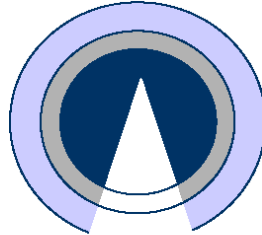


ARiSA First Contact Analysis™

Applied Research In System Analysis™ - ARiSA™



“You cannot control what you cannot measure”

Tom DeMarco

Software	<i>RANDIS</i>
Version	unknown (todo: get version)
Programming Language	Java 1.4

Date	29 th of November, 2006
Contact	Växjö University +46 470 708051 rudiger.lincke@msi.vxu.se
Responsible for Analysis	Rüdiger Lincke
Report version:	1.0

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 *RANDIS*.

1. **This is the final report of the FCA.** The analysis of the system *RANDIS* is completed and documented. The report contains current findings and interpretations. The feedback to this report and an evaluation of our analysis will lead to improvement of future reports.
2. *RANDIS* is a large size system. The averages of 28.9 classes per subsystem and 9.6 methods and 231.1 code lines per class, respectively, are reasonable values and do not uncover deviations from state-of-the-art design rules. Yet the average classes per package seem to be low.
3. It is suggested to split the top-level package `randis` into two packages corresponding to the identified “natural” components. Other top-level packages being connected with each other should be joined under one top-level package (support).
4. The inheritance structure appears to be rather flat and underdeveloped. It should be checked if the degree of reuse through inheritance could be increased, and if the proper data abstractions are used.
5. The following classes are both complex and large and might be subject for refactoring:
 - a. `randis.dynamic.aftn.aftnmetar.MetarMsg`
 - b. `randis.dynamic.aftn.aftntaf.TafMsg`
 - c. `randis.hmi.aftnlist.MasterBean`
 - d. `randis.statics.docimportapplet.DocImportApplet`
6. The following classes are showing a very high coupling to other classes in the system and a very low inner cohesion. Two of them have been already recognized having a very high complexity and size. They might be good candidates for refactoring:
 - a. `randis.dynamic.aftn.aftnawos.AwosMsg`
 - b. `randis.dynamic.aftn.aftnmetar.MetarMsg`
 - c. `randis.dynamic.aftn.aftntaf.TafMsg`
 - d. `randis.hmi.aftnlist.MessKnowledge`
7. Since the packages have been merged out of about 30 separate projects, original project association of the packages has been lost. Knowing the previous structure might uncover valuable information for the interpretation of the results.
8. The following packages show high values in inter package coupling and cycles. They seem to contain key functionality of the system. Building architecture around them and potentially merging packages being included in cycles to components might support the architecture of the system. Packages with high afferent coupling mark servers/libraries providing functionality:
 - a. `randis.common.helpers` (139)
 - b. `randis.common.helpers.version` (80)
 - c. `litexmlparserbean` (57)
 - d. `randis.hmi.serverconnectionbean` (52)

Packages with high efferent coupling mark clients using functionality of servers/libraries:

 - e. `randis.hmi.aftnlist` (38)

Packages being involved in many cycles provide difficulties in build and deployment, since they have a lot of cyclic dependencies. Recognizable is one big cycle involving 12 packages:

 - f. `randis.hmi.aftnlist`
 - g. `randis.hmi.aftnlist.aimbean`
 - h. `randis.hmi.aftnlist.icebean`
 - i. `randis.hmi.aftnlist.igabean`
 - j. `randis.hmi.aftnlist.metarbean`



- k. randis.hmi.aftnlist.notambean
- l. randis.hmi.aftnlist.qnhbean
- m. randis.hmi.aftnlist.sigmetbean
- n. randis.hmi.aftnlist.snowtambean
- o. randis.hmi.aftnlist.tafbean
- p. randis.hmi.aftnlist.upperwindbean
- q. randis.hmi.aftnlist.windshearbean

Apart from this no unusual cycles can be detected.

9. It needs to be discussed, what the actually intended architecture of *RANDIS* is.
10. No special attention was paid to the fact that the application represents a web-application, running on an application server. This should be taken into account when interpreting the results and drawing final conclusions, and to adjust and refine the analysis for future iterations.



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 *RANDIS* system as of November 15, 2006.

Top level packages (subsystem)	27
Packages	176
Classes	665
Interfaces	115
Methods	7.511
Fields	5.769
Lines of code	180.227

Table 1 Global Statistics Measurements

Interpretation: *RANDIS* is a large size system, consisting out of 27 subsystems (top level packages). The averages of 28.9 classes per subsystem and 9.6 methods and 231.1 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.

