Beagle

Design and Architecture

Annika Berger, Joshua Gleitze, Roman Langrehr,
Christoph Michelbach, Ansgar Spiegler, Michael Vogt

10th of January 2016

at the Department of Informatics
Institute for Program Structures and Data Organization (IPD)

Reviewer:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor:	M.Sc. Axel Busch
Second advisor:	M.Sc. Michael Langhammer

Karlsruher Institut für Technologie
Fakultät für Informatik
Postfach 6980
76128 Karlsruhe

Contents

List of Figures ii

Abbreviations iii

1 Architectural Overview 1
1.1 Extension Points . . . 2

2 Beagle's Knowledge: The Blackboard 5
2.1 Measurable SEFF Elements . . . 6
2.2 Evaluable Expressions . . . 6

3 The Analysis 9
3.1 Measurement . . . 9
3.2 Analysers . . . 14
3.3 Final Judge . . . 17

4 Graphical User Interface 21
4.1 Design . . . 21
4.2 Control Flow . . . 21

5 Requirements Specification 25
5.1 Changes to the Software Requirements Specification . . . 25

Terms and Definitions 27

Bibliography 29

List of Figures

1.1 Beagle Core Component 2
1.2 Beagle Core Overview Class Diagram 3
1.3 Beagle Core Package Diagram 4

2.1 Abstraction Layers on the Blackboard 6
2.2 Blackboard Class Diagram 7
2.3 Evaluable Expression Class Diagram 8

3.1 Controller classes 10
3.2 Pseudocode of Beagle Controller#perform Analysis() 11
3.3 Sequence diagram for Analysis Controller#perform Analysis() 12
3.4 Measurement Class Diagram 15
3.5 Sequence diagram for Measurement Controller#can measure() 16
3.6 Sequence diagram for Measurement Controller#measure() 16
3.7 Sequence diagram for Measurement Result Analyser#contribute() . . . 17
3.8 Sequence diagram for Proposed Expression Analyser#contribute() . . . 18
3.9 Sequence diagram for Final Judge#judge() 19

4.1 GUI Class Diagram 22
4.2 PCM Translation Class Diagram 23

Abbreviations

API Application Programming Interface

CTA Common Trace API

GUI Graphical User Interface

PCM Palladio Component Model

RDIA Resource Demanding Internal Action

SEFF service effect specification

SRS Software Requirements Specification

1 Architectural Overview

This chapter gives an introduction to Beagle's high level design. The following chapters will describe conceptual details of different subsystems. For the specification of single types, please refer to Beagle's Javadoc documentation [Berger et al., 2016].

Beagle consists of a core component and interfaces to external components. Components may depend on information provided by another component, but their internal logic works strictly independently. Communication takes exclusively place through the core component. The following are Beagle's key components and interfaces:

Core Component (Mediator Pattern)

In order to manage and synchronise the requests and execution of different jobs, Beagle is controlled by a core component. The core component conducts the order of executable services, distributes information and is responsible for class instantiation. It contains all management logic required to perform dynamic analysis on software and will offer a parameterised Palladio Component Model (PCM) at the end of a successful execution. It does, however, not contain any logic to actually run measurements or analyse parametric dependencies. Instead, it depends on other components providing this functionality.

Measurement Tool

Measurement Tools are responsible for all kinds of measurements that are needed to get the execution time of Resource Demanding Internal Actions(RDIAs), branch decisions of service effect specification (SEFF) Branches and repetitions of SEFF Loops in regard to a certain parametrisation. An adapter instructing Kieker will be the first class to implement this interface.

Measurement Result Analyser

Based on the measurement results, Measurement Result Analysers will suggest Evaluable Expressions that describe the parametric dependencies found in the results. Typical implementations are regression tools.

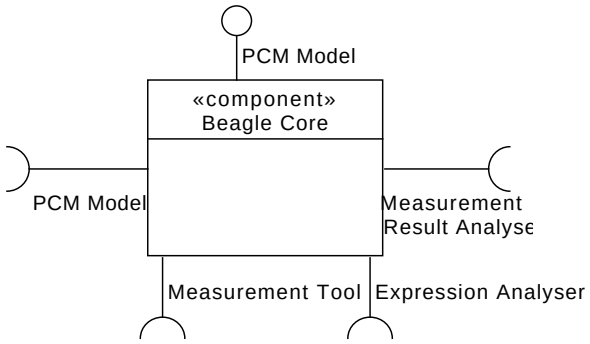

Figure 1.1: The Beagle Core Component and its interfaces.

Proposed Expression Analyser

Proposed Expression Analysers try to improve the results of Measurement Result Analysers. They usually try to combine proposed expressions to build a better one (in terms of Beagle's Fitness Function). As different regression approaches usually have different advantages and shortcomings, combining their results may produce a more accurate expression because it contains "more parts of the truth" [Krogmann, 2011]. The genetic approach described by Krogmann would be implemented as a Proposed Expression Analyser.

Final Judge

This class is responsible to decide which proposed evaluable expression describes the measured results best and will be annotated in the PCM. It also decides if more measurements should be done and when the final solution is found.

1.1 Extension Points

Measurement Tools, Measurement Expression Analysers and Proposed Expression Analysers are connected to Beagle via Eclipse Extension Points. This allows a flexible configuration of the used plugins after compilation. Development of the plugins can take place independently (because the Application Programming Interface (API) is fixed) and which plugins users install can depend on the software they have available.

