h e g
Haute école de gestion de Genève
Geneva School of Business Administration

Evaluating the efficiency of using the Autonomy Ratio Metric for assessing ArgoUML architecture

Bachelor thesis

Author : Mihnea Niculescu
November 2012

Advisor : Professor Philippe Dugerdil, Ph.D

Acknowledgements

My gratitude goes

to Professor Philippe Dugerdil for guiding and supporting me and my work, for his exciting classes and for his precious passion for computer science,

to assistant David Sennhauser for kindly introducing me to the tools he developed in school for this project,

to Professor Peter Daehne for helping me find an interesting bug and for his captivating java programming classes,

to my parents for supporting me during my studies.

Summary

Metrics in software engineering are used to evaluate quantitatively and qualitatively various attributes of (usually large) systems. These figures help synthetizing information such as size, quality or complexity of various element of the analyzed software.

In the past few years, Professor Philippe Dugerdil has developed, at the Geneva School of Business Administration, a new metric, called the Autonomy Ratio, along with an analysis method and related software tools. The AR metric helps measuring the “functional structuring” of a system and indicates how easy is to understand the analyzed system. System understanding is very important for maintenance, which is usually the most expensive task in software engineering.

Up to this day, the analysis method has been tested by Mr. Dugerdil and some other students at Geneva School of Business Administration on one large industrial system. It has revealed weakness in the architecture of this application that has been confirmed by the development team of the application. However, in order to validate the metric, more systems have to be analyzed. In the work presented here, I used the Autonomy Ration metric and the tools developed in the school to asses ArgoUML, a well-designed, mid-size open source application. I evaluated the efficacy of the method, enhanced the existing tools and proposed some improvements.

Table of contents

Introduction to the Autonomy Ratio metric

A software that is easy to understand (and therefore easy to modify and to maintain) should exhibit the tree hierarchical quasi-decomposability characteristic of Simon H.A.: in order to be understandable a complex system needs to be nearly decomposable: to study the inner working of each subsystem one should be able to neglect the influence of other subsystems at the same level. This is what motivates the use of the AR metric: to evaluate the understandability of a system by calculating the degree of autonomy of the components.

To achieve this goal, calculating static coupling and cohesion between classes (these metrics already exists and are commonly used) is not enough, for three reasons:

1) There is the need to define coupling and cohesion at a higher granularity level than classes: the packages level, because this is the level at which business functions are implemented and where functional component are defined. Indeed, understanding of the system begins at this macro level.

2) Static code structures (such as classes or packages) does not necessarily map to functional components. For example, a package can implement functionality that belongs to different components (generally due to a bad design). In this case we must define coupling and cohesion not between this package and other packages, but between the parts of this package which implement the distinct functionality components. To achieve this, the computing of the Autonomy Ratio is done at runtime, evaluating different scenarios (use cases) that are the most commonly used.

3) System understanding is hierarchical between components at the same level. Hence the coupling and cohesion need to be defined hierarchically (see next paragraph to see more details).

Here is how the Autonomy Ratio is computed, for one package and for a scenario:

> *hf_coupling* (hierarchical functional coupling): the number of distinct messages sent by all the instances of the classes in the package and in all of its sub packages, to any instance of any class in any external package. This represents the amount of functional dependency of the package to other packages.
>
> *hf_coheision* (hierarchical functional cohesion): the number of distinct messages sent *among the direct children* of the substructure (which are classes or sub packages). The messages sent *inside the direct children* themselves are not counted. This represents the amount of hierarchical functional cohesion at the level of the package analyzed.
>
> AR = *hf_cohesion* / (*hf_cohesion* + *hf_coupling*)*100 : represents the percentage of hierarchical functional cohesion among the total hierarchical functional calls (coupling and cohesion). This percentage represents the degree of hierarchical functional autonomy of the package.

In those definitions, distinct call means that a particular call from the same method of the same class in the same package is counted only once. The reason for this is obvious: as we count the calls dynamically, at run time, we don't want to count the quantity of the calls, but the quantity of distinct functionality contained in these calls.

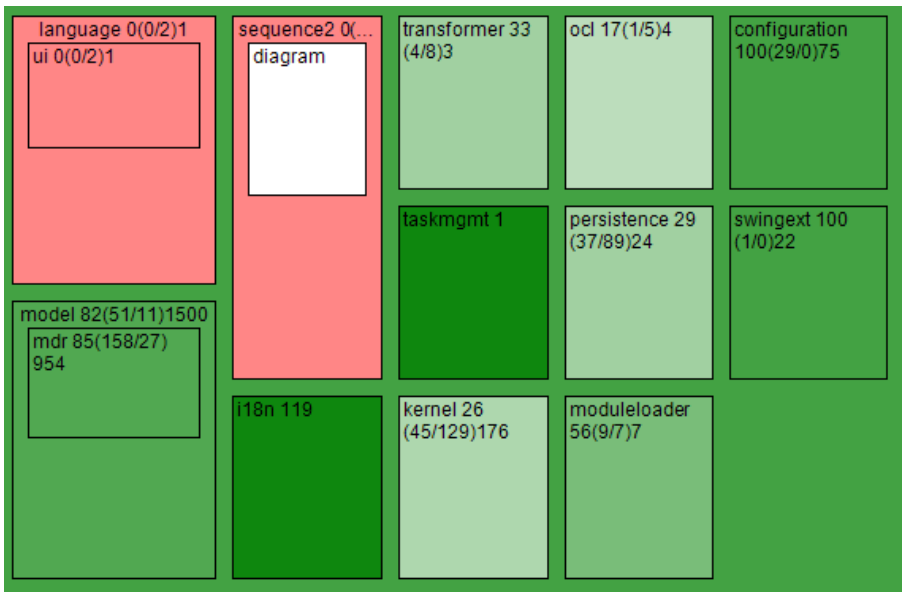
Initially, the method of computing the AR, as proposed by Professor Philippe Dugerdil, is using the dynamic packages and the dynamic classes, in contrast to the static ones (the dynamic package and class for a method are the ones that belongs to the instance of the class, and the static package and class is the one where the method is actually defined; they are not the same in case one class inherit the method form its parent class, without redefining it). In my work I used the static package and classes, and I proposed this to be changed in the initial proposal as well. The reason for this change is the following: the purpose of the AR is to evaluate the functional autonomy from a viewpoint of understandability, hence we must define the dependence on where the functionality is defined, because this is where the source code that we must understand is, i.e. in the static package and class.

For more information on the Autonomy Ratio, see the article “Assessing Legacy Software Architecture with the Autonomy Ratio Metric” by Professor Philippe Dugerdil.

Autonomy Ratio Maps

As suggested by Professor Philippe Dugerdil, a graphical representation of the distribution of the AR among packages should be created in order to help visualize the entire map of a system and easily detect weak autonomy packages.

To achieve this goal I enhanced the existing tools by adding functionality that automatically generates visual maps, called Autonomy Ration maps. These maps are based on the treemapping technique, which enables one to generate graphic maps that show a hierarchical structure by displaying components embedded in one another.



Some of the top packages of ArgoUML in an AR Map

Here is the detailed description of these maps:

- Each box represents a package, a box embedded in another box which represents a package and its parent package. The packages that figure in these maps are the ones that are used in the scenarios, and not all the package from all the source code of the application.
- The title of the box contains the name of the package, followed by the package metrics. The metrics shown and the color of the box depends on the different situations:
    - AR >= 0: all metrics are displayed: AR (Cohesion,Coupling)Export. The color is between light green (for AR=1) and medium green (for AR=100)
    - AR not defined (where cohesion = coupling = 0): only the Export metric is displayed. Its color is dark green.
    - The package is not used in this scenario but it is used in other scenarios: no metrics are displayed. Its color is white.
- The title of top level package contains the name of the scenario for which the AR map has been generated. Each AR map is corresponding to a scenario.

The color represents the degree of autonomy: light green represent weak autonomy, strong green represent strong autonomy. To differentiate the packages when AR = 100 and cohesion <> 0 and the package where cohesion=coupling=0, the AR=100 is displayed in a lighter (medium) green than the other. Both are perfectly autonomous since their coupling is 0. We can interpret the stronger green color for the case in which cohesion=coupling=0, as an indicator that the functionalities contained in each of the direct children are autonomous one to the other.

The cohesion and the coupling metric are displayed for the following reasons:

- When AR=0 we are sure that cohesion is 0, but coupling can be very low (not 0) or very high. It is important to see the coupling because in case this coupling is very low, there is a possibility that the package is quite autonomous (see Shortcoming sections).
- The values of cohesion and coupling give a clue about the amount of functionality used in the package. In addition to the export metric (see next paragraph), it helps evaluate at which extent the package maps onto a functional component (however this in note very accurate, see the Shortcomings section for a proposal of a more precise method to evaluate this).

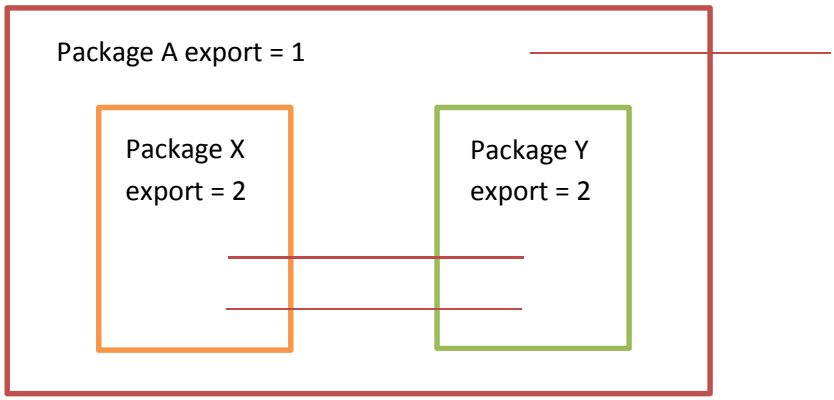
As suggested by Professor Philippe Dugerdil, an additional metric called Export has been added. This metric is computed similarly to the hf_coupling (it is actually the opposite of it):

> *hf_export* (hierarchical functional export): the number of distinct messages sent by any instance of any class in any external package to all the instances of the classes in the package and in all of its sub packages. This represents at which extent the package is used by other external packages.

These are the motivations for adding this new metric:

- It helps identify the packages which are the most heavily used throughout the entire system. This is important because these are the packages that are mostly susceptible to be maintained, so they should have the highest possible autonomy.
- It help evaluate at which extent the package maps onto a functional component, because it indicates if a package is heavily used (however this in note very accurate, see the Shortcomings section for a proposal of a more precise method to evaluate this)

As professor Philippe Dugerdil suggested, I have made an attempt to use the size of packages to show the export metric. These maps are called “proportional AR Maps”, in contrast to the ones described before, for which the size of the boxes is the same for all packages, which are called “absolute” AR maps. There is a problem representing the size of the boxes proportional to the export metric of the corresponding packages: the sizes of the sub-packages can be bigger that the size of their parent package. Since the sub packages are represented by boxes embedded in the box of their father package, this is impossible to represent. Here is an example that illustrates this (the lines represent method calls):



Problem when the size of boxes represents the export metric:
size of box A = 1 must be smaller than size of box X=2 (or Y=2)

I have attempted to use a modified version of the export metric, for which only calls to classes inside the package are considered, without counting the calls to the sub packages. This modified export metric is intended only for displaying the maps, and it is not meant to replace the original export which is displayed in the box titles and serves the purposes mentioned earlier in this chapter. The result of experimenting this with ArgoUML is not reliable enough, because sometimes the boxes are 3 times smaller than they should be, and so the proportions between some boxes and other can be wrong by a factor of tree. However the work has been presented here and the related functionality in the tools has been preserved in order to enable future research on it (the code for generating these maps is in the Tree Map Generator). Currently for each scenario both maps are generated, see “Using the tools to analyze ArgoUML” section for more details)

Installation of the required tools

Here are the instructions on how to install all required tools in order to be able to analyze any software written in java for which you have the sources. The tools are supplied on the CD annexed to this document. For a description of each tool and how to use it, see the "Using the tools to analyze ArgoUML" section.

Eclipse

Eclipse is required. The tools have been tested with the Eclipse SDK Version: 3.7.2 Build id: M20120208-0800. The software you want to analyze must be installed together with its sources and configured as a working java eclipse project. As the sources get modified (instrumented) by the instrumentor, make a backup of the sources (the sources can be easily un-instrumented and reverted to their original form, but for safety, backup is recommended).

Instrumentor

Copy the instrumentor eclipse plugin named *ch.hesge.csim2.instrumentor_1.0.0.jar* from the Instrumentor directory to your eclipse installation plugin directory (typically C:\Program Files\eclipse\plugins). Copy the *instrumentorRuntime.jar* from the same location, to any location you want on your system. A good location to choose is the project folder of the software you want to analyze, since this library will be used by the project.

In eclipse, go to your project settings windows, Library pane, Add External Jars, and choose instrumentorRuntime.jar.

RDDA

Import the 2 projects (DDRA and DDRA-DB) into eclipse from the DDRA\DDRAWORKSPACE directory on the CD. Some parts of the code have to be modified in DDRA-DB\dbconnexion\DBAcessor.java to parameterize the connection to the Oracle DB installed on your system. There is an example DDRA\DBAcessor.java which I used with Oracle 10g Free Express edition.

