Metrics calculation in MAISA

Juha Gustafsson

2nd April 2001

1 Introduction

Usually quality metrics are produced near the end of the software life cycle (typically from source code or program execution). This approach has the advantage that we have all relevant information at our disposal. However, in the MAISA project, we strive to measure several software quality aspects at a much earlier stage. We can better influence the quality of the product, if we have quality indicators available to us while we are still designing the product.

In Chapter 2, we discuss shortly what kind of results we intend to produce with MAISA. Chapter 3 presents the issues of drawing diagrams with an external CASE tool and transferring relevant information about those diagrams into MAISA. In Chapter 4 we discuss briefly the metric configuration. Chapter 5 deals with the techniques used to represent the diagram data and to perform measurements based on this data. Finally, Chapter 6 discusses performance measurement aspects.

2 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]). A short example of the metrics calculation is presented in Appendix B.

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 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 Appendix A). These facts are then read in by the MAISA tool. We are currently using MetaEdit as our CASE tool, but this choice is irrelevant from the MAISA 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 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 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. 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 program in this process).

Much of the information needed by MAISA can not be generated by the CASE tool, thus this is done by MAISA itself (see Chapter 5).

Each element in the diagram structure knows the next-level components in the hierarchy (i.e. a class diagram knows the classes, associations etc. it contains). In the situation presented in Figure 1, this means that the *project* element recognizes and uses the services of the *class*

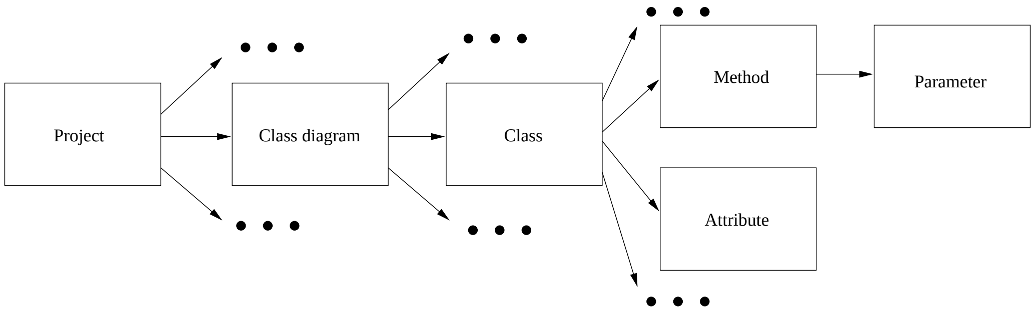

Figure 1: An example of the element hierarchy of a project.

diagram element. Correspondingly the *class* element uses the services of its *method* and *atribute* 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 very easy to extend as one only needs to add appropriate parser elements 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).

4 Metric configuration

The 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.

5 Metric calculation

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 the MAISA 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, 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 respective minimum NOM classes.
2. Each class diagram asks each of its classes for their respective NOM.
3. The class diagram elements then select the minimum value of 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, e.g. ,the classes with minimum NOM, it is no longer necessary to perform step 2.

When searching for patterns we don't have quite such an ideal situation. Most of the time, we need one pass per search over the appropriate set of facts.

6 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 unlike that of the other diagrams, where all modifications are done while drawing the diagram. This modifiability requires also that the user needs to understand the diagram structure. Regular metrics, even pattern mining results can be adequately represented using only a textual format, but for performance analysis a graphical environment is much more important.

Extended activity diagrams have been selected as a modeling tool for performance analysis in the MAISA project. They are familiar enough for anyone accustomed with UML diagrams, while the additional information enables their use for this purpose.

6.1 Nodes

When using activity diagrams for performance evaluation, we need to add timing information to the basic UML activity states. 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 as 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 2). 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*. A loop node begins a loop. We denote a loop node with a rectangle around the ellipse. In addition to the timing values, it contains three limits:

- *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 2). Note that a loop node cannot be expanded.

6.2 Transitions

The transitions are very much like those in common activity diagrams. We add, however, probabilities to them.

