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

List of Algorithms

Abbreviations

API Application Programming Interface

CTA Common Trace API

GUI Graphical User Interface

SRS Software Requirements Specififcation, see [Berger et al., 2015]

1 Architectural Overview

The following chapter is divided into 4 sub-chapters:

- 1.1 Gives a short overview of Beagle's entire system, briefly presenting the whole design ideas as well as the subsystem structure and its out-most functionality.
- 1.2 Completes the internal description of Beagle's system from 1.1, describing the interaction of subsystem components.
- 1.3 Describes the communication process between Beagle and external tools such as measurement software or analyser software.
- 1.4 Explains why some components are developed as Eclipse Extension Points.

1.1 Overview of the entire system

Beagle is divided into components that are distinguished by high-level functionality and service. Components may depend on information provided by another component, but their internal logic works strictly independently. The composition of following components represents Beagle's architecture:

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 depends on proper functionality of the other components and will offer a parametrised PCM at the end of a successful execution.

Measurement Tool

The Measurement Tool is responsible for all kinds of measurements that are needed to get the execution time of Resource Demanding Internal Actions, branch decisions of 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.

Result Analyser

Based on the measurement results, the Result Analyser will suggest evaluable expressions that lead to a parametrisation of Resource Demanding Internal Actions, SEFF Branches or SEFF Loops.

Final Judge

This component is responsible to decide, which proposed evaluable expression fits best to the PCM. It also decides if more measurements should be done and when the final solution is found.

GUI (Model-View-Controller)

The GUI is not a necessary component that provides functionality for parametrisation. But it is necessary for providing interaction between Beagle and the user as the user may want to set up some features of Beagle.

1.2 Components' interaction

The interaction of Beagle's components is guided by the Beagle Core through the Blackboard Pattern. The blackboard contains SEFF specific information, describing what to measure, measurement results and evaluable expression annotations. Measurement Tools and Result Analysers have the possibility to decide for their own, whether they can contribute or not – depending on the information provided on the blackboard. Each component gets a different view of the blackboard, limiting its access to more than what is absolutely necessary. In order to unify the communication, Beagle Core provides its own classes (SEFF characteristics and Evaluable Expression).

1.3 Communication between Beagle and external tools

1.4 Extension Points

The `Measurement Tools`, `Result Analysers` and the `Final Judge` are connected to Beagle via Eclipse Extension Points.

This means they can be developed as separate eclipse plugins and specify in their `plugin.xml` file the classes, which represent the `Measurement Tools`, `Result Analysers` or a `Final Judge`. When there are multiple plugins with a `Final Judge`, the user has to select one in the GUI.

This concept has the advantage, that everybody can write his own or select some existing `Measurement Tools`, `Result Analysers` and `Final Judges` and just install them into his eclipse to use them.

2 Component: Beagle Core

2.1 Overview

2.2 The Blackboard

2.3 Controller Classes

The classes `Beagle Controller` and `Measurement Controller` manage the invocation of `Measurement Tool` or `Result Analyser` components. `Beagle Controller`*`#main`* is the main control loop, managing the control flow throughout Beagle's measuring and analysis activity. There is always exactly one `Measurement Tool`, `Result Analyser` or `Final Judge` running at any given moment during the execution of `Beagle Controller` *`#main`* ("the main loop").

An iteration of the main 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 decide which `Measurement Tools` to run. Usually it will tell every tool to measure as long as there is something left to be measured.

After that, the main loop invokes one arbitrary chosen `Result Analyser` reporting to be able to contribute. This analyser may then propose results for items that have measurement results. If there is no such analyser, the `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 main loop will then be repeated until the `Final Judge` was called and its *`#judge`* method returned **`true`**.

