MAISA metrics tool: Maintenance manual

version 1.3

Lilli Nenonen
Juha Gustafsson

Helsinki December 13, 2001

UNIVERSITY OF HELSINKI
Department of Computer Science
MAISA project

Contact information

Jukka Paakki:	email `Jukka.Paakki@cs.helsinki.fi`
Inkeri Verkamo:	email `Inkeri.Verkamo@cs.helsinki.fi`
Juha Gustafsson:	email `Juha.Gustafsson@cs.helsinki.fi`
Lilli Nenonen:	email `Lilli.Nenonen@cs.helsinki.fi`

TEKES
Matti Sihto: email `Matti.Sihto@tekes.fi`

Kone
Salla Mannerjoki: email `Salla.Mannerjoki@kone.com`

Nokia Mobile Phones
Markku Ilvesmäki: email `Markku.Ilvesmaki@nmp.nokia.com`

Nokia Research Center
Anssi Karhinen: email `Anssi.Karhinen@nokia.com`

Space Systems Finland
Niklas Holsti: email `Niklas.Holsti@ssf.fi`

Contents

1 Overview

This is the first version of the maintenance manual for the MAISA metrics tool. The function of the manual is to give general idea how the system works, how it satisfies its requirements and how it can be modified. More detailed description of usage of MAISA classes can be found from the Javadoc comments of MAISA and formal definition of the file formats can be found in Appendix B in the user manual for MAISA. The purpose of this document is to extend them, by explaining things that go beyond the Javadoc comments which define the responsibilities of the classes and other low-level information.

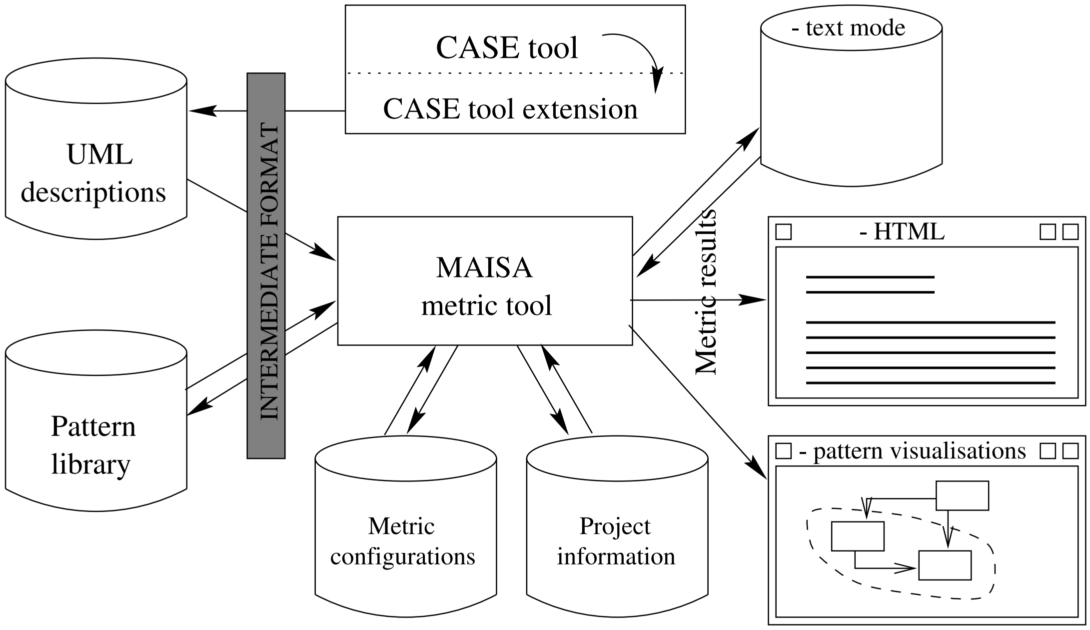

Figure 1: Architecture of MAISA.

Figure 1 presents the overall architecture of MAISA. The MAISA system consists of package `MAISA` which contains the graphical user interface; and its subpackages `MAISA`.datastruct which contains the data structures, and `MAISA.csp` which contains the pattern recognition system. Package `MAISA` forms the upper layer of the system while the others form the lower one. The packages are presented in Chapters 2 – 4. Use cases of MAISA are examined in Chapter 5. In Chapter 6 we describe how some points or hot spots of the tool can be modified or updated. Extending a CASE tool for MAISA is described separately in Chapter 7.

2 Package MAISA

Package `MAISA` contains merely the GUI dependent classes. The GUI of the metrics tool consists of five basic windows: a main window, a metrics window, a pattern library window, a history data window and a help window. The tool can be quitted only in the main window. The main program class is `MAISA.MAISA_tool`. It is the heart of the tool, working as a *Mediator* for the basic windows of the tool. The windows communicate to the *Mediator* by sending messages, while the *Mediator* communicates to the windows by calling their methods. So class `MAISA.MAISA_tool` is also *Observer* of the events that the windows produce whenever they cannot perform the task themselves but need to delegate the task to another class or window. These events are called colleague events, to separate them from the GUI events (which are never passed to the *Mediator*, as they are listened by the window classes themselves). They are instances of `ColleagueEvent` class.

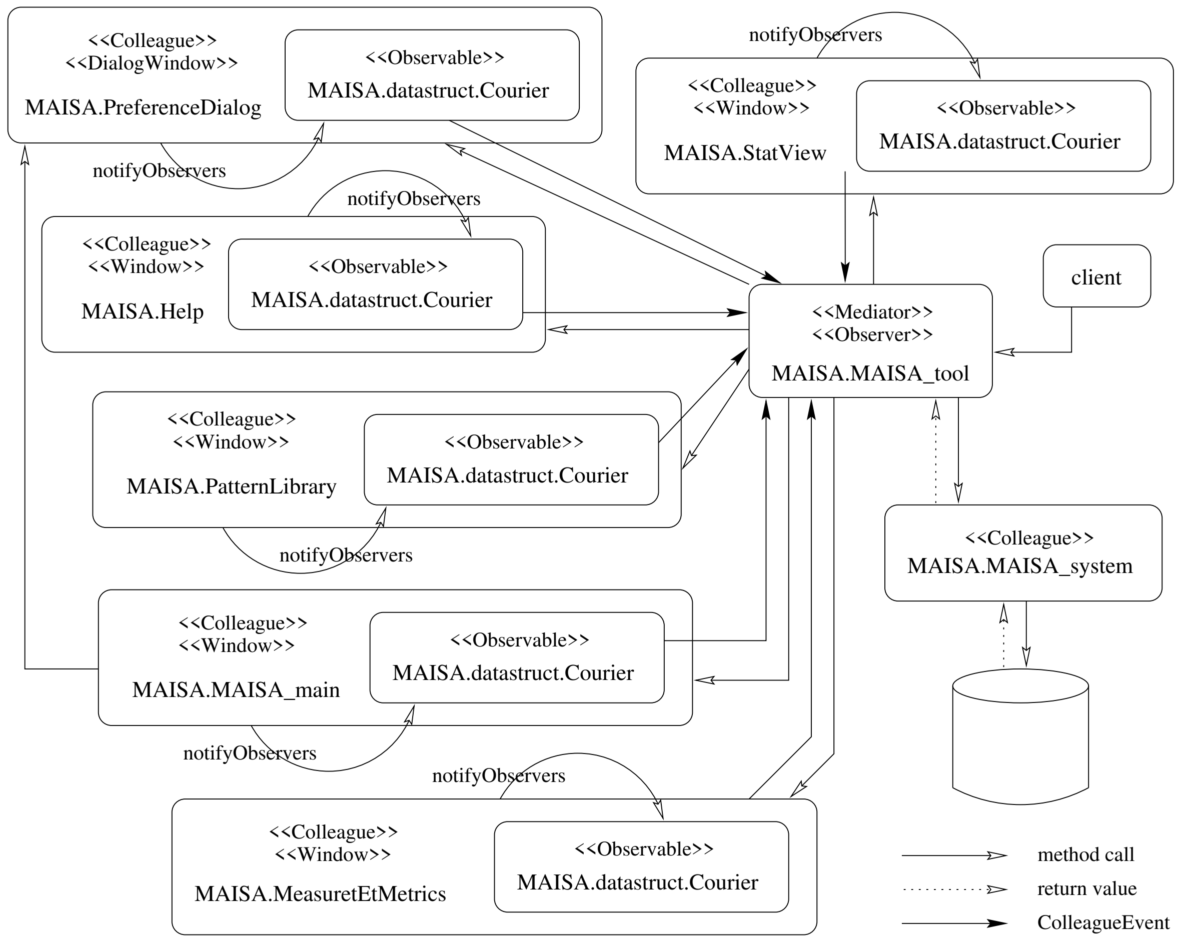

Figure 2: MAISA event handling model.

Figure 2 presents the event model of MAISA as collaboration diagram. The *Observer* pattern is implemented using the ready-made JDK interface `java.util.Observer` andthe class

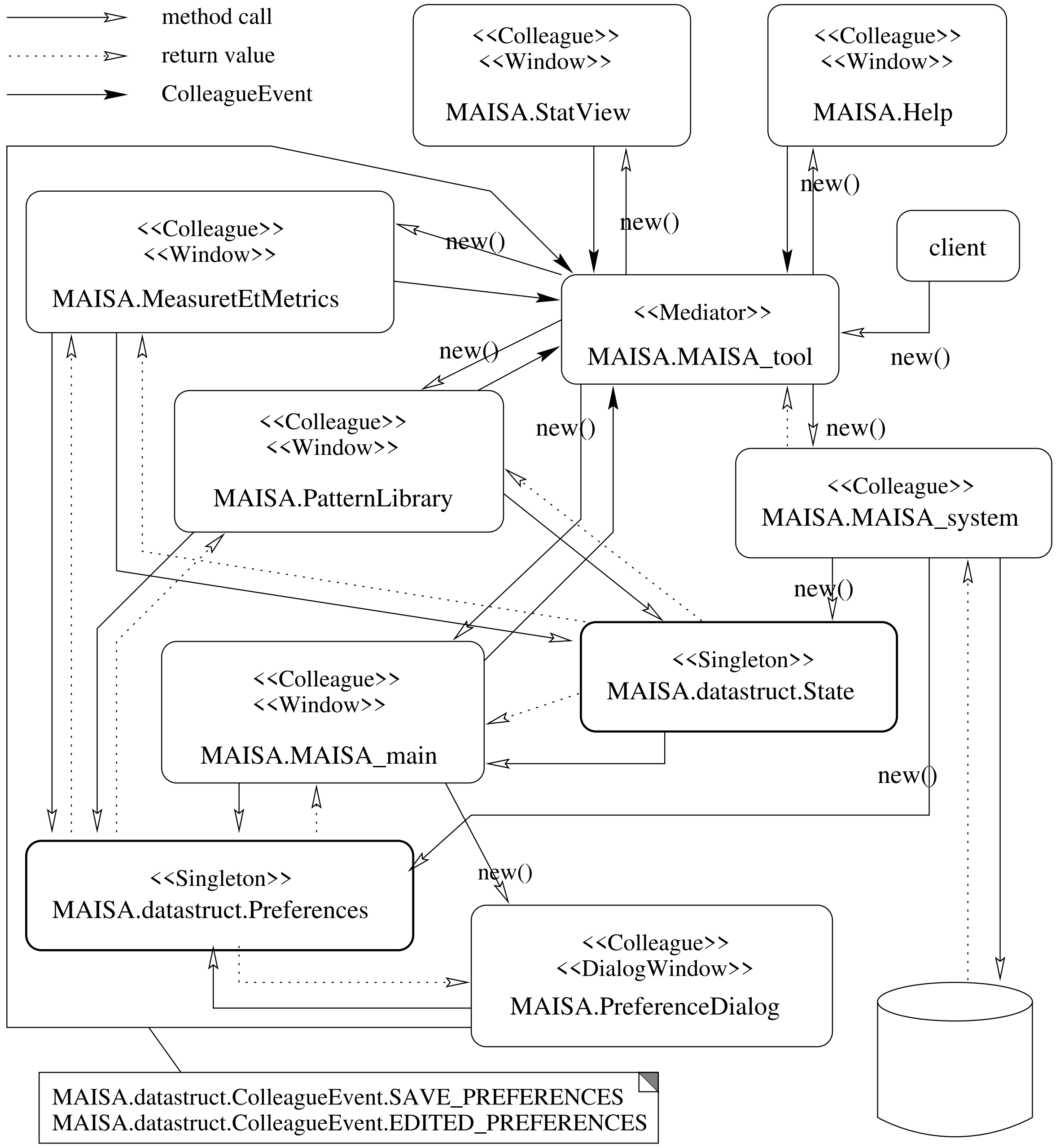

Figure 3: Program state model.

`java.util.Observable`. Because the window classes already extend `javax.swing.JPanel` (and thus cannot be a sublass of `java.util.Observable`, too) the events are transmitted by a courier class which extends `java.util.Observable`. Each window object has its own courier instance, listened by `MAISA.MAISA_tool` which implements `java.util.Observer` and transmits the events between the windows as their *Mediator*.

The collaboration diagram in Figure 3 presents the same classes from the point of view of the *Mediator* pattern. The principles of design of the *Mediator* structure is that the windows know as little as possible about files, file names and other windows, and that the *Mediator* class is as lightweight as possible, knowing nothing about IO, let alone file names. This is why – regardless of how well these principles have become materialized – a caretaker class `MAISA.MAISA_system` is needed. In general, its responsibilities are to know everything that the windows and the *Mediator* do not need to know, namely everything that concerns opening and deleting files, knowing the default file names, keeping a list of unsaved files and reading and writing the program state and other objects in the disk. However, the caretaker class is not a *Colleague*, i.e. it does not know about the *Mediator* class or send it any events – its services are just used by `MAISA.MAISA_tool` when needed.

When a *Mediator* object is created by the main program, it creates all the window objects and the caretaker object. The contents of the windows are read by the caretaker, using `MAISA.Persistent` interface that is implemented by all the windows whose state is saved to the disk between the sessions. The contents of the windows are saved using the same interface when quitting the tool.

The contents of the windows are saved separately from the program state. The state of the program is presented by two classes during the session, both of them being *Singletons*, because the same program state must be available from every other class. `MAISA.datastruct.Preferences` contains the set of user-given preferences, or in other words, the part of the program state that is visible and editable by the user, while `MAISA.datastruct.State` is the other part of the state. Though, at the moment, `MAISA.datastruct.State` contains only the current working directory that is used by the file chooser dialogs. The program state is read from the disk when the tool is started and written to the disk when it is quitted by `MAISA.MAISA_system`. Figure 3 shows the classes that use the two classes.

The following figures are refinements of the previous ones. They present the window classes in more detail, showing which events they pass to the *Mediator*, which user preferences and data structures they use and which methods are called by the *Mediator* object. Those methods are called client methods, as they are called by the client of a window. The *Mediator* calls them in order to let the *Colleague* know that the colleque event has been handled and to return the possible answer to the request presented by the event. The data structures are discussed quite superficially, since they will be explained in the following Chapter. The window class is shown emphasized in the figures. For the sake of clarity, the description of the main window – which is by far the most complex window in the system – is divided into more than one figures. The arrows in the figures express the nature of the association. An arrow with a black tip is colleague event passing relationship. The passed events (event tags) are listed in a note sheet binded to the arrow (see e.g. Figure 4). An arrow with white tip is method call relationship, a dotted arrow with a white tip expresses that the called method returns a value (so the relationship is somewhat stronger and the method caller is more strongly dependent on the method callee). Finally, an arrow with a simple tip is an ordinary association, expressing that a class has a constant reference to another class. The roles that the classes have e.g. in certain design patterns are expressed as stereotypes.

Figure 4 presents the event model from the point of view of the main window, namely the class `MAISA.MAISA_main`. It uses four dialog windows: `PreferenceDialoq`, `PatternDialog`, `ThreadDialog` and `QuestionDialog`. `ThreadDialog` is for stopping metrics calculation threads. `QuestionDialog` is for asking the user a certain question and it returns the answer by calling the corresponding method of its parent of main window. The `PatternDialog` lists the recognized pattern instances of a new calculated the `MetricResult` to ask the user to confirm them, which is why the `PatternDialog` uses `MetricResult` and `PatternInstance` classes. `PreferenceDialog` is for editing the user preferences, in other words for updating the `Preferences` object. The `PreferenceDialog` sends a colleague event if the preferences are modified to inform other classes that are interested and to request to save the preferences to the disk (which is delegated to the caretaker). The main window uses all the boolean valued preferences because they can be updated also in the selectable items in its preference menu.

There is a good deal of colleague events sent from the main window, most of them concerning opening, closing and saving projects and diagrams. Those events are delegated to the caretaker object. In the end of the list (in Figure 4 on the arrow from the main wondow to the caretaker) there is window opening events, which are requests to open one of the other basic windows. These events are risen by clicking the window menu items and they are sent also by every other window. The last events in the list concern quitting the tool. They are sent because before the exit the caretaker object writes the program state and the window contents to the disk and checks if there are unsaved metrics results. This is why the main window cannot do the exit operation alone.

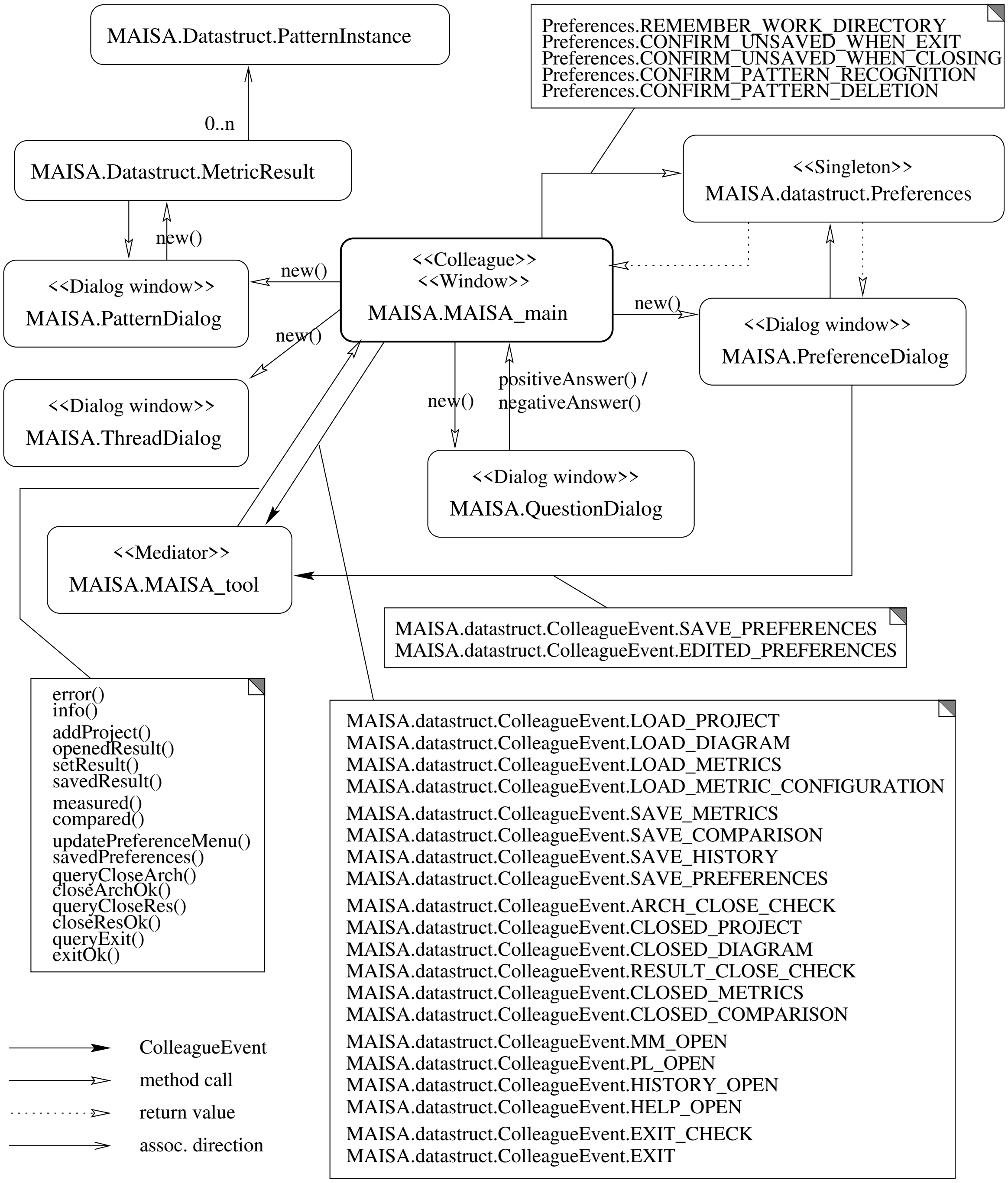

Figure 4: Main window, event model.

