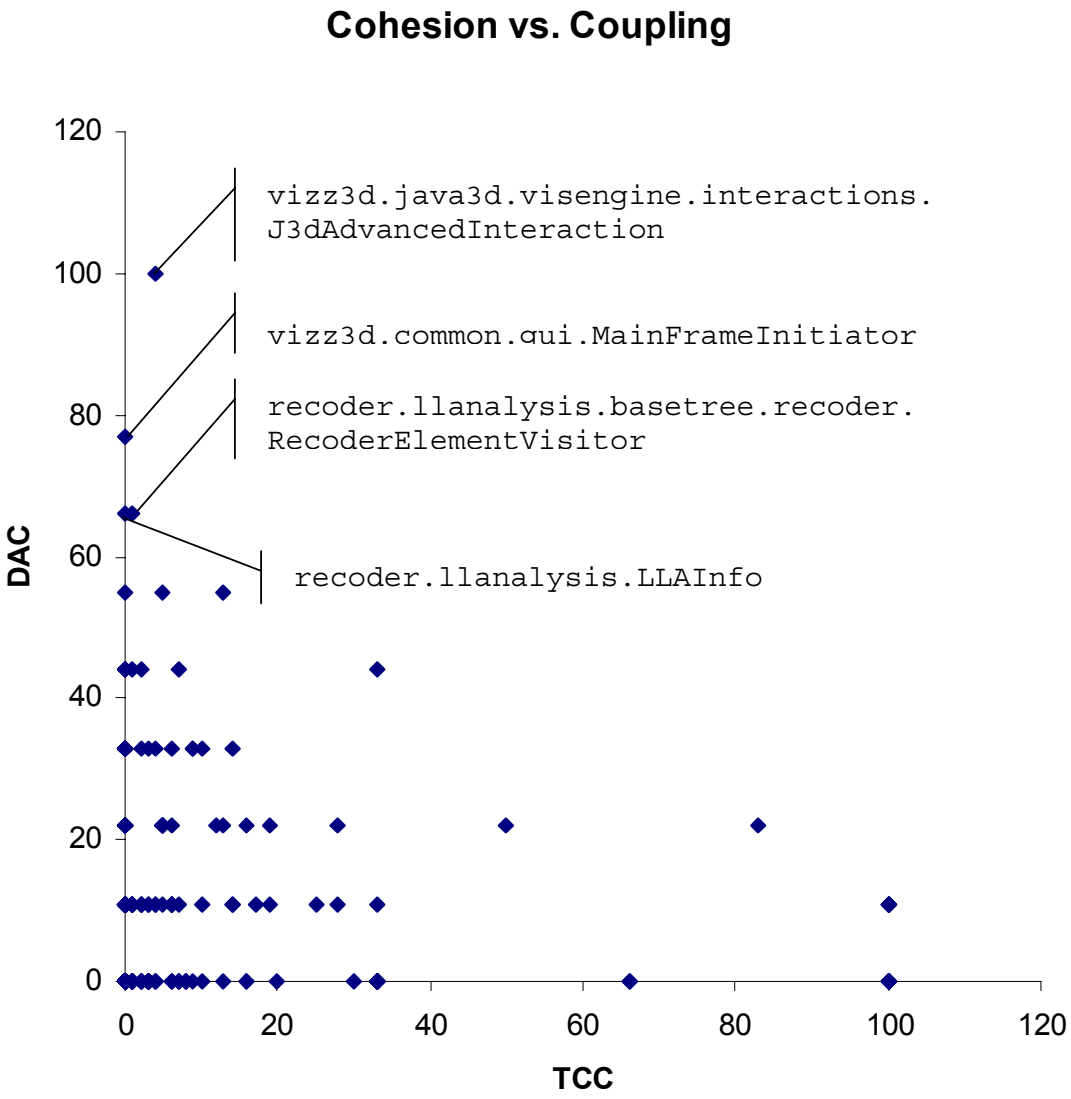

Chart 1. Relative intra-class-cohesion (TCC) and relative inter-class-coupling (DAC).

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.