2.4 Evaluable Expressions

2.5 Conversion from and to Palladio

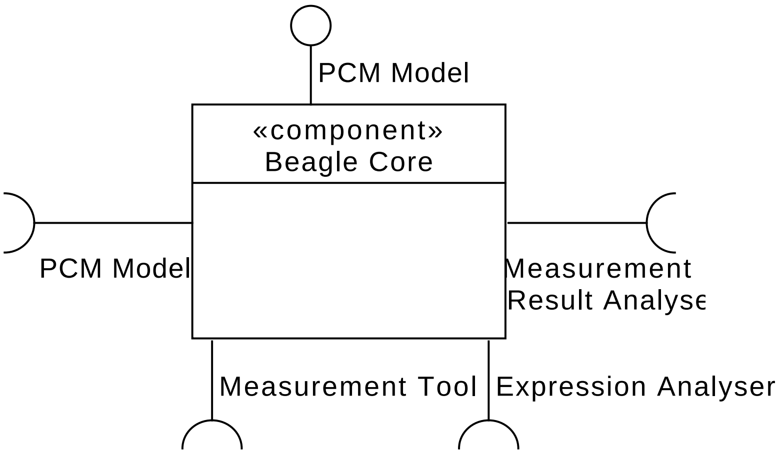

Figure 2.1: The Beagle Core Component and its interfaces

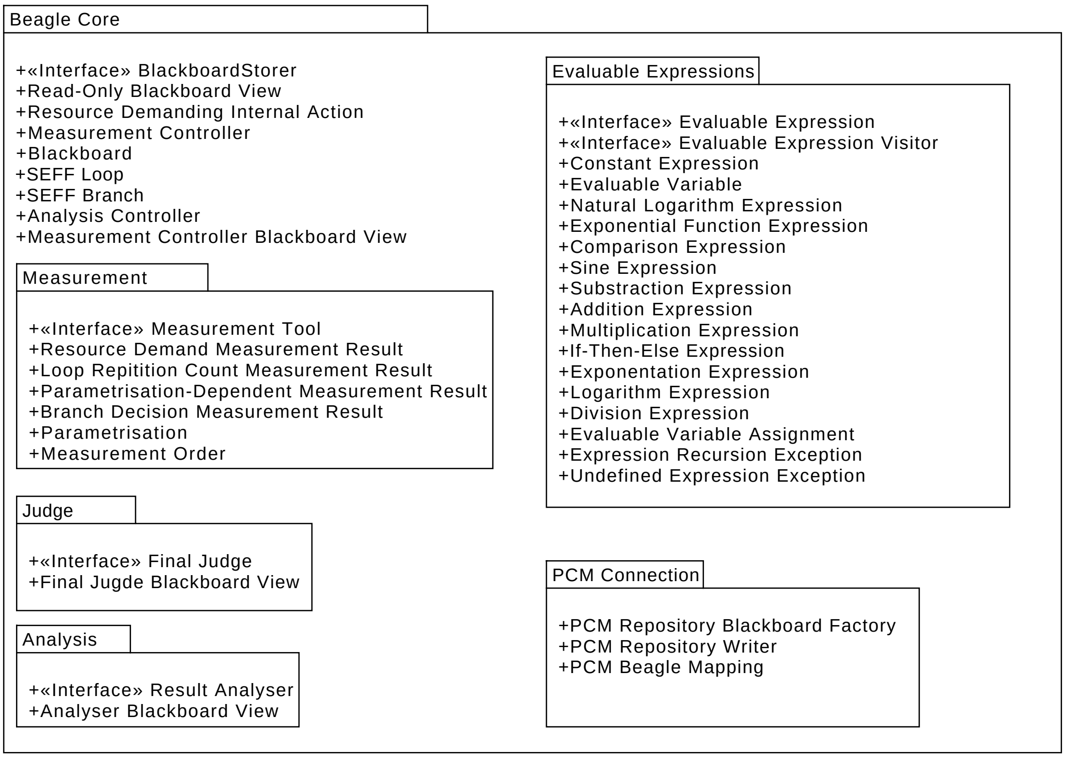

Figure 2.2: Beagle Core's separation into Packages

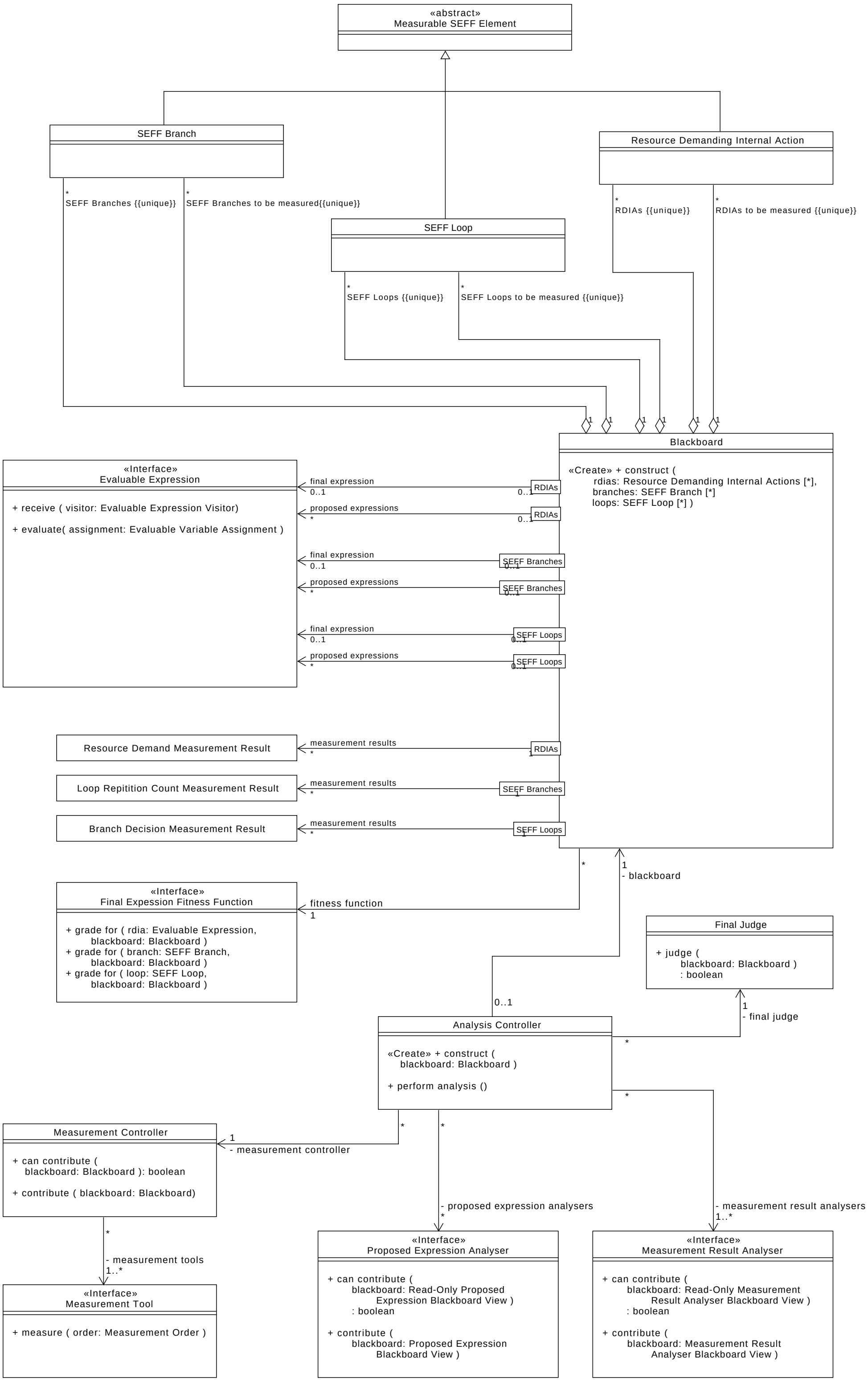

Figure 2.3: Class overview of Beagle Core

Figure 2.4: The Blackboard and its views

Algorithm 2.1 Beagle Controller#perform Anyalysis() in pseudocode

```
finished := false
readOnlyBlackboardView := Read-Only Blackboard View.construct(
    blackboard)
measurementControllerBlackboardView := MeasurementController Blackboard
     View.construct(blackboard)
measurementResultAnalyserBlackboardView := Result Analyser Blackboard
    View.construct(blackboard)
prosedExpressionsAnalyserBlackboardView := Proposed Expressions
    Blackboard View.construct(blackboard)

while(¬finished) do
        measurementsFinished := false
        while(¬measurementsFinished) do
                if(measurementController.can measure(
                    readOnlyBlackboardView)) then
                        measurementController.measure(
                            measurementControllerBlackboardView)
                else
                measurementResultAnalysersIterator =
                    measurementResultAnalysers.iterator()
                while(measurementResultAnalysersIterator.hasCurrent()
                    ∧¬measurementResultAnalysersIterator.current().
                    canContribute(readOnlyBlackboardView)) do
                        measurementResultAnalysersIterator.next()
                od
                if(measurementResultAnalysersIterator.hasCurrent())
                    then
                        measurementResultAnalysersIterator.current().
                            contribute(
                            measurementResultAnalyserBlackboardView);
                else
                        measurementsFinished := true
                fi
        fi
        proposedExpressionsAnalysersIterator =
            proposedExpressionsAnalysers.iterator()
        while(proposedExpressionsAnalysersIterator.hasCurrent() ∧¬
            proposedExpressionsAnalysersIterator.current().canContribute
            (readOnlyBlackboardView)) do
                measurementResultAnalysersIterator.next()
        od
        if(measurementResultAnalysersIterator.hasCurrent()) then
                measurementResultAnalysersIterator.current().contribute
                    (prosedExpressionsAnalyserBlackboardView);
        else
                finished := finalJudge.judge(blackboard)
        fi
od
```