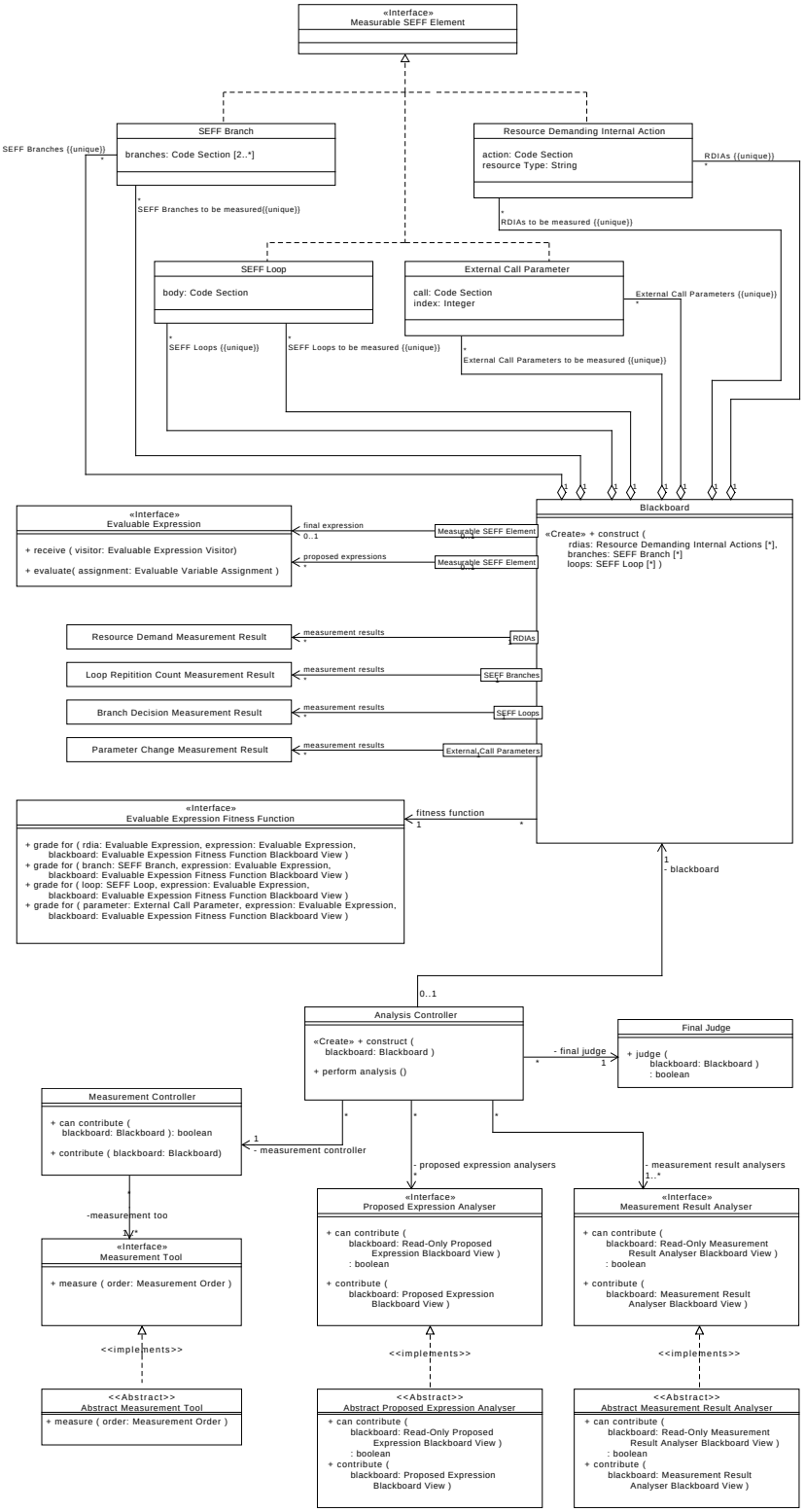

Figure 1.2: Class overview of Beagle Core. For details about specific classes, refer to Beagle's Javadoc [Berger et al., 2016].

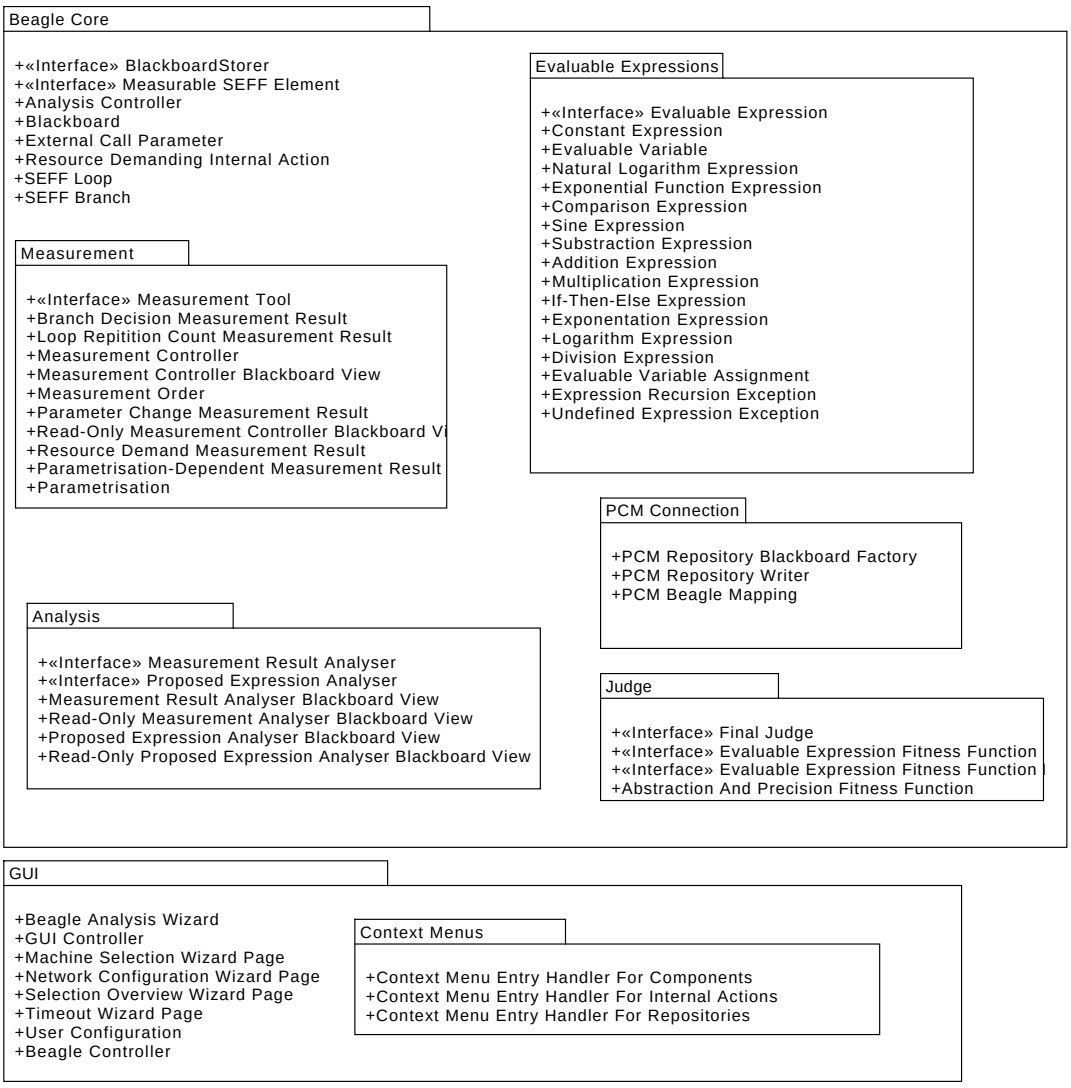

Figure 1.3: Beagle Core's separation into Packages.

2 Beagle's Knowledge: The Blackboard

The Beagle Core uses the Blackboard as described in [Buschmann et al., 1996, Chapter 2.2]. The blackboard is the central data storage which stores all available SEFF elements, which SEFF elements should be measured, all measurement results and the final expression for each measurable SEFF element. It also stores individual information for each Measurement Tool, Measurement Result Analyser, Proposed Expression Analyser and the Final Judge (in the this section: "the tools"), because these tools are not allowed to hold status information. The fact that all status information are stored on the Blackboard allows the serialisation of an analysis, as all elements on the blackboard are serialisable.

The blackboard design also allows communication between the tools, without dependencies between them: Each tool contributes his knowledge to the blackboard, when it can do so and may use information of other tools, but don't need to know about the tool, that contributed them. This enables adding and removing of tools without adapting the Beagle Core and developing them separately. Additional, it is easy to compare different combinations of tools to analyse the software.

As it makes no sense to run a tool that performs regression of the measurement results before any measurements have been made, the different kind of tools have a different priority. Therefore the vocabulary of the blackboard is divided in different layers and the tools can only see some of them. On the bottom layer are the code sections and all tools can access them, on the highest level are the final expressions and only the final judge can access them. In between are the Measurement Result Analysers. They can access access the measurement results (and the code sections). On top of them are the Proposed Expressions Analysers, which can additionally access all already proposed Evaluable Expressions.

The tools contribute content of their highest level or the level above that: Measurement Tools contribute measurement results, Measurement Result Analysers and Proposed Expressions Analysers contribute "proposed" Evaluable Expressions and the Final Judge the "final" Evaluable Expressions, which will be added to the PCM.