2.1 Main window

Figure 5 presents the data structures used by the main window, which are somewhat complex and as a whole cover a substantial part of the package `MAISA.datastruct`. The main window uses three special panels belonging to the package `MAISA`: `MAISA.PatternDrawArea` which visualizes the recognized patterns, and `MAISA.StatPanel` and `MAISA.StatPanel2`, which draw statistic diagrams (and calculate key figures) from certain calculation result. The former handles `MetricResults` while the latter handles `ComparisonResults`. They are drawn as a common object in the figure, because they are not visible at the same time but changed to each other on the fly, depending on which kind of result is selected. Both statistic panels use `MAISA.StatDrawPanel` to draw the actual diagrams. Figures 6 and 8 refine Figure 5 by showing the data structures used by the visualization and statistic panels.

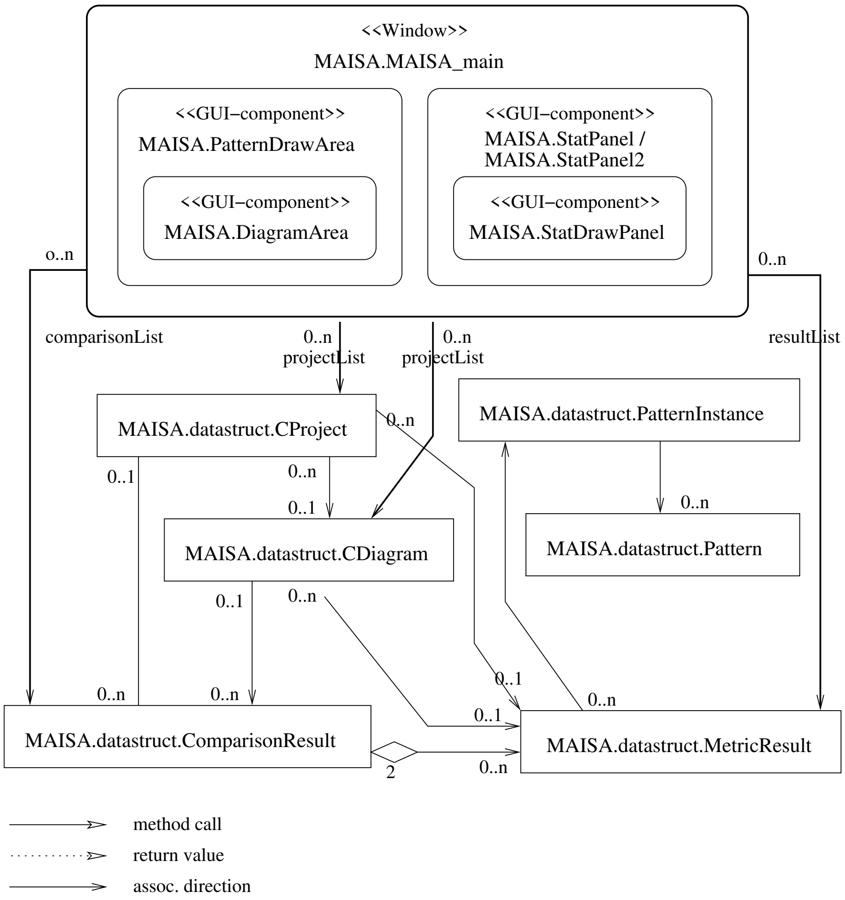

Figure 5: Main window, data structures.

The main window maintains a tree-shaped list of diagrams and diagram groups called projects. A diagram or a project contains a number of metrics results and comparison results. A comparison result consists of two metrics results. Each metrics result belongs to one project or diagram. The main window keeps a list of the metrics results and comparison results of the currently selected project or diagram. Either a metrics result or a comparison result can be selected at a time. So, when a metrics result is selected, the statistic panel shown in the window is a `MAISA.StatPanel`, and if a comparison result is selected, the statistic panel is a `MAISA.StatPanel2`. The two statistic panels could be refactored or parametrized so that it would not be necessary to change the panel on the fly.

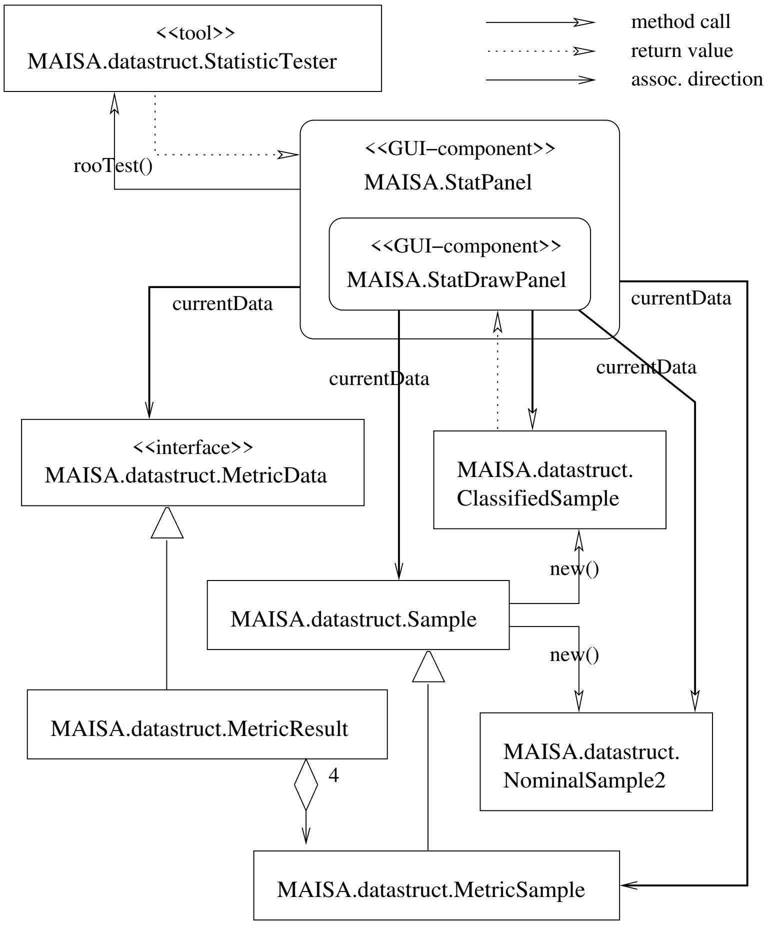

Figure 6: Statistics panel for metrics results, data structures.

What makes the things more complex is that the metrics data was not originally designed for statistical analysis. So, as shown in Figure 6, class `MAISA.datastruct.Sample` is added to the same data model to express the metrics data as an observation matrix. The metrics data consists of four matrixes: metrics values per classes, projects and diagrams and pattern recognition data. `MAISA.datastruct.Sample` is quite a general class. `MAISA.datastruct.MetricSample` is simply a sample with a timestamp. Also nominal and classified samples can be built up from a sample, as instances of `MAISA.datastruct.NominalSample2` and `MAISA.datastruct.ClassifiedSample`. The panel `MAISA.StatPanel` is responsible for listening the user events, while `MAISA.StatDrawPanel` is responsible for drawing the statistic diagram. `MAISA.StatPanel` uses `MAISA.datastruct.StatisticTester` to calculate statistic tests. The only test used by it is the correlation factor (roo) test.

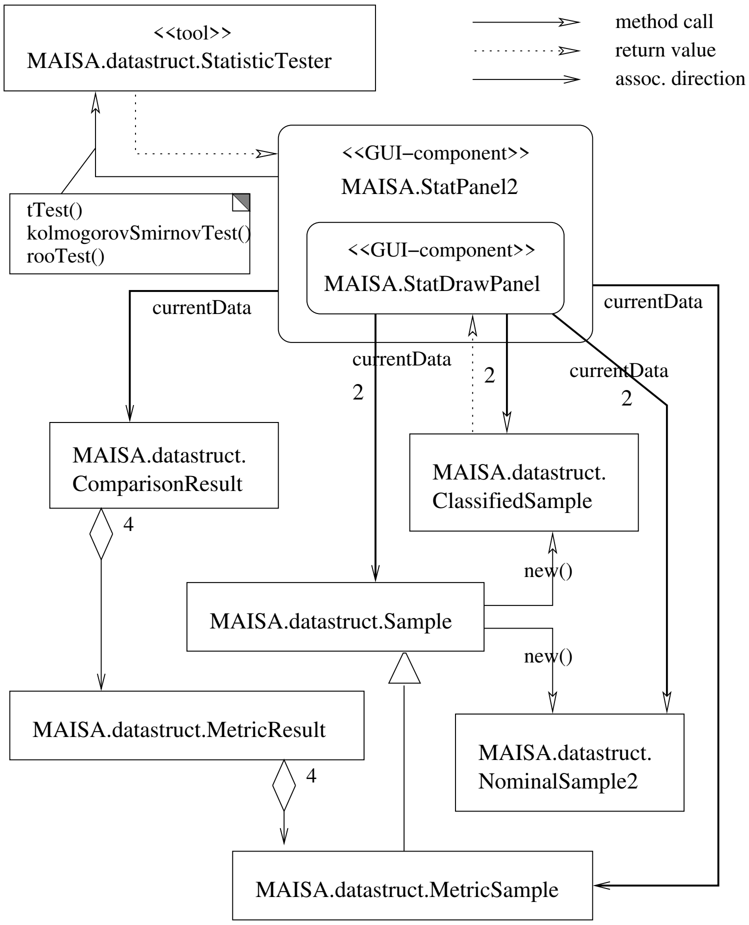

Figure 7: Statistics panel for comparison results, data structures.

Figure 7 shows the data structures used by `MAISA.StatPanel2`. The figure is very similar to the previous one. The only real difference is that `MAISA.StatPanel2` uses `MAISA.datastruct.ComparisonResult` that consists of four `MAISA.datastruct.MetricData` (in practise, `MAISA.datastruct.MetricResult`) objects, instead of using one `MAISA.datastruct.MetricData` object. Thus, there are two data structure references in most of the associations, instead of one. Both stat panels contain GUI components for bounding the sample data for the diagram. So, `MAISA.StatPanel2` contains two sets of those GUI components. This is the reason why the two panels are not refactored and why refactoring or parametrizing the panels beyound a common interface would have been difficult (though not impossible). `MAISA.StatPanel2` uses all three statistic tests available: correlation factor (roo) test, average value (or t-)test and Kolmogorov-Smirnovs similarity test.

In fact, there is a third panel to refactor beyound the same interface, namely `MAISA.StatPanelForHistoryData`. We come back to it in Chapter 2.4.

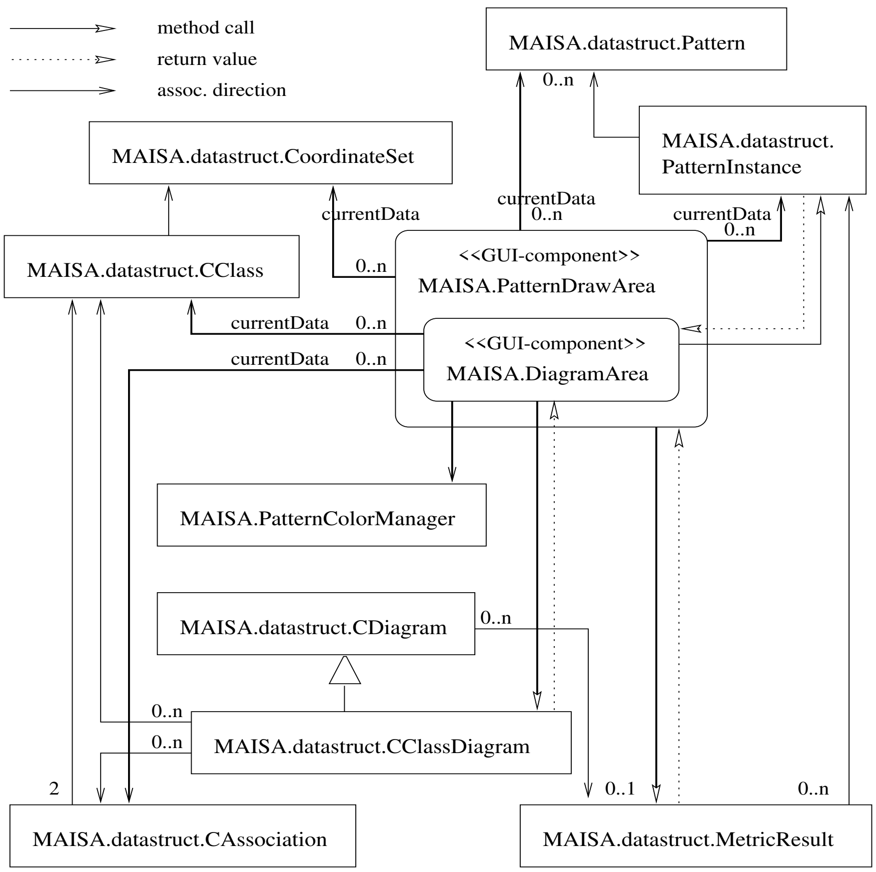

Figure 8: Pattern visualization panel, data structures.

Figure 8 presents the pattern visualization panel, namely the class `MAISA.PatternDrawArea`. The structure of it is quite similar to `MAISA.StatPanel`: the panel listens to the user events and delegates the real drawing of the diagram to another specialized panel, `MAISA.DiagramArea`. `MAISA.PatternDrawArea` maintains lists of the patterns of the diagram, showing them in the user interface. When a user selects a pattern or a class, the panel asks `MAISA.DiagramArea` to emphasize it in the drawn class diagram skeleton. `MAISA.DiagramArea` keeps lists of the elements of the class diagram and uses `MAISA.datastruct.PatternColorManager` to define the emphasizing colour and `MAISA.datastructCoordinateSet` to define the positions of the classes in the diagram. Each class has its own set of coordinates. The `MAISA.datastruct.Pattern` class represents an abstract design pattern, while `Maisa.datastruct.PatternInstance` represents a concrete pattern instance. A pattern instance knows its pattern, but the pattern is ignorant of its instances.

2.2 Metrics window

Figure 9 presents the metrics window, namely the class `MAISA.MeasuresEtMetrics`. The function of the window is to read in the user defined metrics configuration, i.e. the set of metrics that are calculated by the tool (which is typically updated a couple of times during the session), containing the design patterns that are to be recognized. The contents of the window is read in a `MAISA.datastruct.MetricConfiguration` object. The colleague events sent by the metrics window concern saving and loading the current metrics configuration, in addition to the common window opening events. The client methods used by the *Mediator* are requests to get the current metrics configuration or concern modifications to the pattern library. The windows are kept consistent with the pattern library, so when the library is modified, information of it is delivered to every window that handles the library in some way.

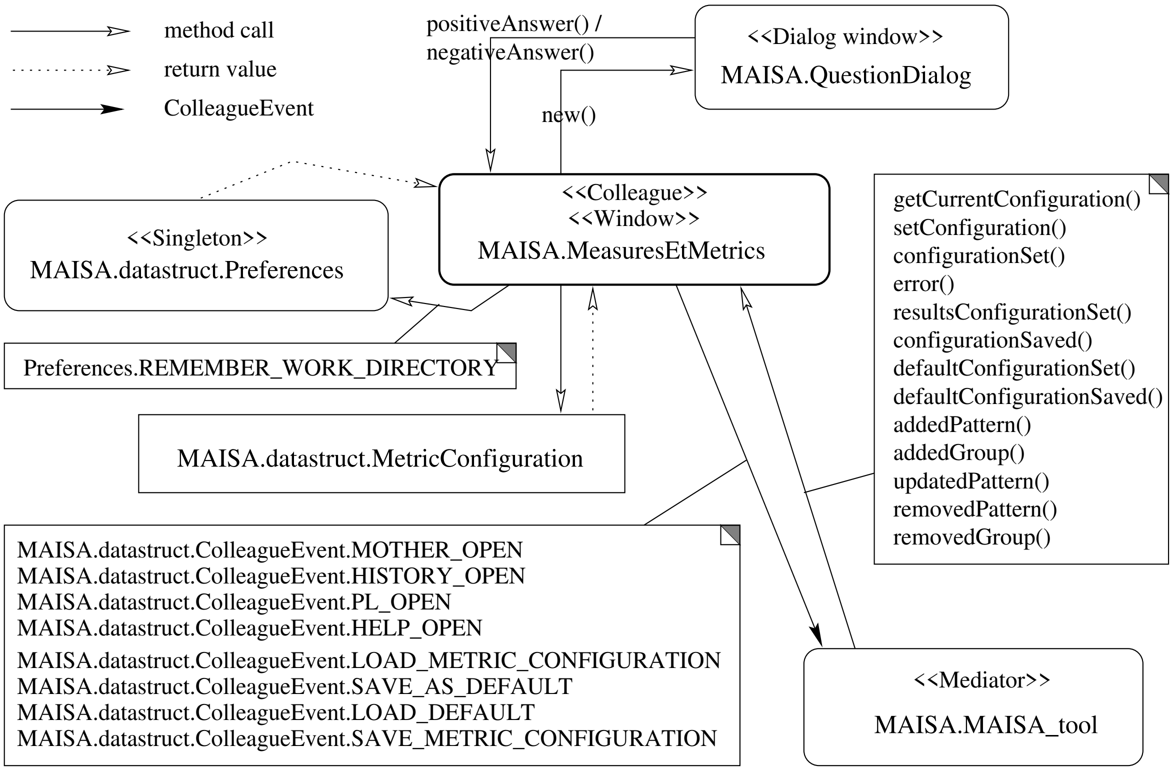

Figure 9: Metrics window.

The reason why the metrics window does not access the pattern class `MAISA.datastruct.Pattern` is that the patterns are referenced as mere strings in the cases it is necessary to handle the real pattern objects. The corresponding pattern object can be queried from the pattern class using a static method, so the patterns can be referenced as strings when it is not necessary to use the pattern objects. Unlike e.g for the `MAISA.PatternDrawArea`, mentioned above, which has to handle also the roles of the pattern and thus has to reference the patterns by pattern objects, the metrics window knows the mere pattern names as strings but does not reference the patterns by other ways. The reason for this kind of reference structure is threading. There is more than one window (which contain one or more threads) who has a tree-shaped list of the patterns, and it would not be necessary to copy the library for every window, because everyting most of the library users need is the mere pattern names.

