UNIVERSITÄT PADERBORN
Die Universität der Informationsgesellschaft

Faculty of Electrical Engineering – Computer Science – Mathematics
Heinz Nixdorf Institute and Department of Computer Science
Software Engineering Group
Zukunftsmeile 1
33102 Paderborn

Quality Analysis Lab (QuAL): Software Design Description and Developer Guide Version 1.1

by
SEBASTIAN LEHRIG[1]
Zukunftsmeile 1
33098 Paderborn

Paderborn, May 2016

[1]The research leading to these results has received funding from the EU Seventh Framework Programme (FP7/2007-2013) under grant no 317704 (CloudScale).

Contents

1 Introduction

This document is a software design description of the Quality Analysis Lab (QuAL), a framework for conducting, storing, and visualizing metric measurements in Palladio [3] analyses (e.g., conducted by simulators or performance prototypes). The main target group for this document are Palladio developers who want to use or to extend QuAL, e.g., by self-defined metrics and custom visualizations. For this purpose, this document covers a general overview of QuAL as well as detailed descriptions of each lab component.

In this introduction, we first introduce a simple running example to illustrate the purpose of QuAL. Afterwards, we give a high-level overview of QuAL in Sec. 1.2. We describe structural and interaction viewpoints of Palladio analyses with QuAL in Sec. 1.3 and Sec. 1.4, respectively. Finally, we outline the rest of this document (where we focus on using and extending QuAL) in Sec. 1.5.

1.1 Running Example: The Alice&Bob System

As running example, we use the *Alice&Bob* system as illustrated in Fig. 1.1. This system consists of two Glassfish servers: GlassfishA allocates the Alice component that provides the interface IAlice with the operation callBob() and GlassfishB allocates the Bob component that provides the interface IBob with the operation sayHello(). The IAlice interface is provided to a user who can invoke its operation through a client-side technology like a browser. This invocation can be received by a

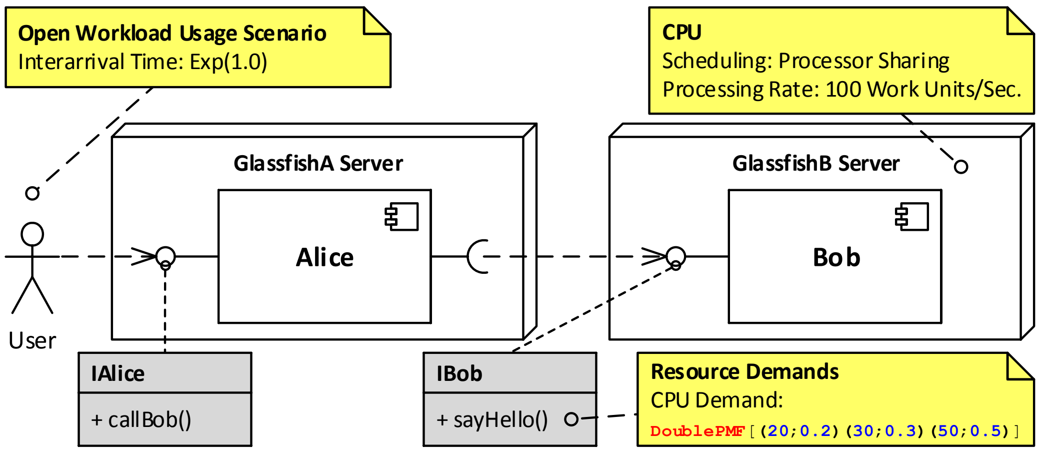

Open Workload Usage Scenario
Interarrival Time: Exp(1.0)
CPU
Scheduling: Processor Sharing
Processing Rate: 100 Work Units/Sec.
GlassfishA Server
Alice
GlassfishB Server
Bob
User
IAlice
+ callBob()
IBob
+ sayHello()
Resource Demands
CPU Demand:
DoublePMF[(20;0.2)(30;0.3)(50;0.5)]

Figure 1.1: Alice&Bob System

component on the server side – in our case, by the Alice component implementing the IAlice interface. Furthermore, we annotated performance-relevant information to the *Alice&Bob* system using yellow sticky notes. These sticky notes state that (1) users arrive the system with an expected interarrival time of 1.0, (2) the CPU of GlassfishB follows a processor sharing (round-robin) scheduling strategy while processing with 100 work units per seconds, and (3) calls to sayHello() cause a CPU demand as specified by the given probability mass function. The latter specifies that 20 CPU work units are demanded in 20% of the cases, 30 CPU work units are demanded in 30% of the cases, and 50 CPU work units are demanded in 50% of the cases.