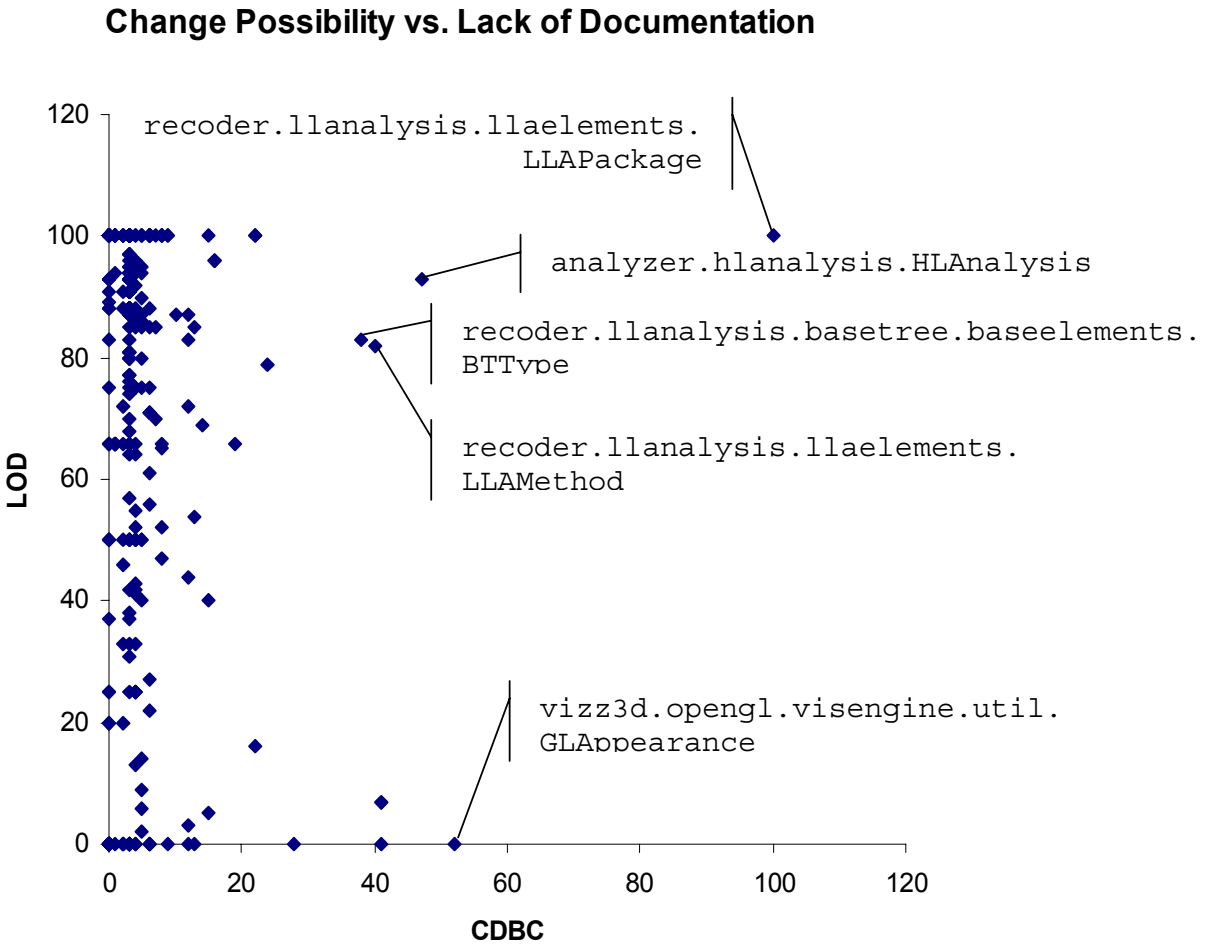

Chart 2. Relative possibility of changes of a class (CDBC) and relative lack of documentation of classes and methods (LOD).