ORACLE

The Oracle database is required. You can use the free version Oracle database Express edition (the tools have been tested with the 10g version)

PL/SQL Developer

You need a tool for using the SQL scripts on the oracle database (the tutorials on how to use the SQL scripts in this document are based on it). The versions tested with the tools is 8.04.1514

This tool is required, because the format of the results.csv file (see "Using the tools to analyze ArgoUML" section) generated with it has a special format that is used by the Tree Map Generator tool. One may use other tools other than PL/SQL to run the SQL scripts, but for generating the AR map, this may not work with another tool, since Tree Map Generator uses the format in the results.csv with is specific to PL/SLQ. If for some reason there is a need to use another tool than

PL/SQL, a solution is to modify the code of the Tree Map Generator application (its java code is well documented) to adapt it to the format of the CSV file (if required).

SQL SCRIPTS

Connect PL/SQL to Oracle.

Run the *Trace_package_list_GRANT_NEEDED.sql* script from the SQL_Scripts\1_Grant_Privileges folder.

Run the *Create_Table_Trace_autonomy_ratio.sql* script from the SQL_Scripts\2_Create_Table folder.

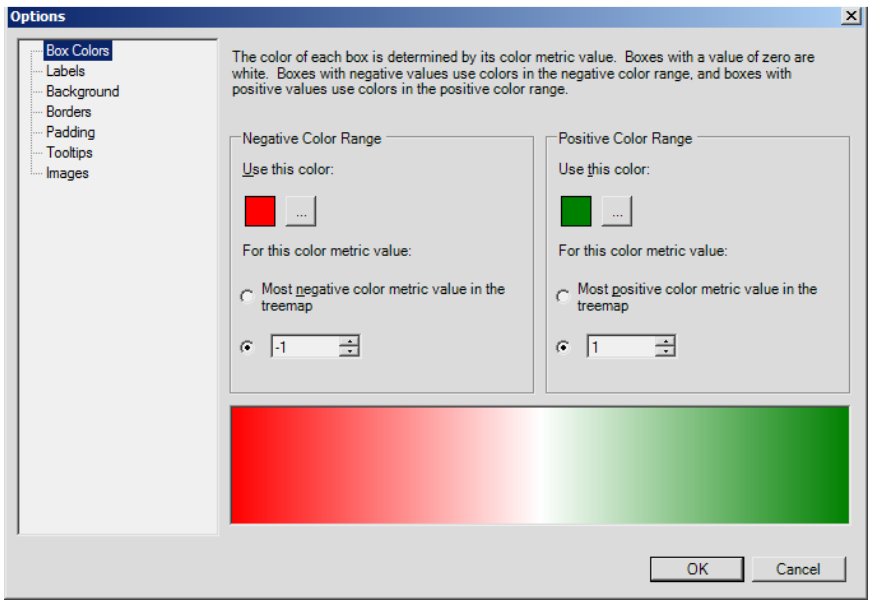
Open and run all scripts found in the SQL_Scripts\2_Oracel_Procedure&Functions folder. This will create stored functions and procedures in the database of your system.

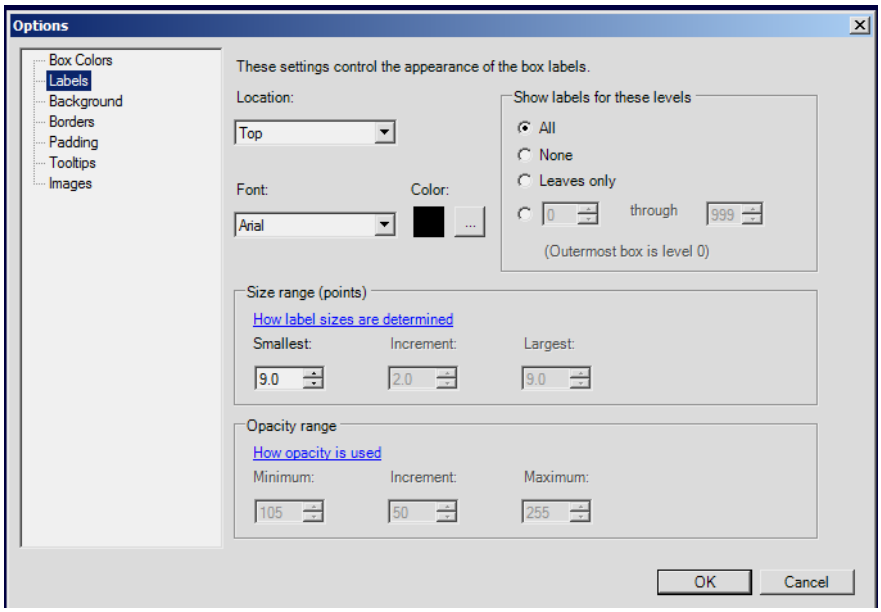
Tree Map Generator

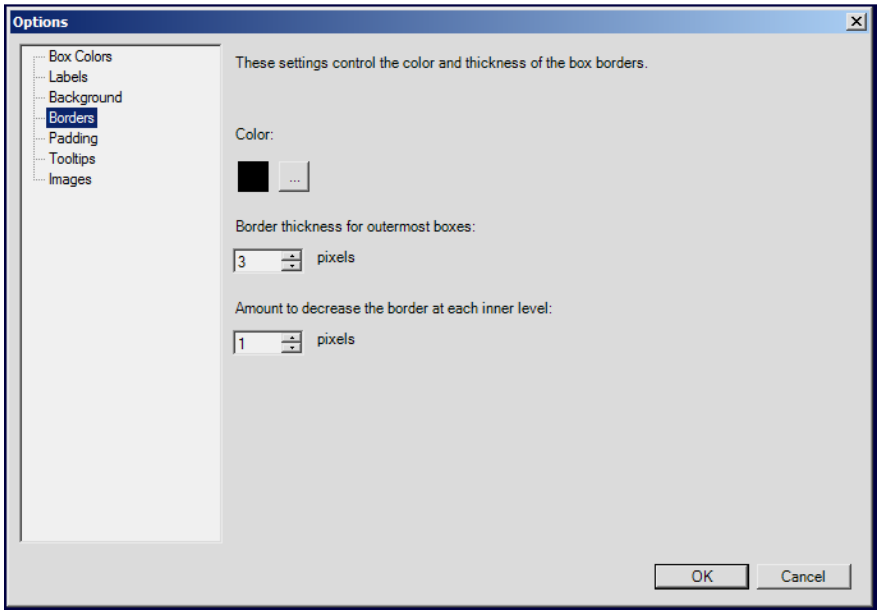
This program will generate the AR maps from the CSV file generated with PL/SQL (see "Using the tools to analyze ArgoUML" section). The program is supplied as an eclipse project with its sources, but you don't need to modify or run it from eclipse. You can simply put the directory somewhere on your system. Create a directory where you want to generate the AR maps. Copy the TreeMapGenerator\generate.bat file in that directory, open this file and modify the path to the .jar so that it matches the directory where you have the Tree Map Generator installed.

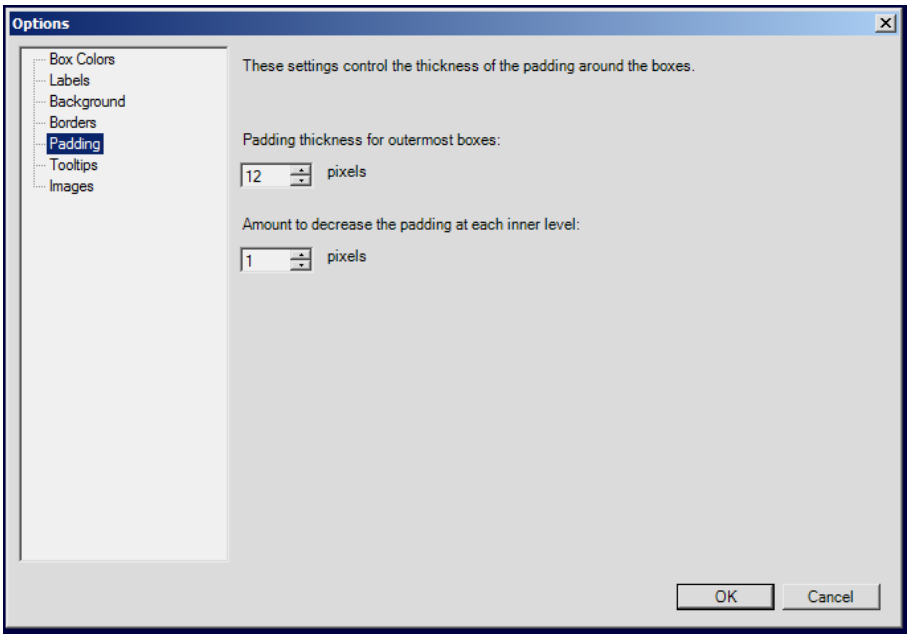
Microsoft Treemapper

This is a free program that is used to visualize the AR maps. I used the 1.0.134 version. In order to render the maps exactly as described in this document, go to View-Options, and match the settings to match the images below:




Options
Box Colors
Labels
Background
Borders
Padding
Tooltips
Images
These settings control the appearance of the box labels.
Location:
Top
Font:
Arial
Color:
Show labels for these levels
All
None
Leaves only
through
(Outermost box is level 0)
Size range (points)
How label sizes are determined
Smallest:
Increment:
Largest:
9.0
2.0
9.0
Opacity range
How opacity is used
Minimum:
Increment:
Maximum:
105
50
255
OK
Cancel


Options
Box Colors
Labels
Background
Borders
Padding
Tooltips
Images
These settings control the color and thickness of the box borders.
Color:
Border thickness for outermost boxes:
3
pixels
Amount to decrease the border at each inner level:
1
pixels
OK
Cancel


Options
Box Colors
Labels
Background
Borders
Padding
Tooltips
Images
These settings control the thickness of the padding around the boxes.
Padding thickness for outermost boxes:
12
pixels
Amount to decrease the padding at each inner level:
1
pixels
OK
Cancel

Using the tools to analyze ArgoUML

Goals to achieve

Here are the different goals to achieve by analyzing ArgoUML with the AR metric:

- Chose a set of scenarios that covers most of the functionality in ArgoUML and imagine real-life scenarios.
- Generate visual maps (called AR maps) with the metrics.
- Verify that that all packages maps onto functional components that emerge when executing the different scenarios. It is known form the start that ArgoUML has a good architecture, so we expect that this would be the case. This evaluation is done through :
    - checking that the packages that contains functionality which is used in a scenario are indeed moderately to heavily used in that scenario
    - checking the source code for each package to see how much of the functionality of the package is used (this work can be done automatically by using a new metric that is proposed as a future work, see the Shortcomings section)
- Verify that the AR metric for each package corresponds to the actual autonomy of the packages as it results from the architecture of the code. This is done analyzing the relationship between the metrics and the architecture of the code.
- Study empirically the understandability of the source code of packages to verify if it matches the values of the AR metric. The AR, which is a percentage, is expected to represent the level of the ease of understandability (as explained in the Introduction section): the higher the AR, the easier the understandability. To realize this evaluation, the following simple method has been used: for each package, we give the ratio between the number of total classes (and interfaces) in the package (and not in the sub packages) (TNOC) and the number of classes (and interfaces) that were examined (which consists in reading the import statements, methods and attributes declarations, and very few code lines) (ENOC): understandability = 100– ((ENOC/TNOC)*100). The rationale for this is that the less classed have to be examined, the easier the package is to be understood. This ratio is expected then expected to be similar to the AR. Note that the understandability evaluated here is at a high level, it represents the comprehension of the structure of the code (functionality of classes and methods), and not the detail of their implementation (methods implementation code). It is interesting to go further and assess the understandability of the methods implementations as well, by counting the number of methods examined, and also integrates the code from the external packages in the calculation of this ratio. This further evaluation was not possible because of the lack of time and therefore it is proposes to be performed in the future.
- Evaluate the stability of the AR metric across all scenarios, by comparing package metrics from a scenario to another. The same functional component that is used in different scenarios is expected to have similar AR metrics.

The scenarios

The scenarios cover a large range of the functionalities of ArgoUML. Two of the main functionalities have not been analyzed for practical reasons (source code for that module is not easily integrated in the eclipse project): code generation (generation of java code from a UML model) and reverse engineering (generating a UML model from java code). The scenarios have been imagined to be the ones that could exist in real life, and the UML diagrams used were taken from real life examples. Due to the large size of the traces generated for the scenarios, these are of a moderate size.

Here are the 7 scenarios:

Press Agency (use case diagram):

- Create a new project "Press Agency"
- Create the principal use case (see Annexes section for the diagram used)
- Save the project

Composite design pattern (class diagram):

- Create a new project "Composite design pattern"
- Create the class diagram (see Annexes section for the diagram used)
- Save the project

Object manager (sequence diagram):

- Create a new project "Object manager"
- Crate the sequence diagram (see Annexes section for the diagram used)
- Save the project
-

Alarm clock (state diagram):

- Create a new project "Alarm clock"
- Create the state diagram (see Annexes section for the diagram used)
- Save the project

Correction Todo's:

- Open the "Composite design pattern" project
- Adjust design critics parameters
- Disable non-relevant critics
- Correct all the automatically generated Todo's items
- Save the project as in a new file.

Printing

- Open one by one the following projects: Press agency, Composite design pattern, Object manager and Alarm clock and print each one of them before closing them.

Exportation:

- Open one by one the following projects: Press agency, Composite design pattern, Object manager and Alarm clock and export to XMI files each one of them before closing them.