Assume we want to analyze systems like the *Alice&Bob* system using an analyzer. Such an analyzer can, for instance, allow us to analyze the performance of a system in terms of response times and CPU utilization. For example, we could be interested in the response time of the callBob operation as observed by the user.

QuAL enables us to specify such metrics (e.g., “response time”) and to record and visualize their measurements (e.g., “response time measured at callBob was 2.0 seconds”). In Fig. 1.2 and 1.3, we provide some example visualizations for the *Alice&Bob* system: Fig. 1.2 shows response times for callBob() over time in an XY plot and Fig. 1.3 a corresponding histogram. Note that such results require to take a series of measuring values that contribute to a full measurement. For example, the XY plot of Fig. 1.2 shows several response time measuring values, taken at different points in time.

Because QuAL heavily uses such metrics and measurements, QuAL comes with mature and precise concepts to specify these in a type-safe way (see Figure 1.4).

The upper part of Fig. 1.4 shows that a measurement consists of a measuring type and a non-empty set of measuring values. The former – measuring types – consist of a measuring metric that states *what* to measure (e.g., “response time”) and a measuring point that states *where* to measure (e.g., “callBob”). The latter – measuring values – measure such a measuring type in terms of a value (e.g., “2”) typed by a unit (e.g., “seconds”) and taken at a certain time (e.g., “t0”). As exemplified above, performance engineers interested in response time typically want to investigate a series of response times (distributed over time) and not only one such response time measuring value. Therefore, the concept of a “measurement” allows to include a series of measuring values at different points in time.

The lower part of Fig. 1.4 shows that (1) *metric descriptions* characterize metric properties to which measuring metrics and measuring values have to conform and (2) *measures* type measuring values. Regarding (1), metric descriptions specify properties such as the name of a metric (e.g., “response time”), its capture type (e.g., “real”), its scale (e.g., “ratio”), its unit (e.g., “seconds”), etc. Regarding (2), measures specify the value type (e.g., “Double”) and the quantity (e.g., “Duration”) for which measuring values have to provide a fitting instance (e.g., “2.0 seconds”).

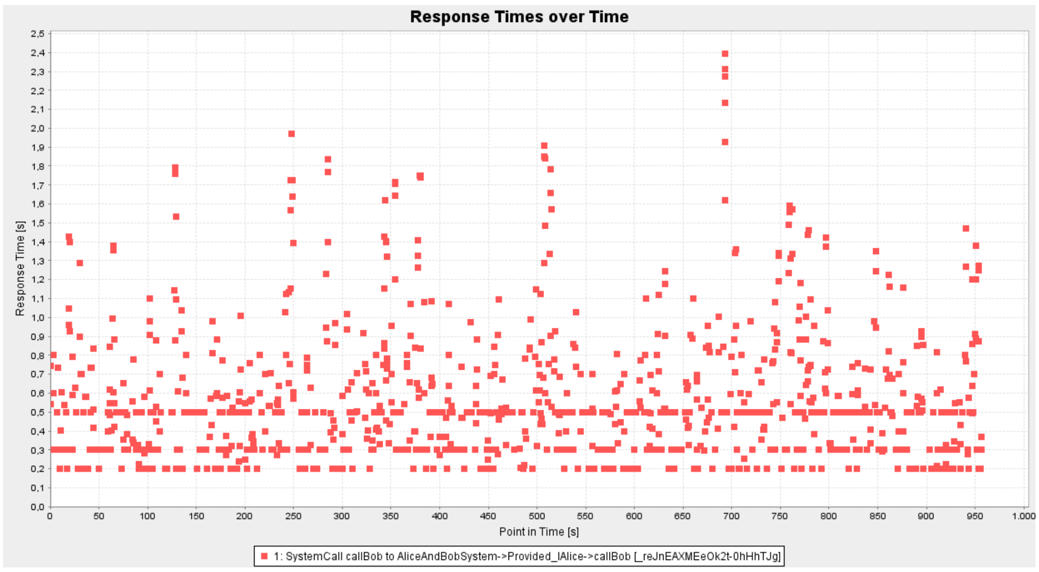


Figure 1.2: Response Times XY Plot for callBob()