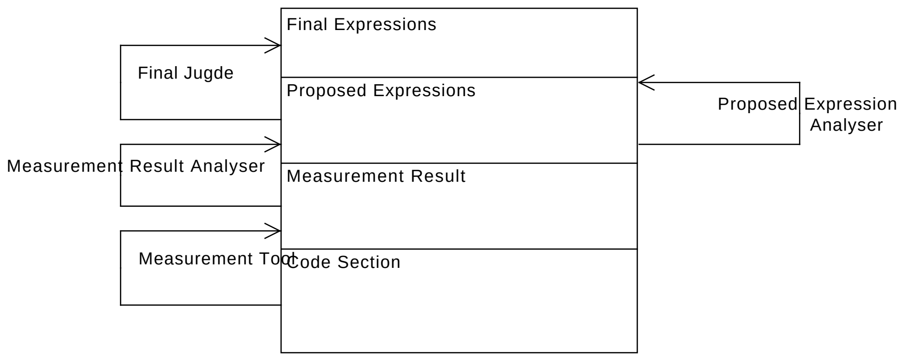

Figure 2.5: Abstraction Layers on the Blackboard

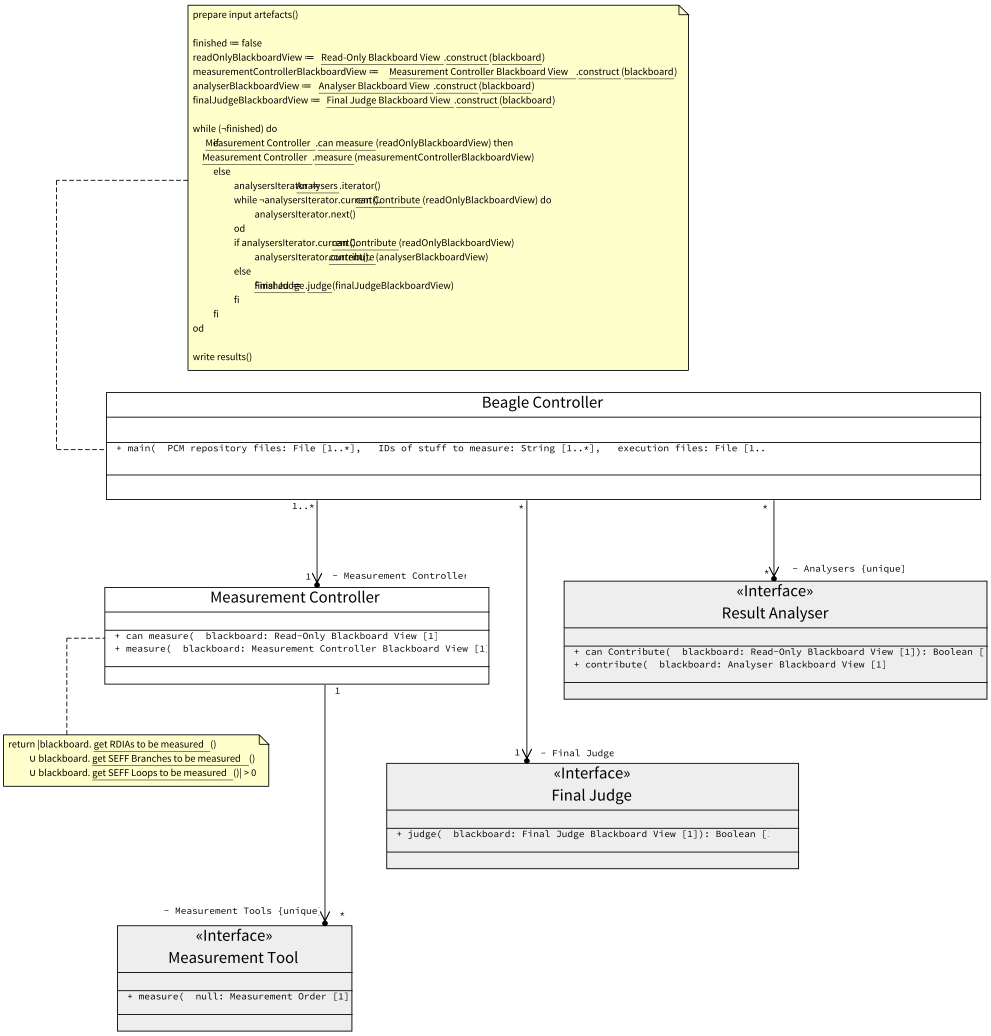

Figure 2.6: UML class diagram of the controller classes.