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, 2, 3, show the architecture of the system. Figure 2, compares „natural“ components and declared packages. Natural components are sets of packages with high cohesion among each other and low coupling to the rest of the system. Declared packages of the systems are packages of the same top-level package (same color). 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 contradicting each other. Looking at figure 1, we observe one huge, one medium and one small component. The huge and the medium component are mainly connected over an “interface package” (yellow, `litexmlparserbean`). All blue packages have the same top-level package “`randis`” and belong therefore to the subsystem “`randis`”.
- The huge component is held together almost exclusively by invokes relations between methods (blue edges). The small amount of accesses relations (green edges) from methods to fields should not be there. It is not good practice to access fields among package boundaries, or even over class boundaries. The medium component involves a higher number of field accesses over package boundaries.
- The small component seems to have only utility functions for the medium component. It consists out of different top-level packages, which are used by one central package.
- Remarkable is further that there seem to be 4-5 centers of gravity pointing out certain packages as highly used by the rest of the packages in each component.



- Further analysis is currently not possible since knowledge about the functions of the different subsystems is necessary to make a more detailed analysis of their relations.
- Figure 2 supports on lower level the impression of the system architecture gained from the package interaction graph. It is clearly visible that there are three main clusters being connected by a few interface classes.

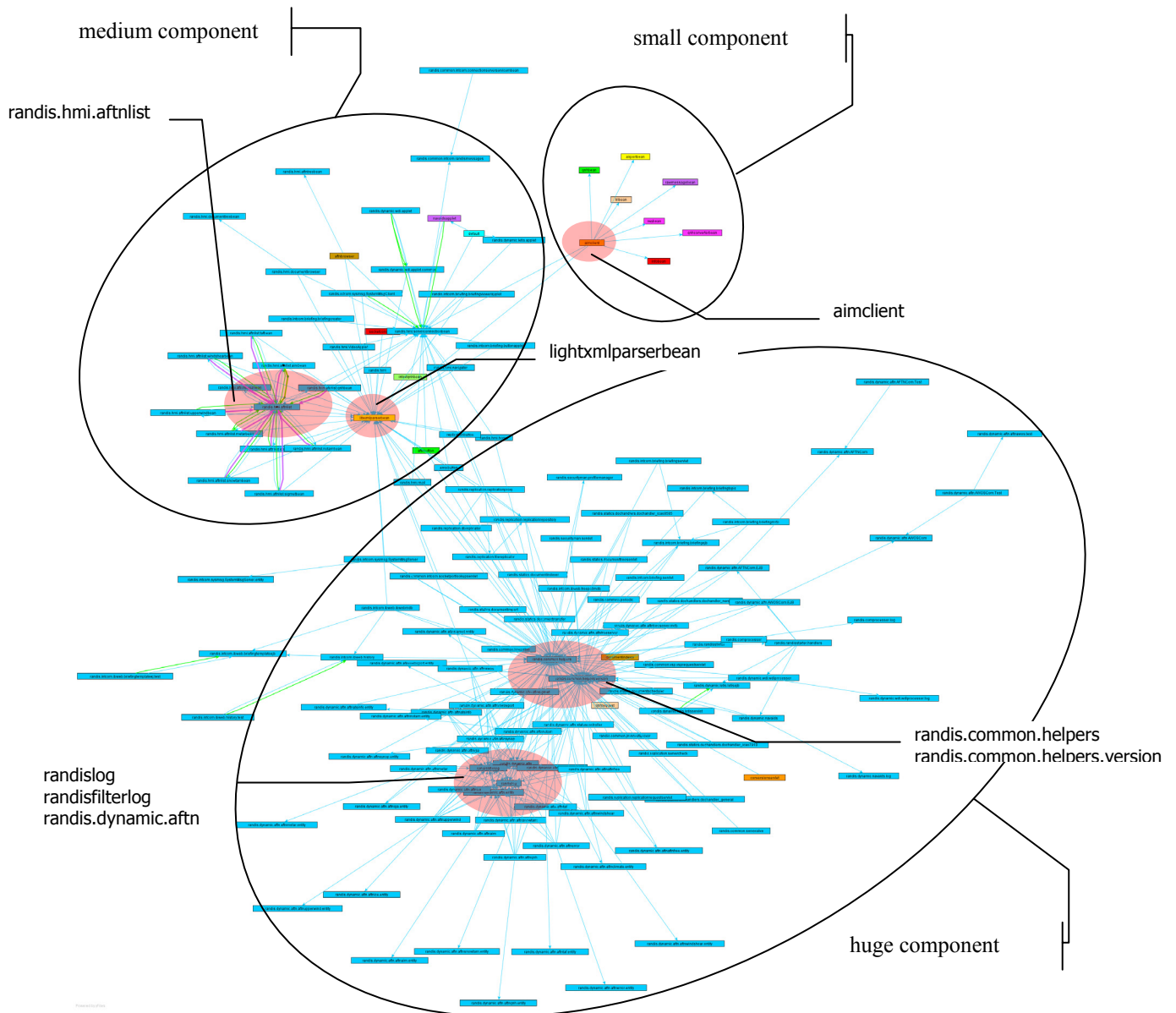


Figure 1. The package interaction graph shows package nodes and their interaction, i.e. calls (blue, purple) and field accesses (green). It uncovers the architecture of *RANDIS*. Ellipsoids characterize „natural“ components, i.e. sets of packages with high cohesion among each other and low coupling to the rest of the system. The colors indicate declared packages of the systems where packages of the same top-level package are depicted with the same color.