Because the metrics window can open a file chooser dialog, it uses the preferences object to find out the most recent user directory to set it in the file chooser dialog. If the user is trying to save the metrics configuration in a file that already exists, he is asked to confirm it in a `MAISA.QuestionDialog`.

2.3 Pattern library window

Figure 10 presents the pattern library window, namely the class `MAISA.PatternLibrary`. Its responsibility is to update the pattern library. In addition to the window opening events, the colleague events sent by the window concern modifying the library. The file operations are delegated by the *Mediator* to the caretaker object. If a pattern or pattern group is removed or added to the library, the information is passed to the metrics window, too. It uses a question dialog object in the same way as the metrics window, to ask confirmation to overwriting already existing files. `MAISA.StringDialog` is used when the user adds a new pattern group, to ask the user the name of the new group. It returns a user given string by calling a method of the pattern library.

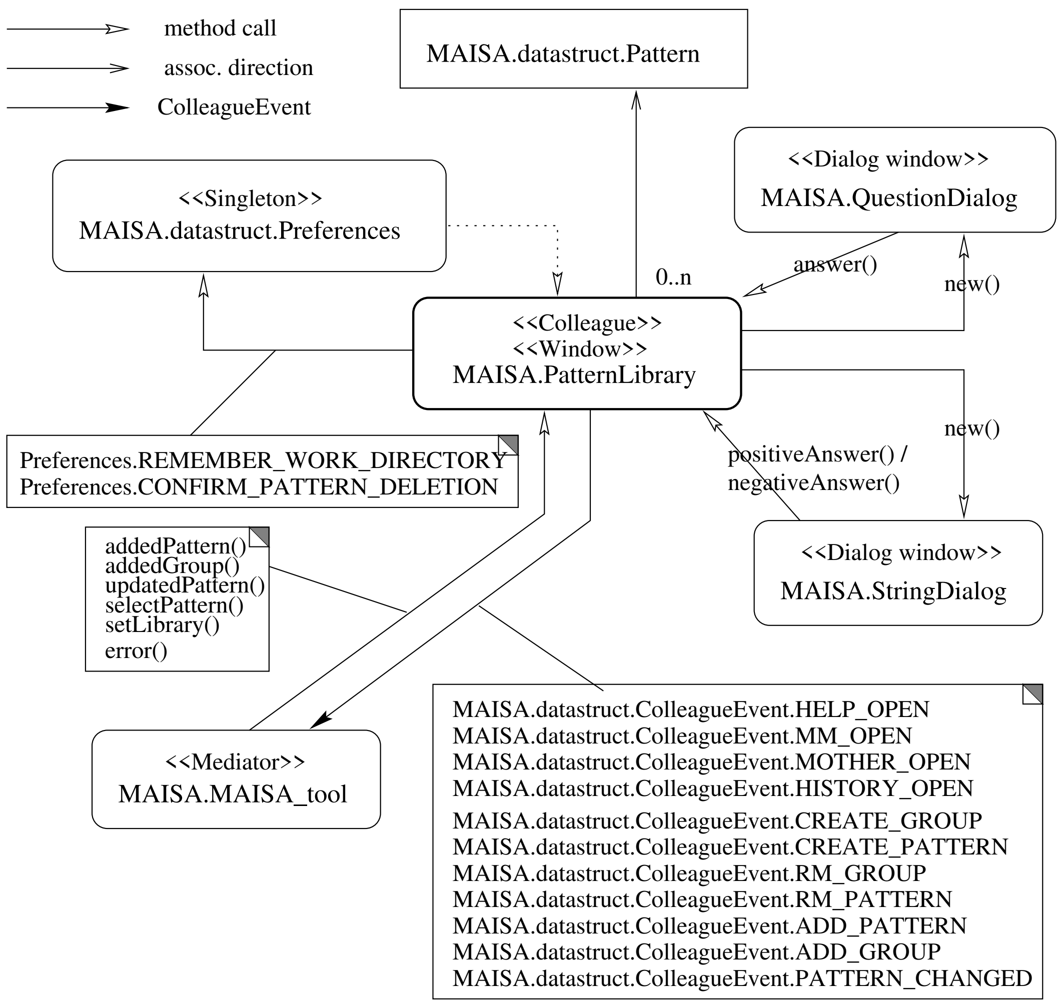

Figure 10: Pattern window.

2.4 History data window

Figure 11 represents the history data window, namely the class `MAISA.StatView`, cf. `MAISA.StatPanel` in Figure 6. The history data window is actually a `MAISA.StatPanelForHistoryData` plugged in its own window, using `MAISA.datastruct.HistoryData` as its data structure instead of `MAISA.datastruct.MetricResult`. When used by the history data window, the statistics panel uses one statistic diagram type more than is used in the main window, because the history data object contains the metrics data in time dimension, while a metrics result object is bound to a certain moment of time.

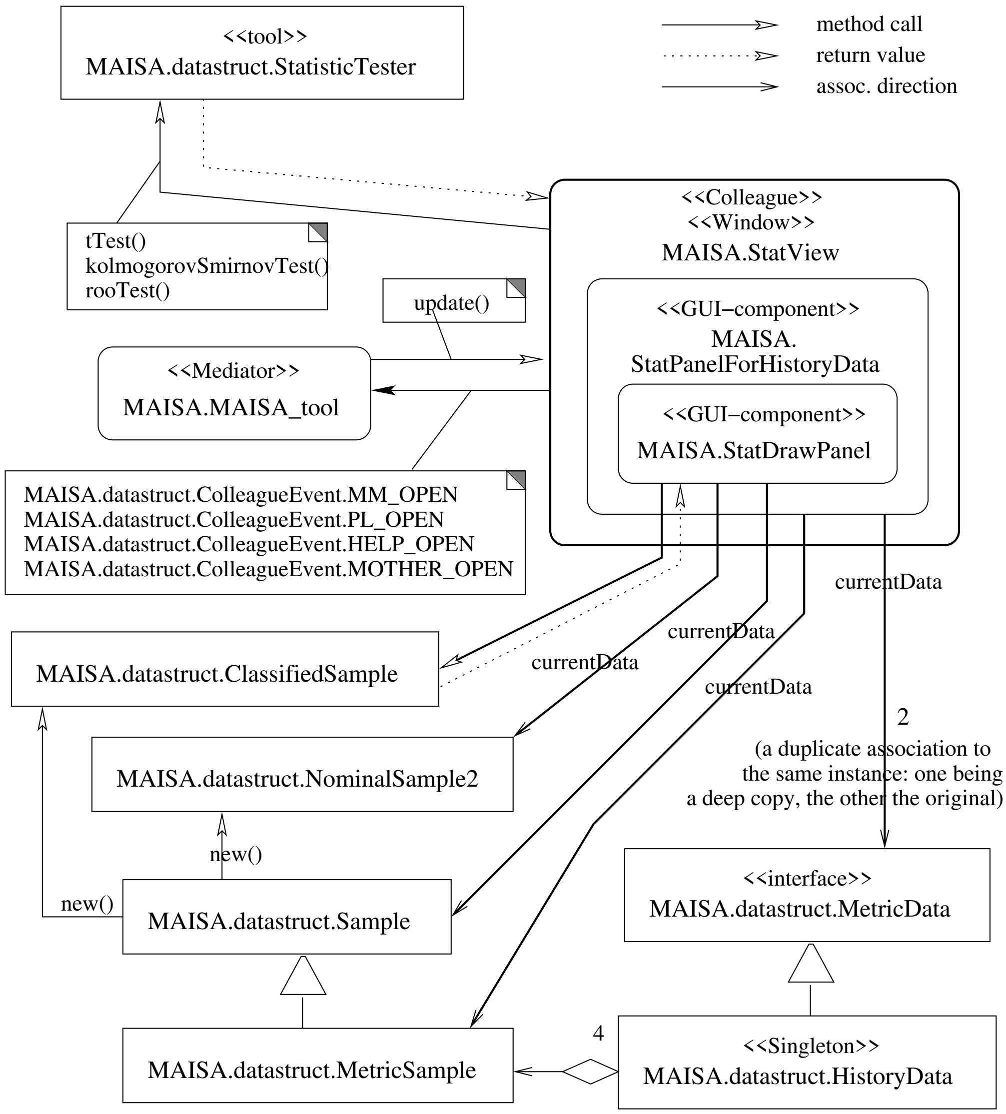

Figure 11: History data window.

The history data window is fairly simple to use by its clients: the only thing the client has to do is to set history data object in the window. Besides, the only colleague events sent by the window are the common window opening requests. `MAISA.StatPanelForHistoryData` uses `MAISA.datastruct.StatisticTester` to calculate statistic tests. It uses all three statistic tests available: correlation factor (roo) test, average value (or t-)test and Kolmogorov-Smirnovs similarity test.

`MAISA.StatPanelForHistoryData` belongs (together with `MAISA.StatPanel` and `MAISA.StatPanel2`) to the group of three statistic panels, each looking and functioning rather (but not quite) similar. It would pay to refactor the similarities of the panels beyound a common interface or parametrize the panels in some way. The data structure interface `MAISA.datastruct.MetricData` would make it possible to combine the statistic panels. However:

- `MAISA.StatPanel` handles `MAISA.datastruct.MetricResult` objects and maintains one reference to the current object.
- `MAISA.StatPanel2` handles `MAISA.datastruct.ComparisonResult` objects and thus maintains two references to the current sample (because a comparison result contains two metrics results), each of which has to be bound separately).
- `MAISA.StatPanelForHistoryData` handles `MAISA.datastruct.HistoryData` and has one more diagram type that the other two ones, namely time dimension diagram. The history data matrixes also have one extra column (timestamp) because of that.

2.5 Help window

Figure 12 represents the help window. It is as simple to use for the client as the history data window: there is only one client method offered, setting the user manual data in the window. The only colleague events are the window opening requests.

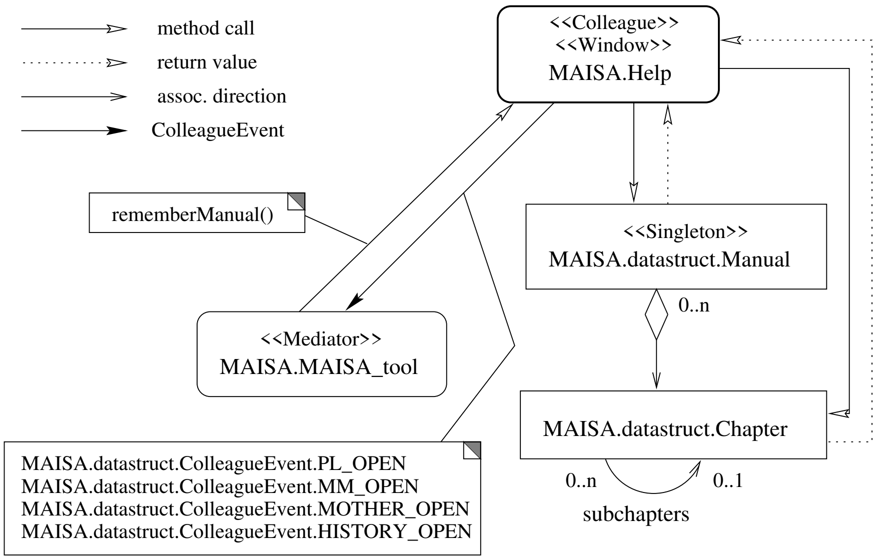

Figure 12: Help window.

The user manual is represented by a `MAISA.datastruct.Manual` object, which, for one, consists of `MAISA.datastruct.Chapter` objects. They represent the chapters of the manual, and contain the chapter text and pictures. The chapter structure is recursive, so a chapter object can consist of subchapters.

3 Package MAISA.datastruct

Package `MAISA.datastruct` contains all the user interface-independent classes excluding the pattern recognition system that is located in package `MAISA.csp`. Actually, the package had better be divided into more than one package, so that statistic classes would form one package and the classes representing architectural elements would form another.

3.1 Measurement and MAISA

The key concept in MAISA is to calculate both traditional metrics (such as the cyclomatic complexity of a state diagram) and pattern based metrics (such as the relative number of classes) as early as possible. In practice this means using design diagrams, as they contain the minimum amount of information on which to base these calculations (See [GuN1999]).

The MAISA tool produces three types of results:

- Simple metrics (e.g., number of methods).
- Pattern mining results (e.g., the number of Singletons present and the respective instances).
- Pattern-based metrics (e.g., relative number of classes).

One advantage of using simple metrics is that many of them can be acccurately calculated already in the design phase. From the moment we have drawn our first class, we are ready to calculate the number of classes. Additionally, most simple metrics don't require any extensive computation, so we can get lots of useful information for the user with relatively little effort.

With pattern mining the situation is slightly different. In principle, we can search for patterns from any diagram (however rough they might be). In practice we need either extremely detailed diagrams or some substantial help from the user, if we want this search to be successful. Another problem is that the patterns themselves are often quite vaguely defined. For more details about pattern mining in MAISA see [NeG2000].

3.2 Generating and exporting the diagrams

As the data in CASE tools has clearly not been designed to be easily exported or even uniform [Gus1999], the output of such tools and the requirements of MAISA are not directly compatible. Thus transferring the diagrams drawn with the CASE tool to MAISA has been a substantial task in the early phases of the MAISA project. A complete diagram is exported as a set of Prolog facts (see [Gus2000]). These facts are then read in by the MAISA tool. The internal data structure for the diagrams is very close to UML structures (See Figure 13 for an overview). We are currently using MetaEdit [Met2001] as our CASE tool, but this choice is

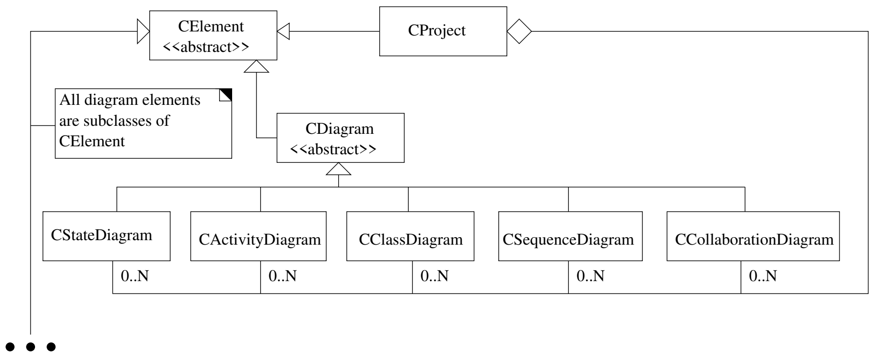

Figure 13: Overview of MAISA's data structure.

irrelevant from MAISA's point of view. Even using a text editor would be sufficient (although quite tedious).

We use the built-in report generator of MetaEdit for generating the facts. Basically, any diagrams drawn with MetaEdit (both standard diagrams and user defined ones) can be exported. The export format can be defined by the user. The definition language of MetaEdit is quite versatile. However, as discussed later, it is not a programming language, so any complex computation has to be done in the MAISA tool.

The procedure itself is fairly straightforward. The diagram to be exported is disassembled into its components, and a fact (or a set of facts) is then generated of each component. Other necessary information (such as the class a particular method belongs to) is included in these facts. For example, for each attribute in a class diagram, we generate a fact that states the identifier and the name of that attribute as well as the class it belongs to. A gif picture of the diagram is also exported. An overview of the general architecture of the MAISA tool and its interaction with MetaEdit is presented in [GuN1999].

To simplify the input process, the facts are required to be in their structural order. When an association is read from a class diagram, both (or all) participating classes must have already been read. Each element is represented by a unique identifier. In principle, any unique string (such as the full package name of a class) is sufficient. At present, the object identifier from MetaEdit is used for this purpose. This is especially convenient as many design elements (such as associations) do not necessarily have a unique name or any name at all in the UML diagram. It also prevents any unintentional errors that could occur, if we were to use names supplied by the user (since we do not use any validity checking in this process).

Much of the information needed by MAISA can not be generated by the CASE tool, so this is done by MAISA itself (see Section 3.3).

Each element in the diagram structure knows the next-level components in the hierarchy (for instance, a class diagram knows the classes, associations etc. it contains). In the situation presented in Figure 14, this means that the *project* element recognizes and uses the services of the *class diagram* element. Correspondingly the *class* element uses the services of its *method*

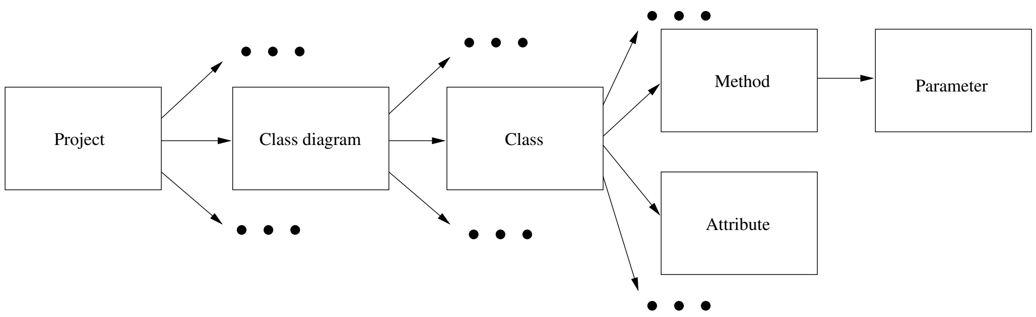

Figure 14: An example of the element hierarchy of a project.

and *attribute* elements.

As the elements are read, they are simultaneously counted. This enables any element to immediately know (and answer questions about) the number of its subelements.

The Prolog files may contain additional information such as comments that are ignored by MAISA. The format is easy to extend as one only needs to add appropriate elements to the `read` methods of the diagram classes so that MAISA recognizes the new facts (even the possible comments could be utilized by MAISA).

Additional, simple measures such as cohesion or coupling metrics are also easy to add when needed by implementing the metrics calculation methods into the appropriate element (e.g. class diagram).

3.3 Metrics calculation

Metric configuration is the link between the data and the results. It defines the metrics the user wishes to see and sets the parameters for these metrics. Both the actual calculation and the presentation of the results is guided by the configuration.

The configuration is defined and set separately through the user interface. Every time any metric calculations are performed, this configuration is accessed. As setting the entire configuration from scratch can be very time consuming, the user can save configurations separately or use a default configuration.

As mentioned earlier, any complex metric computation is done by MAISA. Thus, for example, each class diagram calculates recursively its package hierarchy. In the future, this phase can contain other, diagram type specific processing. Currently, the information that is not needed in the pattern search in also pruned during this phase. An example of such information would be the diagram header.

Most calculations are done once per diagram per session as the diagrams are static from MAISA's point of view. The situation is different for the project as new diagrams can be added to or removed from it at will during the session. Even if this is the case, using any of the old diagrams doesn't require additional computing.

More complex processing is needed with any hierarchical diagrams, such as activity diagrams. With these we must be able to handle an arbitrary number of subdiagrams. The following basic procedure, however, applies also to this type of diagrams:

- Each element knows its subelements.
- When a metric is required, an element polls its subelements for the relevant data.
- If the required information is already calculated, it is returned immediately, otherwise it will be calculated.

Consider an example where we want to know the class(es) with the minimum number of methods (NOM) in a project. The procedure is as follows:

1. The project element asks each class diagram for their minimum NOM classes.
2. Each class diagram asks each of its classes for their NOM values.
3. The class diagram elements then select the minimum value of class-wise NOM and each of them returns this information to the project element.
4. The project element then selects those classes that have the globally minimum value of NOM.