Guideline for using the tools

Here is a guideline describing the steps to be followed in order to generate the AR maps for any software that you want to analyze (the same steps were used for analyzing ArgoUML):

1) Choose the different scenarios. These one should be use cases that have value to the business concerned by the application, and they should cover all the functionalities that want to be analyzed.

2) Instrument the source code of the application to be analyzed. See Instrumentor\ Instrumentation_plugin_user_manual.pdf for detailed instructions.

3) Run the application form eclipse and realize the scenario one by one. At the end of each scenario, go to the application' eclipse project folder, and rename the trace.txt to a name that reflects the scenario. See Instrumentor\Instrumentation_plugin_user_manual.pdf for detailed instructions.

4) Open the DDRA project in Eclipse and run the application by running the *Run As a java application* command on the \presentation.MainScreen\TraceWindow class. In the DDRA application, go to File-Manage traces and add the trace(s) that has been made in Step 3). You can load several traces at a time, but one must be aware that traces can be very large (several GBs) so this can be a problem if the size of the database is limited (as in the Oracle Express edition, limited to 4GB). Because of that, it is recommended to load with DDRA one or two traces at once. Once the trace(s) has been loaded (it takes approx. 1h for 1GB), you can directly quit RDDA (there is no need to select the trace in the Manage Traces dialog, this is for further analysis of traces with the DDRA tool, which does not offer functionality for the AR metric, instead we will use SQL scripts to do that). RDDA will create a *tracenodeparam* table containing the list of the traces, and for each trace it will create a *tracenodeX* table (x is the number of the trace).

5) In PL/SQL, go to Procedures folder and right click on Process_AR_ALLTRACE and select Test (it is possible that the procedure might have not compiled yet, in this case just compile it by right clicking on it and selecting compile). In the Test window, go to DBMS Output and change Buffer size to 100000 (this is for allowing debug messages to be printed on the console, if the buffer is not big enough there will be an error and the script will stop). Run the script by hitting the F8 key. Wait for the script to complete. The duration can vary depending on the settings of your Oracle database installation. This will create for each trace a tracenodeX_package_list, and it will append to the *Trace_Autonomy_Ratio* table the results with the AR metrics, for all traces (note that the table must exist, empty or not, the creation (or initialization) is done with the *Create_Table_Trace_autonomy_ratio.sql*).

6) If there are other traces that needs to be analyzed, go to RDDA, to File-Manage traces and remove all existing traces and add the ones that you want to add. This will remove the *tracenodeX(s)* table(s) corresponding to the trace(s) and also remove the references to these traces from the *tracenodeparam* table. Go to step 4) to add the remaining traces.

7) Once all traces have been loaded with RDDA and processed with the Process_AR_ALLTRACE, the AR metrics results for all traces are found in the Trace_Autonomy_Ration table. In PL/SQL, open a SQL empty window and type the command “select * from trace_autonomy_ratio” and run it by hitting the F8 key. Right-click on the results pane and select *export results as csv file* and save to a file named, for example, results.csv, in the directory where you want to generate the AR maps (see the “Installation of the required tools” section). Open a terminal and go to that directory and type “generate results.csv”. This will create the AR maps files for all scenarios. You can open each file for each map (one file contains one map) by opening them with Microsoft Treemapper (File-Open as “*Comma-delimited non-cumulative files*”). Go full screen to optimize the display. You can double-click on a package to zoom on it. You can also render the map to the resolution you want (this might be useful is there are many packages and their labels are not shown entirely) by selecting the resolution in View/Options/images and then by creating a graphic with the *File-Save as* command.

Analyzing the results

The main scenario that has been used for analyzing the results is “Exportation” (which consists in importing an existing use case diagram, a state diagram and a class diagram and exporting them to XMI files (see the Scenarios section). As you can see from comparing the different treemaps (see Annexes the section) scenarios are very similar in terms of metrics results, so analyzing packages with the metrics from one scenario is enough for almost all packages. Some packages are an exception to that, because their metrics change significantly in some scenarios. In that cases the scenario used is mentioned in the paragraph with the packages results.

For each package, metrics are given (for the Exportation scenario, unless another scenario is specified), as well as the number of classes examined (read) (ENOC) and the number of total classes (TNOC), in order to evaluate the understandability (see the “Goals to achieve” section), and the calculated understandability. The functionality for the entire package (classes and sub packages) is resumed, as well as the functionality of the classes contained in the package (in each “Classes:” sub-section; for the detail of each sub package, one should look for the paragraph related to the sub-package). This is followed by an interpretation of the package’s autonomy ratio (AR), with details on the package’s coupling, cohesion and export metrics. The dependency the package has with other packages is evaluated by analyzing the detailed coupling data. To evaluate if the package X depends strongly or weakly to another package Y, we compare the coupling of X to Y (name it couplXY) with the cohesion of X (name it cohX). If couplXY >> cohX or If couplXY is more or less identical to cohX, then X depends heavily on Y. if couplXY << cohY, X depends weakly on Y. When we evaluate the impact on the AR of the dependency of package X to other packages, we only consider packages for which coupling is strong, because only those are relevant.

The names of the package are without “org.argouml”, and sometimes they are written in italic to help distinguish them from the text. For example, the org.argouml.uml.diagram package is referred as “*uml.diagram*”. In the detailed results for the metrics, an “.*” is appended to the package’s name to indicate that the described metric (ex. number of coupling calls) is intended the package and all its sub packages.

uml

AR: 5 (cohesion 72, coupling 1383); Export: 459

Understandability: 26 (TNOC 13, TNOC 5)

Functionality:

- All functionalities of ArgoUML that have a direct relationship with UML are implemented in this package (except for the storage of all UML elements which is found in the *model* package). Often, this package (and his sub packages) extends other packages (such as *ui* or *cognitive*) to add UML specific behavior in top of existing base functionalities.
- Classes: UML general utilities (add a stereotype to a collection of model elements, get UML element properties, UML element documentation, generate base/subclasses, get list of composite classes, etc.

1383 coupling calls *for uml.** (of which 15 calls are from the classes of *uml*)

- 933 to model.* (150 to the Model.getFacade(), see the Shortcoming section)
- 98 to application.*
- 81 to cognitive.*
- 79 to i18n
- 75 to ui.*
- 37 to notation.*
- 23 to kernel
- 17 configuration
- 17 to application
- 18 to sequence2.*
- 10 to util
- 6 to profile
- 4 to swingext
- 3 to pattern.*
- 2 to ocl
- 1 to language

72 cohesion calls fo*r uml.**

- 47 from uml.diagram to uml.ui
- 13 from uml.ui to uml.diagram
- 6 between classes of uml
- 4 from uml.diagram to uml.cognitive
- 1 from uml.cognitive to uml.diagram
- 1 from uml.diagram to uml

The package has a very low AR=5, here are the reasons why:

- The classes and sub-packages of *uml* are rather independent one from the others (with the exception of *uml.diagram* which uses functionalities in *uml.ui*). Indeed, the *uml* package exists to regroup functionalities that have in common only the fact that they are related to UML, but there is no functional relatedness between these different entities (again, with one exception). The consequence is a very low cohesion, which combined with the high coupling (see next paragraph) results in a very low AR. Note that in this particular case, where the direct children of the package (classes and sub-packages) are independent, the fact that the AR is low does not necessarily mean that all these direct children are very non autonomous as well. For example uml.diagram has an AR of 24 (which is significantly higher than the AR of *uml*), and the classes in *uml* have a low coupling of 15 altogether (which shows that they are rather autonomous). This is a shortcoming of the AR metrics (see the Shortcomings section)
- The package has an important dependency on a large number of other packages (model, application, cognitive, i18n, ui). Their large number (5 packages) creates a very high coupling, even if each dependency considered independently is not very strong (indeed, each significant coupling per package has an average of 80 calls : 98 to application.* ,81 to cognitive.*, 79 to i18n,75 to ui.*, with the exception of the model package).
- There is a strong dependency on the *model* package (counts for the 3/4 of the total coupling). This strong coupling comes from the functional relatedness of the Model-Control-Viewer architecture pattern.
- Some parts of the package use a reusability-based architecture: they extend functionality from external packages, hence a high coupling. Examples: uml.cognitive extends cognitive, *uml.ui* extends *ui* (see those package description for more details).

In conclusion, the very low AR=5 does reflect the actual functional structure (studied in the source code) of the package. Most of the parts of the package have strong functional dependency on other packages, making it almost impossible to comprehend without understanding the other packages on which it depends. However it does not show that some parts are less dependent on the external packages and also easier to understand. This is a shortcoming of the AR metrics which arises when its direct children are functionally independent (See the Shortcomings section). The high values for the coupling and export metrics shows that the package is heavily used in the scenario and that it implements a functional component.

uml.diagram

AR: 24 (cohesion 191, coupling 595); Export: 297

Understandability: 64 (TNOC 22, ENOC 8)

Functionality:

- Present the diagrams to the user and allow the user to manipulate the diagram through the view.
- Creates classes to represent the diagram to the rest of the project, decouple the rest of the application from the GEF framework (Interface(s), Factory)
- Classes: Base abstract class (interface, Factory) for all diagrams types (ArgoDiagramImpl, ArgoDiagram, DiagramFactory); centralized methods dealing with diagram appearance; undo manager; Apply changes to the UML model with a GEF listener & events; abstract class that extends graphs functionality in GEF to add UML specific functionality (UMLMutableGraphSupport)

595 coupling calls for *uml.diagram.**:

- 290 to model.* (15 from uml.diagram, 55 to Model.getFacade(), see the Shortcoming section)
- 71 to application.*
- 60 to i18n
- 54 to uml.*
- 37 to notation.*
- 31 to ui.*
- 17 to sequence2.diagran
- 11 to configuration
- 11 to kernel
- 5 to util
- 7 to cognitive
- 2 to swingext

The package has a low to average AR of 24. This reflects the fact that it has a certain amount of autonomy, since all diagram functionalities (except for the model functionality, see next sentence) are implemented here. Diagrams are implemented with the Model-View-Controller architecture pattern: the uml.diagram package contains the Controller and the View, and the model package contains the Model. This explains the heavy dependency of *uml.diagram* on *model*, and this is the main reason why *uml.diagram* does not have a high AR. The high values for the coupling and export metrics shows that the package is heavily used in the scenario and that it implements a functional component.

uml.diagram.ui

AR: 37 (cohesion 184, coupling 310); Export: 279

Understandability: 82 (TNOC 114, ENOC 20)

Functionality (all contained in classes, there is no sub package):

- Common User Interface functionality (Actions) to all diagrams (copy/paste/delete, abstract modifier, multiplicity modifier, input modes),
- Basic graphic figures (extended from the GEF framework) used by diagrams (rectangles, associations, etc.)
- Base abstract class UMLDiagram that extends ArgoUMLImpl that provide base to all Diagrams
- Base class for Property panels for diagrams

310 coupling calls for *diagram.ui*:

- 100 to model.* (25 to Model.getFacade(),see the Shortcoming section)
- 39 to uml.diagram
- 36 to uml.diagram.static_structure.ui (all to extended classes)
- 30 to notation.*
- 15 to diagram.state.ui (all to extended classes)
- 15 to sequence2.*
- 13 to i18n
- 13 to diagram.use_case.ui (all to extended classes)
- 10 to uml.ui
- 10 to application.helpers
- 9 to ui.*
- 7 to cognitive
- 2 to swingext.*
- 1 to application.events

The package represents the View (UI elements and figures) and some of the part of the Controller (Actions) of the Model-Controller-View architecture pattern used for UML diagrams, the rest of the Controller part being implemented in the *uml.diagram* and the Model in *model*. On top of that there is a reusability-based architecture: *uml.diagram* and *uml.diagram.ui* represent the basic common Controller and View functionality for all diagrams, and sub-packages *uml.diagram.use_case* and *uml.diagram.use_case.ui* (and similar packages for all other diagrams: *uml.diagram.static_strucutre* for class diagrams, *uml.diagram.state* for state diagrams and so on). The *uml.diagram.ui* package depends heavily on other packages mainly because of these two architectures. Its functionality is quite cohesive: user interface elements control Actions and graphic figures for UML diagrams. The average value of AR=37 shows correctly the autonomy ratio level of the package. The high values for coupling, cohesion and export shows that the package is heavily used in the scenario and that it implements a functional component.

uml.diagram.use_case

AR: 0 (cohesion 0, coupling 127); Export: 39

Understandability: 0 (TNOC 1, ENOC 1)

Functionality:

- All functionality specific for the UML use case diagram (except for the model part)
- Classes : only one class UseCaseDiagramGraphModel that extends uml.diagram.UMLMutableGraphSupport : bridge between the UML meta-model representation of the design and the GraphModel interface used by GEF

For the only class in the package there is 0 coupling (and 0 cohesion since there is only one class). All coupling comes from the sub package uml.diagram.use_case.ui. The uml.diagam.use_case package represent the use case diagram specific Controller and View part of the Model-Controller-View architecture pattern used for UML diagrams (the Controller part being implemented in the *uml.diagram* and the Model in *model*). On top of that there is a reusability-based architecture: the class and the sub-packages of the uml.diagram.use_case extend the functionality in uml.diagram and uml.diagram.ui. For these reasons, the package depends heavily on external packages. Additional to that, the package is located deeply into a reusability architecture, where functionality only represents one type of diagram. For those reasons, AR is very low. And since there is no cohesion (there is only one class), AR is actually 0. The average values for coupling and export shows that the package is used in the scenario and that it implements a functional component. The fact that these values are average and not high is explained again by the fact that the package is situated deep in the reusability architecture.

uml.diagram.use_case.ui

AR: 2 (cohesion 3, coupling 127); Export: 39

Understandability: 55 (TNOC 13, TNOC 6)

Functionality (all contained in classes, there is no sub package):

- All UI functionality specific for the UML use case diagram
- Classes: UMLUseCaseDiagram that extends UML Diagram and represent the diagram; classes (extend uml.diagram.ui classes) for specific use case UML elements (use case, actor, extend, include, etc.)