Interpretation: The packages of the top-level package `randis` contains two “natural” components. Since they are contained in the same top-level package it needs to be further investigated if this is wanted. Splitting the package `randis` into two top-level packages, each for

one component could 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 package `lightxmlparserbean` seems to be the interface between the two “natural” components, but it is bypassed in 30% of invokes relations between the two big “natural” components. It should be investigated if `lightxmlparserbean` is really an interface package, and if it is a violation to bypass it.

The top-level packages making up the smallest “natural” component should be grouped together under the same top-level package to increase understandability.

Going down on class level, we see that the clusters remain, and that the three main components are communicating with each other over basically one central class, the `LiteXMLParser` class. Other heavily used classes are: `Aftn`, `ServerConnectionBean`, `VersionHelper`, and `Logger`.

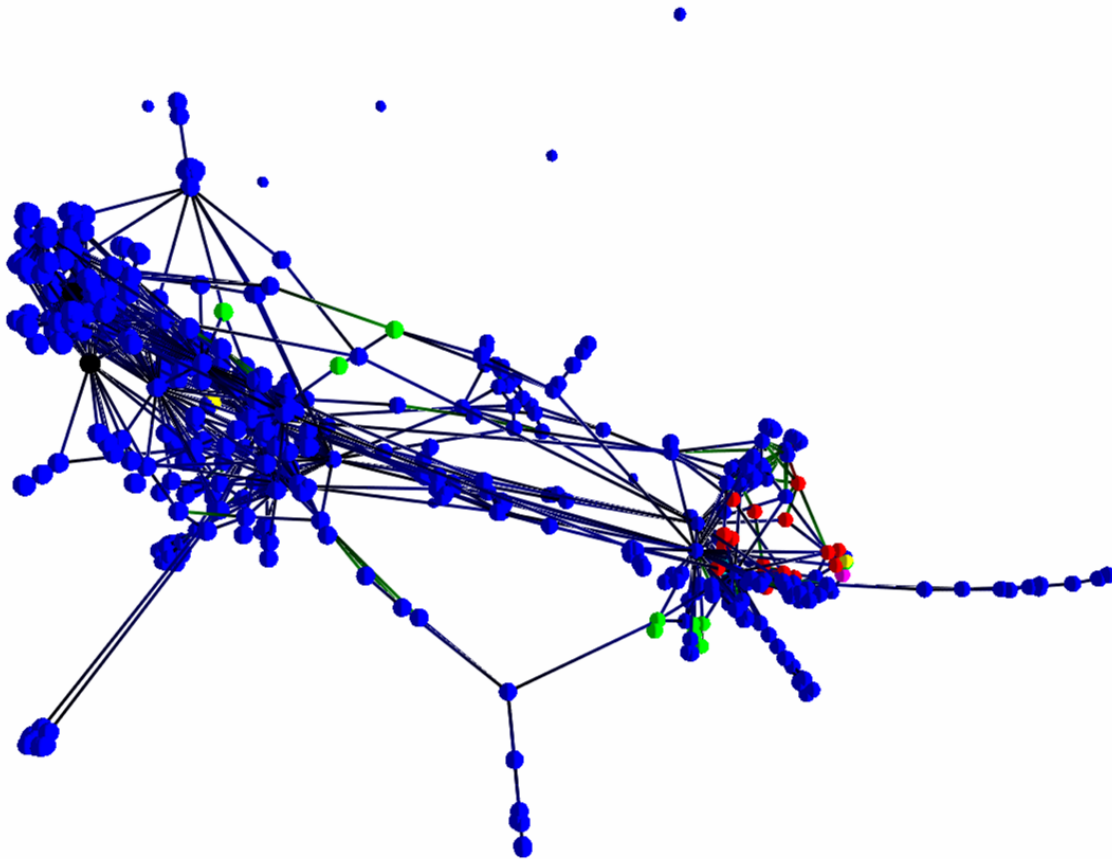


Figure 2. The class interaction graph 3D, shows class nodes and their interaction, i.e. calls (blue, purple) and field accesses (green). It uncovers the architecture of *RANDIS*. Ellipsoids characterize „natural“ components, i.e. sets of classes with high cohesion among each other and low coupling to the rest of the system. The colors indicate declared packages of the systems where classes of the same top-level package are depicted with the same color.