Each kind of tool has a different priority, contrary to their access level: The Measurement Tools have the highest priority. Only when no Measurement Tool can contribute something, a Measurement Result Analysers is invoked and so on. This ensures, that the layers a tool depends on, have already been filled with the best, possible content. E.g. when a Proposed Expressions Analyser is invoked, all tools that could contribute

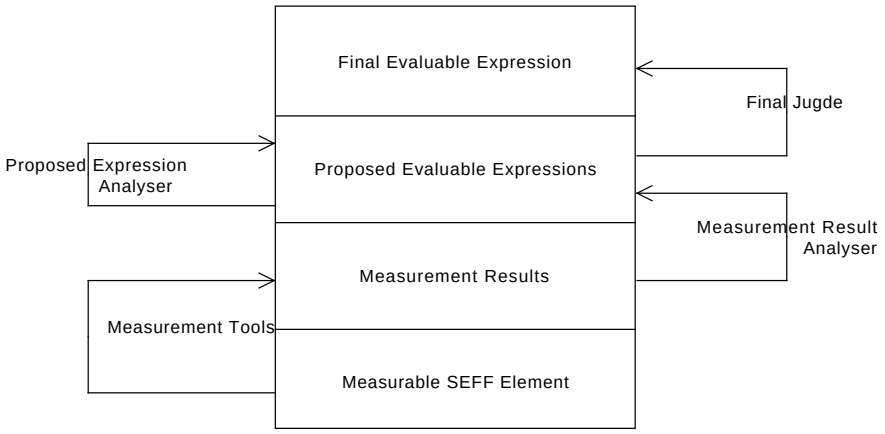

Figure 2.1: Abstraction Layers on the Blackboard

content for it and do not depend on the Proposed Expressions Analyser's kind of results, the "proposed" evaluable expressions, have already done so.

Typical Measurement Result Analysers are regression tools. Proposed Expressions Analyser are, fore example, a tool that takes the average of all available Evaluable Expressions or that builds new ones for an genetic programming approach.

This concept is supported by the Blackboard views: To ensure that each tool – except the Final Judge – can access only certain layers, they never get access to the blackboard. They only get a Blackboard View which provides the subset of the Blackboard's methods they are allowed to use. Each view can navigate to the blackboard and each method on the view delegates to the equivalent method on the blackboard. The outcome of this is, that each tool can only use the blackboard's method, it is allowed to use.

2.1 Measurable SEFF Elements

The blackboard distinguishes between measurable SEFF elements and SEFF elements in general. Tools access the set they need through their corresponding views.

2.2 Evaluable Expressions

Figure 2.2: The Blackboard and its views. The accessor methods are explicitly stated to visualise the view's usage.

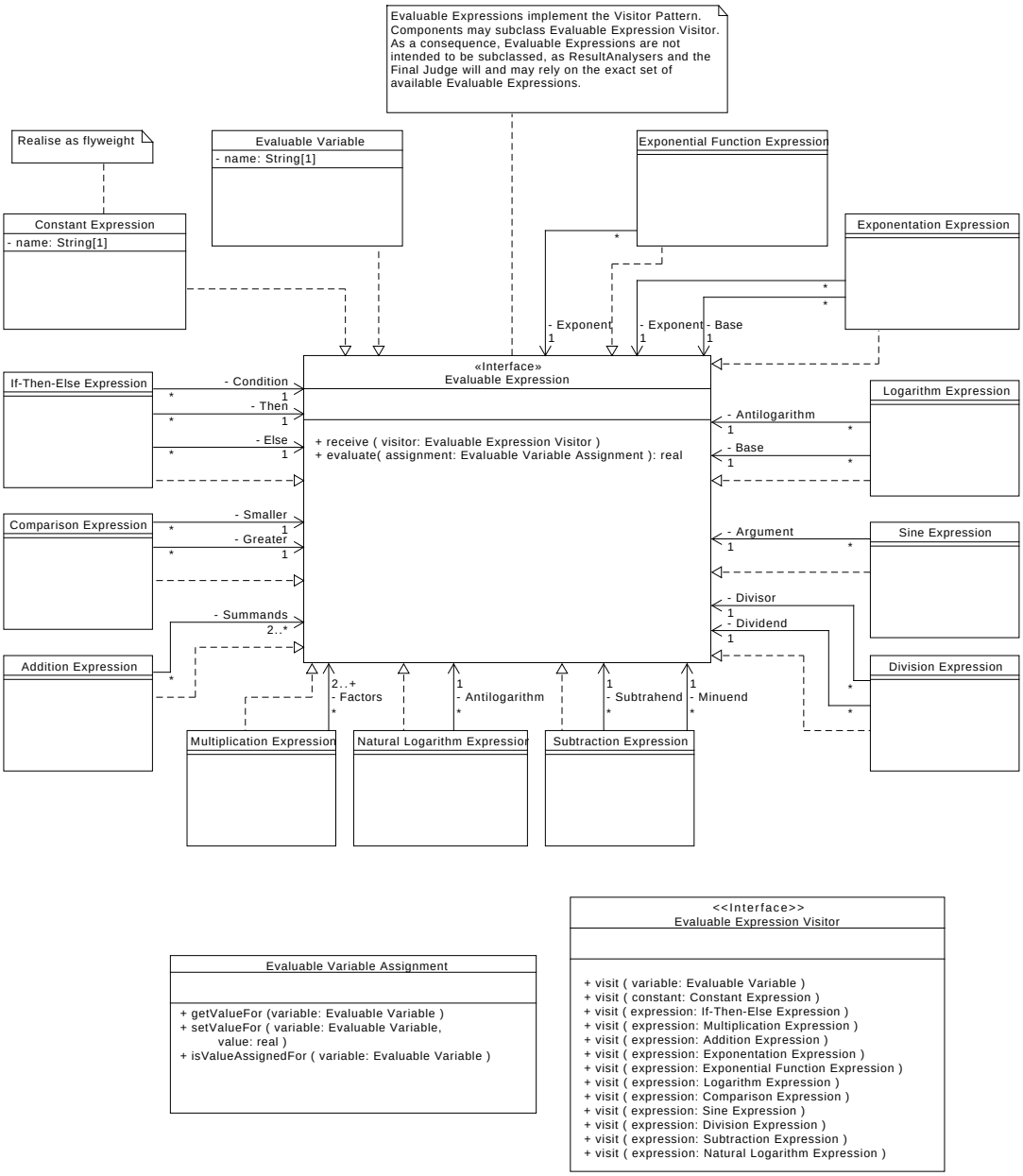

Figure 2.3: The Evaluable Expression Interface and its implementations realise the Visitor pattern.

3 The Analysis

At Beagle's core, the Analysis Controller controls all analysis activity by instructing the Measurement Controller, the Measurement Result Analysers, the Proposed Expression Analysers and the Final Judge. While not contributing itself, it is charge of all control flow during analysis.

Analysis Controller#perform Analysis performs a complete analysis, starting by measuring the examined software, and continuing to analyse until the Final Judge reports that the analysis is finished. There is always at most one Measurement Tool, Measurement Result Analyser, Proposed Expression Analyser or Final Judge having the control flow at any given moment during the execution of Analysis Controller#perform Analysis ("the analysis loop").

An iteration of the analysis loop starts by asking the Measurement Controller whether it wants to conduct measurements for the current blackboard state—which will usually be the case if there is something not yet measured—, and if so, calling its #measure method. The Measurement Controller will then instruct the Measurement Tools to measure. Usually, it will tell every tool to measure all new SEFF Elements.

After that, the main loop invokes one arbitrary chosen Measurement Result Analyser reporting to be able to contribute to the current blackboard state. This analyser may then propose expressions describing the parametric dependencies of SEFF Elements' measurement results. If there is no such analyser, an arbitrary chosen Proposed Expression Analyser reporting to be able to contribute will be invoked. It may then propose more expressions based on the ones the ones Measurement Result Analysers added to the blackboard, usually trying to improve them. If there Final Judge will be called. It decides whether enough information has been collected and Beagle can terminate. If this is the case, it also creates or selects the final result for each item that has proposed results.

The analysis loop will then be repeated until the Final Judge was called and its #judge method returned true. Figure 3.2 sketches the procedure.