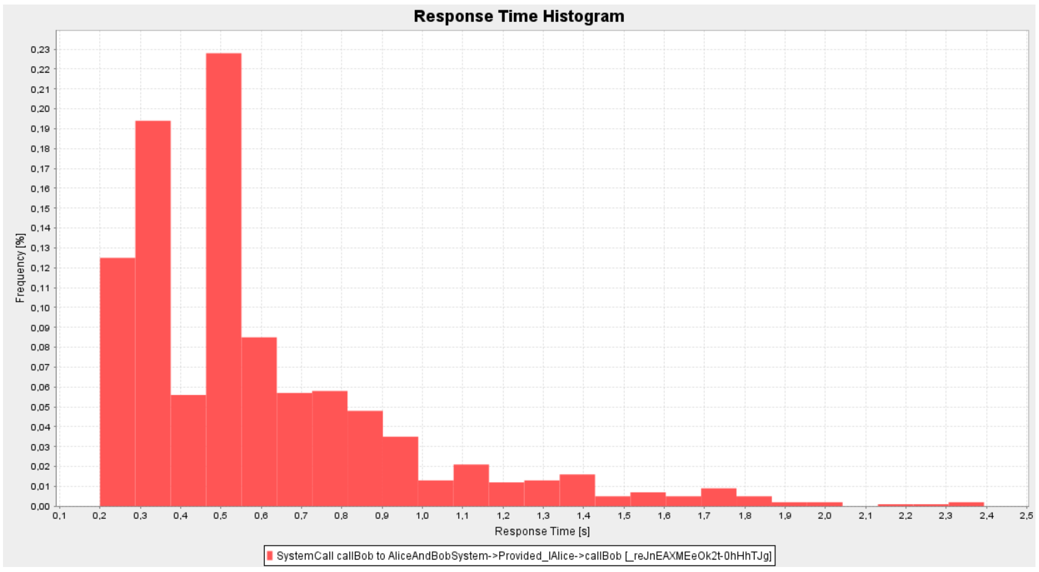


Figure 1.3: Response Time Histogram for callBob()

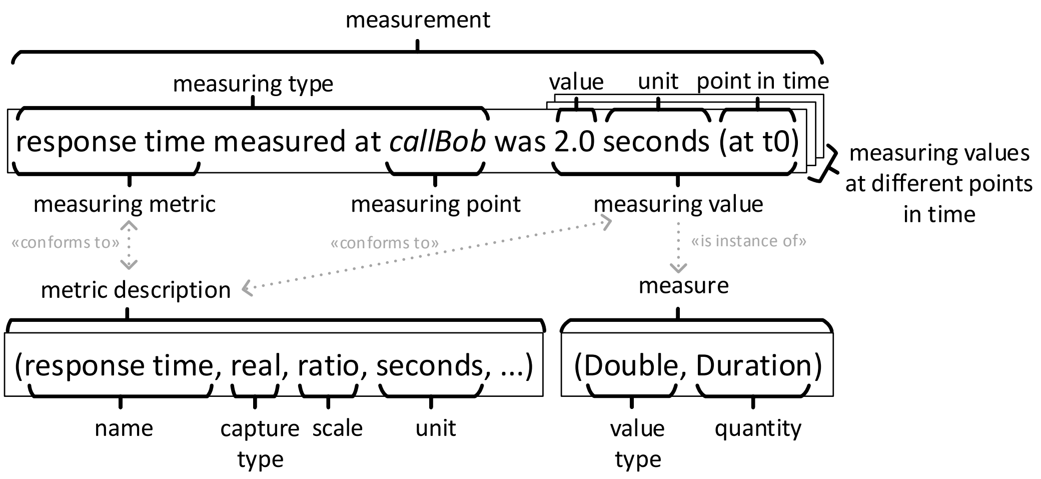


Figure 1.4: Measurements of Palladio's Quality Analysis Lab

1.2 Overview of the Quality Analysis Lab

As stated in the previous section, QuAL enables us to specify metrics and to record and visualize their measurements. Figure 1.5 gives a high-level overview of the corresponding data flow through QuAL. The lower part illustrates the parts of QuAL that are responsible for data specification while the upper part illustrates how QuAL uses this data for metric measurements in Palladio analyses.