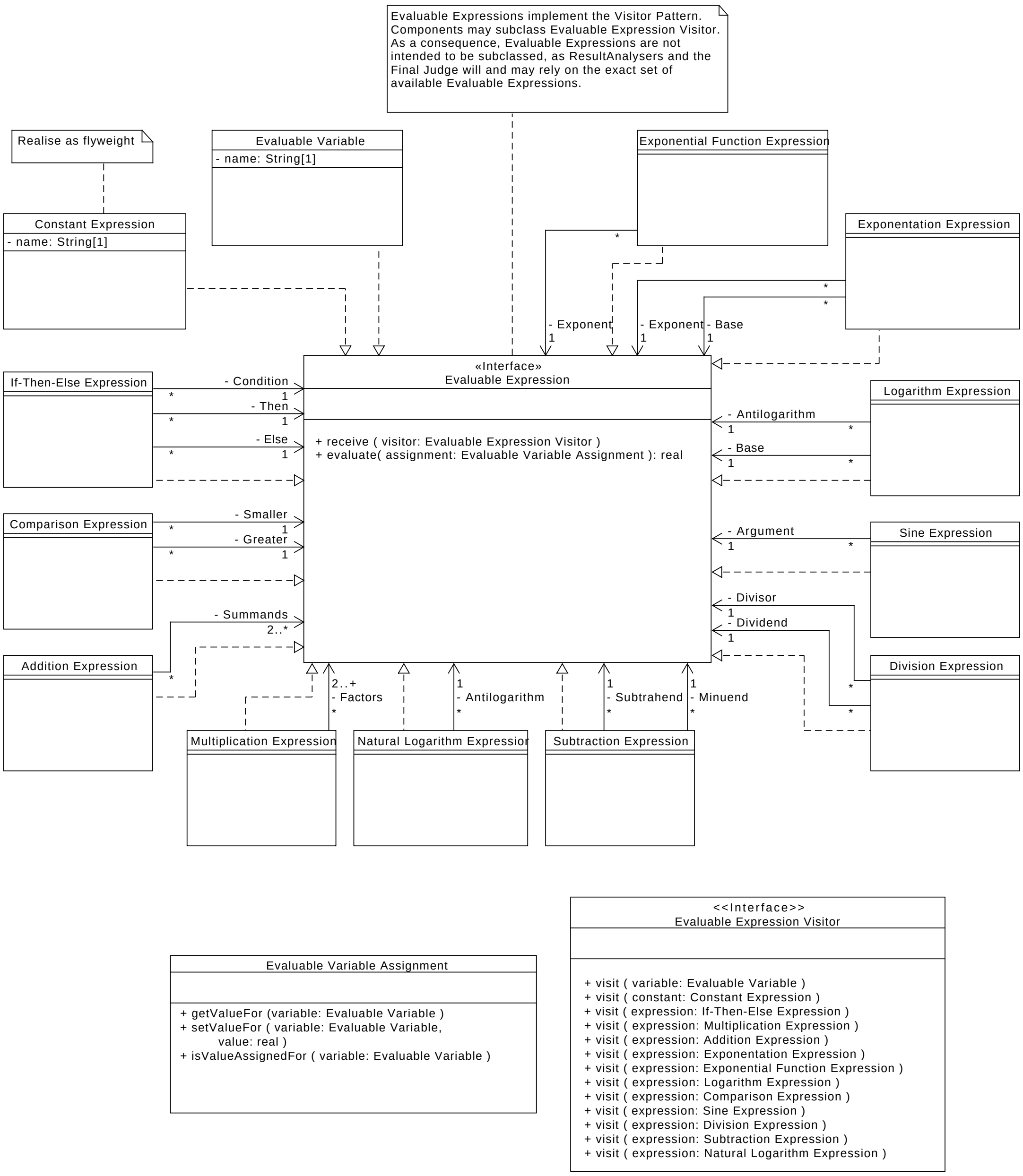

Figure 2.7: Evaluable Measurement Interface and its implementations

3 Component: Beagle GUI

3.1 Overview

3.2 Most Important Classes

3.2.1 GUI Controller

Graphical User Interface (GUI) Controller is the heart of the GUI. An object of it is created (and the method open() is called) to open the GUI. This class controls the flow of actions over the entire analysis but gives control to Beagle Controller after the user started the analysis.

3.2.2 Beagle Controller

Beagle Controller becomes active after the user started the analysis and unless the analysis is paused is active until the analysis is finished or aborted. Beagle Controller is commanded by GUI Controller to start, pause, continue, and abort the analysis.

3.3 Control Flow

The GUI Controller is called via actions originating from context menu clicks. An object of GUI Controller is created there and the method open() is called on it. It creates an object of User Configuration which it from now on will be associated with. It then creates a new Beagle Analysis Wizard object associated with the same User Configuration object. Beagle Analysis Wizard then creates its Wizard Pages which are also associated with the same User Configuration objects. That's because this object stores everything the user configures, so all pages of the Wizard need to read from and/or write on it.

After the wizard has finished, control is returned to the GUI Controller object which then opens a dialog providing the user with the information that the analysis is running and the options to pause/continue or to abort the analysis. To get the analysis running, pausing, continuing, or to abort it, the GUI Controller objects calls the corresponding methods of Beagle Controller when it receives information that the wizard has finished