0 cohesion calls from uml.diagram.use_case

127 coupling calls for uml.diagram.use_case.ui:

- 41 to model.* (2 to Model.getFacade(),see the Shortcoming section)
- 40 to uml.diagram.ui (to base classes)
- 17 to application
- 14 to i18n
- 7 to uml.ui
- 4 to uml.diagram (none from base classes)
- 3 to ui.targetmanager
- 1 to util

The analysis for this package is similar to uml.diagram.use_case. The AR very low (=2) but not 0 as for the uml.diagram.use_case, because there is a small cohesion (= 3).

uml.diagram.activity

AR: 0 (cohesion 0, coupling 1); Export: 4,
Understandability: 0 (TNOC 1, ENOC 1)

uml.diagram.deployment

AR: 0 (cohesion 0, coupling 1); Export: 4,
Understandability: 100 (TNOC 1, ENOC 1)

uml.diagram.collaboration

AR: 0 (cohesion 0, coupling 1); Export: 4
Understandability: 0 (TNOC 1, ENOC 1)

The architecture of these three packages (and their sub package) is similar to the uml.diagram.use_case (and its sub package). Metrics are not similar though. Indeed, in the scenario used for the analysis (Exportation), these types of diagrams were not used. In other terms, these packages do not represent a functional component used in the scenario. This is reflected in the very low values for the coupling, cohesion and export. The AR=0 of these packages (and all their sub packages) is a consequence of this. Note that if these packages are used in this scenario is only for the functionality of empty diagrams initialization.

uml.diagram.state

AR: 1 (cohesion 2, coupling 246); Export: 60,
Understandability: 0 (TNOC 1, ENOC 1)

uml.diagram.static_structure

AR: 0 (cohesion 0, coupling 126); Export: 126
Understandability: 0 (TNOC 1, ENOC 1)

The architecture of these two packages (and their sub package) are similar to the uml.diagram.use_case (and its sub package). Their metrics and analysis is also similar to the uml.diagram.use_case.

uml.ui

AR: 19 (cohesion 42, coupling 176); Export: 103

Understandability: 82 (TNOC 115, TNOC 20)

Functionality:

- Basic UI functionality related to UML
- Classes: abstract class (PropPanel) provides the basic layout and event dispatching support for all Property ; Interface for the tabs containing panels (Todo, Properties, Documentation, Presentation, Source, …); basic UI elements related to UML, (extended from Swing) ; Base class for undoable actions; ; Action classes (create/save/export projects, create/remove diagrams, etc) ; Extends some basic functionality from ui

176 coupling for uml.ui.*:

- 51 to model (17 to Model.getFacade(),see the Shortcoming section)
- 45 to ui.* (3 to base classes)
- 25 to application.*
- 14 to uml.diagram.*
- 4 to configuration
- 17 to i18n
- 8 to kernel
- 6 to persistence
- 2 to swingext
- 1 to sequence 2

103 export for uml.ui:

- 25 from ui.* (none from extended classes)
- 15 from Arbitrary Root Node
- 10 from uml.diagram.ui
- 7 from uml.diagram.static_structure.ui
- 7 from uml.diagram.use_case.ui
- 5 from application
- 1 from kerrnel
- 1 from profile.init
- 1 from uml.diagram.activity.ui
- 1 from uml.diagram.collaboration.ui
- 1 from uml.diagram.deployment.ui

The package represents some part of the View functionality and some part of the Controller functionality form the Model-View-Controller architecture pattern. It contains all base UI related elements (View) and all Actions related to UML (Controller). A more specific UI functionality for UML is contained in the uml.diagram package. On top of that there is reusability architecture: uml.ui uses the functionality in *ui* (this architecture is not implemented by heavy class extension, but rather by simple reuse of functionalities between non extended classes). The package is dependent mainly on the *model* and on the *ui* packages. The low-to average value of AR=19 reflects this dependency.

The average values of coupling and export show that the package is used in the scenario and that it implements a functional component. Note that in this scenario (Exportation), functionality for editing diagrams is not used. This can be observed if we compare the metrics for this scenario to the metrics for other scenarios where the creation of diagram is done (such as Press Agency, Composite Design Pattern or Object Manager scenarios). The metrics for the package in these scenarios are sensibly higher (especially the coupling, cohesion and export metrics; the AR is also higher but the difference is less significant: this shows a certain stability of the AR metrics).

It is interesting to compare these three packages:*uml.diagram.ui, ml.ui* and *ui*. By comparing their metrics, we see that they reflect the reusability architecture of the UI functionality of ArgoUM: the *uml.ui* package (which holds common UI UML functionality) contains less functionality than ui (which contain basic UI non-UML functionality) and less functionality than *uml.diagram.ui* (which contains UI functionality for UML diagrams). This is coherent with the functionality in ArgoUML: most of UML related UI functionality is for UML diagrams.

uml.ui.behaviour

AR: 0 (cohesion 0, coupling 20); Export: 11,

Understandability: not defined (TNOC 0, ENOC 0)

Functionality:

- Mainly Actions and some UI elements (extended from uml.ui) for UML-related functionality classified in the “behavior” category.

20 coupling for uml.ui.behaviour:

- 10 to application.*
- 8 to i18n
- 2 to model
- 2 to ui

The package contains no classes, only sub-packages with classes. Its purpose it to create a decomposition in categories and sub categories. Cohesion is low, hence AR = 0. This is due to the fact that the functionality almost consists of independent control Actions (in contrast to *uml.ui* where more UI elements are implemented, which is more cohesive). The low values of the metrics shows that the functionality of the package is not heavily used in the scenario (that happens in all scenarios as well). Indeed, the diagrams used in the scenarios are quite simple, using a limited set of UML elements. Hence little of the functionality for creating these elements is used (implemented through Actions). Consequently, the value of the AR can be different in another scenario if the package gets more used. Still, as we conclude by comprising different scenarios, the AR metric is quite stable from one scenario to another and differences are often not very significant.

In conclusion, the very low AR=0 reflects the structural architecture of the source code. It is highly dependent on other packages.

uml.ui.foundation

AR: 0 (cohesion 0, coupling 19); Export: 4,

Understandability: not defined (TNOC 0, ENOC 0)

The architecture of these packages (and their sub package) is similar to the uml.ui.behavior (and its sub package).Their metrics and analysis is also similar.

cognitive

AR: 58 (cohesion 91, coupling 66); Export: 202

Understandability: 68 (TNOC 25, ENOC 8)

Functionality:

- Base functionality for design critics and todo items, design issues, design goals. These are general in nature critics and todo items, not UML specific (these are in uml.cognitive)

66 coupling for cognitive.*:

- 27 to uml.cognitive.* (most to extended classes)
- 11 to ui
- 9 from application.*
- 8 to configuration
- 5 o i18n
- 4 to util
- 2 to model.*
- 1 to swingext

The package depends mainly on extended classes in uml.cognitive, which extends the functionality of the package to specific UML treatments. The only architecture used here is the reusability pattern. There is no Model-View-Controller architecture pattern between the package and other packages. This is reflected in the average to high value of the AR=58. Indeed, the package is rather autonomous and it is possible to understand its main functionality without understanding the other packages in ArgoUML.

The high values of coupling and export show that the package is heavily used in the scenario and that it implements a functional component (it happens that in all scenarios as well this is the case). Indeed, design critics and todo items are enabled by default and constantly generated by a separate thread in ArgoUML.

cognitive.checklist

AR: 13 (cohesion 1, coupling 7); Export: 13

Understandability: 75 (TNOC 4, ENOC 1)

Functionality:

- Classes to implement checklists to be used with designs critics

7 coupling for cognitive.checklist:

- 2 to application.api
- 2 to cognitive
- 2 to ui
- 1 to i18n

The package depends moderately on application, cognitive and ui packages. The weak cohesion is partly due to the small number of classes (4) that actually exists in the package. This is reflected by the low value of the AR=13. However, the package is rather autonomous, indeed it possible to understand its main functionality without understanding other packages. This is a shortcoming of the AR metric and it sometimes arises where there is a small number of classes (See the Shortcomings section). Low values for the metrics show that this functionality has been weakly used in the scenario. Indeed, in all scenarios, the cognitive checklist has not been used, only initialized. Consequently, the value of the AR can be different in another scenario if the package gets more used.

cognitive.checklist.ui

AR: 27 (cohesion 3, coupling 8); Export: 10

Understandability: 0 (ENOC 2, TNOC 2)

Functionality (all contained in classes, there is no sub package):

- UI for the design critics checklists. Extends functionality in application.api.AbstractArgoJPanel to render the UI for the checklist

8 coupling for cognitive.checklist.ui:

- 3 to cognitive
- 2 to ui
- 2 to application.api
- 1 to ui

The package depends moderately on application, cognitive and ui packages. It has also a small number of classes (2). However, in contrast to the parent package cognitive.checklist, there is a stronger cohesion (3), and consequently, the average value of AR=37 reflects more correctly the moderate degree of autonomy of the package. Low values for the metrics shows that this functionality has been weakly used in the scenario. Indeed, in all scenarios, the cognitive checklist has only been initialized, not used. Consequently, the value of the AR can be different in another scenario if the package gets more used.

cognitive.ui

AR: 26 (cohesion 16, coupling 45); Export: 32

Understandability: 23 (ENOC 9, TNOC 39)

Functionality (all contained in classes, there is no sub package):

- Base UI functionality for all design critics, ToDo items and wizards, not UML specific.

45 coupling for cognitive.checklist.ui:

- 24 to cognitive
- 9 to ui
- 6 to application
- 2 to configuration
- 2 to model
- 1 to swingext
- 1 to i18n

The package depends mainly on the *cognitive* parent package. The architecture here is responsibility based. Cognitive.ui separates all UI classes that support the design critics and "To do"'s implemented in the *congtive* package. *Cognitive.ui* has a moderate average AR=26 because of this dependency. . The moderate values for coupling, cohesion and export shows that the package is used in the scenario and that it implements a functional component. Indeed, design critics and todo items are enabled by default and constantly generated by a separate thread in ArgoUML.

cognitive.critics

AR 0 (cohesion 0, coupling 28); Export: 26 (for the “CorrectionTodos” scenario)

Understandability: 0 (TNOC 3, ENOC 3)

Functionality:

- Classes : base class for wizards used to automatically solve design critics;
  Class that implements snooze functionality for design critics

28 coupling for cognitive.critics:

- 14 to cognitive
- 11 to uml.cognitive.critics
- 6 to application
- 2 to configuration
- 2 to util
- 1 to ui

The package contains an abstract class extended in uml.cognitive.critics, as well as a concrete class that provides a snoozing functionality. The package is dependent on the parent cognitive package and on the extended classes in uml.cognitive.critics. The cohesion is 0 because there is only one class used. There is a high coupling so the AR correctly shows the low level of autonomy of the package. The moderate value of the export metric shows that the package is used in this scenario more than in the others (where the export is only 4 or 5). In this scenario (as well on all other scenarios), the class who implements the snooze functionality is not used.

cognitive.critics.ui

AR 40 (cohesion 10, coupling 15); Export: 40 (for the “CorrectionTodos” scenario)

Understandability: 60 (TNOC 5, ENOC 2)

Functionality (all contained in classes, there is no sub package):

- Functionality for the UI dialog to manage critics and their settings

15 coupling for cognitive.critics:

- 12 to cognitive
- 2 to util
- 1 to ui

The package has a moderate value for AR=40 because it has a moderate cohesion between the 2 classes that implements the dialog. Coupling is moderate to the cognitive package, so the AR shows the moderate autonomy of this package.

uml.congnitive

AR 1(cohesion 5, coupling 669); Export: 117

Understandability: 50 (TNOC 4, ENOC 2)

Functionality:

- Extends base cognitive classes from the cognitive package, add base UML functionality

669 coupling for uml.cognitive:

- 567 to model.* (71 to Model.getFacade(),see the Shortcoming section)
- 72 to cognitive.* (mostly from base classes)
- 6 to profile
- 5 to kernel
- 5 to profile
- 4 to util
- 3 to pattern.*
- 2 to configuration
- 2 to ocl
- 1 to i18n

The package has e very low AR because it depends strongly on *model* and *cognitive*. The dependency on *model* is due the Model-View-Controller architecture pattern, and the dependency on *cognitive* is due to a reusability-based architecture, since *uml.cognitive.critics* heavily extends *cognitive* classes. The very low value of AR=1 shows this strong dependency.

uml.congnitive.critics

AR 12 (cohesion 85, coupling 648); Export: 108

Understandability: 92 (TNOC 10, ENOC 115)

Functionality:

- All UML design critics and wizards, extend classes in the cognitive and uml.cogntive package

The architecture of the *uml.cognitive* package (and its sub-packages, with the exception of the *ui* package) is similar to the architecture of the cognitive package. The separation between uml.cognitive and uml.cognitive.critics is based on reusability: *uml.cognitive* implements a few base classes and uml.cognitive.critics a large number of critics and wizards classes that uses uml.cognitive and heavily extends it. The *uml.cognitive.critics* has a strong dependency to the *model* package, mentioned in its father package (*uml.cognitive.critics*) description. The low value of AR=12 shows this strong dependency. In contrast to its father, the package is more cohesive hence it's superior AR.