Note that any further recalculations are only necessary for the project. When a class diagram already knows the classes with minimum NOM, it is no longer necessary to perform step 2.

On the method level the previous example would be as follows (the numbers refer to Figure 15):

- create a new `MetricResult`
- for each diagram:
  - call `calculateMetrics` method (1)
  - call the appropriate `handleMetrics` method depending on the diagram type, which in this case would be of class `classMetrics` (2)
    - collect the requested data from the diagram (4)
    - update information (3)
- for each requested metric:
  - update the `MetricResult` using the data gathered (in this case, by `classMetrics`)
- return `MetricResult`

It is important to keep in mind that there are several levels of metrics. Minimum (and other) NOM values are needed *per project*, *per diagram* and *per class*, thus, for example, there are two separate versions of the method `checkMinAndMax`. However, for *Number of Classes* there are only project and diagram levels.

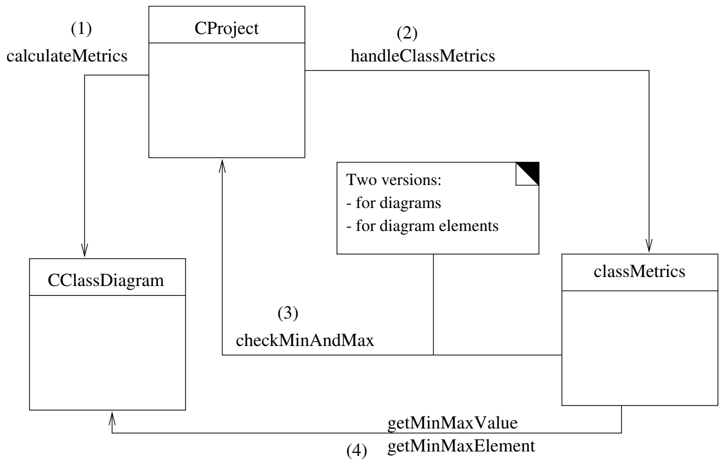

Figure 15: Calculating class metrics in a project.

3.4 Performance analysis

Performance analysis requires detailed access to the parameters of a particular diagram. At least timing information and the probabilities attached to transitions must be modifiable within MAISA. This means that the structure or layout of the diagram is also different from other diagrams, where all modifications are done while drawing the diagram. This modifiability requires also that the user needs to understand the diagram structure.

Extended activity diagrams (`CActivityDiagram`) have been selected as a modeling tool for performance analysis in the MAISA project. They are well-known enough for anyone accustomed with UML diagrams, while the additional information enables their use for this purpose. In addition to the structure of the activity diagrams, a short description of the current performance analysis mechanism will be provided in section 3.4.3 for easier modifiability. As the extended activity diagrams differ from regular UML, their architecture is shown in Figure 16.

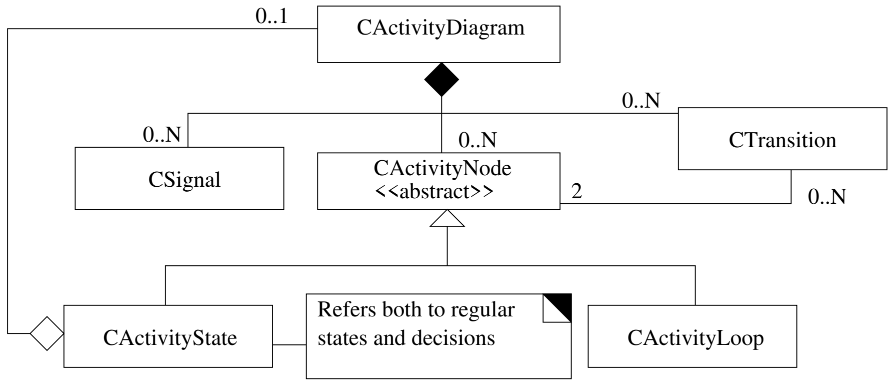

Figure 16: The elements of activity diagrams.

3.4.1 Nodes

The simplest elements in an activity diagram are signals (class `CSignal`). They do not have any of the following properties, although any new properties can easily be added to them.

When using activity diagrams for performance evaluation, we need to add timing information to the basic UML activity states (class `CActivityState`). In our model each node contains (at least) three time attributes:

- *Time_min*: the minimum execution time of this node
- *Time_max*: the maximum execution time of this node
- *Time_typ*: the typical execution time of this node

In order to create hierarchical models, we allow activity nodes to be expanded to further activity diagrams. If a node is expanded (i.e. has a subdiagram), then we obtain all time information from the subdiagram. This means that we always calculate first the subdiagrams. Graphically we denote an expanded node with a special node symbol (see Figure 17 for the symbol). The timing information for all simple nodes is shown as *Time_min* - *Time_typ* - *Time_max*. If an expanded node contains explicit minimum and maximum times, these can be used as constraints for the values obtained from the subdiagram.

As with regular activity diagrams, we have several special types of nodes:

- Start nodes: Each diagram (or substate) has a single start-node (each concurrent substate has its own start-node).
- Stop nodes: Each diagram (or substate) has one or several stop-nodes.
- Decision points: Contains a condition according to which the outgoing transition is chosen.

A special type of node is also the *loop node* (class `CActivityLoop`), which begins a loop. We denote a loop node with a rectangle around the ellipse. In addition to the timing values, it contains three thresholds:

- *limit_min:* the minimum number of repetitions
- *limit_typ:* the typical number of repetitions
- *limit_max:* the maximum number of repetitions.

These limits are shown along the lower edge of the rectangle (see Figure 17). Note that a loop node cannot be expanded.

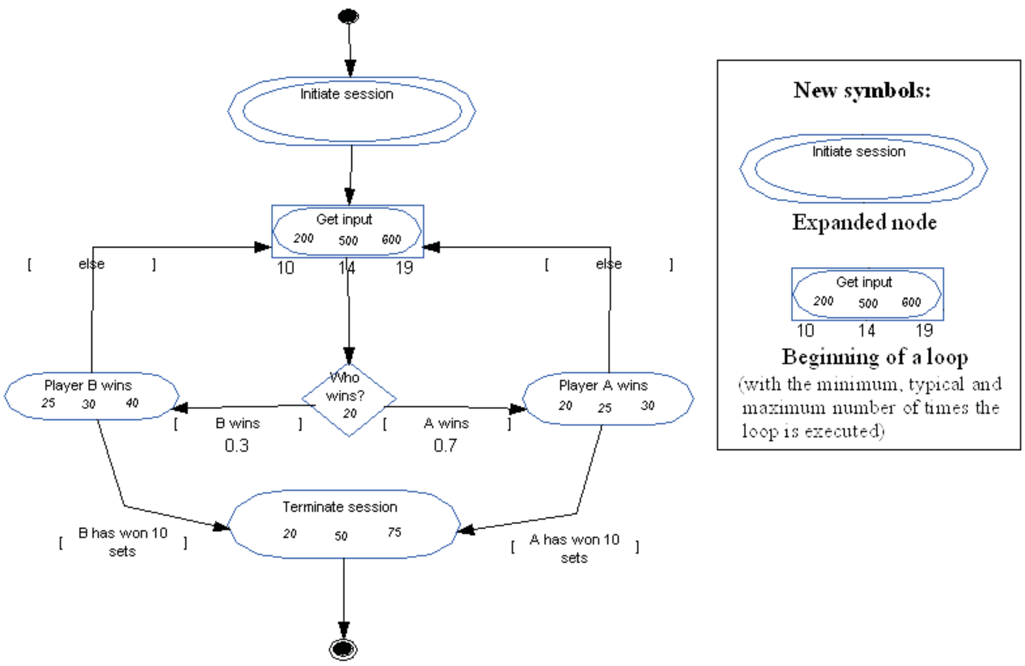

Figure 17: The node types of the extended activity diagram.

3.4.2 Transitions

The transitions (class `CTransition`) are very much like those in standard activity diagrams. We add, however, probabilities to them.

Each transition may contain a probability (how likely it is that the respective condition is fired). This naturally makes sense only if there are more than one outgoing transition from the node. The probabilities of all the transitions attached to a given node can be less than or equal to 1. If one of the transitions has no probability attached, its probability is assumed to be 1-p, where p is the sum of all other transitions originating from the same node. However, if every node is given a probability, the sum must be 1.

Loops are also allowed, but not all loop types are legal:

- Nested loops: One loop is contained by another. The basic case.
- Loops containing non-independent paths: These loops have two or more separate execution paths that are not independent of each other. Consider the situation where two players play a game one round at a time until either has won ten times. Although the whole loop may be executed nineteen times, a single path may only be executed ten times. To obtain a tight bound for the execution time the minimum time should be calculated by executing one path (having the shortest execution time) its minimum number of times. The maximum time should respectively be calculated by executing one

path (having the longest execution time) its maximum number of times (in this case ten times) and the other path nine times. However, in MAISA, we present only the upper and lower limit to the number of times a loop is executed. Thus we may get a larger maximum execution time for this loop.

- Loops crossing each other: These are not allowed. If a loop contains the end node of another loop, then it must also contain the start node.

A loop is always accessed via its start node. Thus all transitions to a node inside a loop come from nodes that are inside it as well. However, any node inside a loop may contain transitions that lead outside the loop. In these cases any probabilities attached to these transitions are ignored when calculating the execution time of the loop. This is because only the limits given in the start node affect the number of times a loop is executed. Any end node (i.e. a node that has a transition to the beginning of the loop) may also have one or more transitions that lead out of the loop.

3.4.3 Basic time analysis

When we use the extended activity diagrams for time analysis, we begin with a set of nodes and a set of successors for each node (i.e. the nodes to which there is a transition from this node). We want to find a path from a given node (the start node) to some end node. In our case we are interested in very specific paths (e.g. the one having the shortest execution time). We define these paths recursively. The halting condition of this recursion is an end node. Otherwise the paths originating in a given node contain the node combined with the paths from each of its successors.

Correspondingly, the time value of a path originating in a given node is the time value[1] of the node itself combined with the values of its successors. The exact procedure depends on the metric needed. For the minimum time, we select that outgoing path which gives the smallest value. For maximum time, the path producing the maximum value is selected. For typical time, we multiply the typical time value of each successor with the probability of the respective transition and add up the results.

The expanded nodes as well as loops are calculated separately. For loops we also need to determine the nodes belonging to it (i.e. the nodes that have a path leading to the start node of the loop).

3.4.4 Further issues

At the current state, the basic tools and techniques for performance analysis have been developed in the MAISA project.

Some important questions for further development include:

[1] Using minimum, typical or maximum value as appropriate.

- What kind of a mechanism do we need (or want to use) for constraint modeling? The Object Constraint Language (OCL) [WaK1999] is versatile, but it would require a separate program or module to enforce those constraints.
- What other attributes do we need for specifying all the resource requirements (e.g CPU usage, disk accesses, memory management, RPC overhead)?
- What algorithm would be the most sensible for performance evaluation purposes? Integer Linear Programming (ILP) [BeT1997] could be a viable alternative to the bottom-up method that is currently being implemented.

3.5 Some reusable classes

In addition to the previously mentioned classes, the datastruct package contains some general classes that are designed to be reusable.

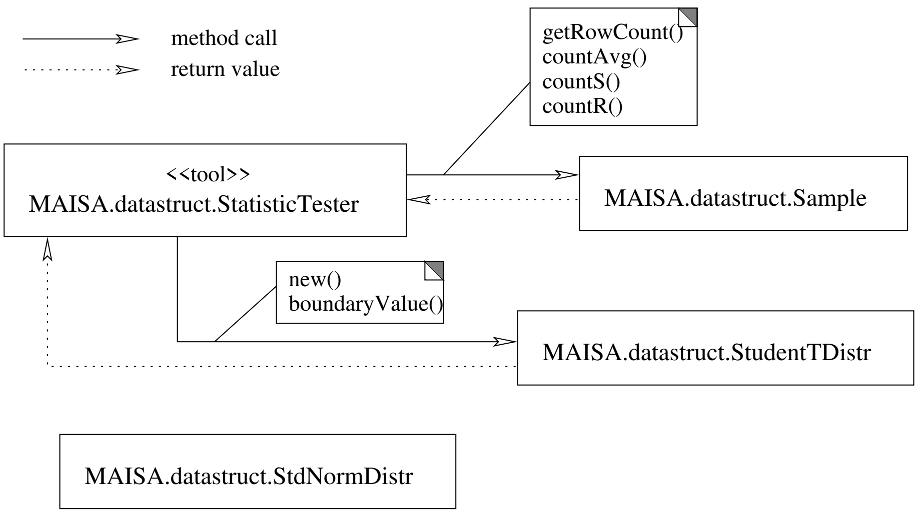

Figure 18: Help window.

The sample classes `MAISA.datastruct.Sample`, `MAISA.datastruct.MetricSample`, `MAISA.datastruct.NominalSample2` and `MAISA.datastruct.ClassifiedSample` are described in Chapter 2. However, there is a detail invisible to the GUI classes. Figure 18 presents the class `MAISA.datastruct.StatisticTester`, who is responsible of calculating different statistic tests. It is a tool class, i.e. it contains only static properties and cannot be instantiated. There are two classes corresponding to distributions: `MAISA.datastruct.StdNormNistr` presenting the standard normal distribution and `MAISA.datastruct.StudentTDistr` presenting Student's T distribution. The former is not in use, because the implemented statistic tests do not use it. But, if more tests are implemented, there is very likely need for the class.

It might be a good idea to separate all classes belonging to statistic calculation to their own package. On the other hand, metrics data classes are quite closely coupled to the sample classes.

Class `MAISA.datastruct.MetricSample` uses a lightweight date class `MAISA.datastruct.Date` as the data type of the timestamp. It is referred also by `MAISA.StatDrawPanel`. The data class has also a comparator, namely `MAISA.datastruct.DateComparator`. It, too, is referred by `MAISA.StatDrawPanel`.

4 Package MAISA.csp

The MAISA.csp package has originally been designed and implemented by Pauli Misikangas as a part of his Master's thesis [Mis1998] and it is also available as a standalone application. To be used in Maisa, two new classes, `MaisaCustomizedSystem` and `MaisaSolutionSaver`, have been added (See Figure 19).

The `MAISA.csp` package is basically a black box, which is extremely easy to exchange to another implementation as the rest of Maisa is only connected to it via the `MaisaCustomizedSystem` class. This class is constructed with the following information:

- the directory of the pattern library as `String`
- the project directory as `String`
- the files containing the Prolog descriptions as `String` array (one per diagram)
- the `MetricResult` object for the results
- the names of the requested patterns as `String`
- the algorithm to be used as `String` (only relevant if more than one algorithm is in use)

The core class in the `MAISA.csp` package is the abstract `Delegator` class, which handles the pattern matching problem (`getProblem`), the input files (`getContext`), and the algorithm (`getAlgorithm`). It also defines the solution handling mechanism through the `SolutionHandler` class. Only the interfaces are defined here, so any one of these can be redefined via the `CollectionMaker`, `AlgorithmMaker`, and the `SolutionHandler` classes. The `CustomizedSystem` class defines the user interface (from MAISA's point of view the interface to MAISA) and the initialization functionality. The `MaisaSolutionSaver` class implements the `SolutionHandler` interface. It receives the results one at a time, creates appropriate pattern instances, binds the roles and adds the instance to the `MetricResult`.

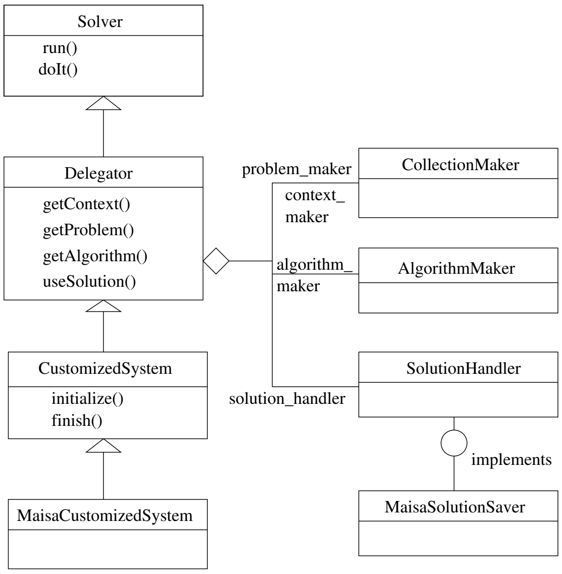

Figure 19: The core of the application framework for pattern detection.

5 Use case scenarios

Figure 20 presents the use case diagram of MAISA, as defined in the requirements analysis. The same diagram is shown in the requirements specification of MAISA. One use case is added to the system after the requirements specification phase, namely statistic analysis. Thus, it is not present in the figure. In the following, the use cases are refined by sequence diagrams which show how the use cases were implemented.

The arrow notations in the following figures are the same as in Chapter 2. The method calls are shown without their parameters, to save the space. The parameters shown in the method calls can be considered as pseudo code. The "return values" returned to the user are typically the occurences the user can see in the GUI and from which the user can notice that his/her commands are performed. Threads are marked with a round thread symbol.

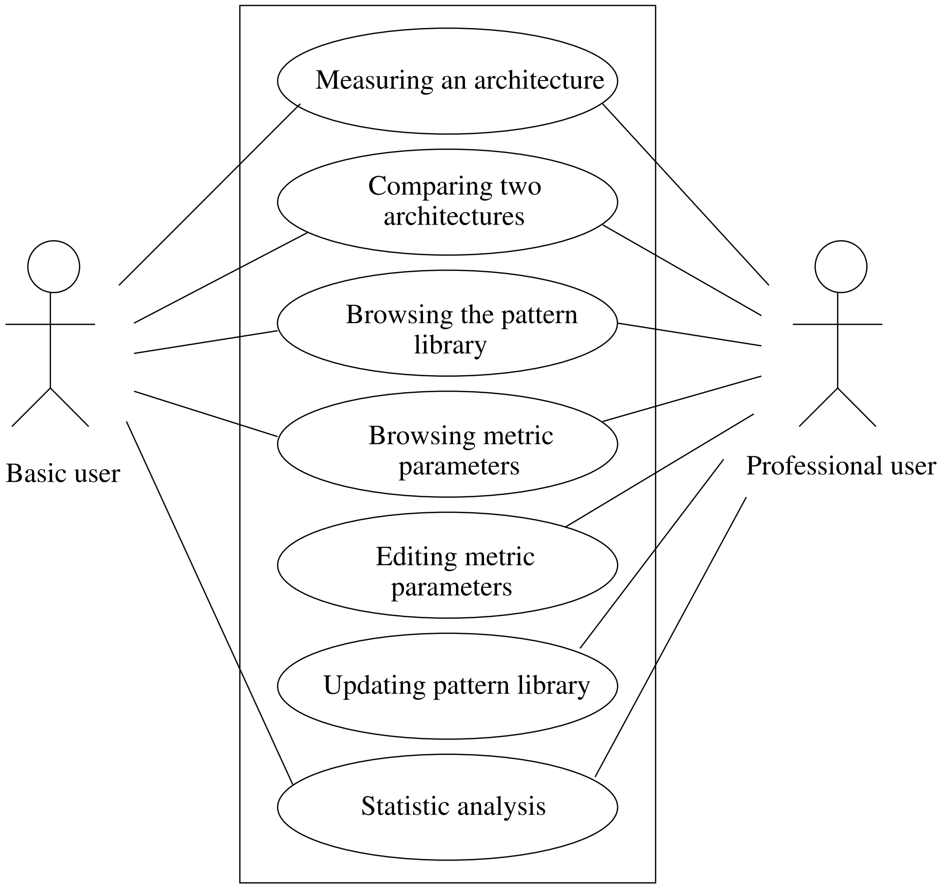

Figure 20: MAISA use cases.