Each transition may contain a probability (how likely it is that the respective condition is triggered). 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

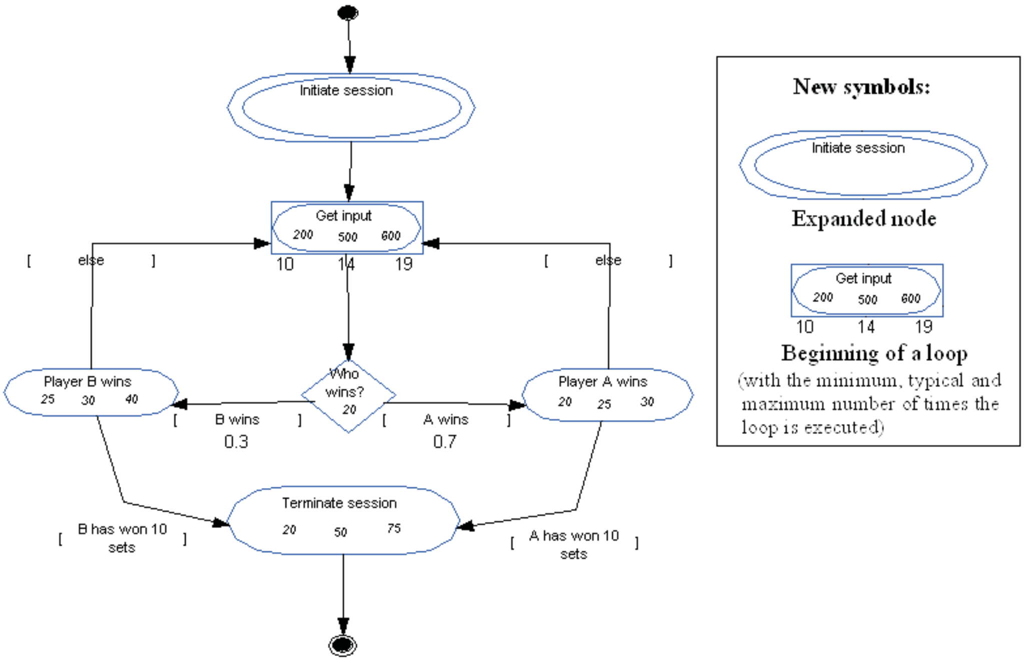

Figure 2: The node types of the extended activity diagram.

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 bounds 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 also inside it. 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.

6.3 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 ending 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).

6.4 Further issues

In the first phase of the MAISA project, we implement the basic tools and techniques for performance analysis.

Some important questions for further development include:

- What kind of a mechanism do we need (or want to use) for constraint modeling? The Object Constraint Language (OCL) is versatile, but it would require a separate program or module to enforce those constraints.
- What other attributes do we need to specify 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) could be a viable alternative to the bottom-up method that is currently being implemented.

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

7 Conclusions

In this paper we have discussed the transfer of design information (the UML diagrams in particular) from a CASE tool to our metrics tool. The mismatch between these two environments is fortunately small enough, so that we can use them together with reasonable effort.

We have also presented the techniques the MAISA tool uses when calculating regular (i.e. not pattern based) metrics. These include maintaining the natural hierarchical structure of UML diagrams and reducing the number of times any given calculations need to be performed.

Open topics include performance analysis (as discussed in the previous chapter) and the requirements for the user interface (especially regarding the usability of the MAISA tool).

References

[GuN1999] Juha Gustafsson, Lilli Nenonen: *Requirements specification for the MAISA metric 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.

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

A Facts used in Prolog-files

The parameters of the facts are presented in angle brackets. The marking <id> denotes the element's identifier. The first letter of object identifiers generated with MetaEdit specifies the type of the element (e.g. C is a class element). The meaning of each letter is explained with each diagram type. This list is by no means complete.

A.1 Class diagrams

Element types:

D: diagram
K: package
C: class
A: attribute
M: method
P: parameter
S: association
Y: dependency

Facts:

→ **abstract(<id>).**
The class (or method) <id> is abstract:
e.g.: abstract("C17_135").

→ **aggregation(<id1>,<id2>).**
The association <id1> is an aggregation, with the class <id2> as the containing participant:
e.g.: aggregation("S17_2846","C17_607").

→ **association(<id>,<class1>,<constraint1>,<class2>, <constraint2>).**
Classes <id1> and <id2> participate in the association[2] <id>
with <constraint1> and <constraint2> as their respective constraints:
e.g.: association("S18_1668","C17_2096","1..*","C18_1617","0..*").