3.1 Measurement

Measurement Tools are responsible for actually running the software examined by Beagle. They'll usually instrument the source code, execute it and collect the results from the instrumentation points. Because this competence is fundamentally different from

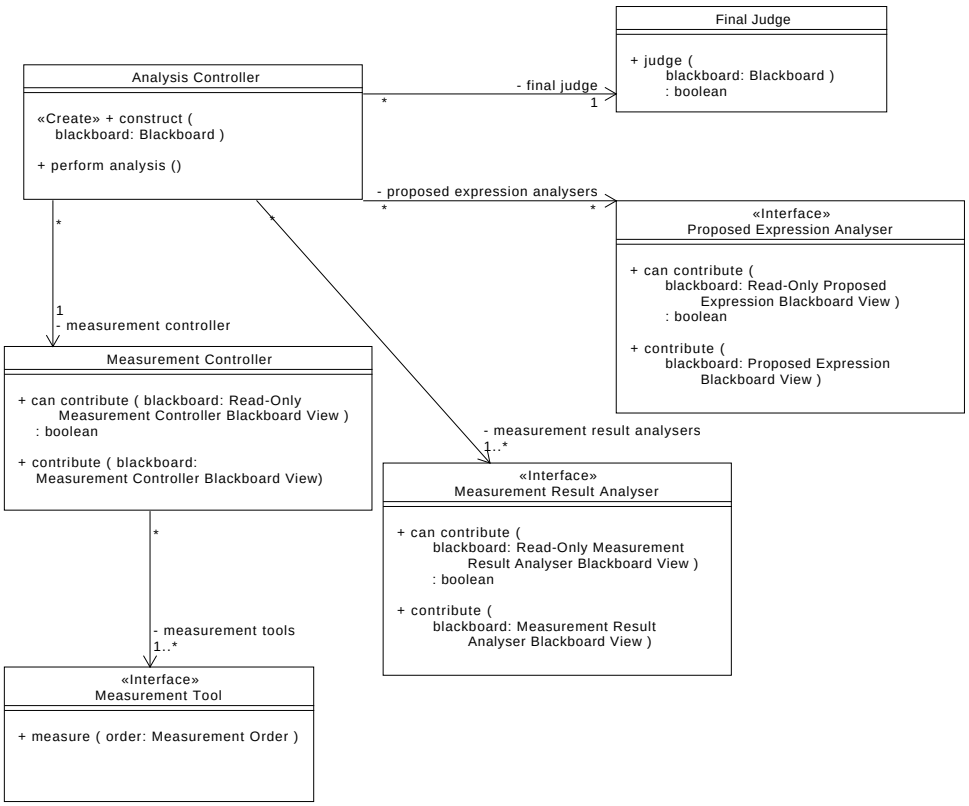

Figure 3.1: UML class diagram of the controller classes.

```
finished := false

romraBlackboardView := Read-Only Measurement Result Analyser Blackboard
View.construct(blackboard)
mraBlackboardView := MeasurementController Blackboard View.construct(blackboard)
ropeBlackboardView := Read-Only Proposed Expression Analyser Blackboard
View.construct(blackboard)
peBlackboardView := Proposed Expressions Blackboard View.construct(blackboard)

while ¬finished do
    if measurement controller.can measure(blackboard) then
        measurement controller.measure(blackboard)
    else if ∃ analyser ∈ measuremet result analysers :
    analyser.can contribute(romraBlackboardView) then
        analyser.contribute(mraBlackboardView)
    else if ∃ analyser ∈ proposed expression analysers :
    analyser.can contribute(ropeBlackboardView) then
        analyser.contribute(peBlackboardView)
    else
        finished := final judge.judge(blackboard)
    end
end
```

Figure 3.2: Beagle Controller#perform Analysis() in pseudocode.

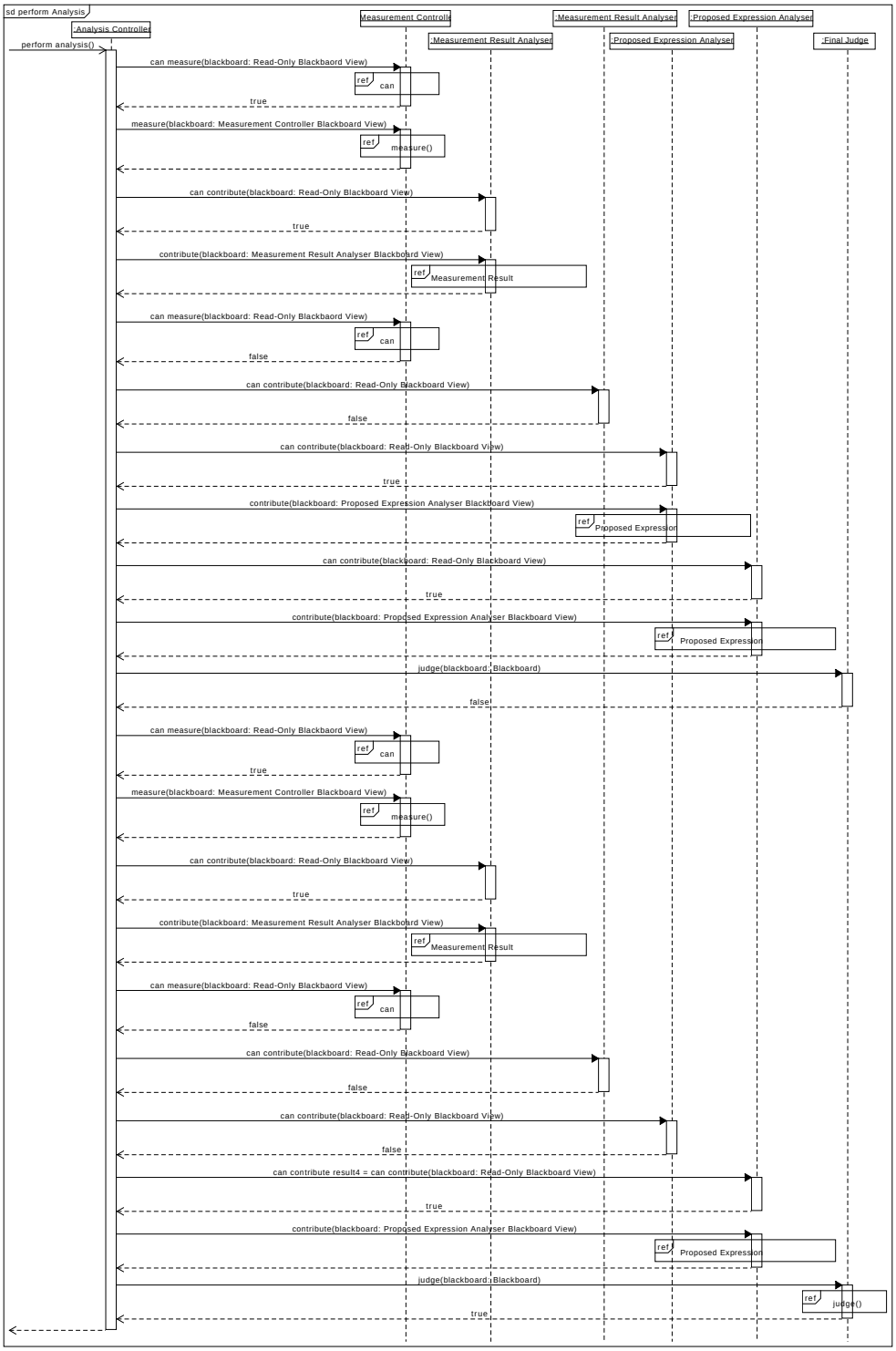

Figure 3.3: An exemplary run of Analysis Controller#perform Analysis() with a concrete setup of Analysers. Various method calls will be shown in following diagrams.

analysing SEFF Elements, it has a different vocabulary and different business objects. The Measurement Controller translates between Measurement Tools and Beagle Core. For instance, measuring code does not involve any control flow abstraction like the SEFF, but operates directly on source code. On the other hand, the Analysers require already interpreted results to propose expressions for them.