1.2.1 Data Specification: Metrics & Measurements

The data specification of QuAL (lower part of Fig. 1.5) includes the following main components:

Metric Specification Framework: Metric Descriptions Metric descriptions characterize metrics to be measured (cf., Fig. 1.4). For example, a *point in time* metric represents the point in time when a measurement value is taken and can be measured with real numbers (capture type) and in seconds (unit). Another example is the response time metric of the *Alice&Bob* system: the response time metric represents the duration between start of an operation call and its end; it can be measured with real numbers (capture type) and in seconds (unit).

In Palladio, metric descriptions are realized in the "Metric Specification Framework" (Chap. 2). Metric Specification Framework comes with a library of commonly used metrics. Moreover, it allows to specify custom metrics.

Measurement Framework: Measurements Measurements specify data objects for storing measuring values at a given measuring point; typed by a corresponding metric description. For example, "12 seconds wall clock time of an analysis" is a valid measuring value/point combination corresponding to the *point*

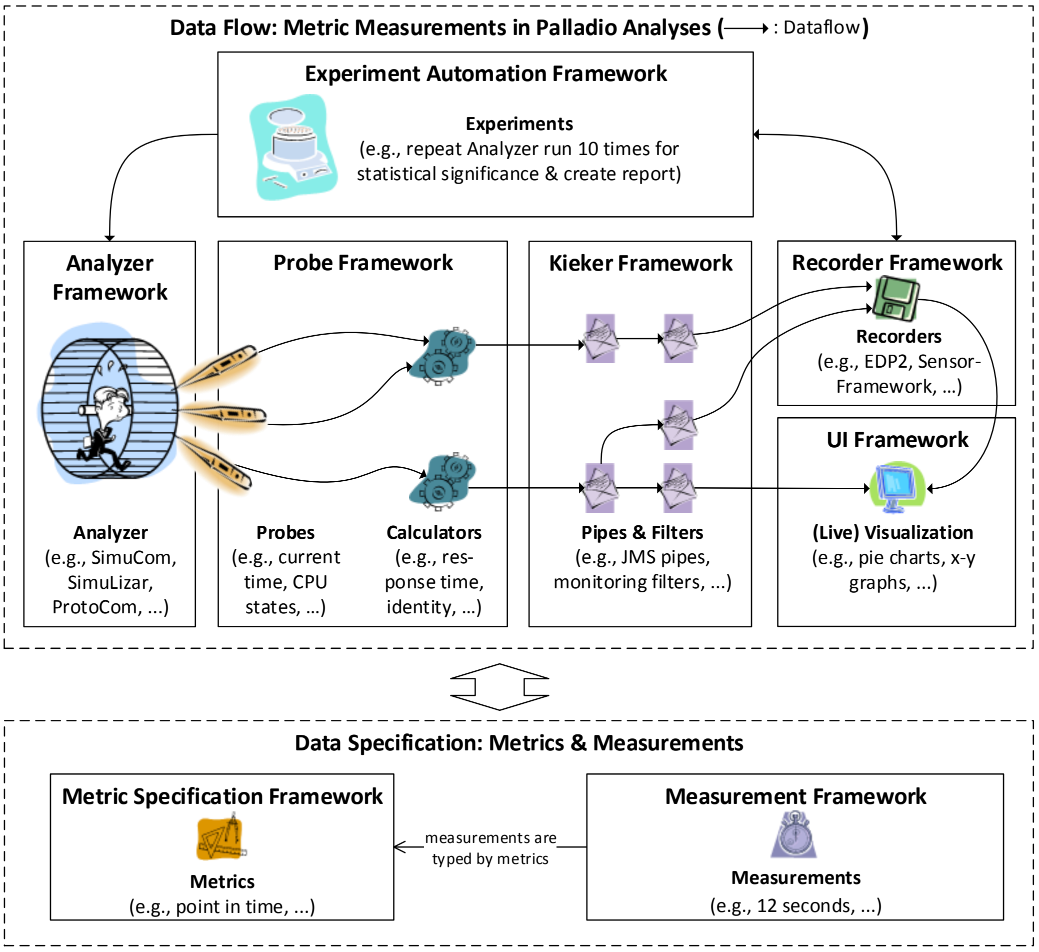

Data Flow: Metric Measurements in Palladio Analyses (→ : Dataflow)
Experiment Automation Framework
Experiments
(e.g., repeat Analyzer run 10 times for statistical significance & create report)
Analyzer Framework
Probe Framework
Kieker Framework
Recorder Framework
Recorders
(e.g., EDP2, Sensor-Framework, ...)
UI Framework
Analyzer
(e.g., SimuCom, SimuLizar, ProtoCom, ...)
Probes
(e.g., current time, CPU states, ...)
Calculators
(e.g., response time, identity, ...)
Pipes & Filters
(e.g., JMS pipes, monitoring filters, ...)
(Live) Visualization
(e.g., pie charts, x-y graphs, ...)
Data Specification: Metrics & Measurements
Metric Specification Framework
Metrics
(e.g., point in time, ...)
measurements are typed by metrics
Measurement Framework
Measurements
(e.g., 12 seconds, ...)

Figure 1.5: Overview of Palladio's Quality Analysis Lab

in time metric. An example measurement for the response time metric of the *Alice&Bob* system is "12 seconds for a callBob operation call".

In Palladio, measurements are realized in the "Measurement Framework" (Chap. 3). Measurement Framework provides a library to measure basic measurements (one value) or measurement sets (set of basic measurements). These two types of measurements should generically cover all measurement needs, thus, the framework does not provide extension points for custom measurements.

1.2.2 Data Flow: Metric Measurements in Palladio Analyses

The data flow of metrics and measurements within Palladio analyses (upper part of Fig. 1.5) involves the following main components:

Analyzer Framework An analyzer runs the environment to be measured (e.g., a simulation or a performance prototype of the *Alice&Bob* system). Currently, typical Palladio simulators are SimuCom [2], SimuLizar [1], and EventSim [7]; a typical performance prototype is provided by ProtoCom [2, 6].