→ **attribute(<class_id>,<attribute_id>,<name>,<type>).**
The class <class_id> has an attribute <attribute_id>, with the name <name>
and type <type>:
e.g.: attribute("C18_1617","A18_1547","number","real").

→ **belongsTo(<id>,<owner_id>).**
The class (or package) <id>, belongs to the package <owner_id>:
e.g.: belongsTo("C18_1617","K18_4517").

→ **class(<id>,<name>).**
The diagram has a class <id> with the name <name>:
e.g.: class("C18_1617","Subclass").

→ **classdiagram(<id>,<name>).**
The project has a class diagram <id> with the name <name>:
e.g.: classdiagram("D18_131","MAISAdemo").

[2] Note that aggregations and compositions are also associations

→ **composition(<id1>,<id2>).**
The association <id1> is a composition, with the class <id2> as the containing participant:
e.g.: composition("S17_229","C17_6485").

→ **constructor(<id>).**
The method <id> is a constructor:
e.g.: constructor("M17_3165").

→ **dependency(<id>,<supplier_id>,<client_id>).**
The diagram contains a dependency <id> with supplier <supplier_id> and client <client_id>:
e.g.: dependency("Y18_15836","C18_623","C18_537").

→ **extends(<class_id>,<superclass_id>).**
The class <class_id> is a subclass of class <superclass_id>:
e.g.: extends("C17_343","C17_135").

→ **formalParameter(<method_id>,<id>,<name>,<type>).**
The method <method_id> has a parameter <id>. The name of the parameter is <name> and it's type is <type>:
e.g.: formalParameter("M17_853","P17_842","amount","integer").

→ **has(<class>,<method>).**
The class <class> has a method <method>:
e.g.: has("C18_1617","M18_1582").

→ **interface(<class_id>,<interface_id>).**
The class <class_id> implements the interface <interface_id>:
e.g.: interface("C17_1387","C17_1914").

→ **method(<class_id>,<id>,<name>).**
Class <class_id> has a method <id>, with the name <name>:
e.g.: method("C18_1617","M18_1582","Do_something").

→ **package(<id>, <name>).**
The diagram contains a package <id> with the name <name>:
e.g.: package("K18_4517","MyPackage").

→ **publicMethod(<id>).**
The method <id> is public:
e.g.: publicMethod("M18_1582").

→ **returns(<method_id>,<value>).**
The method <method_id> returns the value <value>:
e.g.: returns("M18_1582","string").

→ **stereotype(<id>,<stereotype>).**
The element's <id> stereotype is <stereotype>:
e.g.: stereotype("C17_135","actor").

→ **type(<id>,<type>).**

The dependency <id> is of type <type>:
e.g.: type("Y13_192","creates").

→ **virtualMethod(<id>).**
Method <id> is virtual:
e.g.: virtualMethod("M17_3165").

A.2 State diagram

Element types:

D: diagram
S: state
T: transition
V: operation

Facts:

→ **action(<id>,<name>).**
An action <name> belongs to the transition <id>:
e.g.: action("T18_14259","C1").

→ **entry(<state_id>,<operation_id>).**
The operation <operation_id> is an entry action of the state <state_id>:
e.g.: entry("S18_13251","V18_14259").

→ **exit(<state_id>,<operation_id>).**
The operation <operation_id> is an exit action of the state <state_id>:
e.g.: exit("S18_13251","V18_14311").

→ **do(<state_id>,<operation_id>).**
The operation <operation_id> is a do action of state <state_id>:
e.g.: do("S18_13251","V18_14363").

→ **guard(<id>,<condition>).**
The transition <id> has a guard <condition>:
e.g.: guard("T18_14049","G3").

→ **operation(<id>,<name>).**
The diagram contains an operation[3] <id> with the name <name>:
e.g.: operation("V18_14259","A1").

→ **state(<id>,<name>).**
The diagram has a state <id> with the name <name>:
e.g.: state("S18_13981","Decision").

→ **statechart(<id>,<name>).**
The project contains a state diagram <id> with the name <name>:
e.g.: statechart("D18_13235","Chart3").

→ **transition(<id>,<state1>,<state2>).**
The diagram contains a transition <id> from <state1> to <state2>:
e.g.: transition("T18_14049","S18_13981","S18_13287").