When the Blackboard is created by PCM Repository Blackboard Factory, all Measurable SEFF Elements get a Code Section pointing to where they are defined in the source code, as read in from the PCM files. Based on this information, the Measurement Controller creates a Measurement Order. This order describes the source code sections Beagle needs information about in the language of the Measurement Tools. The description is analogous to the Instrumentation Points described in [Krogmann, 2011], 5.10.2:

- Before and after each external call, parameter value sections will be put on the Measurement Order to capture the transferred and returned parameter's state.
- An execution section will be added to the Measurement Order for each branch in a SEFF condition (represented by one SEFF Branch) to determine which branch was executed.
- An execution section will be created for the body of every SEFF loop, to count how often it will be executed.
- For each Resource Demanding Internal Action, a resource demand section will be created to measure the resource demands of that action and type.

The order also contains Launch Configurations that can be used to run the examined software's code. These launch configurations are provided by the user. They are not Eclipse's launch configurations, although an adapter to them would make a good implementation of Launch Configuration.

When receiving the order, Measurement Tools can instrument the source code based on this information. Executing the instrumented code (through the Launch Configurations) hence gives them information about the executed code sections. They give this information back in the form of a list of Measurement Events ordered by the time they occurred. The Measurement Controller has no expectations about what a Measurement Tool can measure, tools are allowed to support only a subset of the instrumentation instructions on the Measurement Order. A measurement can also lead to no Measurement Events at all, for example because of a Launch Configuration only running code the user is not interested in.

The Measurement Controller then uses a Measurement Event Parser to create Parameterisation-Dependent Measurement Results that Measurement Result Analysers can operate on out of the captured Measurement Events:

- The last consecutive occurrences of Parameter Value Captured Events from the same code section sets the parameterisation for all following results. If the events occur after an External Call, they are used to create a Parameter Change Measurement Result.
- The sum of consecutive executions of a SEFF Loop's body is used to create a Loop Repetition Count Measurement Result.
- Which code section was executed determines the branch index for the created Branch Decision Measurement Result.
- A Resource Demand Captured Event can directly be translated to a Resource Demand Measurement Result.

The Measurable SEFF Elements the created Parameterisation-Dependent Measurement Results will be assigned to are determined by the code sections the Measurement Events occurred for.

The measurement order also provides a Parameter Characteriser, Measurement Tools shall use to characterise parameters in parameter value sections. This functionality, including the Parameterisation and Parameter Characterisation classes, is intentionally only sketched in the current design. Implementing them is out of this project's scope and will only be targeted if time is left after implementing Beagle's mandatory requirements. However, they are included in the class diagram to show that the functionality will seamlessly integrate into the design.

The first implementation of a Measurement Tool will be an adapter to the Kieker performance monitoring framework.

3.2 Analysers

Measurement Result Analysers and Proposed Expression Analysers ("Analysers") are not limited in their functionality. There exist many different approaches to analyse parametric dependencies, all having their advantages and disadvantages. Especially if Krogmann's genetic programming approach is implemented (as a Proposed Expression Analyser), more, different Analysers may drastically improve the final result and the computation time needed to find it [Krogmann, 2011]. The Final Judge will always pick the best result (in regard of the Fitness Function) and no proposed expression is ever deleted, so the result can only get better by adding more analysers.

Analysers have to carefully implement their #can contribute() methods. There intentionally exist very little restrictions to when and how often an Analyser may run to not restrict the variety of approaches that can be implemented. However, Analysers have to make sure that the analysis can terminate. This is especially crucial with multiple

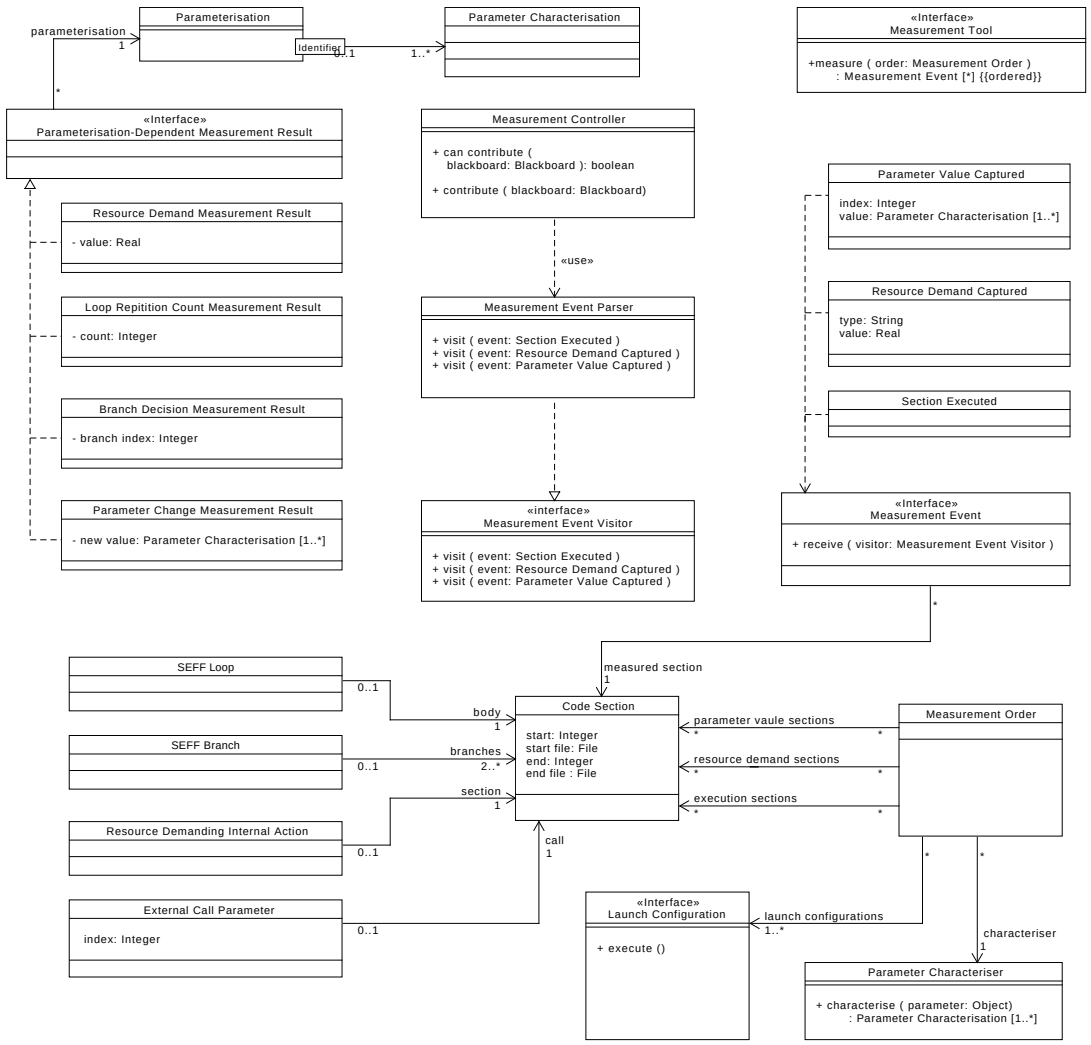

Figure 3.4: Classes involved in Measurement. Classes on the left hand side belong to Beagle Core's View, classes on the left hand to Measurement Tools' view. Classes in the centre are used by both sides or translate between the views.