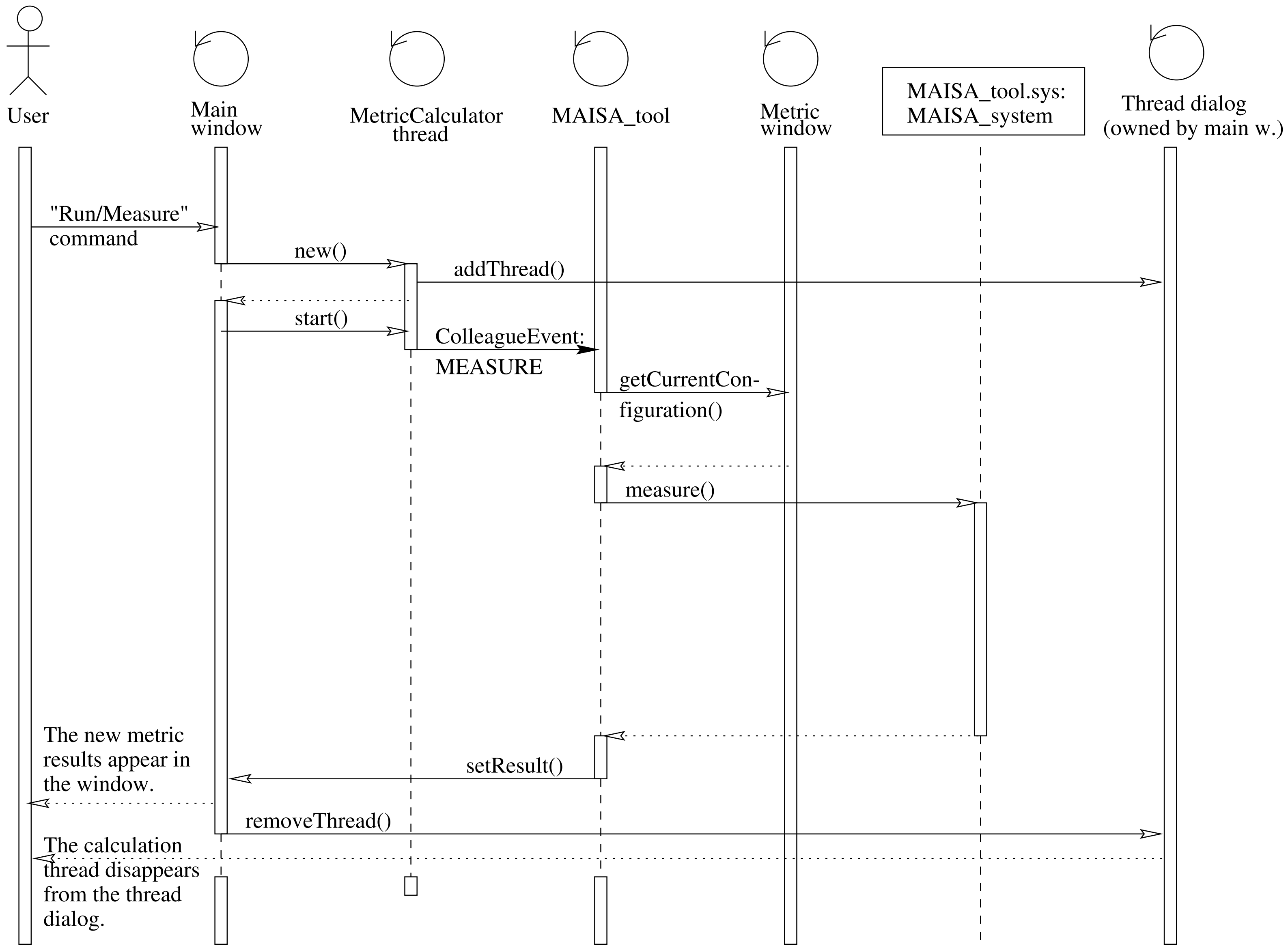

Figure 21: Measuring an architecture.

5.1 Measuring an architecture

Figure 21 presents the general architecture measuring scenario. The user selects the measuring command in the File menu. The figure shows how the calculation thread is created and started due to the command. The thread adds itself in the thread dialog that shows all the ongoing threads. The thread sends the measure event to the *Mediator*, using the same *Observable* object (courier) as its parent, or the main window. The selected architecture object (project or diagram) is transmitted within the colleague event object. When the *Mediator* handles the event, it asks the metrics window to get the current metrics result. The resulting metrics configuration object is passed to the caretaker (`MAISA_system`) with the architecture object. The caretaker calculates the metrics and adds the unsaved metrics results in its "unsaved files" list. When the work is done, the resulting metrics result is returned to the *Mediator*, who passes the results to the main window by calling `setResult` method. The method adds the new results in the window and removes the calculation thread from the thread dialog. This can be considered illogical and could be done otherwise; for instance, the threads could be put in the thread dialog by the main window instead of the thread itself.

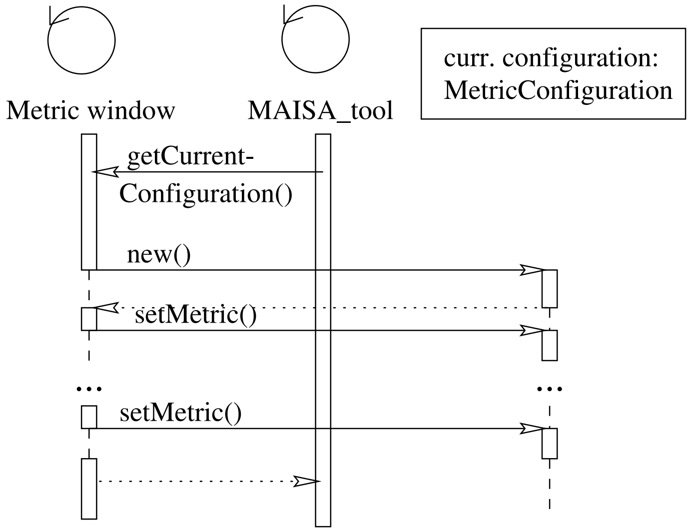

Figure 22: Refinement: querying the metrics configuration.

Figures 22 – 24 refine Figure 21 by presenting how the metrics configuration is set and how the metrics are calculated. In Figure 22 the metric window is asked for the current metrics configuration by calling `getCurrentConfiguration`. The current values are read from the window components and a new metrics configuration object is constructed from them and returned. The `setMetric` method is called as many times as necessary, that is what the three dots stand for.

In Figure 23 a diagram is measured. The `measure` method is overloaded for both diagrams and projects, so the diagram version is used here. It calls the `CDiagram.calculateDiagramMetrics` method, which creates the new metrics result and sets the metrics values by using `MetricResult.addResult` for the metrics that are defined in the metrics configuration object (created in the previous figure). `MetricConfiguration.getMetric` is a boolean method that indicates the metrics that belong to the current configuration. The resulting metrics result is returned by tracking the delegation chain back to the *Mediator*.

Figure 24 presents the other option: the target of the measurement is a project. The `CProject.calculateMetrics` method calls `CDiagram.calculateMetrics` of each diagram belonging to the project (notice the difference to `CDiagram.calculateDiagramMetrics`, which is used when measuring a single diagram). The metrics result object is passed as a parameter of the method, to enable the diagrams to update the same metrics result object.

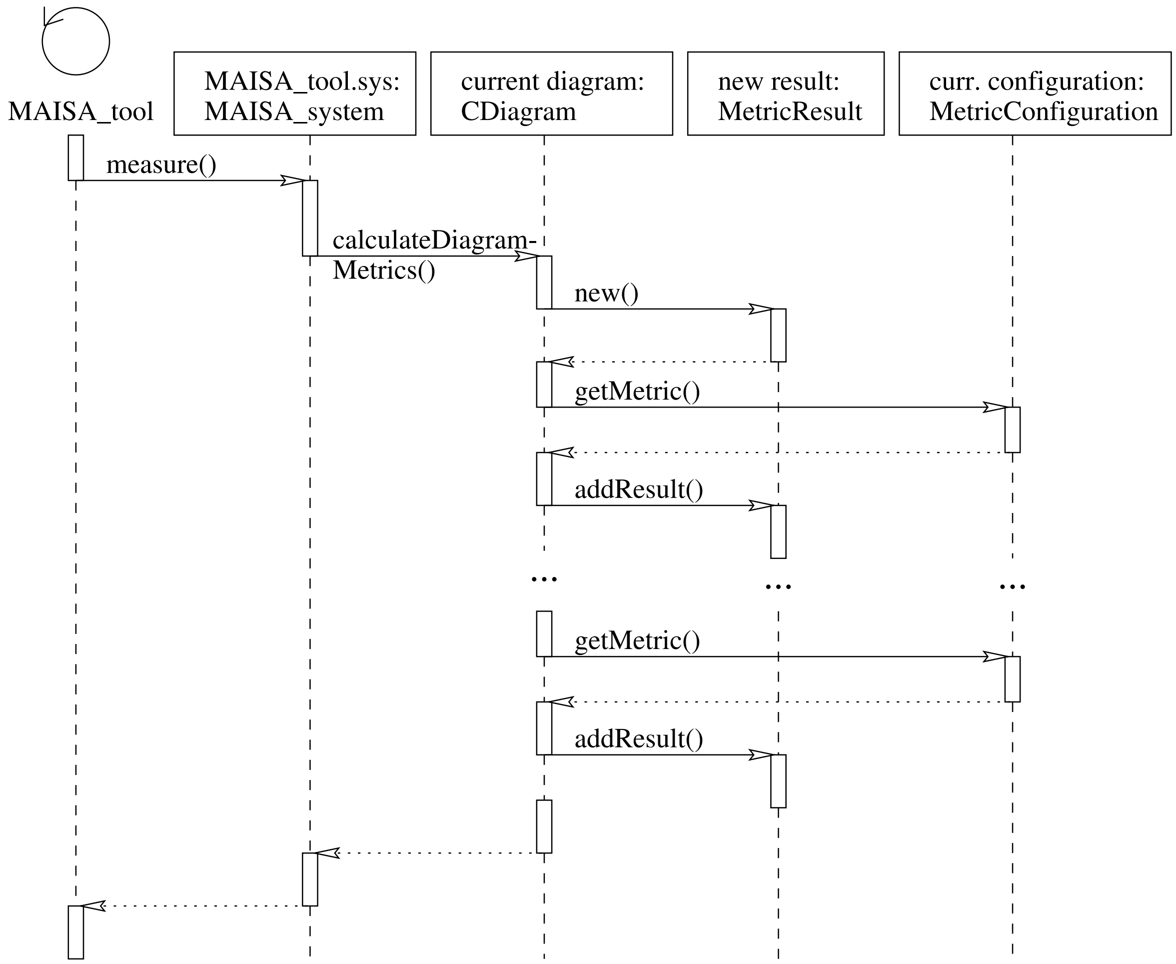

Figure 23: Refinement: measuring a diagram.

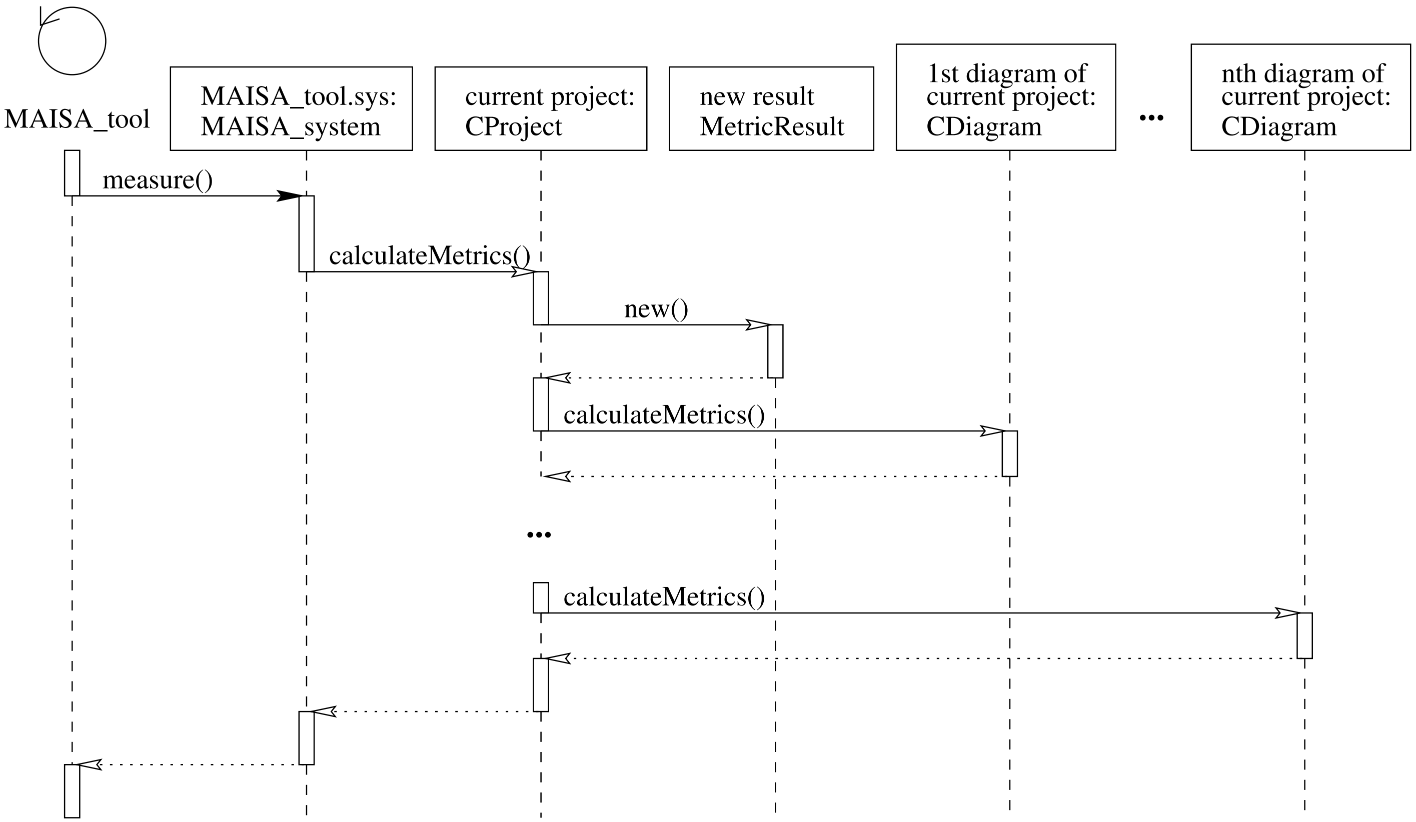

Figure 24: Refinement: measuring a project.

5.2 Comparing two architectures

The architecture comparison operation uses the measurement described above. Figure 25 presents the operation in general. Calculation threads that perform the comparison are created, started and shown in the thread dialog in the same manner as the measurement threads.

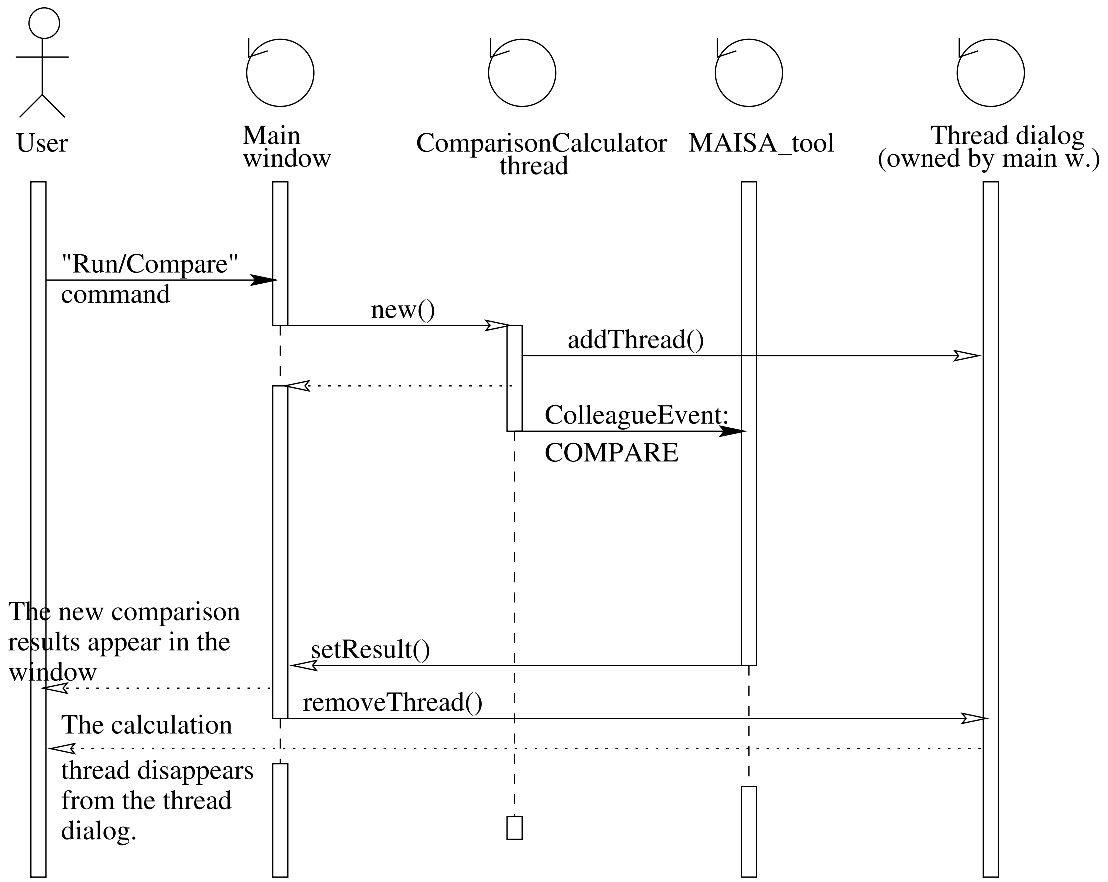

Figure 25: Comparison operation.

The method `MAISA_system.compare` is overloaded for a pair of metrics results, a pair of archicectures (projects of diagrams) and a metrics result and a diagram. Figures 26 and 27 present the first two of the three cases. A comparison result consists of two metrics results. If they are not calculated in advance, which is the basic case, they are calculated now, which is the second case. The third case, where one of the metrics results is calculated in advance, is a combination of the former ones.

Note, that the two metrics results are expected to be based on the same metrics configuration. This is no problem in the second case, because both metrics results are calculated now, with the current configuration. In the first and third case some metrics data may disappear when the metrics results are combined into a comparison result, because only those metrics that are common to both results are taken into account. The original results are not violated, however.

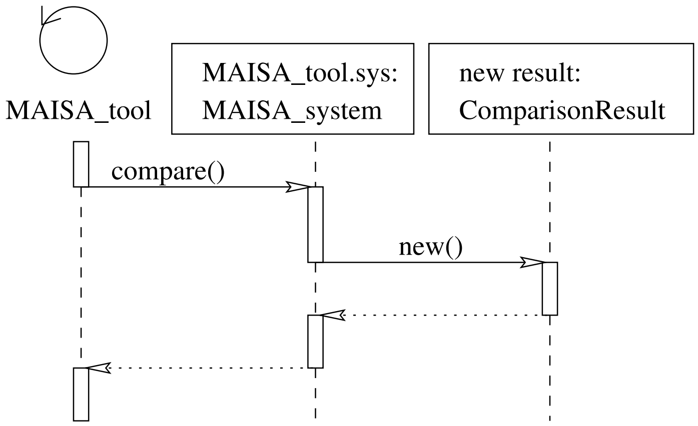

Figure 26: Refinement: comparing two metrics results.