[3] Either entry, exit or do

A.3 Activity diagram

Element types:
D: diagram
A: state
G: signal
T: transition

Facts:

→ **activitydiagram(<id>,<name>).**
The project contains an activity diagram <id> with the name <name>:
e.g.: activitydiagram("D18_2129","MAISA_activity").

→ **activityLoop(<id>,<name>).**
The diagram contains a loop node <id> with the name <name>:
e.g.: activityLoop("A18_29331","Loop1").

→ **activityState(<id>,<name>).**
The diagram contains a state <id> with the name <name>:
e.g.: activityState("A18_2160","Get input").

→ **activitysubdiagram(<id>,<name>).**
The definition of the subdiagram <id> (that begun with a subdiagram statement) ends here. The name of this subdiagram is <name>:
e.g.: activitysubdiagram("D14_2993","sub1").

→ **counter(<id>,<count_min>,<count_typical>,<count_max>).**
The node <id> (which must be a loop node) marks a loop and the minimum, typical and maximum number of times that loop is executed are <count_min>, <count_typical> and <count_max> respectively:
e.g.: counter("A18_29331","5","20","30").

→ **decision(<id>,<name>).**
The diagram contains a decision point <id> with the name <name>:
e.g.: decision("A18_2187","who wins?").

→ **guard(<id>,<condition>).**
The transition <id> has a guard condition <condition>:
e.g.: guard("T18_2355","A has won 10").

→ **probability(<id>,<value>).**
The probability for the transition <id> is <value>:
e.g.: probability("T18_2355","0.5").

→ **signal(<id>,<name>,<type>).**
The diagram contains a signal <id> with the name <name> that is of type[4] <type>:
e.g.: signal("G18_2463","Call for help","Send").

→ **subdiagram(<owner>,<id>).**
The following facts define the refinement <id> of state <owner>. The refinement is a

[4] The type is either 'Send' or 'Receive'.

subdiagram and its definition ends with an activitysubdiagram statement:
e.g.: subdiagram("A18_2918","D15_288").

→ **time(<id>,<min>,<typical>,<max>).**
The minimum, typical and maximum execution times of the state <id> are <min>, <typical> and <max> respectively:
e.g.: time("A18_2160","200","400","600").

→ **transition(<id>,<state1>,<state2>).**
The diagram contains a transition <id> from <state1> to <state2>:
e.g.: transition("T18_2355",A18_2178","A18_2193").

A.4 Collaboration diagram

Element types:
O: object
E: message

Facts:

→ **belongsTo(<id>,<owner>).**
Object <id> is an instance of class <owner>:
e.g.: belongsTo("O18_17661","C18_537").

→ **collaborationdiagram(<id>,<name>).**
The project contains a collaboration diagram <id> with the name <name>:
e.g.: collaborationdiagram("D18_141","MAISAcollaboration").

→ **guard(<id>,<condition>).**
The message <id> has a guard condition <condition>:
e.g.: guard("E18_17703","B=10").

→ **message(<id>,<sender>,<receiver>).**
The diagram contains a message <id> from <sender> to <receiver>:
e.g.: message("E18_17769",O18_17661","O18_17748").

→ **msginfo(<id>,<name>,<number>).**
The diagram contains a message <id> with the name <name>,
the sequence number of this message is <number>:
e.g.: msginfo("E18_17769","newmessage","2").

→ **object(<id>,<name>).**
The diagram contains an object <id> with the name <name>:
e.g.: object("O18_192","instance1").

→ **predecessor(<id>,<number>).**
The message <id> has a predecessor with the number <number>:
e.g.: predecessor("E18_17769","1").

→ **stereotype(<id>,<type>).**
The stereotype of the message <id> is <type>:
e.g.: stereotype("E18_17688","local").

B An example of metrics calculation in a class diagram

B.1 The example diagram

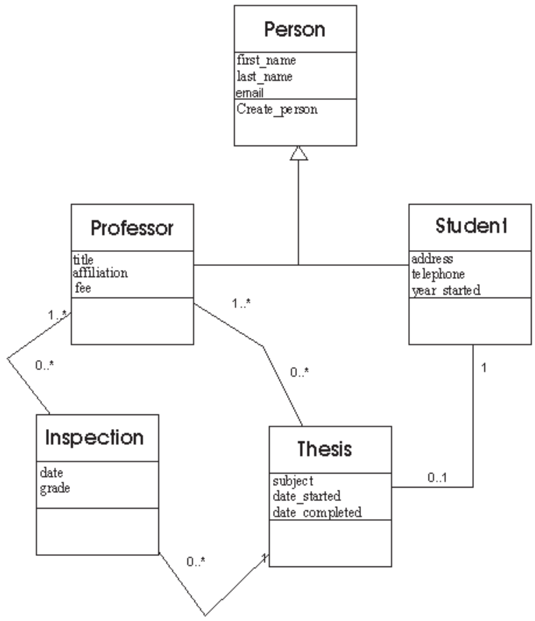

Figure 3: An example of a class diagram.