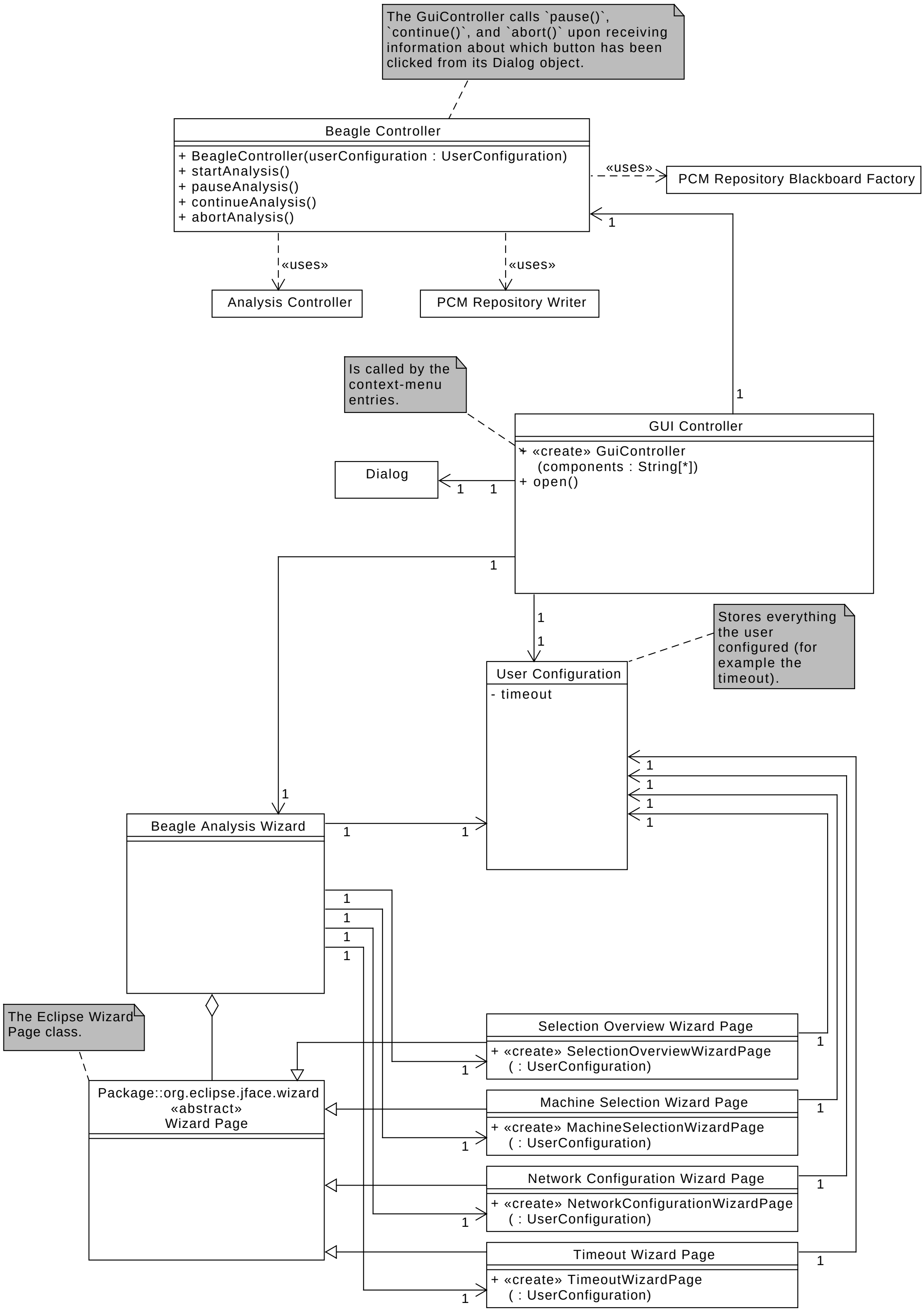

Figure 3.1: UML class diagram of the GUI classes.

or a button in the dialog the GUI Controller objects opens after the wizard has finished has been clicked.

3.4 Reasons for Chosen Design

This design has been chosen for the clarity of its flow of information and control. Information is stored either locally or in a single object every different object knows (the User Configuration object) because only the settings of the user 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 controls the analysis.

3.5 Chosen Design Patterns

4 Component: Measurement Tool

4.1 Reasons for chosen design

4.2 Adapter to Kieker

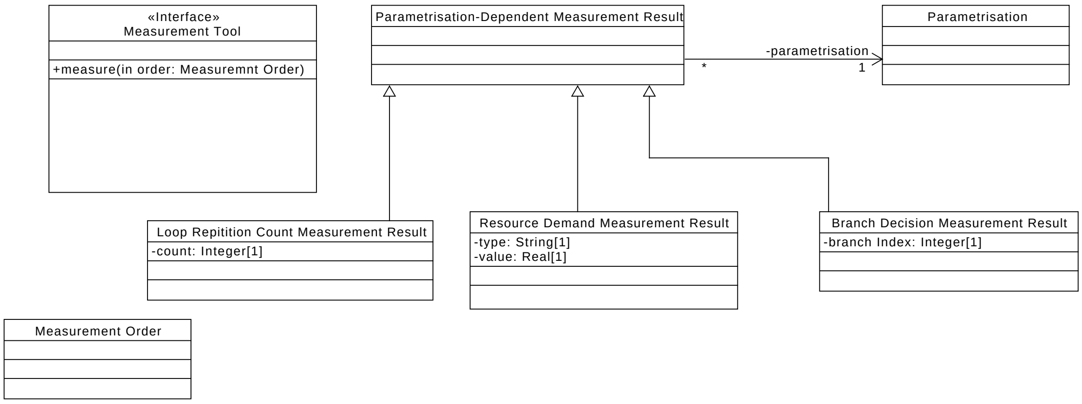

Figure 4.1: The Measurement Results

5 Component: Result Analyser

5.1 Reasons for chosen design

6 Component: Final Judge

6.1 Reasons for chosen design

6.2 “Averaging” Final Judge

7 Requirements Specification

Beagle's software design directly follows the SRS. Most design decisions are proposed to fulfil mandatory requirements while allowing easily supplementing optional requirements. Section 7.1 describes changes to the SRS that proved necessary while designing Beagle. Section 7.2 describes how Beagle's requirements are reflected in its design. However, some mandatory criteria have changed, concerning the Common Trace API (CTA) .

7.1 Changes to the SRS

The CTA

The CTA was planned to be used by Beagle to communicate with measurement software (/B10/, /F40/). While starting to investigate the Application Programming Interface (API), it became apparent that it does not offer the expected functionality. The CTA is designed to return measurement results on method level, while Beagle's measurements need to be performed on sub-method (statement) level. Furthermore, there is no possibility to instrument source code, or 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 the 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.

7.2 Relating Requirements to the Design

Terms and Definitions

Common Trace API

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

Kieker

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

A measurement software Beagle aims to support.

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).

[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.