uml.congnitive.checklist

AR 0 (cohesion 0, coupling 19); Export: 1

Understandability: 0 (TNOC 2, ENOC 2)

Functionality:

- Extends classes from *cognitive.checklists* to add UML functionality

19 coupling:

- 14 to model.*
- 3 to cognitive.checklists
- 1 to i18n
- 1 to util

The package heavily depends on *model* due to the Model-View-Controller architecture pattern (see parent package). It depends on cognitive.checklists because it extends classes from this package (reusability based architecture). It contains a small number of classes (2), which explains partly the value of 0 for cohesion. The reason for the very low AR=0 might be due to the shortcoming of the AR which arises when the package contains a very low number classes (see the Shortcoming section). However, we are not sure that this is the case because the low values for the metrics indicates that the functionality has been weakly used in the scenario (indeed, in all scenarios, the cognitive checklist has not been used, only initialized). Consequently, in other scenarios there could be more cohesion and the value of the AR can be different (higher), so in this case there would be no shortcoming because of the small number of classes, and AR would show correctly a larger value.

model

AR 93 (cohesion 38, coupling 3); Export: 1518

Understandability: 95 (TNOC 85, ENOC 5)

Functionality:

- Remove knowledge from the rest of ArgoUML of what model repository is in use (eg. MDR, EMF/UML2, NSUML) and to give a consistent interface for manipulating data within those repositories. In the ArgoUML version used in this work, the MDR implementation is used.
- Classes : Set of interfaces to be implemented by classes in the model.mdr (Facede, Helpers, etc); A class Model which provides the façade of the model.mdr implementation (through the Model.getFacede() function)

The package implements the model part of the various Model-View-Controller architecture patterns used in ArgoUML. In the architecture of the MVC pattern, the model has some weak coupling to implement notifications for changes. In ArgoUML, these notifications are implemented in other packages (uml.diagram, application.events). A consequence of this is the very high AR=93 for the package, which reflects indeed that it is almost perfectly autonomous. The very high value of the export metric (the largest export of all packages) shows that the package is used in the scenario (it happens that it is the case in all scenarios) ant it implements a functional component. The very high value of the export also indicates that this is a key component to ArgoUML (because it is heavily used by other components), hence it is important to have a high AR so that it is easily understandable and easily maintainable (see section Guidelines for the use of the AR metric).

model.mdr

AR 93 (cohesion 59, coupling 7); Export: 1040

Understandability: 91 (TNOC 43, ENOC 4)

Functionality (all contained in classes, there is no sub package):

- Implements functionality of interfaces in model. This includes getters for UML elements, creating UML elements, copy UML elements, event pump (notify registered listeners of changes made to the model), etc.

The architecture used by this package is explained in the father package model. Metrics and analysis is similar.

util

AR 17 (cohesion 2, coupling 10); Export: 41

Understandability: 45 (TNOC 27, ENOC 15)

Functionality:

- Classes: some basic UI functionality; general collection utilities; string operations; file extensions utilities

The package contains small functionalities which cannot be categorized into another existing package.

10 coupling for classes in util package (there is no coupling for *util.osdep* and *util.logging*):

- 3 to model.
- 3 to i18n
- 2 to cognitive
- 2 to configuration

This package does not heavily depend on other packages, but its AR is low because its cohesion is low. The export is not low (41), which mean that in this scenario the packages is not underused, and so this scenario's AR=10 probably reflects the structure of the whole packages. If we check the source code of the package we can confirm indeed that the there is no relationship between all the functionalities implemented, hence the lack of cohesion (all classes in util have no cohesion, and packages *util.osdep* and *util.logging* have no relationship between them). In this case the low value for the AR does not reflect the fact that the functionality in package is rather autonomous. This is a shortcoming or the AR metric that arises when the direct children are functionally independent (see the Shortcomings section).

util.osdep

Export: 41

Understandability: 33 (TNOC 3, ENOC 2)

Functionality (all contained in classes, there is no sub package):

- Operating system specific functionalities

Since the package has no coupling, it is perfectly autonomous. Cohesion is 0, but since export is low there are chances that there might be some cohesion that can appear if the export is higher in other scenarios. If we check the source we see that the package contains a small number of autonomous classes. In this case then, the scenario AR reflects the AR of the structure of the whole package.

util.logging

Export: 5

Understandability: 33 (TNOC 3, ENOC 2)

Functionality (all contained in classes, there is no sub package):

- Logging (debug) functionalities

The architecture, metrics and analysis for this package is similar to the *util.osdep* package.

i18n

Export: 136

Understandability: 0 (TNOC 1, ENOC 1)

Functionality:

- Localization (string translation and format handling to match locale settings)

Since the package has no coupling, it is perfectly autonomous. Export is higher so chances are that the scenario's AR= not defined (means coupling=cohesion=0) reflect the AR of the structure of the whole package. If we check the source we see that the package contains one single autonomous class, so indeed the AR of the whole package (under all possible scenarios) is the same as the scenario's AR, which reflects a perfect autonomous package.

pattern

AR 17 (cohesion 0, coupling 21); Export: 7

Understandability: not defined (TNOC 0, ENOC 0)

Functionality:

- Functionality related do design patterns

21 coupling for pattern (all coupling comes from sub package *patern.cognitive.critics*):

- 20 to model.*
- 1 to cognitive

The package contains no classes and serves only for classification in different categories. For the analysis of the metrics, see the *patern.cognitive.critics* package,

pattern.cognitive

AR 17 (cohesion 0, coupling 21); Export: 7

Understandability: not defined (TNOC 0, ENOC 0)

Functionality:

- Cognitive (design critics and “To do”’s) related do design patterns

The package contains no classes and serves only for classification in different categories. For the analysis of the metrics, see the *patern.cognitive.critics* package,

pattern.cognitive.critics

AR 17 (cohesion 0, coupling 21); Export: 7

Understandability: 60 (TNOC 5, ENOC 2)

Functionality (all contained in classes, there is no sub package):

- The designs critics for some patterns (Singleton and Façade)

21 coupling for pattern.cognitive.critics:

- 20 to model.*
- 1 to cognitive

The package has an AR = 0 because it has 0 cohesion and dependency to *model*. Since export is low, there is a chance that the AR of the scenario does not reflect the whole package AR. In this case, the AR 0 of the scenario does not reflect the AR of the structure of the whole package, so under other scenarios the AR could be > 0. Low values for the metrics shows that this functionality has been weakly used in the scenario. Indeed in the “Correction todos” scenarios, the design critics generated were not related to patterns. Consequently, the value of the AR can be different in another scenario if the package gets more used. Indeed, when we look at the source code, we observe that there is cohesion between the classes (which also extends *uml.cognitive.critics* classes).

application

AR 6 (cohesion 12, coupling 173); Export: 153

Understandability: 25 (TNOC 4, ENOC 1)

Functionality:

- helper functions specific to ArgoUML that does not fit on any other package (application events, etc.)
- Classes: Entry point for the application; startup and initialization of the application

173 coupling:

- 39 to uml.*
- 36 to cognitive
- 26 to ui.*
- 16 to notation.*
- 14 to configuration
- 13 to model.*
- 9 to utill.*
- 7 to kernel
- 5 to moduleloader
- 4 to i18n
- 4 to pattern.cognitive.critics
- 1 to profile

The package depends moderately on many other packages. This is one of the reasons why AR is low. Another reason is because cohesion is low. The export is not low (153), which means that in this scenario the packages are not underused, and so this scenario's AR=6 probably reflects the structure of the whole packages. If we check the source code of the package we can confirm indeed that the there is no relationship between all the functionalities implemented, hence the lack of cohesion. In this case the low value for the AR does not reflect the fact that the functionality in package is rather autonomous. This is a shortcoming or the AR metric that arises when the direct children are functionally independent (see the Shortcomings section).

The high values for the coupling and export metrics shows that the package is heavily used in the scenario and that it implements a functional component.

application.helpers

AR 18 (cohesion 2, coupling 9); Export: 118

Understandability: 33 (TNOC 3, ENOC 2)

Functionality (all contained in classes, there is no sub package):

- Application version number management; UI resource management (icons)

9 coupling for application.helpers:

- 8 to model.*
- 1 to i18n

The package depends weakly on the *model* package. The weak cohesion is mainly due to the small number of classes (a total of 3: 1 for Application version number management and 2 for UI resource management) that actually exists in the package. This is reflected by the low value of the AR=18. However, the package is rather autonomous, indeed it possible to understand its main functionality without understanding other packages. This is a shortcoming of the AR metric and sometimes arises where there is a small number of classes (SEEEE for more details). The high values for the export metric shows that the package is heavily used in the scenario and that it implements a functional component.

application.events

AR 67 (cohesion 8, coupling 4); Export: 16

Understandability: 87 (TNOC 15, ENOC 2)

Functionality (all contained in classes, there is no sub package)::

- Events management

4 coupling for application.helpers: to ui.*

The package depends weakly on the ui package. It has a moderate cohesion. There are numerous classes in this package. The AR=67 correctly reflects the strong autonomy of the package.

The values of the coupling, cohesion and export for this package are low on all scenarios. This gives the indication that maybe the whole package is not made to be heavily used in ArgoUML. Looking at the code indeed this is the case (a hundred of references throughout all the code). In this case, despite the small coupling, cohesions and export metrics, the package implements a functional component.

application.api

AR 67 (cohesion 8, coupling 4); Export: 16

Understandability: 58 (TNOC 7, ENOC 3)

Functionality (all contained in classes, there is no sub package):

- Small set of static methods and definitions about the Argo installation; Application initialization

14 coupling for application.helpers: to *configuration*.

The functionality used in the package depends weakly on the configuration package. It has cohesion of 0, which is due to the fact that the classes are independent. In this case the very low value of AR=0 does not reflect the fact that the classes are rather autonomous. This is a shortcoming of the AR metric that arises when direct children are independent (see the Shortcomings section)

The export value is moderate and reflects the fact that the package implements a functional component.

Shortcomings of the Autonomy Ration Metric

- In the situation where a package has functionally independent first children (classes or packages), the AR metric may not show correctly the autonomy of the package. Indeed, if the coupling for each of these children is low (or moderate), the coupling of the father package can be high (since it is the sum of all the children's coupling). Since the cohesion of the father package is very low (or even 0), the AR is very low as well. However, the package is autonomous from the viewpoint of understandability, because each of the package's children can be understood in isolation since it has low coupling with the external packages. As a solution for this shortcoming I propose that when analyzing low AR packages, one must look at the cohesion of the package, to see if it is *extremely* low compared to the coupling. If it is the case, then one should look at the AR of each sub-package to estimate the global autonomy of the parent package (instead of looking at the parent package AR). Examples of packages in such situations in ArgoUML: *uml*, *util*.

- Another similar situation to the one mentioned above, is when there is a very small number of (large) classes in a package. In this case, since cohesion is not computed inside classes (considered as a black box), the cohesion of the package is made of the cohesion between classes, and since there is a small number of them, the cohesion is low. As a consequence, the AR of the package may not reflect correctly the autonomy of the package, for the same reasons as described in the situation where children are functionally independent. As a solution for this shortcoming, I propose that for the package in this situation, one could decide to compute (and display on the AR maps) classes with their AR metric (computing a AR for a class is similar to computing an AR for a package). Examples of packages in such situations in ArgoUML: *cognitive.checklist, uml.congnitive.checklist***,** a*pplication.helpers*.

- Some methods that do not represent functionality are called an important number of times. Because of this, the metrics (coupling or the cohesion) may be bigger that it should be, since these metrics should represent the amount of functional dependency. For example, the Model.getFacade() method in ArgoUML is such an example. For each call to a method in the model, this method is called first do get the façade of the model implementation, and then the actual method to the desired functionality is called trough the façade. The reasons for calling getFacede() every time (in contrast to calling it only in a class constructor and then keeping in an attribute) has not been investigated. However, the impact of this function call is insignificant for all packages in ArgoUML. Indeed, all packages have maximum average of only 10% of the total coupling generated by the call to getFacade(). If this 10% were removed, the AR would be only 1% bigger, so this is not significant. As a solution for this shortcoming, we propose to use the functions in the Shortcoming.sql script which offers functionality to detect most used function calls and also to calculate the impact they have on the metrics. GetFacade() is the only method in ArgoUML in this situation.

- With the AR metric alone, one could not evaluate at which extend the functional components map onto the program substructures. If we analyses the details of the AR (the coupling and the cohesion metrics), and also the export metric for the package, one could evaluate with more certitude the extent of mapping. Indeed, if the coupling, cohesion and export metrics are large, chances are that a lot of the existing code in the package has been used as functionality in the analyzed scenario, hence probably most of the package code

represents a functional component. However, this method is not 100% accurate because we don't know how much of the code in the package has been used. To solve this issue, I propose to create a metric, called, for example, the Component to Structure Mapping Ratio (CSMR) metric, calculated like this: (Number of distinct methods in the package (and sub packages) that are used in a scenario / number of total distinct methods in the entire code of the package (and sub packages))*100. Using this metric, one could evaluate with 100% accuracy, for each package, at which extend the functional components map onto the package concerned.