B.2 Facts generated from the example diagram

```
classdiagram("D18_11146","demo2").
class("C18_11800","Thesis").
attribute("C18_11800","A18_11695","subject").
attribute("C18_11800","A18_11730","date_started").
attribute("C18_11800","A18_11765","date_completed").
class("C18_11302","Person").
attribute("C18_11302","A18_11162","first_name").
attribute("C18_11302","A18_11197","last_name").
attribute("C18_11302","A18_11232","email").
method("C18_11302","M18_11267","Create_person").
publicMethod("M18_11267").
has("C18_11302","M18_11267").
constructor("M18_11267").
class("C18_11458","Professor").
attribute("C18_11458","A18_11353","title").
attribute("C18_11458","A18_11388","affiliation").
attribute("C18_11458","A18_11423","fee").
class("C18_11614","Student").
attribute("C18_11614","A18_11509","address").
```

```
attribute("C18_11614","A18_11544","telephone").
attribute("C18_11614","A18_11579","year_started").
class("C18_11921","Inspection").
attribute("C18_11921","A18_11851","date").
attribute("C18_11921","A18_11886","grade").
extends("C18_11458","C18_11302").
extends("C18_11614","C18_11302").
association("S18_12074","C18_11800","0..*","C18_11458","1..*").
association("S18_11972","C18_11921","0..*","C18_11458","1..*").
association("S18_12023","C18_11800","1","C18_11921","0..*").
association("S18_12125","C18_11800","0..1","C18_11614","1").
```

B.3 Example results calculated from the diagram

Note:
These results are produced by the command-line interface of the MAISA tool. Hence the formatting is crude. The basic concept should, however, be evident.

```
MAISA report by JPG
August 15, 2000 2:28:10 PM GMT+03:00
==========================================================
Class diagram demo2
   Number of packages: 0
   Number of public methods: 1
   Number of public methods (incl. inherited): 1
   Number of all methods: 1
   Number of all methods (incl. inherited): 1
       min= 0, in class(es):
  Thesis
          Professor
          Student
          Inspection
       max= 1, in class(es):
          Person
       avg= 0.2
   Number of attributes (incl. inherited): 20
       min= 2, in class(es):
          Inspection
       max= 6, in class(es):
          Professor
          Student
       avg= 4.0
   Number of generalizations: 2
   Number of interfaces: 0
   Number of associations: 4
   Depth of inheritance tree: 1
----------------------------------------------------------

This class diagram contains the following 5 classes:

Class Thesis
```

```
=>This class has 3 attributes:
    Attribute subject
    Attribute date_started
    Attribute date_completed
=>This class has no methods

Class Person
=>This class has 3 attributes:
    Attribute first_name
    Attribute last_name
    Attribute email
=>This class has 1 method:
    Public method Create_Person()
      Method is a constructor

Class Professor
=>This class inherits class Person
=>This class has 6 attributes:
    Attribute title
    Attribute affiliation
    Attribute fee
    Attribute first_name
    Attribute last_name
    Attribute email
=>This class has no methods

Class Student
=>This class inherits class Person
=>This class has 6 attributes:
    Attribute address
    Attribute telephone
    Attribute year_started
    Attribute first_name
    Attribute last_name
    Attribute email
=>This class has no methods

Class Inspection
=>This class has 2 attributes:
    Attribute date
    Attribute grade
=>This class has no methods
--------------------------------------------------
Association S18_12074
   => Role1 Thesis: 0..* -- Role2 Professor: 1..*
Association S18_11972
   => Role1 Inspection: 0..* -- Role2 Professor: 1..*
Association S18_12023
   => Role1 Thesis: 1 -- Role2 Inspection: 0..*
Association S18_12125
   => Role1 Thesis: 0..1 -- Role2 Student: 1
```