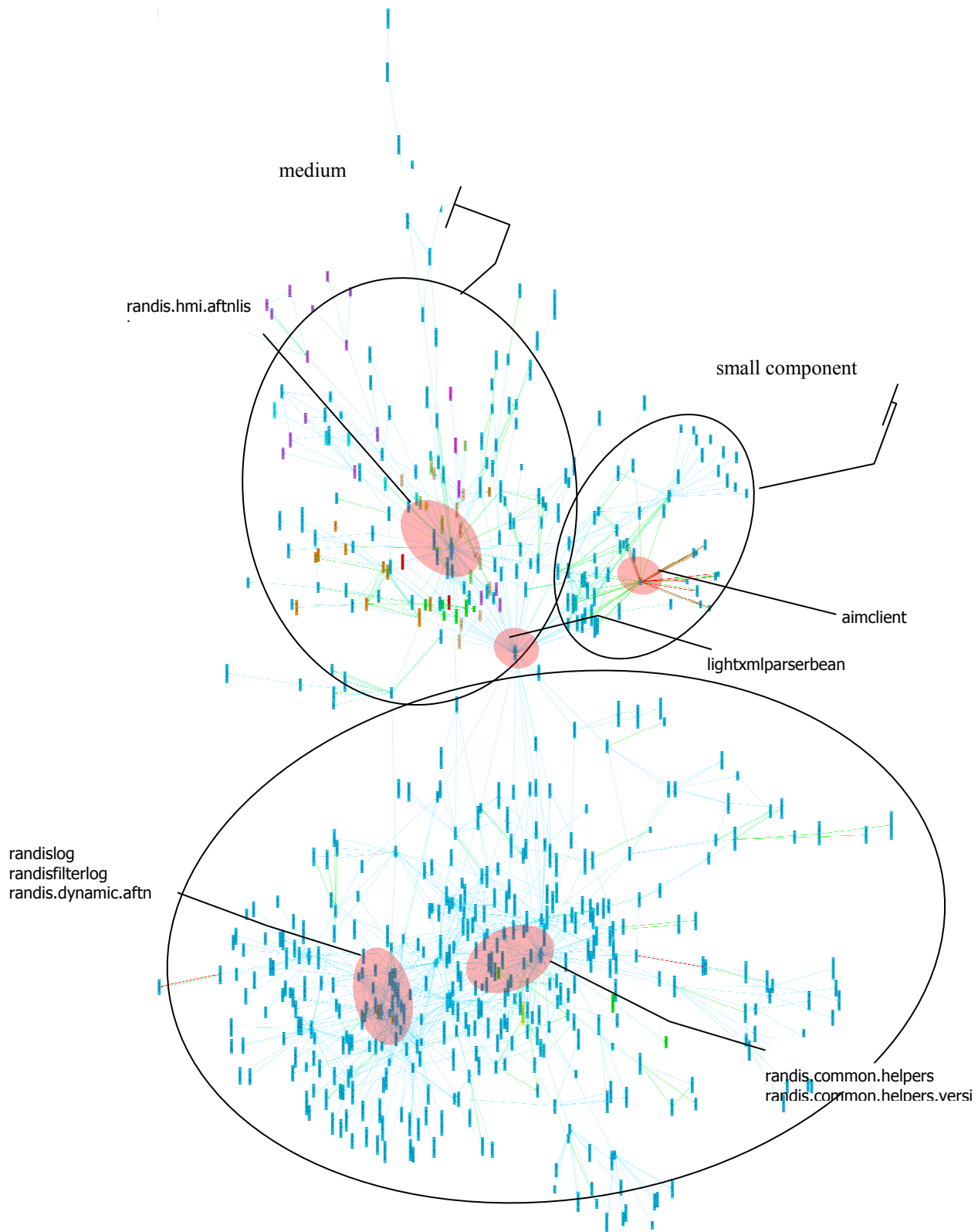


Figure 3. The class interaction graph shows class nodes and their interaction, i.e., calls (blue, purple) and field accesses (green). It uncovers the architecture of *RANDIS*. Ellipsoids characterize „natural“ components, i.e., sets of classes with high cohesion among each other and low coupling to the rest of the system. The colors indicate declared packages of the systems where classes of the same top-level package are depicted with the same color.

4.1.2 Intended vs. actual Architecture

A discussion with the designers of *RANDIS* is required to find out about the actually intended architectural style and to see if the “natural” components match with it.