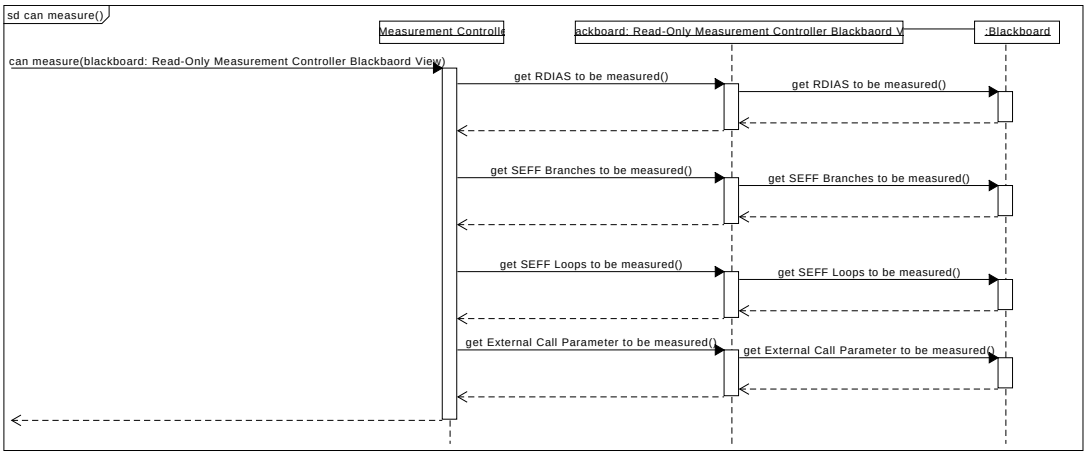

Figure 3.5: An exemplary run of Measurement Controller#can measure(), checking whether there are new SEFF Elements to be measured.

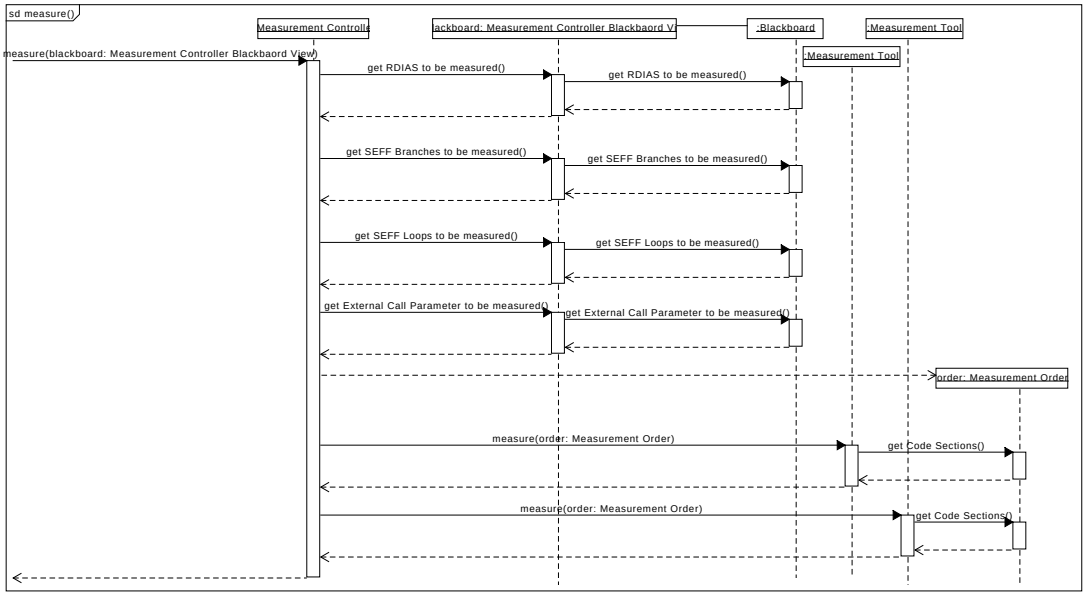

Figure 3.6: An exemplary run of Measurement Controller#measure, preparing the Measurement Order and commissioning it to the registered Measurement Tools.

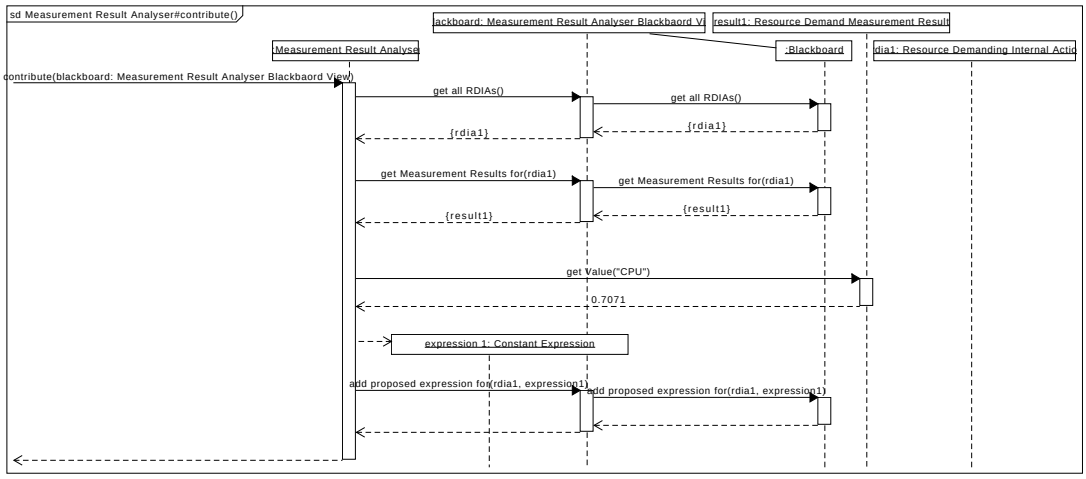

Figure 3.7: An exemplary run of Measurement Result Analyser#contribute() for a Measurement Result Analyser contributing the measurement results's values as Constant Expressions.

analysers, all constantly generating new results. It can be achieved by regularly not contributing, and hence letting the Final Judge decide from time to time. The Final Judge will not terminate the analysis if there is reasonable hope for a better result, so analysers are likely to be called again after returning false in their #can contribute method.

The first implementation will include a Measurement Result Analyser proposing all measurement results as a Constant Expression and a Proposed Expression Analyser proposing the average of all proposed Constant Expressions. These will still be useful when implementing more complex analysers like the genetic analyser.

3.3 Final Judge

The Final Judge is in charge of deciding when to end the analysis. As explained above, this is not trivial, as stopping it to soon will likely generate bad results. Therefore, it will apply a combination of heuristics, including how much the best and average result improved in the last analysis iterations, how long the analysis is running, how many optimal results have been found, etcetera. Prompting users is an option, too. There are only two cases in which the analysis will be ended immediately:

- All Measurable SEFF Elements have an optimal proposed Evaluable Expression on the Blackboard, or

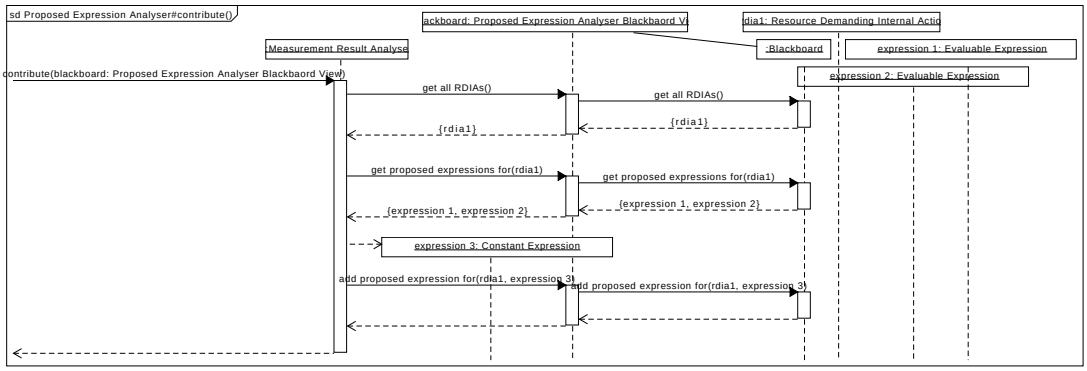

Figure 3.8: An exemplary run of Proposed Expression Analyser#contribute() for a Proposed Expression Analyser contributing the average of all Constant Expressions proposed for a SEFF Element.