Guidelines for using the Autonomy Ratio Metric

1) By using the AR Maps, find the packages with the most important export metric, among all analyzed scenarios. These are the packages that implement the functionality which is the most used throughout the entire system. Because of that, these packages are the most susceptible to be in the need of maintenance, so they have to be as most autonomous as possible, in order to be easily understood. If for these packages the AR is low, one should find the reasons why (see next paragraph) and try to improve the autonomy if possible.

2) For each package, understand the reasons why the AR is low. If the reason is not found in this following list, then maybe the architecture of the system must be revised (see the Conclusion section for more details about these reasons):
    - The package is dependent (weakly or moderately) on a large number of relatively autonomous packages, and there is a good reason to keep these packages separated.
    - The package is part of an architecture based on other criteria than functionality: reusability (the most used criteria), modifiability, performance (the two most commonly used besides reusability), usability, testability, availability and others.
    - The package has independent direct children and coupling for each one is low or moderate.
    - The package has a very small number of classes

3) Find the packages for which cohesion, coupling, and export metrics are all low under all scenarios. These packages might not be corresponding to a functional component, or they may implement parts of different functional components. Analyze further the code and the structure of these packages. If so is the case, then the architecture of the system should be revised. Ideally, each package must implement one functional component.

Conclusion

As it was explained in the introduction section, ideally, an (easily) understandable system should be made of packages that all have a high AR metric.

In theory, the coupling of a package is smaller (or equal) to the sum of the coupling of all its sub packages (the demonstration of this, based on the set algebra, is trivial). This implies that higher level (in the tree hierarchy) packages tend to have smaller coupling. The functional decomposition of packages implies that higher level packages tend to have less cohesion between direct children as well (because at higher level, cohesion between packages corresponds to coupling between strong functionally distinct packages, and this coupling is low). Since AR is a ratio between cohesion and coupling, this means that AR does not necessarily become smaller at higher or at lower levels. This implies that theoretically and under the assumption that packages are decomposed on functionality basis, having all packages with a high AR could be possible.

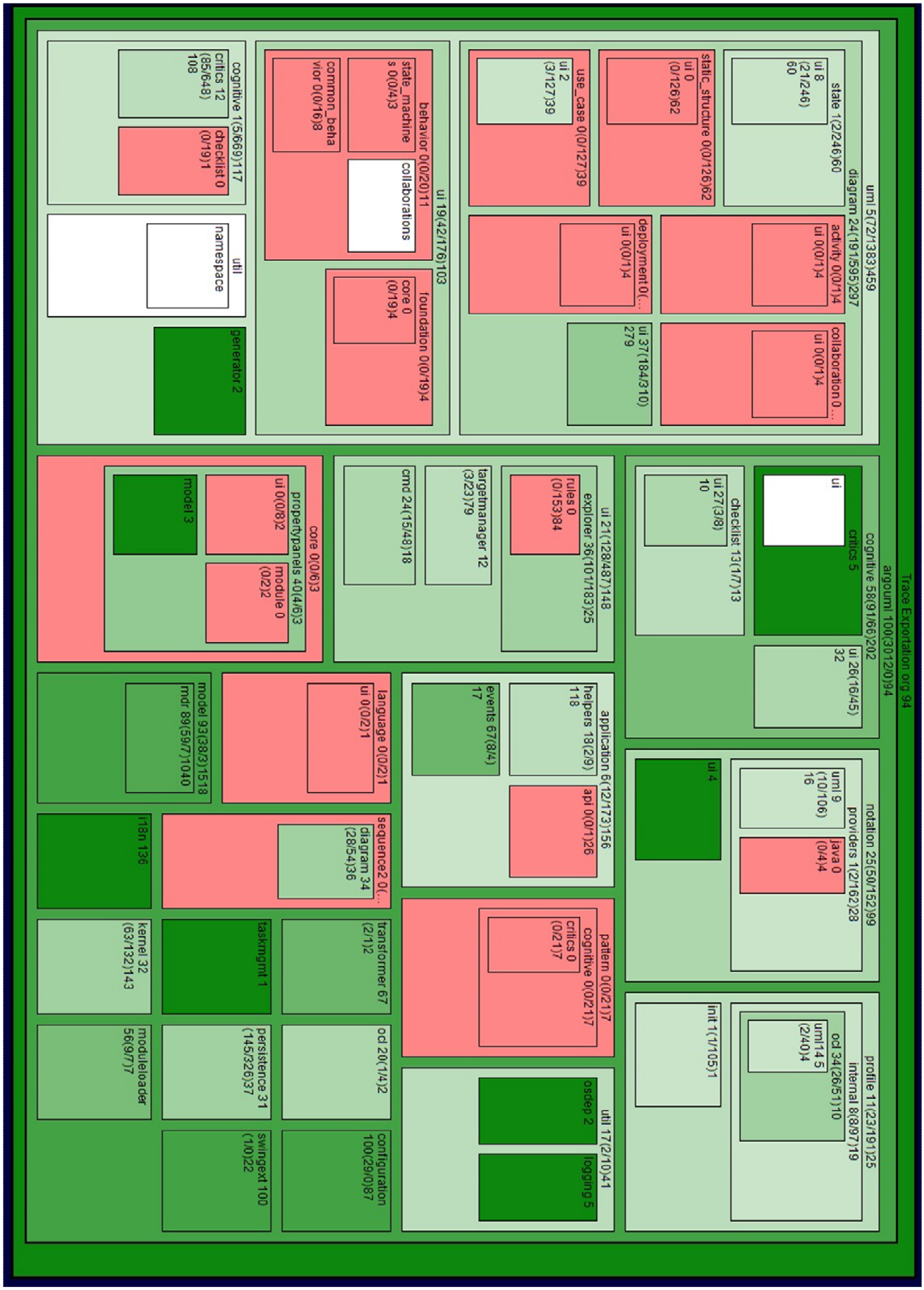
To resume, in theory, we expect from a software which is easy to understand to have all packages with high autonomy ratio, and we expect higher level packages to have a higher autonomy ratio than lower level packages.

In practice, the analysis of the ArgoUML (which has a good architecture) shows that the ideal situation of having all packages with a high AR ration is difficult to achieve. As well, packages on the low level sometimes have higher AR than packages on high level. I think there are three reasons for this:

1) At some point in the hierarchy, high coupling between distinct functional packages must exist. This occurs for example when a functionality is decomposed in a consequent number of distinct responsibilities which all need functionality one from another. In ArgoUML, for example, the functionality of the program is divided at a high level into 20 main packages which implement distinct responsibilities. As a consequence of this, the *uml* package, for example, has strong coupling due to collaboration with numerous other packages such as *application*, *cognitive*, *configuration*, *i18n*, *model*, *notation*, *ui* and *util*.
2) Often packages decomposition is based on nonfunctional (or non-responsibility) criteria. This includes reusability (the most used criteria), modifiability, performance (the two most commonly used besides reusability), usability, testability, availability and others. The use of these criteria can produce high coupling between packages, Example in ArgoUML: The architecture of the package which implements the visual UML diagrams (*uml.diagram* and its sub packages) is based on reusability (see the Results section for more details). Coupling between packages that extend other packages can be high.
3) Sometimes packages contain functionality in sub-packages that are not functionally related (they are functionally independent). In this case, cohesion is very low or 0, and consequently, even with a small or moderate coupling, the AR is very slow. In this case, the autonomy ratio does not reflect the actual autonomy of the packages (since the package is rather autonomous with a small or moderate coupling). This is a shortcoming of the autonomy ration metric (see the Shortcomings section). The same problem may arise, for similar reasons, when a package has a very low number of classes (see the Shortcomings section).

To conclude, the analysis of ArgoUML has confirmed that the AR metric efficiently shows (with some exceptions, see the Shortcoming section) the level of functional autonomy of code sub structures (packages), as well as the degree of the empirically evaluated understandability of each of these sub structures. For packages with low AR, the reasons for the lack of autonomy are found in the different necessary architectures that are used. The packages were found to be functional components, as expected from ArgoUML, which has a good design. The AR metric has proven to be stable: each package has similar AR when its functionality is utilized across different scenarios of using ArgoUML. Colored maps have been created (AR maps) that helps with the overview of the repartition of the AR metric to the entire system and facilitates detecting eventual problems. Some future work has been proposed as a research that may lead to add more functionality to these maps. A metric has been created (export) to facilitate both the detection of important functional packages and to improve the assessment of the mapping between program substructures and functional components. A new metric has been proposed (see the Shortcoming section) to further improve the assessment of the mapping between program substructures and functional components. Future work has been proposed to evaluate even further the mapping between the AR metric and empirically evaluated understandability of each substructure.

Annexes


Trace Exportation.org 94
argouml 100(30120)94
uml 5(72/1383)459
diagram 24(191/595)297
state 1(2/246)60
ui 8 (21/246) 60
activity 0(0/1)4
ui 0(0/1)4
static_structure 0(0/126)62
ui 0 (0/126)62
use_case 0(0/127)39
ui 2 (3/127)39
deployment 0(...
ui 0(0/1)4
collaboration 0...
ui 0(0/1)4
ui 37(184/310) 279
ui 19(42/176)103
behavior 0(0/20)11
state_machine s 0(0/4)3
collaborations
common_beha vior 0(0/16)8
foundation 0(0/19)4
core 0 (0/19)4
cognitive 1(5/669)117
critics 12 (85/648) 108
checklist 0 (0/19)1
util
namespace
generator2
cognitive 58(91/66)202
critics 5
ui
ui 26(16/45) 32
checklist 13(17/13)
ui 27(3/8) 10
ui 21(128/487)148
explorer 36(101/183)25
rules 0 (0/153)84
targetmanager 12 (3/23)79
cmd 24(15/48)18
application 6(12/173)156
helpers 13(2/9) 118
api 0(0/1)26
events 67(8/4) 17
pattern 0(0/21)7
cognitive 0(0/21)7
critics 0 (0/21)7
core 0(0/6)3
propertypanels 40(4/6)3
ui 0(0/8)2
module 0 (0/2)2
model 3
notation 25(50/152)99
providers 1(2/162)23
uml 9 (10/106) 16
java 0 (0/4)4
ui 4
util 17(2/10)41
osdep 2
logging 5
language 0(0/2)1
ui 0(0/2)1
model 93(38/3)1518
mdr 89(59/7)1040
sequence2 0(...
diagram 34 (28/54)36
i18n 136
transformer 67 (2/1)2
taskmgmt 1
kernel 32 (63/132)143
profile 11(23/19)125
internal 8(8/97)19
ocl 34(26/51)10
uml14 5 (2/40)4
init 1(1/105)1
ocl 20(1/4)2
persistence 31 (145/326)37
configuration 100(29/0)87
swingext 100 (1/0)22
moduleloader 56(9/7)7