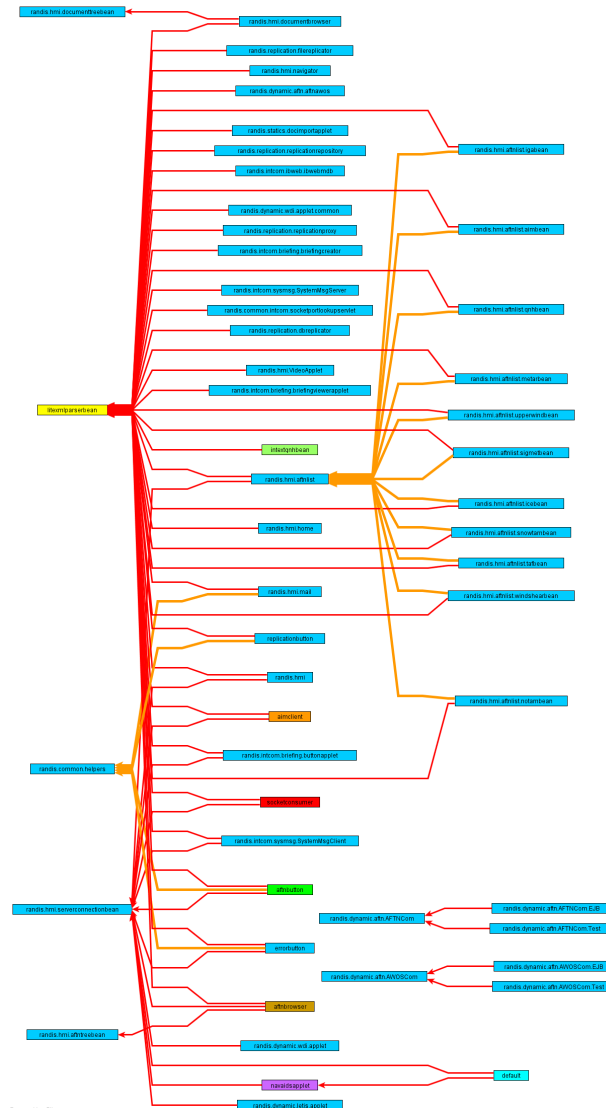


Figure 4. The package inheritance structure of *RANDIS*. The color scheme encodes the top-level packages. Red edges are implements relationships, and orange edges are extends relationships.

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 recommendations. Even though, one should observe the average in a system and have a closer look at escapes.

Figure 4, depicts the inheritance structure of *RANDIS* on package level. Nodes represent packages; edges represent extends/inherits relations. Again, the color scheme encodes the top-level packages of classes and interfaces. It only depicts packages that are in an inheritance relation.

Figure 5, depicts the inheritance structure of *RANDIS* on class level. Nodes represent classes; edges represent extends/inherits relations. Again, the color scheme encodes the top-level packages of classes and interfaces. It only depicts classes that are in an inheritance relation.

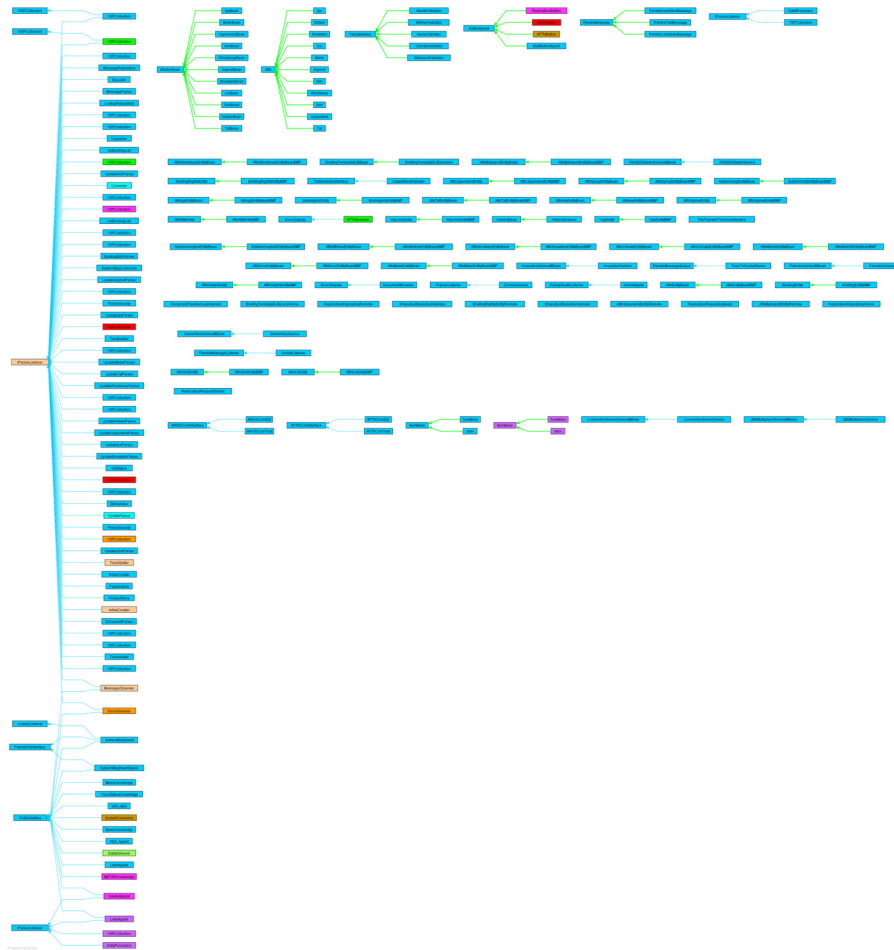


Figure 5. The class inheritance structure of *RANDIS* on class level. Classes are colored according to top-level package. Implements relations are blue, extends are green.