- The Measurement Controller, all Measurement Result Analysers and all Proposed Expression Analysers returned false in their #can measure or #can contribute method in the last iteration of the analysis loop.

The Final Judge grades Evaluable Expressions based on a fitness function. Because analysers may use a fitness function to generate better results, the it is available to them from the Blackboard. The Final Judge will always pick the proposed expression graded best for the final expression. If there are multiple such expressions, is it undefined which one will be chosen.

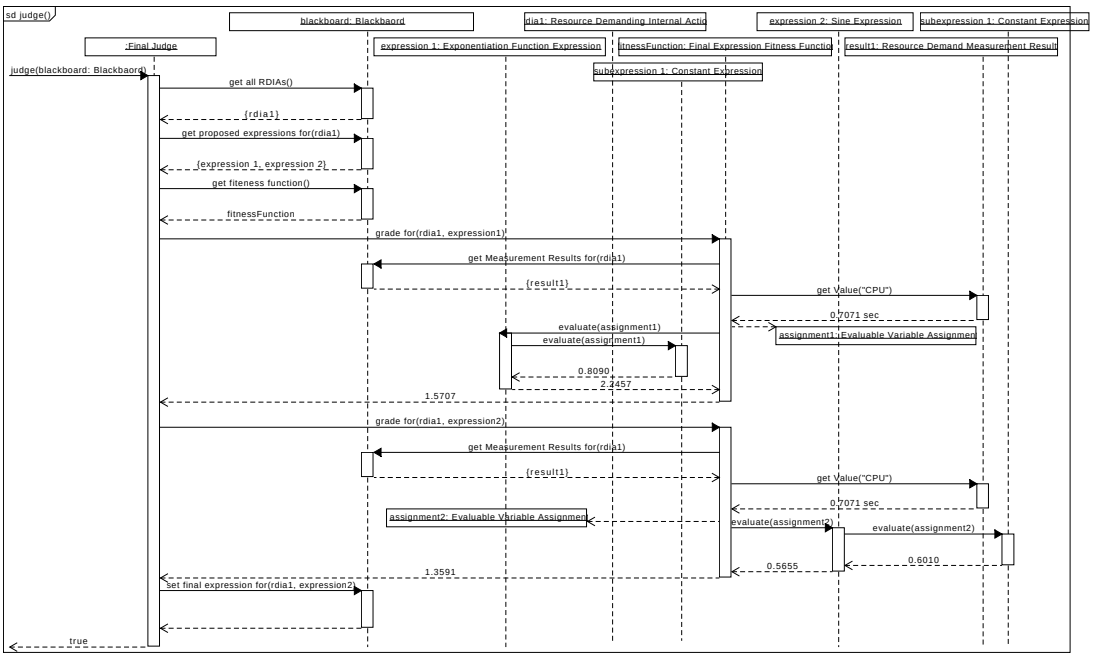

Figure 3.9: Exemplary run of Final Judge#judge(). Two expressions have been proposed for a Resource Demanding Internal Action having one Measurement Result of its CPU usage.

4 Graphical User Interface

4.1 Design

The GUI follows the Model-View-Controller paradigm. It accordingly clearly separates responsibilities: Displaying GUI parts, controlling these parts, setting up the model (Beagle Core), and the model itself. This leads to a clear flow of information and control. Information is stored either locally or in a single object every GUI class knows (the User Configuration) because only the user's settings need to be transported through all objects. The user always has control over the analysis because information can be transported from the dialog, though the GUI Controller object, to the Beagle Controller which can instruct the model.

4.2 Control Flow

GUI Controller is the heart of Beagle's GUI. It is responsible to start the GUI. The class controls the flow of actions on the GUI and gives control to Beagle Controller once the user started the analysis. Beagle Controller is commanded by GUI Controller to start, pause, continue, and abort the analysis.

There are three context menu entries added by Beagle with eclipse extension points: One for analysing a complete repository in the package explorer and the project explorer on the .repository and .repository_diagram files. Another one for analysing a single components in the repository diagram on the Basic Component Edit Parts. The last one is for analysing a single internal actions in the SEFF diagram for the Internal Action Edit Parts. Each of them has its own Command Handler class, which is handles clicks on the context menu entries.

The GUI Controller is created and called from actions originating from context menu clicks. The GUI stores all data obtained from the user in the User Configuration. Thus, the GUI Controller creates a User Configuration which it from now on will be associated with. It then creates a new Beagle Analysis Wizard which in return will create its Wizard Pages. All objects obtain the User Configuration to be able to directly provide their gathered information to it. Calling GUI Controller's open() method will display the GUI and show the Analysis Wizard, prompting the use for all information necessary to perform an analysis.