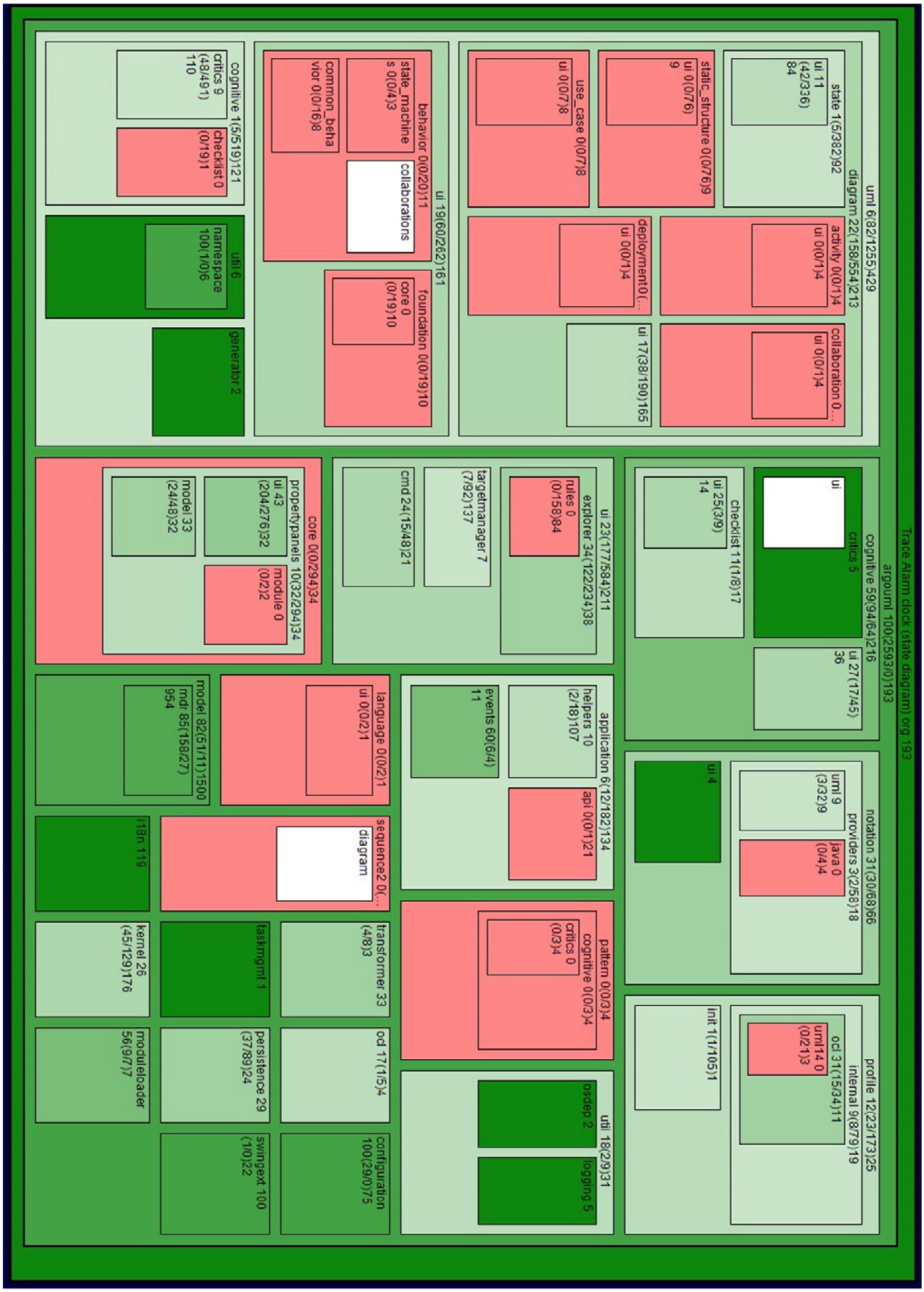
Trace Printing org 97
argouml 100(2960/0)97
uml 5(76/1380)462
diagram 24(187/594)305
state 1(2/246)61
ui 8 (21/246) 61
static_structure 0(0/126)61
ui 0 (0/126)61
use_case 0(0/127)41
ui 2 (3/127)41
deployment 0(...
ui 0(0/1)4
activity 0(0/1)4
ui 0(0/1)4
collaboration 0...
ui 0(0/1)4
ui 37(184/314) 283
ui 20(43/176)103
behavior 0(0/2)11
state_machine s 0(0/4)3
collaborations
common_beha vior 0(0/16)8
foundation 0(0/19)4
core 0 (0/19)4
cognitive 1(5/672)120
critics 12 (86/651) 113
checklist 0 (0/19)1
util
namespace
generator 2
cognitive 58(91/67)202
critics 5
ui
ui 26(16/45) 32
checklist 13(1/7)13
ui 27(3/8) 10
ui 21(121/463)147
explorer 36(101/183)29
rules 0 (0/153)84
targetmanager 12 (3/23)79
cmd 19(19/80)20
core 0(0/6)3
propertypanels 40(4/6)3
ui 0(0/8)2
model 3
module 0 (0/2)2
application 7(12/172)153
helpers 18(2/9) 117
events 73(8/3) 15
api 0(0/1)26
language 0(0/2)1
ui 0(0/2)1
model 93(37/3)1514
mdr 89(55/7)1037
sequence2 0(...
diagram 34 (28/54)36
i18n 128
notation 25(50/152)99
providers 1(2/162)28
uml 9 (10/106) 16
java 0 (0/4)4
ui 4
pattern 0(0/21)7
cognitive 0(0/21)7
critics 0 (0/21)7
transformer 67 (2/1)2
taskmgmt 1
kernel 31 (57/129)131
profile 11(23/191)25
internal 8(8/97)19
ocl 34(26/51)10
uml14 5 (2/40)4
init 1(1/105)1
util 17(2/10)38
osdep 2
logging 5
ocl 20(1/4)2
persistence 31 (135/304)21
moduleloader 56(9/7)7
configuration 100(29/0)82
swingext 100 (1/0)22

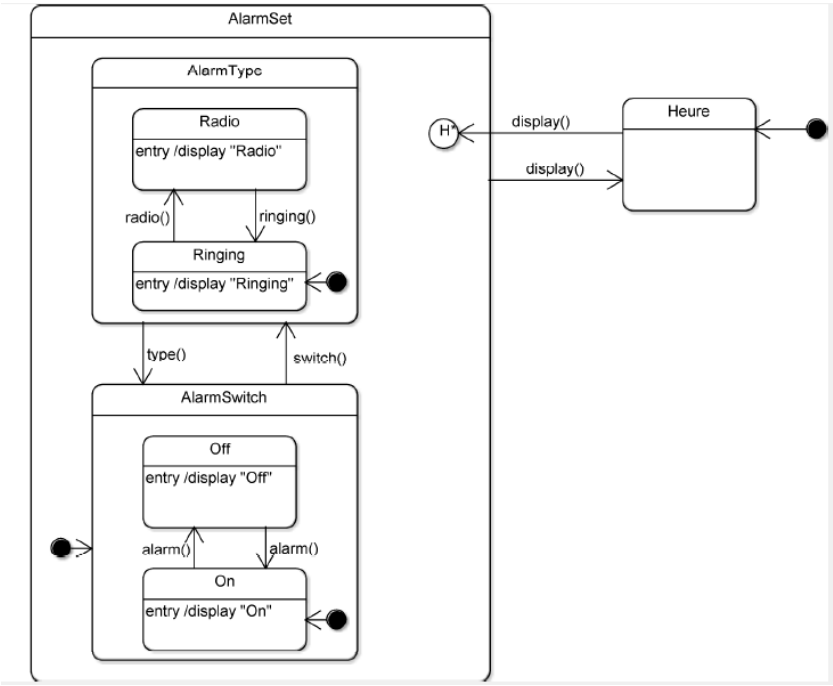
Trace Press agency (usecase diagram) org 205
argouml 100(2360)0)205
uml 6(65/1045)441
diagram 29(210/511)217
state 0(0/1)4
ui 0(0/1)4
activity 0(0/1)4
ui 0(0/1)4
collaboration 0...
ui 0(0/1)4
static_structure 0(0/76)9
ui 0(0/76)
9
deployment 0(...
ui 0(0/1)4
use_case 0(1/284)54
ui 6
(15/245)
52
ui 36(168/294)
229
ui 25(81/241)165
behavior
state_machine
s
collaborations
common_beha
vior
foundation 0(0/19)10
core 0
(0/19)10
cognitive 1(2/353)106
critics 7
(25/322)95
checklist 0
(0/19)1
util 6
namespace
100(1/0)6
generator 100
(2/0)3
cognitive 60(87/58)196
critics 4
ui
ui 27(17/45)
34
checklist 10(1/9)18
ui 23(3/10)
15
ui 24(186/576)216
explorer 37(130/219)37
rules 0
(0/158)84
targetmanager 8
(9/98)139
cmd 24(15/48)21
core 0(0/178)28
propertypanels 13(27/178)28
ui 46
(149/178)25
module 0
(0/2)2
model 36
(14/25)28
application 6(12/174)109
helpers 18(2/9)
80
api 0(0/1)21
events 58(7/5)
13
language 0(0/2)1
ui 0(0/2)1
model 78(43/12)1215
mdr 83(96/20)733
notation 35(31/57)81
providers 0(0/48)20
uml 11
(4/32)13
java 0
(0/4)4
ui 4
pattern 0(0/3)4
cognitive 0(0/3)4
critics 0
(0/3)4
sequence2 0(...
diagram
i18n 94
transformer 33
(4/8)3
taskmgmt 1
kernel 29
(53/129)164
profile 11(23/190)23
internal 8(8/96)18
ocl 34(26/51)11
uml 14 5
(2/40)4
init 1(1/105)1
util 38(5/8)31
osdep 2
logging 5
ocl 17(1/5)4
persistence 31
(37/82)24
moduleloader
56(9/7)7
configuration
100(29/0)77
swingext 100
(1/0)26

Trace Object Manager (sequence diagram).org 172
argouml 100(2145/0)172
uml 6(5147/42)455
diagram 12(36/272)239
state 0(0/1)4
ui 0(0/1)4
activity 0(0/1)4
ui 0(0/1)4
state_machine 1
collaborations 1
common_behavior
behavior 1
ui 21(61/228)145
use_case 0(0/7)7
ui 0(0/7)7
deployment 0(...
ui 0(0/1)4
static_structure 0(0/76)88
ui 0(0/76)
collaboration 0...
ui 0(0/1)4
ui 21(49/184)156
foundation 0(0/19)10
core 0
(0/19)10
generator 2
util 6
namespace
100(1/0)6
cognitive 1(2/291)106
critics 5
(15/260)95
checklist 0
(0/19)1
cognitive 61(87/56)188
critics 4
ui
ui 27(17/45)
33
checklist 13(1/7)15
ui 27(3/8)
12
ui 25(167/498)177
explorer 39(115/183)34
rules 0
(0/149)84
targetmanager 7
(6/86)109
cmd 24(15/48)21
application 6(12/173)93
helpers 18(2/9)
65
events 60(6/4)
12
api 0(0/1)21
pattern 0(0/3)4
cognitive 0(0/3)4
critics 0
(0/3)4
notation 26(24/67)63
providers 0(0/53)16
uml 10
(4/36)8
java 0
(0/4)4
ui 4
profile 12(23/173)23
internal 9(8/79)18
ocl 31(15/34)11
uml14 0
(0/21)3
init 1(1/105)1
util 22(2/7)26
osdep 2
logging 5
core 0(0/198)29
propertypanels 12(27/198)29
ui 45
(167/205)26
model 44
(14/18)28
module 0
(0/2)2
language 0(0/2)1
ui 0(0/2)1
model 82(46/10)1053
mdr 84(102/20)
sequence2 0(...
diagram 25
(64/192)42
i18n 79
transformer 67
(2/1)2
taskmgmt 1
kernel 28
(43/112)150
ocl 17(1/5)4
persistence 32
(37/79)24
moduleloader
56(9/7)7
configuration
100(29/0)75
swingext 100
(1/0)21

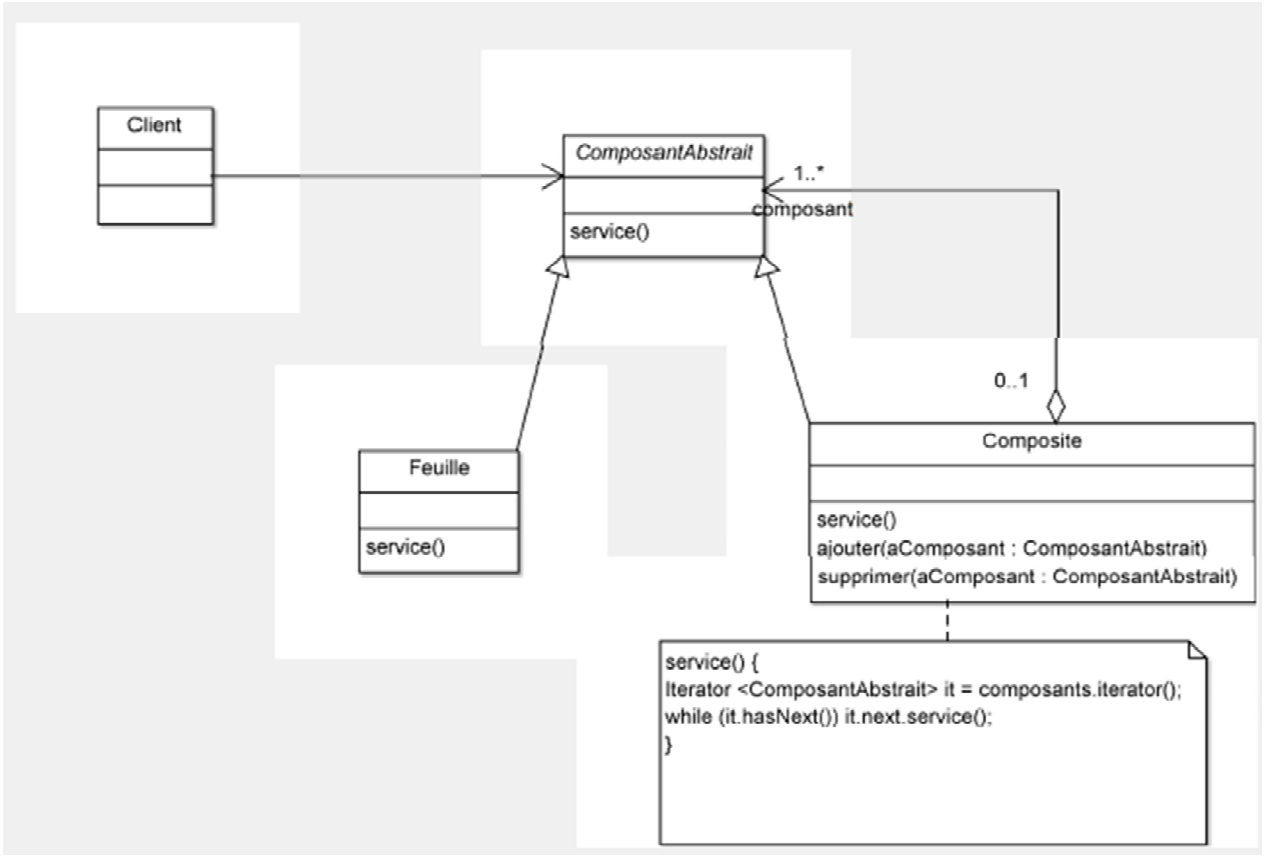
Trace Correction todos org 200
argouml 100(2958/0)200
uml 7(89/1199)562
diagram 35(246/465)315
state 0(0/1)4
ui 0(0/1)4
activity 0(0/1)4
ui 0(0/1)4
collaboration 0...
ui 0(0/1)4
static_structure 0(1/286)89
ui 1
(2/253)87
deployment 0(...
ui 0(0/1)4
ui 38(206/332)
287
use_case 0(0/7)9
ui 0(0/7)9
ui 20(59/234)147
behavior
state_machine
s
collaborations
common_beha
vior
foundation 0(0/19)10
core 0
(0/19)10
cognitive 1(8/572)166
critics 13
(83/540)
151
checklist 0
(0/19)1
util 6
namespace
100(1/0)6
generator 2
cognitive 56(170/133)288
critics 0(0/28)26
ui 40
(10/15)8
ui 19(31/135)
69
checklist 11(1/8)18
ui 25(3/9)
15
ui 23(203/671)213
explorer 34(115/224)39
rules 0
(0/148)84
targetmanager 7
(7/90)121
cmd 24(15/48)21
core 0(0/146)17
propertypanels 16(27/146)17
ui 44
(110/142)14
module 0
(0/2)2
model 34
(15/29)28
application 7(12/172)100
helpers 18(2/9)
63
api 0(0/1)25
events 67(8/4)
17
pattern 0(0/22)7
cognitive 0(0/22)7
critics 0
(0/22)7
notation 35(55/100)104
providers 0(0/103)30
uml 8
(6/73)19
java 0
(0/4)4
ui 4
profile 11(23/181)30
internal 8(8/87)22
ocl 28(13/34)11
uml14 0
(0/21)1
init 1(1/105)1
util 17(2/10)49
osdep 2
logging 5
language 0(0/2)1
model 84(48/9)1457
mdr 85(104/19)
934
sequence2 0(...
diagram
i18n 92
transformer 33
(4/8)3
ocl 17(1/5)4
configuration
100(29/0)86
taskmgmt 1
kernel 28
(66/166)240
persistence 35
(157/288)37
moduleloader
56(9/7)7
swingext 100
(1/0)27