Recognizable in both figures (4, 5) is a very flat and broad inheritance structure, both on package and on class level. Further very many implements relations are recognizable from packages/classes (figure 4/5) that implement the `IParserListener` interface from the `litexmlparserbean` package. These are depicted in the left part of Figure 4 and 5. There are also some extends relations from a few classes in the `randis.hmi.afnlist` package to a few classes in other packages. These are depicted in the upper part of Figure 5. Additionally there

are quite some small inheritance relations having just two classes participating (center part of Figure 5).

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

It should be pointed out, that the inheritance relations are involving many packages; the maximum inheritance depth is 2, maximum width is 56, which appears on the first look rather unusual.

Interpretation: The inheritance structure seems on the first look rather underdeveloped considering the size of the system (66 inheritance relations for 780 classes and interfaces). Where inheritance is used, the structure is rather flat and very broad. 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 56 classes implementing the interface `IParserListener`. 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 coupling 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

$$TCC = \frac{ndp}{np} \text{ with } np = \frac{n(n-1)}{2}$$

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. 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.

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 *RANDIS*. Each class/interface is a point in the diagram; its x -position indicates the class'/interface's DAC value, its y -position indicates the class'/interface's TCC value.

Since, high cohesion and low coupling are desired; classes in the bottom-right corner/part are to review.



Cohesion vs. Coupling

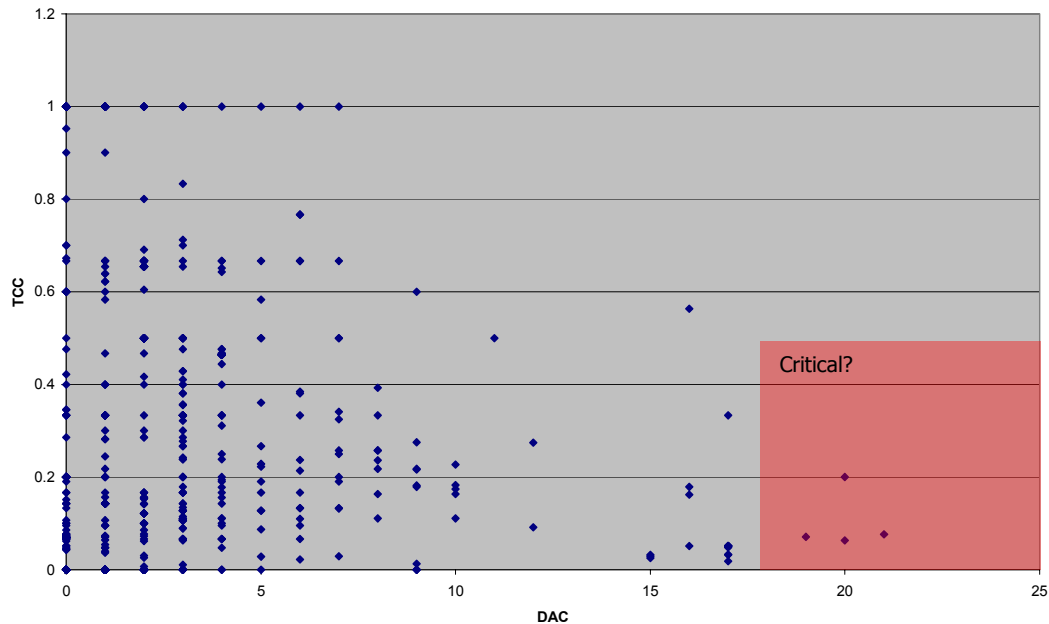


Chart 1. Relative intra-class-cohesion (TCC) and relative inter-class-coupling (DAC).

Coupling Between Classes (CBO) indicates the strength of a coupling and thereby determines the potential amount of follow-up work to be done in a client class when a server class is modified. 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 CBO value is defined between a client and server classes as the number of methods, fields and types, 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.

Interpretation: The following classes have both a very high coupling and a low cohesion:

- a. randis.dynamic.aftn.aftnawos.AwosMsg
- b. randis.dynamic.aftn.aftnmetar.MetarMsg
- c. randis.dynamic.aftn.aftntaf.TafMsg
- d. randis.hmi.aftnlist.MessKnowledge

They should be further instigated and seem good candidates for refactoring. Through their high coupling with other classes, changing these classes effects bigger parts of the system. They are difficult to test and to understand.