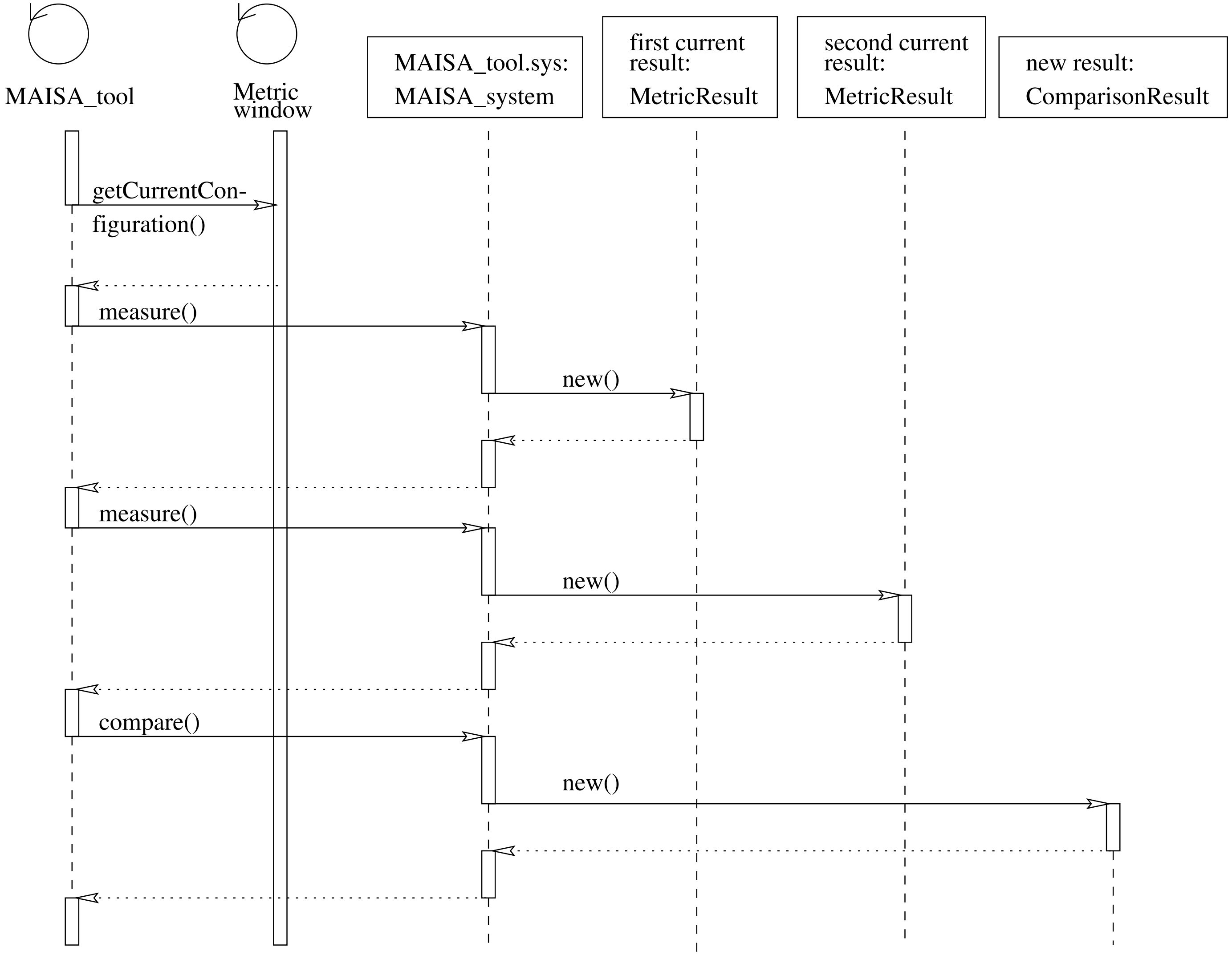

Figure 27: Refinement: comparing two architectures.

5.3 Browsing the pattern library

The functionality of "Browsing the pattern library" concerns mostly just the pattern library window. It is opened by selecting the pattern library window in the Window menu of any window of the tool. The *Mediator*, who keeps a reference to the windows of the tool, simply asks the pattern library to draw itself on the screen, deiconify itself and put itself in front of other windows on the screen.

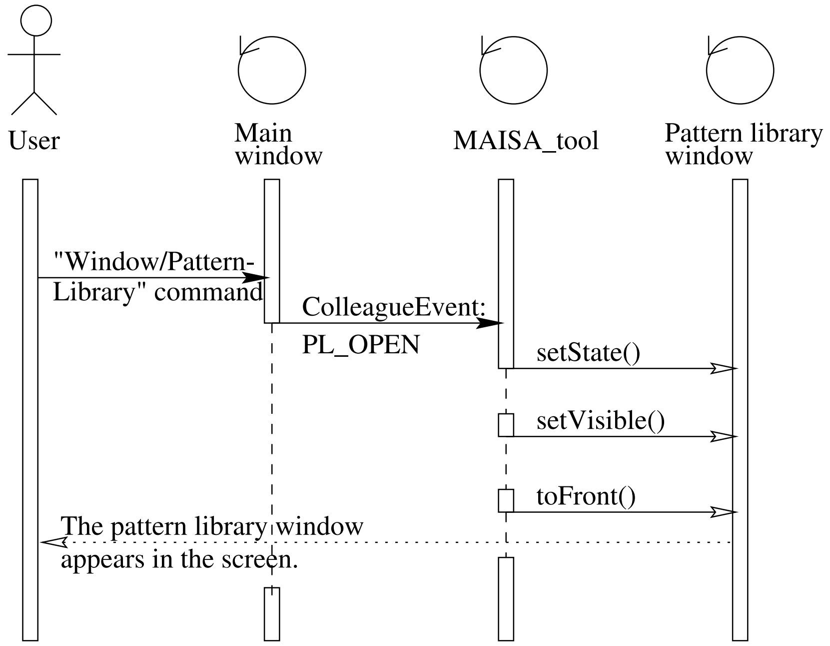

Figure 28: Opening the pattern library and moving the focus there.

The patterns are browsed by selecting them in the tree shaped list in the pattern library window. Figure 29 describes what happens then. As explained in Chapter 2, the window itself refers to the patterns with their names, which are strings. When the user clicks a pattern in the list, the pattern object is queried and its description is put in the window with the `setData` method. The pattern description consists of some text fields, an image, an image file name, statistic information (numeric fields), Prolog description and Prolog file name. The common file of the library (containing information that concerns pattern recognition order) is also loaded in the window each time a pattern is selected. This could be optimized by doing it only when necessary.

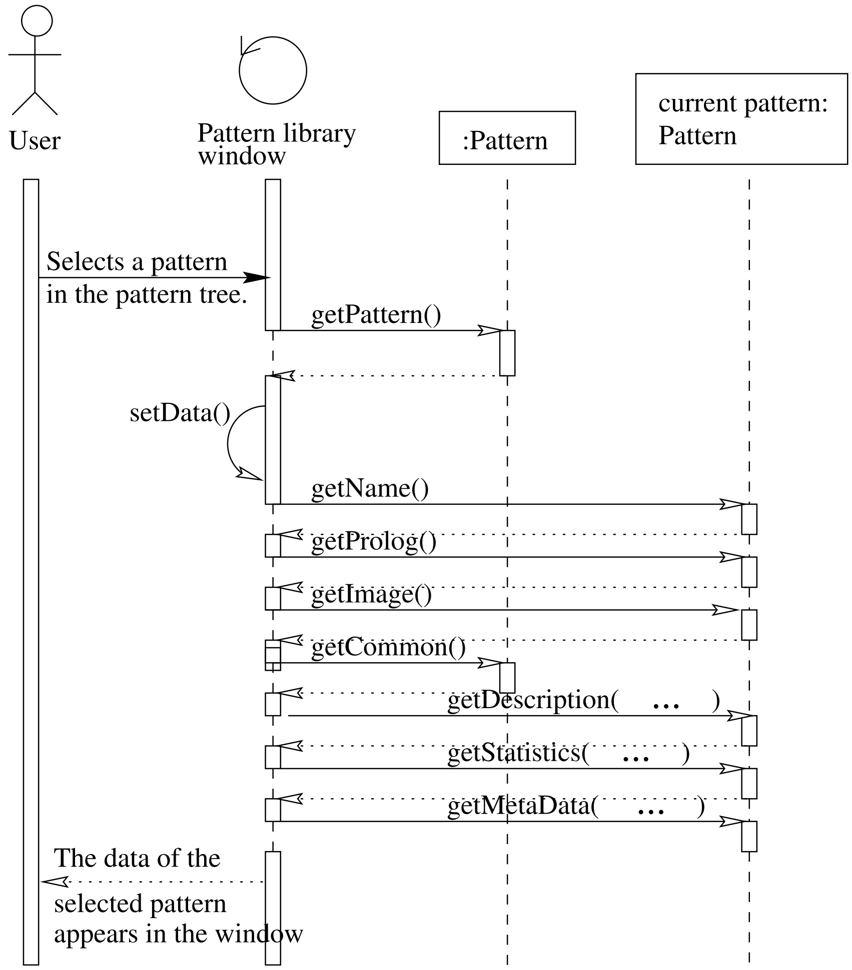

Figure 29: Browsing patterns in the library.

5.4 Browsing metrics parameters

The setting of metrics parameters begins with opening the metrics window. The procedure is similar to opening the pattern library window (or any window of the tool). It is shown in Figure 30.

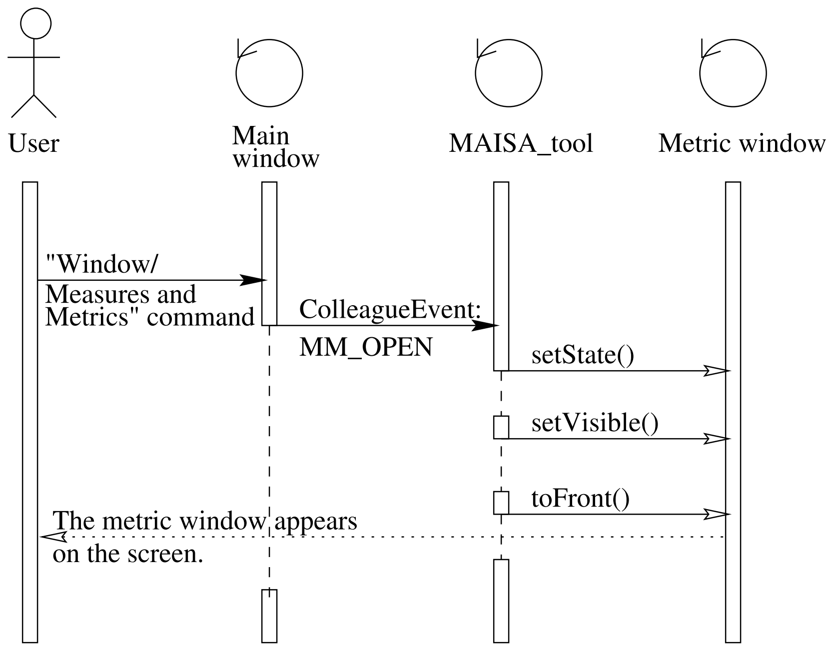

Figure 30: Browsing the metric parameters: opening the metric vindow.

When the tool is started, the current metrics configuration is set in the window, if the user preference “Remember program state” is set on. If not, the default metrics configuration is set. When starting the tool for the first time, the default configuration contains all the metrics available.

5.5 Editing metrics parameters

The actual editing process is not very interesting. It is done just via the GUI components of the window. When the checkbox of a measure is selected, the checkboxes of the individual metrics of that measure are enabled. And when the checkbox of a metrics is selected, the text fields of the thresholds of that metrics are enabled. And vice versa, unselecting goes likewise.

The metrics configuration is read from the window when saving the configuration or measuring an architecture. A new metrics configuration is created and the configuration is constructed according to the contents of the GUI components, as shown in Figure 22 (on page 31).

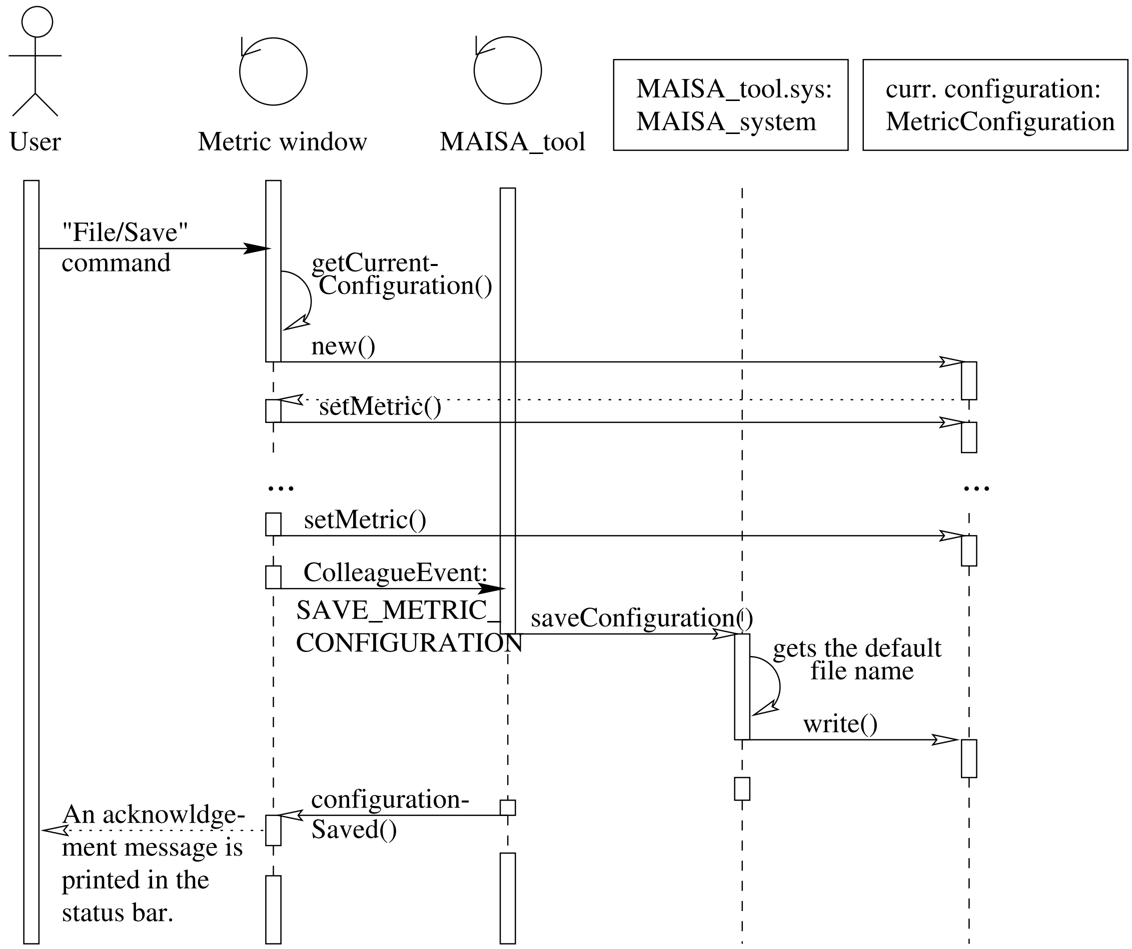

Figure 31: Saving the metrics configuration.

Figure 31 shows what happens when the metrics configuration is saved. The current configuration is read and passed to the *Mediator* in `SAVE_METRIC_CONFIGURATION` event. This is also an example of using the caretaker object. In theory, the metrics window could save the current configuration itself, but the operation is delegated to the caretaker because it knows the name of the default configuration file.

5.6 Updating the pattern library

Three updating operations are described here: adding a pattern (group), saving a pattern and deleting a pattern (group). One of the main differences between editing the pattern library and handling architectures or editing metrics parameters is that deleting a pattern is permanent while closing an architecture is not. This is why the explicit edit mode is offered for those who need to edit the library. Only saving operation (a pattern can be copied elsewhere) is available in read-only mode.

Figure 32 presents the operation of adding a pattern group. Adding a single pattern goes the same way. String Dialog is used for querying the name for the new pattern group. Its `setInfo` method sets the query text in the window. String Dialog returns the string by calling the `answer` method of the pattern library window. When the name is known, a colleague event is sent as a request for adding a new group to the library. This could be done by the pattern library window itself, but the metrics window needs to know whenever the library structure is modified, and the windows are ignorant of each other. So (again) adding the pattern group is delegated to the caretaker (because the *Mediator* is designed to be as dummy as possible) and the *Mediator* handles sending the info to the metrics window (`addedGroup`). The pattern library window is informed of adding a new pattern group (the other `addedGroup`) because the tree shaped list of the window is not updated before it is known that creating a new subdirectory to the pattern library directory has succeeded.

Figure 33 shows how a pattern is saved in the library. The "Save" command is disabled if a pattern group is selected. The command could be added to the groups, too. The patterns know their default file names, to optimize event sending. Thus, the operation can be performed by the pattern library window itself in case the name of the pattern is not changed. If it is, as in Figure 33, a colleague event is sent to inform the metrics window. The info is sent both to the metrics window and to the pattern library window, just like in Figure 32.

Figure 34 presents deleting a pattern from the library. Deleting a pattern group is performed in the same way. The `doDel` method does the trick. But first, it is checked if the "Confirm pattern deletion" preference is set on. If it is, the user is asked if he really wants to remove the pattern, and `doDel` is only called if he gives a positive answer. Otherwise it is called directly. And like in the previous cases, the pattern deletion is done by sending a colleague event, the operation is delegated to the caretaker, and finally information of the successful operation is sent to the two windows that want to update their pattern lists according to the modification.