Trace Composite Design Pattern (class diagram) org 236
argouml 100(2479/0)236
uml 8(95/1156)446
diagram 32(253/529)214
state 0(0/1)4
ui 0(0/1)4
activity 0(0/1)4
ui 0(0/1)4
static_structure 1(3/343)90
ui 3
(10/300)
89
deployment 0(...
ui 0(0/1)4
use_case 0(0/7)7
ui 0(0/7)7
collaboration 0...
ui 0(0/1)4
ui 39(221/348)
227
ui 21(57/209)149
behavior
state_machine
s
collaborations
common_beha
vior
foundation 0(0/19)11
core 0
(0/19)11
cognitive 1(4/490)144
critics 13
(67/461)
133
checklist 0
(0/19)1
util 0(0/2)8
namespace
100(1/0)6
generator 2
cognitive 56(93/73)213
critics 6
ui
ui 27(17/47)
36
checklist 11(1/8)16
ui 25(3/9)
13
ui 25(173/527)197
explorer 39(119/185)33
rules 0
(0/148)84
targetmanager 8
(7/85)116
cmd 24(15/48)21
core 0(0/321)60
propertypanels 8(29/321)60
ui 47
(243/279)57
module 0
(0/2)2
model 30
(29/69)30
notation 29(44/106)79
providers 0(0/106)29
uml 8
(7/82)20
java 0
(0/4)4
ui 4
application 6(12/174)89
helpers 18(2/9)
61
api 0(0/1)20
events 58(7/5)
13
pattern 0(0/21)7
cognitive 0(0/21)7
critics 0
(0/21)7
profile 11(23/180)24
internal 9(8/86)19
od 31(15/34)11
uml14 0
(0/21)3
init 1(1/105)1
util 68(15/7)33
osdep 2
logging 5
language 0(0/2)1
ui 0(0/2)1
model 81(73/17)1525
mdr 75(136/45)
967
sequence2 0(...
diagram
i18n 82
transformer 67
(2/1)2
taskmgmt 1
kernel 25
(43/132)165
ocl 17(1/5)4
persistence 30
(36/85)24
moduleloader
56(9/7)7
configuration
100(29/0)75
swingext 100
(1/0)22

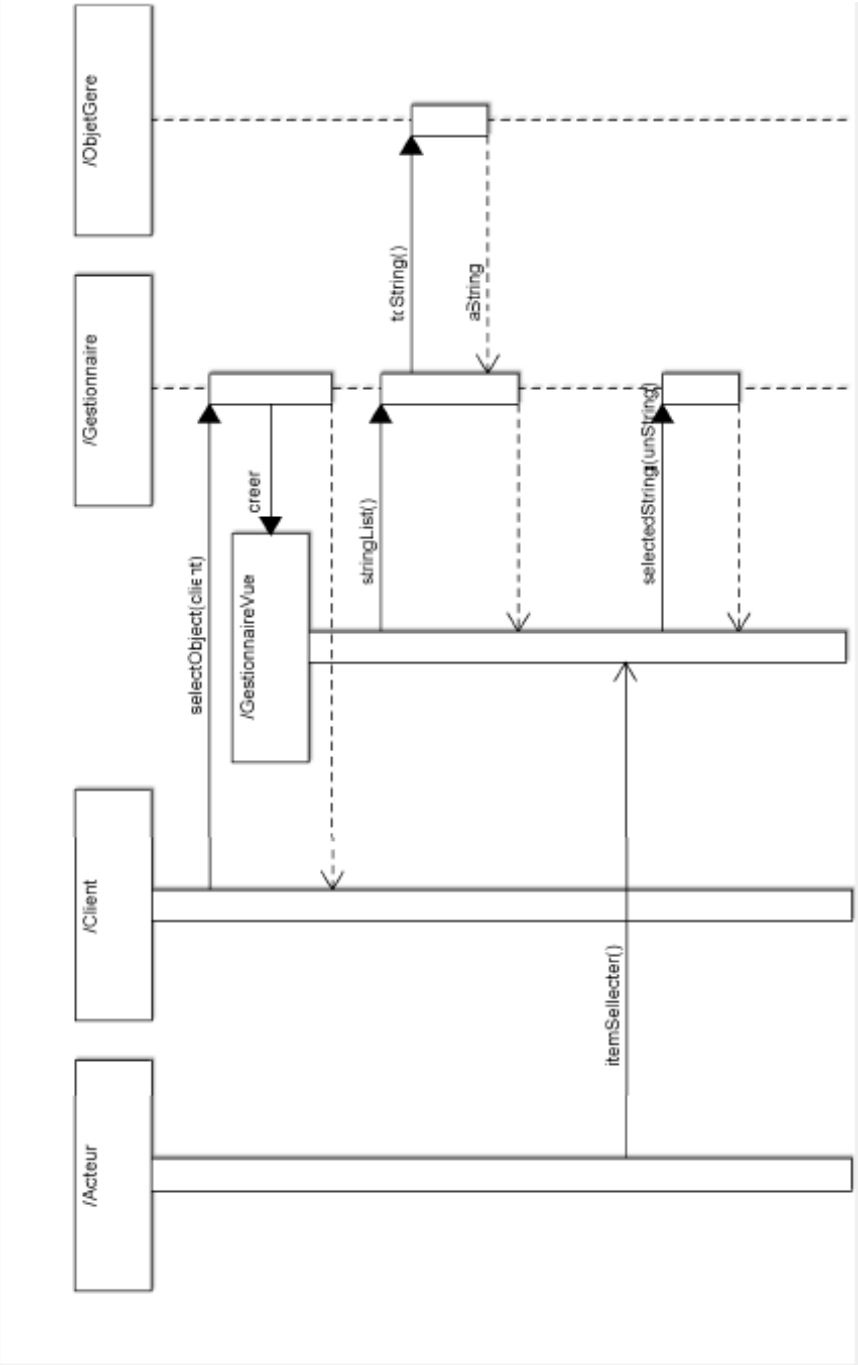

Trace Alarm clock (state diagram) org 193
argouml 100(2593/0)193
uml 6(821255)429
diagram 22(158/554)213
state 1(5/382)92
ui 11 (42/336) 84
activity 0(0/1)4
ui 0(0/1)4
static_structure 0(0/76)9
ui 0(0/76) 9
collaboration 0...
ui 0(0/1)4
use_case 0(0/7)8
ui 0(0/7)8
deployment0(...
ui 0(0/1)4
ui 17(38/190)165
ui 19(60/262)161
behavior 0(0/20)11
state_machine s 0(0/4)3
collaborations
common_beha vior 0(0/16)8
foundation 0(0/19)10
core 0 (0/19)10
cognitive 1(5/519)121
critics 9 (48/491) 110
checklist 0 (0/19)1
util 6
namespace 100(1/0)6
generator 2
cognitive 59(94/64)216
critics 5
ui
ui 27(17/45) 36
checklist 11(1/8)17
ui 25(3/9) 14
ui 23(177/584)211
explorer 34(122/234)38
rules 0 (0/158)84
targetmanager 7 (7/92)137
cmd 24(15/48)21
core 0(0/294)34
propertypanels 10(32/294)34
ui 43 (204/276)32
module 0 (0/2)2
model 33 (24/48)32
application 6(12/182)134
helpers 10 (2/18)107
events 60(6/4) 11
api 0(0/1)21
language 0(0/2)1
ui 0(0/2)1
model 82(51/11)1500
mdr 85(158/27) 954
notation 31(30/68)66
providers 3(2/58)18
uml 9 (3/32)9
java 0 (0/4)4
ui 4
pattern 0(0/3)4
cognitive 0(0/3)4
critics 0 (0/3)4
sequence2 0(...
diagram
i18n 119
profile 12(23/173)25
internal 9(8/79)19
ocl 31(15/34)11
uml14 0 (0/21)3
init 1(1/105)1
util 18(2/9)31
osdep 2
logging 5
transformer 33 (4/8)3
ocl 17(1/5)4
configuration 100(29/0)75
taskmgmt 1
persistence 29 (37/89)24
swingext 100 (1/0)22
kernel 26 (45/129)176
modulelcader 56(9/7)7



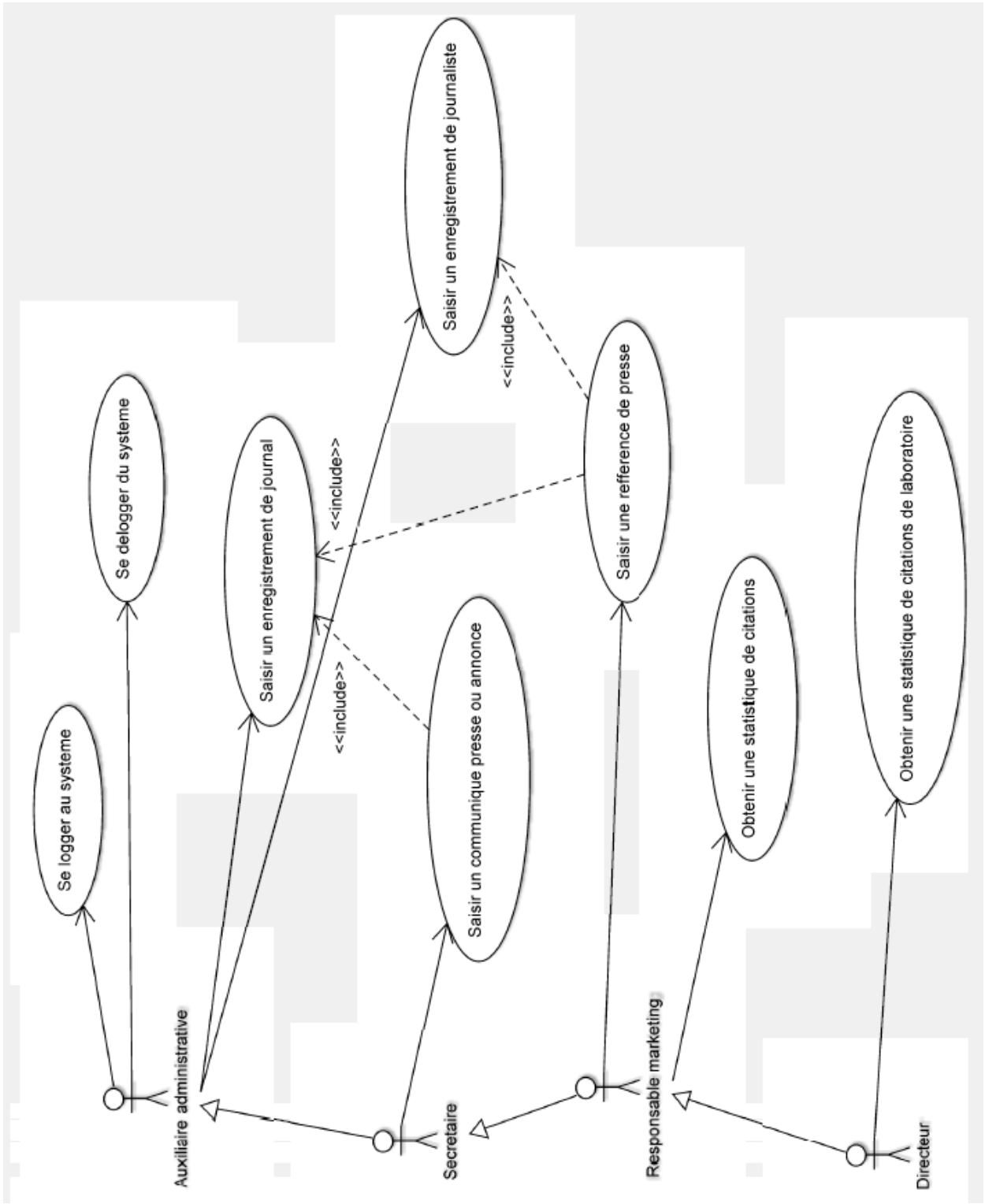
Alarm clock UML state diagram



Composite design pattern UML class diagram



Object Manager UML sequence diagram



Press Agency UML Use case