Afferent vs. Efferent Coupling vs. Cycles

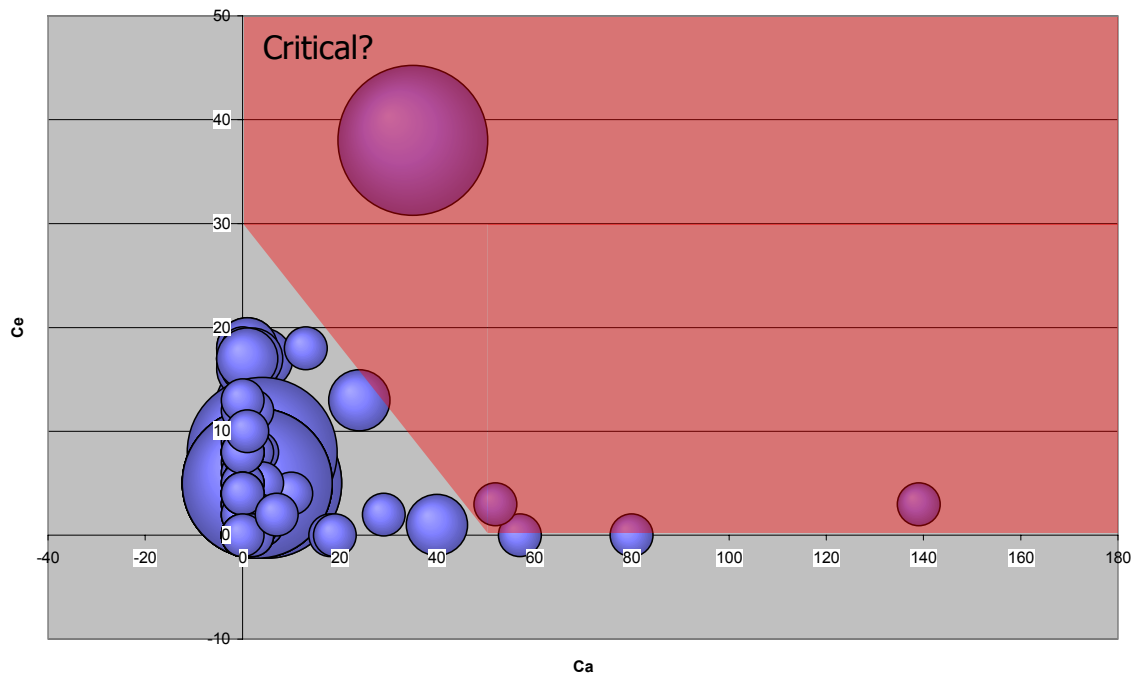


Chart 2. Afferent (Ca) and efferent (Ce) coupling between packages and cycle length between packages (CYC).

Afferent (ingoing) and efferent (outgoing) coupling (Ca, Ce) indicates the strength of a coupling and thereby determines the potential amount of follow-up work to be done in a client class when a server class is modified. Cyclic relations between packages (CYC_Package) affect the build and deployment dependencies as well as maintainability and understandability. 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.

As we can see in Chart 2, there are two packages having over average couplings to the other packages in the system. The following packages show high values in inter package coupling and cycles. They seem to contain key functionality of the system. Building architecture around them and potentially merging packages being included in cycles to components might support the architecture of the system. Packages with high afferent coupling mark servers/libraries providing functionality:

- e. randis.common.helpers (139)
- f. randis.common.helpers.version (80)
- g. litexmlparserbean (57)
- h. randis.hmi.serverconnectionbean (52)

Packages with high efferent coupling mark clients using functionality of servers/libraries:

- i. randis.hmi.afnlist (38)

Packages being involved in many cycles provide difficulties in build and deployment, since they have a lot of cyclic dependencies. Recognizable is one big cycle involving 12 packages:

- j. randis.hmi.afnlist
- k. randis.hmi.afnlist.aimbean
- l. randis.hmi.afnlist.icebean
- m. randis.hmi.afnlist.igabean



- n. randis.hmi.aftnlist.metarbean
- o. randis.hmi.aftnlist.notambean
- p. randis.hmi.aftnlist.qnhbean
- q. randis.hmi.aftnlist.sigmetbean
- r. randis.hmi.aftnlist.snowtambean
- s. randis.hmi.aftnlist.tafbean
- t. randis.hmi.aftnlist.upperwindbean
- u. randis.hmi.aftnlist.windshearbean

Apart from this no unusual cycles can be detected.

Interpretation: The packages listed above are recognized because of their high afferent and efferent coupling. This is not necessarily a problem, yet it should be investigated if this reflects the intended functional distribution in the system. More serious are the cyclic relations. They can have serious effects on the maintainability of a system. Recognizing that all packages being in a cycle of length 12 are in the same subpackage: *randis.hmi.aftnlist* this is probably not an issue and having effects on maintainability of the overall system.