In Palladio, analyzers are realized in the "Analyzer Framework" (Chap. 4). Analyzer Framework provides interfaces and abstract classes serving as a starting point for custom analyzers. The above mentioned concrete analyzers are examples for such custom analyzers.

Probe Framework

Probes Probes know how to measure values (e.g., "current analysis time" or "CPU state/queue size") for a given analyzer, i.e., they come with a suitable, analyzer-specific implementation. Once implemented, probes can be placed all over the analyzer to steadily probe.

In Palladio, probes are realized in the "Probe Framework" (Chap. 5). Probe Framework comes with an analyzer-independent library of abstract probes. These abstract probes allow to easily specify custom, analyzer-specific probes.

Calculators Calculators attach themselves to a set of probes to enrich their measurements by the investigated metric (e.g, response time) and the measuring point (e.g., `callBob` operation). Calculators can then be used further, e.g., for visualization or recording. Therefore, calculators specify which measurements are of interest outside from the analyzer. For example, the response time calculator provides response time measurements that are calculated based on current time probes for start and end time (response time = end − start time). Another example is the identity calculator that directly lets probe measurements pass through, without changing the measured value.

In Palladio, calculators are part of the "Probe Framework" (due to their strong dependency to probes; see Chap. 5). Probe Framework comes with a factory for commonly used calculators. Moreover, it allows to add custom calculators.

Kieker Framework: Pipes & Filters Pipes and filters allow to transfer and filter measurements of calculators to data sinks such as recorders and live visualization. For example, measurements can be transferred via JMS to dedicated servers for measurement storage.

In Palladio, pipes and filters are realized via the push-based, third-party framework "Kieker" [4] (elements are pushed through the pipeline, not pulled; see Chap. 6). Kieker allows for an easy creation of custom filters.

Recorder Framework: Recorders Recorders allow measurements received via the pipes & filters framework to be stored, e.g., in a database, an XML file, or a binary file.

In Palladio, recorders are realized in the "Recorder Framework" (Chap. 7), allowing to specify custom recorders. Currently, two of such custom recorders exist: "SensorFramework" and "Experiment Data Persistency & Presentation (EDP2)". Note that EDP2 is the preferred recorder and will fully replace SensorFramework soon (this document is currently the main documentation of EDP2).

UI Framework: (Live) Visualization A visualization allows to present measurements in graphical form. For example, CPU utilization can be visualized via a pie chart or the response times of the *Alice&Bob* system in a x-y graph. Such a visualization can either be based on recorded measurements (recorders provide data input) or based on live monitoring data (pipes & filters framework provides data input).