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. 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 following classes show in general a high external coupling and low internal cohesion at the same time:

1. vizz3d.java3d.visengine.interactions.J3dAdvancedInteraction
2. vizz3d.common.gui.MainFrameInitiator
3. recoder.llanalysis.basetree.recoder.RecoderElementVisitor
4. recoder.llanalysis.LLAInfo

The following classes show in general a high change possibility and lack of documentation at the same time:

1. recoder.llanalysis.llaelements.LLAPackage
2. analyzer.hlanalysis.HLAnalysis
3. recoder.llanalysis.basetree.baseelements.BTType
4. recoder.llanalysis.llaelements.LLAMethod

The class vizz3d.opengl.visengine.util.GLAppearance has a high change possibility but appears to be well documented.

These classes in the packages recoder.llanalysis and vizz3d might be considered worth being reengineered.

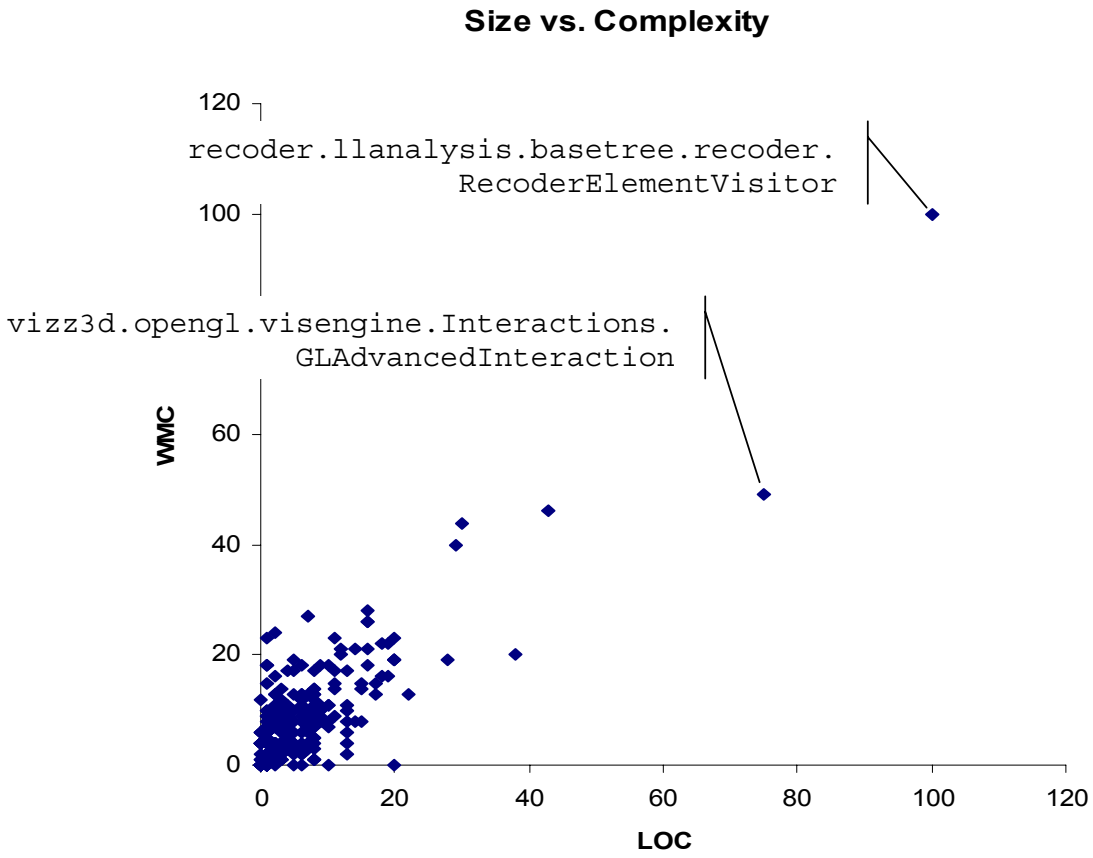

Chart 3. Relative size in lines of code (LOC) and relative complexity in weighted method complexity (WMC).

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. `recoder.llanalysis.basetree.recoder.RecoderElementVisitor`
2. `vizz3d.opengl.visengine.Interactions.GLAdvancedInteraction`

These classes might be considered splitting. Special attention should be paid to the class `recoder.llanalysis.basetree.recoder.RecoderElementVisitor`. This class was recommended to re-engineer because of its high external coupling (DAC) and low internal cohesion (TCC).

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.