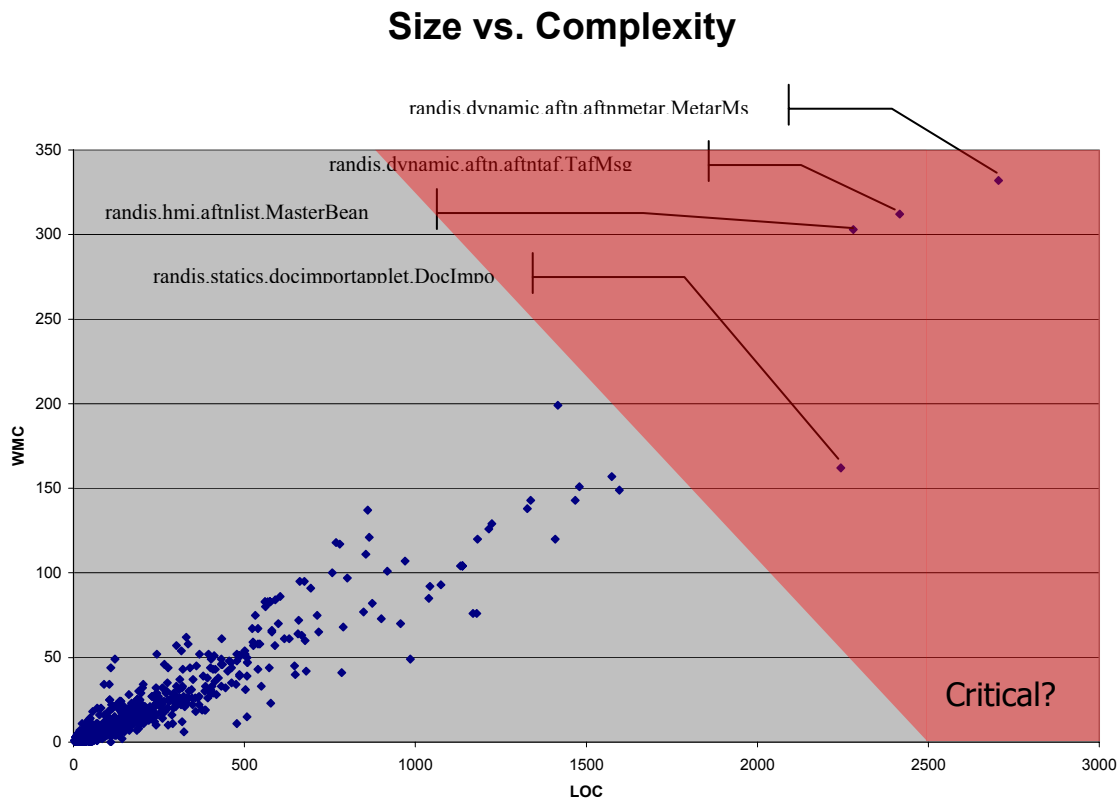


Chart 3. Absolute size in lines of code (LOC) and absolute complexity in weighted method complexity (WMC).



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: `do`, `for`, `if`, `switch`, `while`. It is assumed that each branch has the same complexity/weight.

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

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 *RANDIS*. 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.

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:

- a. *randis.dynamic.aftn.aftnmetar.MetarMsg*
- b. *randis.dynamic.aftn.aftntaf.TafMsg*
- c. *randis.hmi.aftnlist.MasterBean*
- d. *randis.statics.docimportapplet.DocImportApplet*

The big size and complexity of these classes makes them hard to understand and test. They seem to be good candidates for refactoring, in particular since two of the classes (*italic*) have been recognized while assessing design metrics.



7 Conclusions

This report documented the results of the ARiSA First Contact Assessment of *RANDIS*. The report is the foundation for a dialog in which the metrics and the analysis process shall be tailored to the project specific needs in order to assess the quality of *RANDIS* in a way which is useful for its designers and maintainers.

Our assessment as part of the First Contact Analysis, included an architecture analysis in general, yet a comparison between intended and actual architecture is still on the *todo* list. The designers/maintainers of *RANDIS* need to be contacted. An analysis of the inheritance structure is completed, but some detailed analysis of the subsystems is required after the current results have been discussed with the *RANDIS* maintenance team. Analyses of coupling and cohesion on class level, and analyses of the understandability and maintainability of the software is completed and some classes as starting point for discussions have been identified.

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, but still the actual and the intended design need to be compared. It is now the task of the *RANDIS* maintenance team to evaluate the results documented by this report using their expert knowledge. Together we can draw conclusions if the current design and implementation of the software provides a solid foundation for future development cycles and how the used analysis and metrics should be adjusted to measure the quality factors being important for the maintenance best.