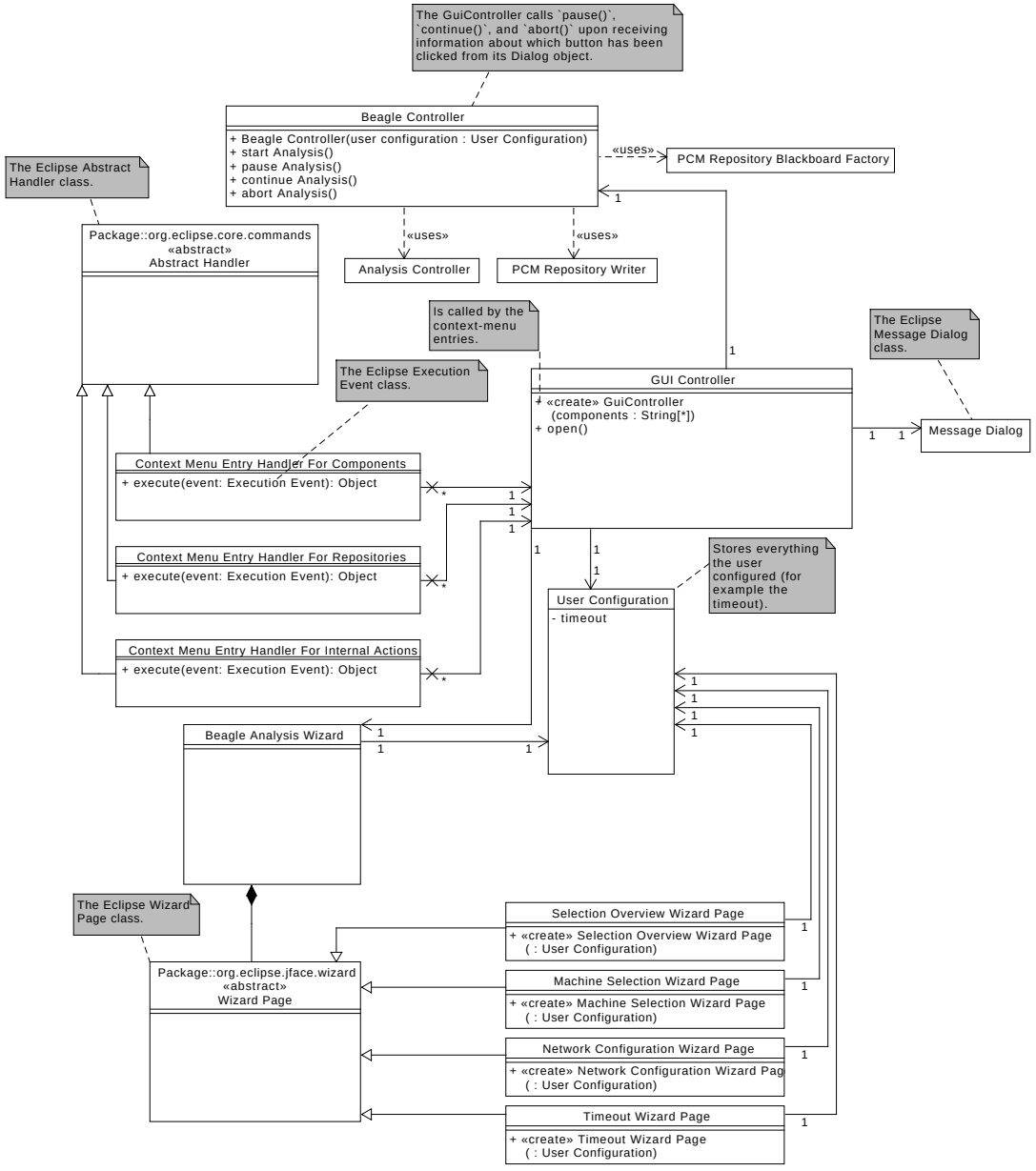

Figure 4.1: UML class diagram of the GUI classes.

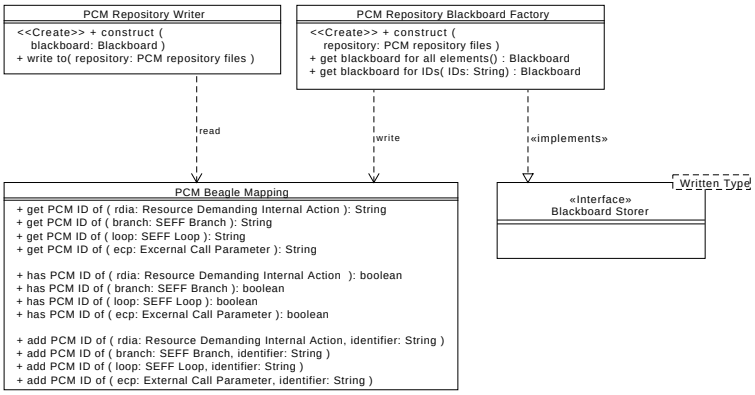

Figure 4.2: Classes used to read from and write to PCM repository files.

After the wizard has finished, control is returned to the GUI Controller which will now (concurrently) instruct the Beagle Controller to set up the Beagle Core Component. To do so, Beagle Controller sets up a Blackboard based on the information collected in the User Configuration. For example, it uses a PCM Repository Blackboard Factory to translate PCM objects to the affected Measurable SEFF Elements. The prepared Blackboard is then passed to Beagle Core's Analysis Controller, which will perform the actual analysis.

In the meantime, GUI Controller opens a dialog providing the user with information about the analysis and options to pause, continue or abort the analysis. GUI Controller bequeaths these calls the corresponding methods of Beagle Controller, which handles the Beagle Core. Once the Analysis Controller has finished the analysis, the Beagle Controller uses a PCM Writer to write the results back to the PCM files.

5 Requirements Specification

Beagle's software design directly follows the Software Requirements Specification (SRS) [Berger et al., 2015]. Most design decisions are proposed to fulfil mandatory requirements while allowing to easily add optional requirements. Section 5.1 describes changes to the Software Requirements Specification (SRS) that proved necessary while designing Beagle. Section **??** describes how Beagle's requirements are reflected in its design. However, some mandatory criteria have changed concerning the Common Trace API (CTA) .

5.1 Changes to the Software Requirements Specification

The Common Trace API

The CTA was planned to be used by Beagle to communicate with measurement software (/B10/, /F40/). While starting to investigate the API, it became apparent that it does not offer the expected functionality. The CTA is designed to return measurement results on method level, whereas Beagle's measurements need to be performed on sub-method (statement) level. Furthermore, there is no possibility to instrument source code or to control measurement software. The CTA can only return measurement results. Beagle's Measurement Tools will now be directly connected to specific measurement software like Kieker without the CTA as intermediary.

Accordingly, the following modifications where made to the SRS:

/B10/ Results are not transferred through the CTA.

/F40/ Has been removed without substitution.

/T30/ Has been removed without substitution.

/T210/ The CTA will not be tested.

Measurement Timeout

The measurement timeout functionality described in /F50/ and /F60/ is a valuable function to Beagle. However, it is not required to successfully use Beagle for the purpose described in Chapter 1 of the SRS. Denoting it as mandatory was a mistake.

Hence, the following modifications where made to the SRS:

/F50/ was moved to /OF70/

/F60/ was moved to /OF80/

Layout

Further minor changes have been made to the SRS, fixing broken links and layout issues.

Terms and Definitions

Common Trace API

an API developed by NovaTec GmbH for measuring the time, specific code sections need to be executed.

internal action

sequence of commands a component executes without leaving its scope (e.g. without calling other components). Part of a component's SEFF.

Kieker

> "a Java-based application performance monitoring and dynamic software analysis framework." [van Hoorn et al., 2012]

A measurement software Beagle aims to support.

Palladio Component Model

a domain-specific modelling language (DSL) used by Palladio.

> It is designed to enable early performance predictions for software architectures and is aligned with a component-based software development process. [Kounev, 2009]

resource demand

how much of a certain resource—like Central Processing Unit (CPU), Network or hard disk drive—a component needs to offer a certain functionality. In the PCM, resource demands are part of the SEFF. They are ideally specified platform independently, e.g. by specifying required CPU cycles, megabytes to be read, etc. If such information is not available, resource demands can be expressed platform dependent, e.g. in nanoseconds. In this case, a certain degree of portability can still be achieved if information about the used platforms' speed relative to each other is available.

SEFF condition

conditions (like Java's `if`, `if-else` and `switch-case` statements) which affect the calls a component makes to other components. Such conditions are—contrary to conditions that stay within an internal action—modelled in the component's SEFF.

SEFF loop

loops (like Java's `for`, `while` and `do-while` statement) which affect the calls a component makes to other components. Such loops are—contrary to loops that stay within an internal action—modelled in the component's SEFF.

service effect specification

description of a component's behaviour in the PCM. SEFFs contain information about the component's calls to other components as well as its resource demands. This information is used to derive the component's performance for simulation and prediction.

Bibliography

[Berger et al., 2015] Berger, A., Gleitze, J., Langrehr, R., Michelbach, C., Spiegler, A., and Vogt, M. (2015). Beagle—software requirements specification. Technical report, Karlsruhe Intitute of Technology Department of Informatics Institute for Program Structures and Data Organization (IPD).

[Berger et al., 2016] Berger, A., Gleitze, J., Langrehr, R., Michelbach, C., Spiegler, A., and Vogt, M. (2016). Beagle—javadoc. Technical report, Karlsruhe Intitute of Technology Department of Informatics Institute for Program Structures and Data Organization (IPD).

[Buschmann et al., 1996] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *Pattern Oriented Software Architecture - A System of Patterns*. Wiley.

[Kounev, 2009] Kounev, S. (2009). *Automated extraction of palladio component models from running enterprise Java applications*. PhD thesis, University of Wuerzburg.

[Krogmann, 2011] Krogmann, K. (2011). *Reconstruction of Software Component Architectures and Behaviour Models using Static and Dynamic Analysis*. PhD thesis, Karlsruhe Institute of Technology.

[van Hoorn et al., 2012] van Hoorn, A., Waller, J., and Hasselbring, W. (2012). Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd joint ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*, pages 247–248. ACM.