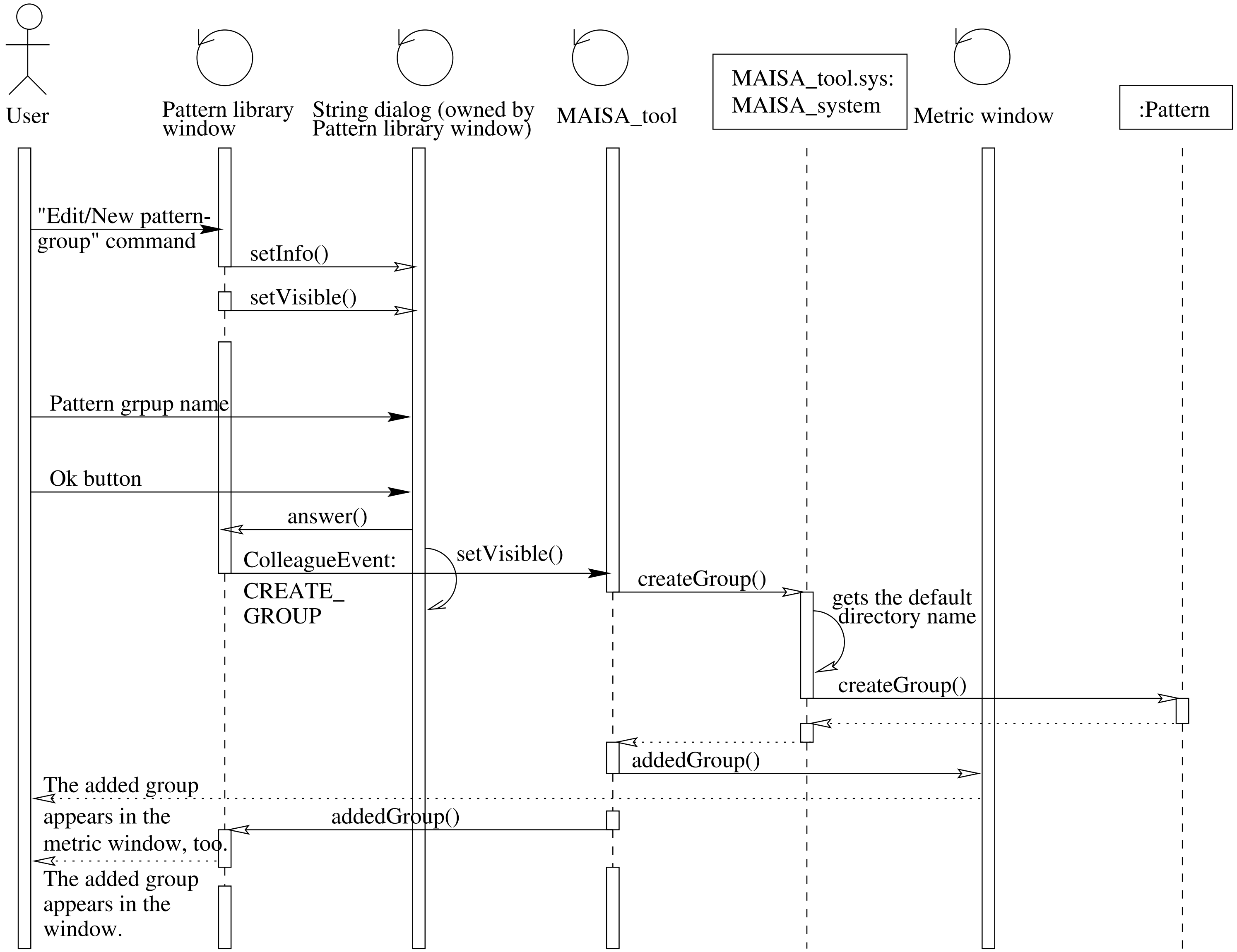

Figure 32: Adding a pattern group.

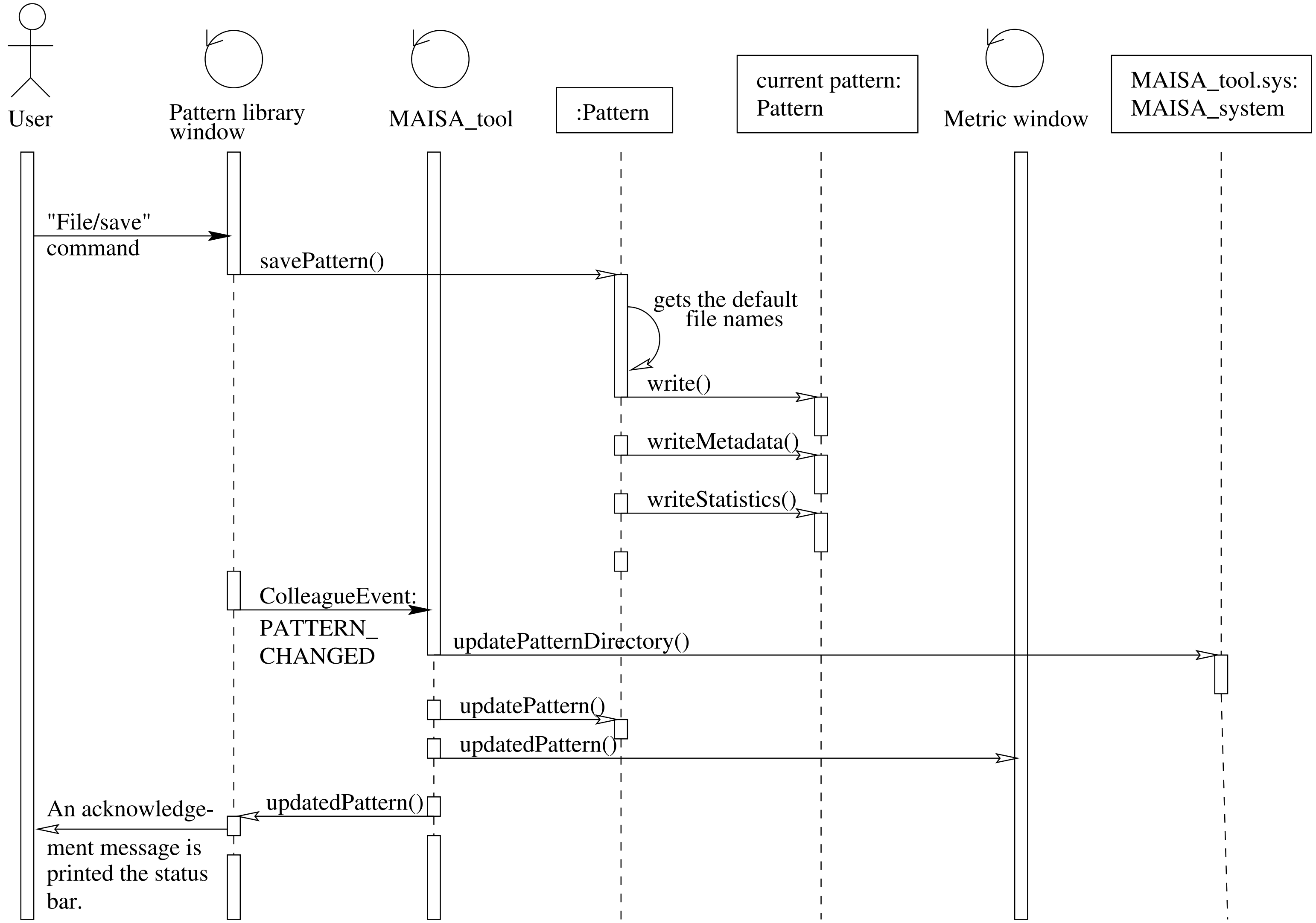

Figure 33: Saving the selected pattern.

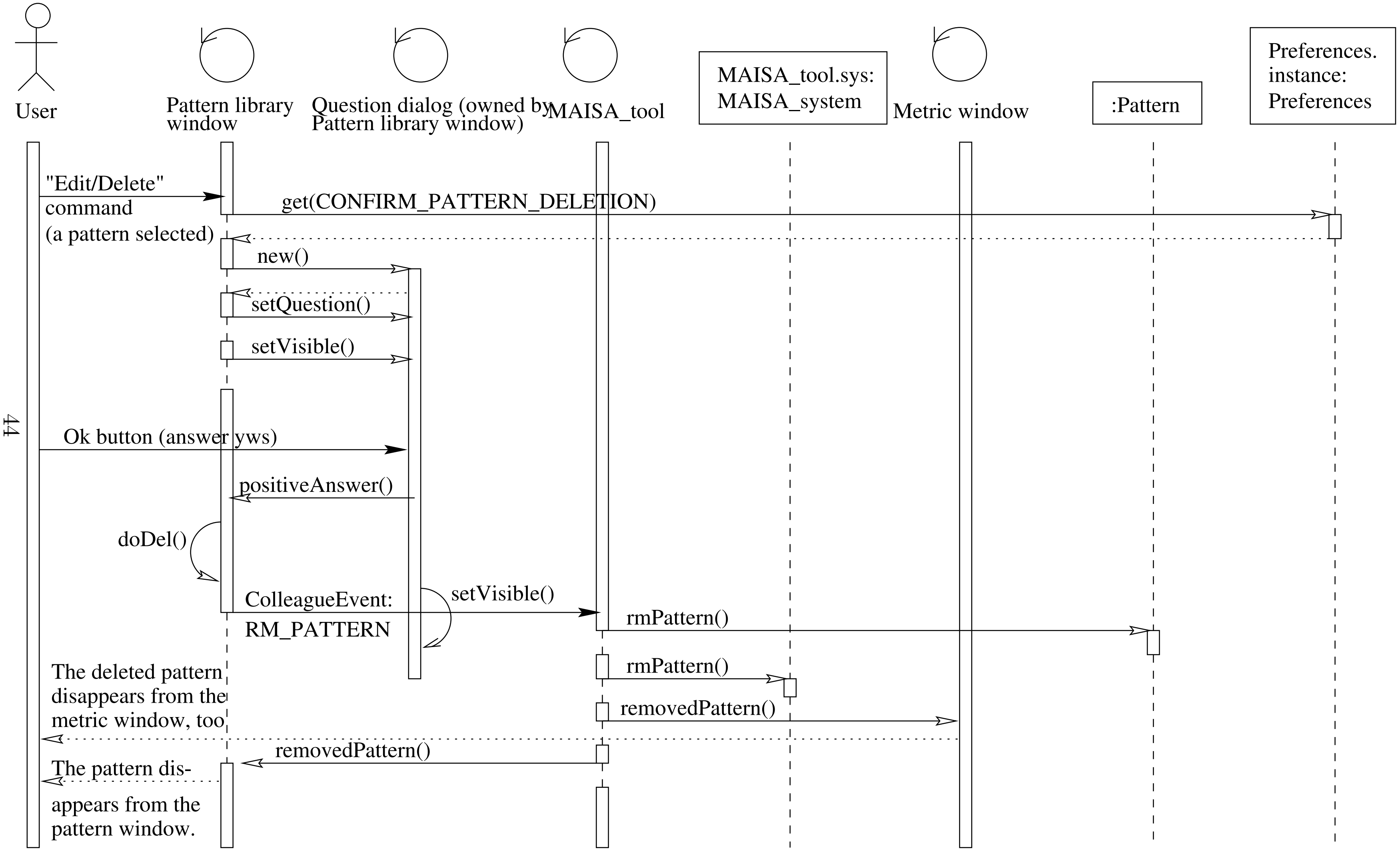

Figure 34: Deleting a pattern.

5.7 Statistic analysis

The statistic analysis begins with collecting the history data. Figure 35 presents the scenario of adding a metrics result in the history data repository. The user selects a result and gives the add command, after which the main window packs the current metrics result in a colleague event which it sends to the *Mediator*. The procedure is staightforward: the *Mediator* delegates the task to the caretaker object, which simply adds the `MetricSample`s of the result to the history data object one by one. If the added sample already exists in the repository, the `addData` methods return false, otherwise they return true. The same values are returned to the *Mediator*. When the job is done, the *Mediator* informs the history data window that the repository is changed by calling the `MAISA_tool` the with `update` method. This is the only way to set the repository data in the history data window. It is done when the program is started and every time the history data repository is modified, that is, when the user clears the whole repository or adds a metrics result there.

Figure 36 presents this `update` method. The history data window, which is a mere dummy wrapper itself, delegates the job to the `StatPanel`, which in turn is the heart of the window, containing both the functionality and the GUI components. The panel checks the current sample unit definition (per projects, classes of certain diagrams, plus the possible bounds) and updates the offered variable collection according to it and the current history data. This is done by the `updateXY` method which gets the available variable names by calling `getHeader` as many times as there are variables. This is what the three dots stand for in the figure.

An example of drawing the diagram and calculating the key figures is shown in Figure 37. First, the user selects the diagram type and defines the sample unit and the variable of the diagram (i.e. the metrics whose values are to be examined). When ready, he presses the "Update" button to draw the diagram and calculate the key figures. The sample unit is checked, and the diagram parameters are read from the GUI components. The actual diagram generation is delegated to `StatDrawPanel` which is the drawing canvas contained by `StatPanel`. Each diagram type has a corresponging method in that class. The method is overloaded, there being two versions of it: one called by `StatPanel` and the other called by `StatDrawPanel` itself. In the figure the diagram type is one variable distribution, which is presented as histogram. The corresponding drawing method is `drawDiagramDISTR`. It updates the status fields of the `StatDrawPanel` and calls its `repaint` method, which causes `paintComponent` to be called. It checks the status fields and calls the corresponding diagram drawing method.

Because the numeric histograms are drawn from a classified sample, `drawDiagramDistr` creates one from the current metrics sample. The method gets count of all the categories (i.e. classes) of the classified sample, and draws the diagram according to them. The control is returned to `StatPanel`, which queries the key figure values from the current metrics sample and writes them in the window.

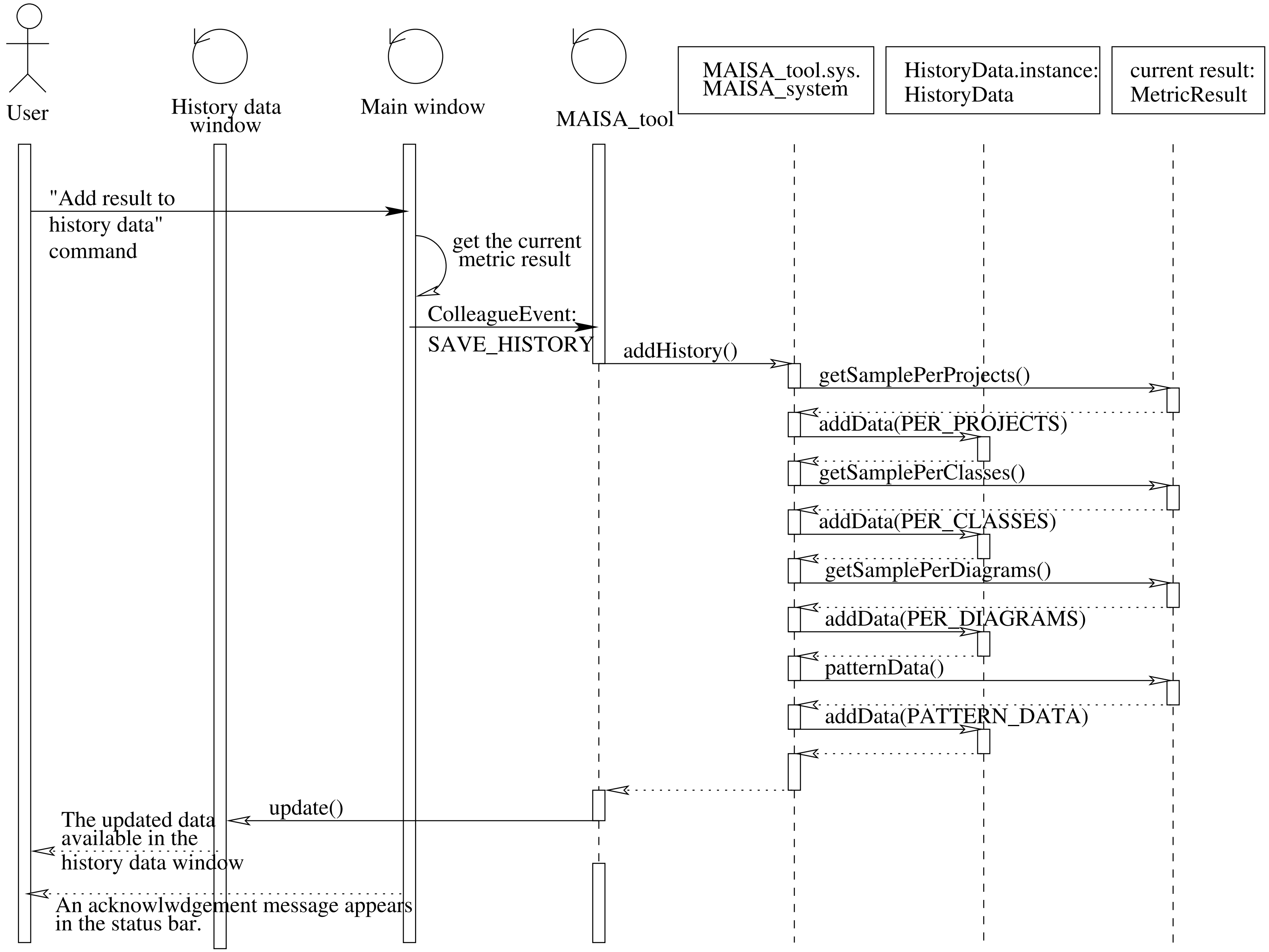

Figure 35: Adding a metrics result in the history data.

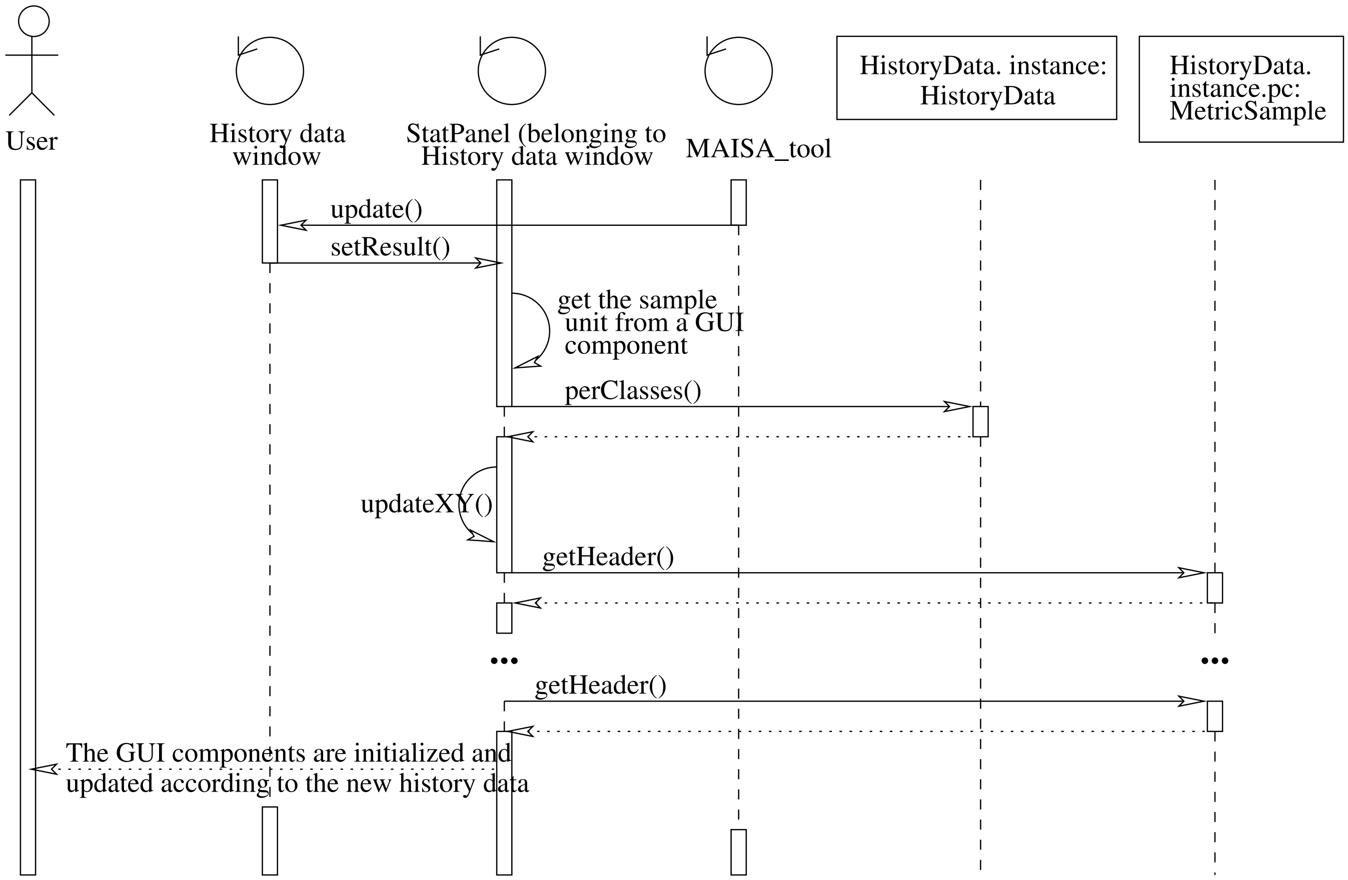

Figure 36: Updating the history data.

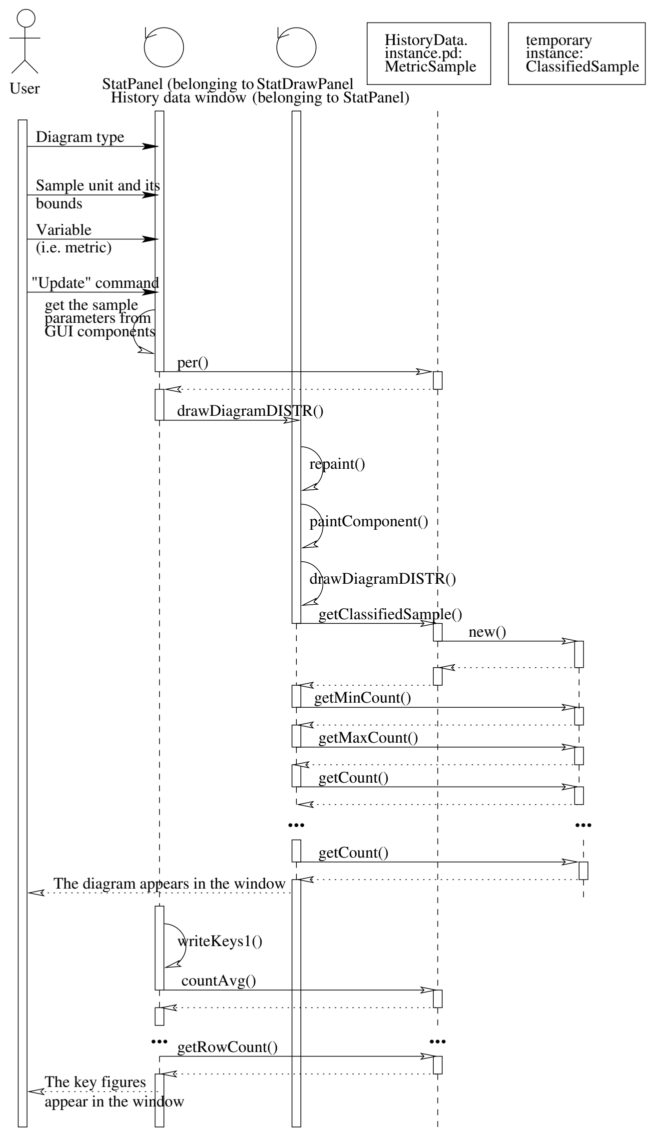

Figure 37: Drawing a statistic diagram.

6 How to modify or update MAISA

The objective of this chapter is to give hints of how certain features of MAISA can be modified. The chapter does not try to be a complete description, but only explains the points that we have regard as called hot spots. The system has been designed to be (easily) modifiable in these points.

6.1 Events

The event tags are coded in class `MAISA.datastruct.ColleagueEvent`. The events are transmitted by `MAISA.datastruct.Courier`. The events are handled by the `update(Observer, Object)` method of the *Observer* class `MAISA.MAISA_tool` and that is the point of code you have to edit, if any.

6.2 Colleagues

If you want to add a new `Colleague` to the structure, remember to create an *Observable* object for the new *Colleague* class. The *Colleagues* are created by the *Mediator* object in its constructor. If the new *Colleague* is a GUI element and you want to save its contents between the sessions, use interface `MAISA.Persistent`. Then you can use `MAISA.MAISA_system.readObject (String, Persistent)` and `MAISA.MAISA_system.saveObject(String, Persistent)` to save and read its contents from the disk (in the same way as all the other *Colleagues* are handled).

6.3 Metrics

Whatever you do to the offered set of metrics, you should never need to edit class `MAISA.datastruct.MetricResult`. It uses method parameters and the numeric tags of `MAISA.datastruct.MetricConfiguration` to construct its data structures, so you don't need to touch it.

1. Start from `MAISA.datastruct.MetricConfiguration` with updating the metrics tags (the static final integer values, there being more than 30 starting from `MAISA.datastruct.MetricConfiguration.LOC` and ending to `MAISA.datastruct.MetricConfiguration.ICOUNT` in MAISA version 1.3). If you are adding a new metric, you have to know which measure the new metrics belongs to. The tag numbers are ordered, first being the size metrics, then the performance metrics, the understandability metrics and the pattern recognition metrics. Don't mess up the order but put your metrics in the right place (and make sure that there are no gaps in the numbering nor any number existing twice). Remember to update the counter tags, namely those ending with `_FIRST` and `_LAST`, expressing the first and last metrics tag values belonging to a certain measure. Then, update the static `metric_pool` vector. This is done in the static initialization block of the class.

2. Then update the window class `MAISA.MeasuresEtMetrics`. It does not use the tags (though it might be improved to do so – feel free :-), so you have to put the metrics labels and text fields in the window by yourself. It is done in the method `jbInit`. You possibly also have to edit the methods that put the contents of the metrics configuration in the window, (`setConfiguration(MetricConfiguration)`), the method that reads the current metrics configuration from the GUI components (`getCurrentConfiguration()`) and the event methods that correspond to the checkboxes of the updated metrics and measures. The latter methods merely enable or disable GUI components, and all the methods of the class are fairly staightforward, so they are quite simple to update.

3. This the most complex point here. You have to update the metrics calculation methods of the class `MAISA.datastruct.CProject` and the subclasses of `MAISA.datastruct.CDiagram`. If the new metrics is independent of pattern recognition, you edit methods `MAISA.datastruct.CProject.calculateMetrics(MetricConfiguration)` and/or `MAISA.datastruct.CDiagram.calculateDiagramMetrics(MetricConfiguration)` and `MAISA.datastruct.CDiagram.calculateMetrics(MetricResult)`. If the new metrics depends on the pattern recognition, you edit methods `MAISA.datastruct.CProject.calculatePatternCoverage(MetricResult)` and/or `MAISA.datastruct.CDiagram.calculatePatternCoverage(MetricConfiguration, MetricResult)`. This is because if the user wants to confirm the pattern recognition, the latter methods are called separately, after the former ones.

6.3.1 Adding a new metric

Basic metrics can be calculated by the appropriate `CElement` object. A `CClass` always knows, how many `CAttribute` objects belong to it. Some metrics require recursive computation, e.g. the depth of inheritance tree for a `CClass` object.

- Make sure that the new Prolog facts are recognized by the diagram in question (the `read` method) and add the necessary data structures into the diagram.
- Add the metrics into the `MetricConfiguration` and `MetricResult` classes.
- Add the calculation to the `calculateMetrics` method of the element in question. If any recursive computation is needed, make the necessary adjustments into the appropriate `doRecursion` method.

6.4 Statistic key figures

The key figures are calculated in classes `MAISA.datastruct.Sample`, `MAISA.datastruct.NominalSample2` and `MAISA.datastruct.ClassifiedSample` (and other sample classes, though they are not used by the tool). Each key figure is calculated by one method. Remember to update classes `MAISA.StatPanel`, `MAISA.StatPanel2` and/or `MAISA.StatPanelForHistoryData`, too, to print the right key figures in the GUI. It is done in methods `writeKeys1(Sample, int)` and `writeKeys2(Sample, int, int)`, which call the key figure methods.

6.5 Statistic diagrams

The diagrams are drawn by `MAISA.StatDrawPanel`. The diagram drawing is requested by calling its diagram drawing methods. The diagram drawing the methods take a `Sample` and some integers as their parameters and their names begin with `drawDiagram`. Those methods are called by `MAISA.StatPanel`, `MAISA.StatPanel2` and `MAISA.StatPanelForHistoryData`, in the event method of the update button, so you might need to update that methos, too, if you update the diagram types. In that case, remember to put the corresponding radio buttons in the statistics panel. If you only edit the diagram drawing methods of `MAISA.StatDrawPanel`, using the same method signatures, you don't need to edit the statistics panel at all.

6.6 Statistic tests

To add a new statistic test, simply implement a corresponding method in the class `MAISA.datastruct.StatisticTester`. There are two ready-made distribution classes for you: `MAISA.datastruct.StdNormDistr` (standard normal distribution) and `MAISA.datastruct.StudTDistr` (Student's T distribution). They have the necessary accessors for calculating the boundary values.

6.7 UML diagram type

All the UML diagram classes are subclasses of `MAISA.datastruct.CDiagram`. The diagram objects are created in `MAISA.datastruct.CProject.readProject(File)` or `static MAISA.` datastruct.CProject.readDiagram(File)1, so you have to edit both the modified diagram class and the project class. Remember to check the metrics calculation methods (see Chapter 6.3 above).

6.8 Result file format

Metrics results

1. Metrics result files are read by the method `MAISA.datastruct.MetricResult.read(File, int)` and saved by the method `MAISA.datastruct.MetricResult.write(File, int)`, where the integer parameter presents the file format. In MAISA version 1.3 there are two file formats for reading (MAISA-format and unix-format) and three formats for writing (MAISA-format, unix-format and HTML). The integer flags used as parameters are defined in the metrics result class.

2. If you add or remove a format, you have to edit also the method `MAISA.MAISA_tool.update(Observable, Object)`, because the file format is recognized there and an error message is sent to the main window if the format is unknown to the system. The format is recognized according to the suffix of the file name.

3. The file name suffices are defined in the metrics result class, so you might want to add your suffix there, too.

Comparison results

1. The comparison results are read by the method `MAISA.datastruct.ComparisonResult.read(File)` and saved by the method `MAISA.datastruct.ComparisonResult.write(File, int)`.where the integer parameter presents the file format. There are two formats for writing the results (MAISA-format and HTML) but as there is only one format for reading comparison results. Thus, if you want to provide more formats, you have to override the method.
2. If you add or remove a format, you have to edit also the method `MAISA.MAISA_tool.update (Observable, Object)`, for the reason explained above.
3. In addition to that, if you add a new read format, you have to edit the caretaker class `MAISA.MAISA_system` who calls the read and write methods of the comparison result class.
4. The file name suffices are defined in the metrics result class, so you might want to add your suffix there, too, if you add one.

6.9 Pattern definition

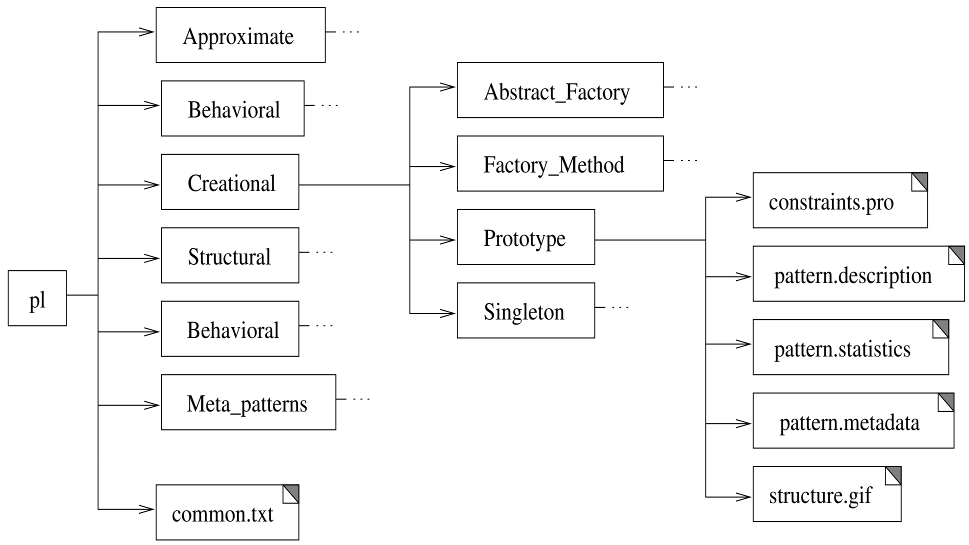

Figure 38: Structure of the pattern library directory.

Figure 38 (found also in the user manual) presents the structure of the pattern library directory. The patterns are read by the method `MAISA.datastruct.Pattern.addPattern(String,`

`String, String)`. You also have to modify the pattern library window to be consistent with the modified pattern structure. Remember that if you modify the Prolog syntax, facts or rules used in pattern description files, you also have to edit the pattern recognition mechanism in package `MAISA.csp`, since it uses the Prolog files. If you only modify the format of non-Prolog files, all you have to do is modify the method `MAISA.datastruct.Pattern.setDescription(int, Object)`.

6.10 Pattern group as a feature

The skeleton of class `MAISA.datastruct.PatternGroup` already appears in the class hierarchy, though it is not used by anyone. If it were in use, it should be used by the pattern library window and it should contain some textual description of the group. The text should be shown in the pattern library window when the pattern is selected in the tree-shaped list.

6.11 Help system

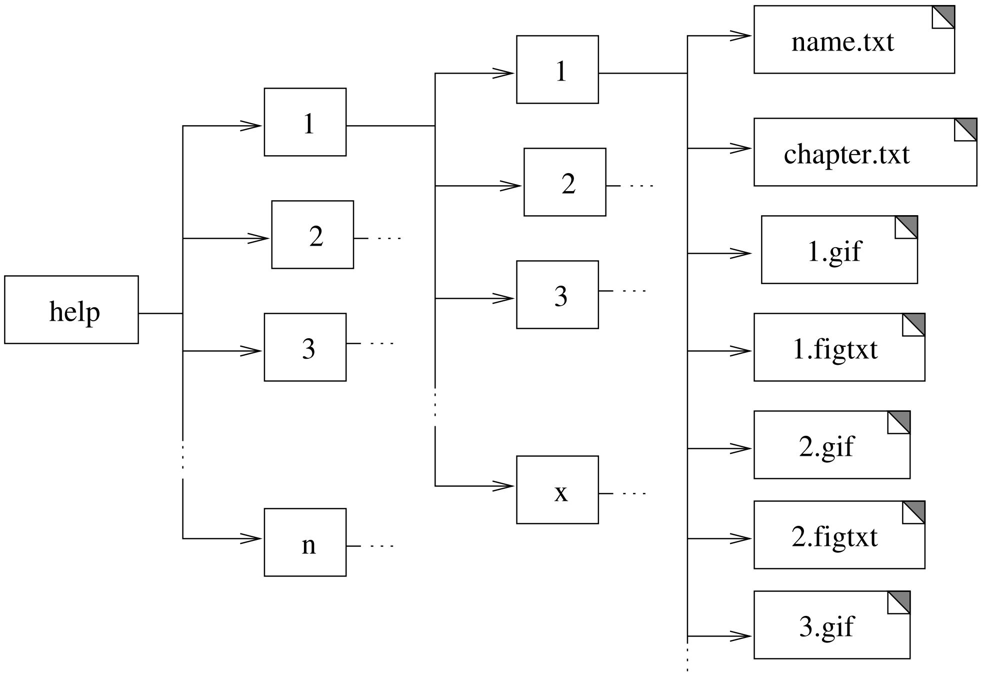

Figure 39: Structure of the help directory.

Figure 39 presents the structure of the help directory. The manual object is read by `MAISA.datastruct.Manual.read(String)` and `MAISA.datastruct.Chapter.read(String)`, the latter being called by the former one. The read method of the manual class is called only by the caretaker object.

6.12 Pattern visualization

At the moment, pattern visualization is provided only for class diagrams, so adding support for other diagram types is a potential hot spot.

The visualization panel in the main window is defined in the class `MAISA.PatternDrawArea`. It keeps a list of the classes and associations of the diagram and knows the selected class in the class list. The data structures should be generalised changing the `MAISA.datastruct.CClass` references to refer `MAISA.datastruct.CElement` instances (superclass of `CClass` and other diagram element classes). The pattern draw area could keep a flag to express which type of diagram the current one is.

The class that actually draws the diagram skeleton is `MAISA.DiagramArea`. `MAISA.PatternDrawArea` calls its method `MAISA.DiagramArea.initDiagram(CDiagram)` to initialize the diagram skeleton in the panel. The method tries to cast the given `CDiagram` object to `CClassDiagram` and clears the panel if the operation throws an exception. The method should be modified to handle the diagram types by e.g. overloading.

`MAISA.DiagramArea` uses `MAISA.datastruct.CoordinateHandler` to get the coordinate data of the classes, which is essential to draw a diagram. It is got by calling `MAISA.datastruct.CDiagram.getHandler`. The other diagram classes override the `getHandler` method by returning null, so the corresponding `getHandler` method should be fully implemented. The coordinate handler class has a private integer field that expresses the type of the diagram.

7 Extending an external CASE tool

When an integrated CASE tool is modified for MAISA, two key aspects must be taken into account. First, the export facility or report generator of the CASE tool must be extended so that it can produce MAISA export format. The facts currently used by MAISA are presented in [Gus2000]. The diagram files can also contain other facts than the predefined ones. In this case, MAISA will simply ignore these new facts. Note that the identifier field can be the internal identifier used by the CASE tool itself, or any other unique string.

Second, if the user wishes to add new diagram types or features into MAISA, also the diagram parser must be added to MAISA.

7.1 Coordinates

Currently, MAISA supports the use of coordinate data for class diagrams. The data is exported as an HTML image map file (with the same name as the diagram). In this file there should be coordinates for classes and packages.

Should the user want to change the way coordinates are presented or add new diagram types to MAISA, the `CoordinateHandler` class in the `MAISA.datastruct` package needs to be updated.

7.2 XMI

As the quality of export facilities vary a great deal among CASE tools, we are considering the use of XMI as an interchange format. The fact, that XMI has been planned as a potential lingua franca of CASE tools, also supports this idea. However, XMI support will probably not become fully integrated into the CASE tools until a new UML standard is released. This new standard may support other features that are central to MAISA, such as an explicit notation for design patterns.

References

[BeT1997] Dimitris Bertsimas, John Tsitsiklis: *Introduction to Linear Optimization*. Athena Scientific, 1997.

[GuN1999] Juha Gustafsson, Lilli Nenonen: *Requirements specification for the MAISA metrics tool*. University of Helsinki, Department of Computer Science, MAISA project, 1999.

[Gus1999] Juha Gustafsson: *UML-perustaiset ohjelmistotyökalut*. (in Finnish) University of Helsinki, Department of Computer Science, MAISA project, 1999.

[Gus2000] Juha Gustafsson: *Metrics calculation in MAISA*. University of Helsinki, Department of Computer Science, MAISA project, 2000.

[Met2001] MetaCase Consulting: `http://www.metacase.com/`

[Mis1998] Pauli Misikangas: *Suunnittelumallien automaattinen tunnistaminen olio-ohjelmista*. (in Finnish) University of Helsinki, Department of Computer Science, C-1998-1.

[NeG2000] Lilli Nenonen, Juha Gustafsson: *Pattern recognition in the MAISA tool*. University of Helsinki, Department of Computer Science, MAISA project, 2000.

[WaK1999] Jos Warmer, Anneke Kleppe: *The Object Constraint Language. Precise Modeling with UML*. Addison-Wesley, 1999.