In Palladio, visualization is realized in the "UI Framework" (Chap. 10). For recorder-based visualization, the UI framework provides a pull-based pipes and filters framework to request data from recorders. For live visualization, we are currently developing a suitable component (not released yet). Finally, note that UI Framework is currently part of EDP2.

Experiment Automation: Experiments Experiments allow to conduct a series of analyzer runs and to aggregate recorded measurements to new measurements (so-called experiment reports). For example, an experiment can repeat an analyzer run for ten times. The variance over these runs can be reported, indicating the statistical significance of these analyzer runs. Another example is to conduct the same analysis using different analyzers and to report on the analysis time per analyzer.

In Palladio, experiments are realized in the "Experiment Automation Framework" (Chap. 11). Experiment Automation Framework allows to specify analyzer configurations, to run analyzers, and to store experiment reports based on investigating analyzer measurements within recorders. In particular, the framework provides extension points for adding custom analyzers.

1.3 Analyses: Structural Viewpoint of Dependencies

The analysis-relevant part of QuAL has the dependencies shown in the component diagram in Fig. 1.6. Such dependencies are needed for the initialization and configuration of the measurement pipeline, later on used for sending measurements from analyzers to recorders.

Analyzer is the main component, allowing quality engineers to run analyses. For such analyses, Analyzer uses the Probe Framework to equip itself with probes and

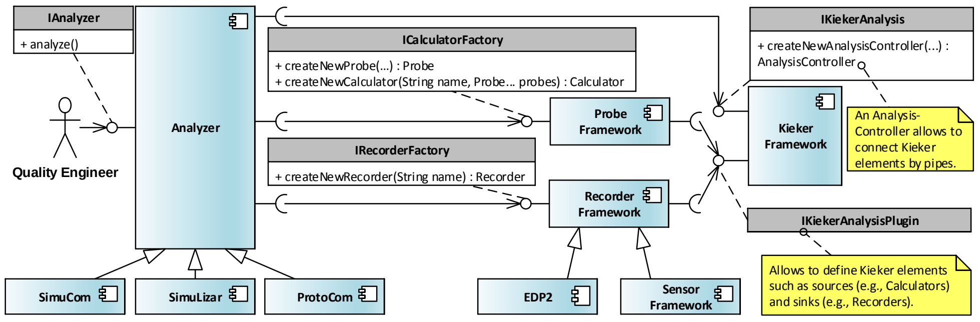

IAnalyzer
+ analyze()
Quality Engineer
Analyzer
ICalculatorFactory
+ createNewProbe(...) : Probe
+ createNewCalculator(String name, Probe... probes) : Calculator
IRecorderFactory
+ createNewRecorder(String name) : Recorder
Probe Framework
Recorder Framework
IKiekerAnalysis
+ createNewAnalysisController(...) : AnalysisController
Kieker Framework
An Analysis-Controller allows to connect Kieker elements by pipes.
IKiekerAnalysisPlugin
Allows to define Kieker elements such as sources (e.g., Calculators) and sinks (e.g., Recorders).
SimuCom
SimuLizar
ProtoCom
EDP2
Sensor Framework

Figure 1.6: Component Dependencies of QuAL

calculators as well as the Recorder Framework to receive a suitable recorder for measurement storage. To connect calculators and the recorder, Analyzer uses the Kieker Framework.

Typical examples for concrete Analyzer components are SimuCom, SimuLizar, and EventSim. Typical examples for recorders are EDP2 and SensorFramework.

Note that, besides the illustrated dependencies, each component also depends on the Metric Specification Framework and the Measurement Framework. We did not visualize this dependency for clarity.

1.4 Analyses: Interaction Viewpoint

Analyses in QuAL follow the interactions shown in the sequence diagrams in Fig. 1.7, Fig. 1.8, Fig. 1.9, and Fig. 1.10.

Figure 1.7 illustrates the general structure of an analysis run: it consists of an initialization, a running, and a finishing phase. Each of these phases is described in separation in one of the other sequence diagram, respectively.

Figure 1.8 illustrates the initialization phase of an analysis. In this phase, an Analyzer first creates suitable probe and calculator objects where the latter are directly linked to the former. For example, an Analyzer creates probe objects for measuring the point in time when a system entry call started (startProbe) and when it finished (endProbe).[1] A corresponding calculator object that measures response time (responseTimeCalculator) links to these two probes. Secondly, an Analyzer can create a recorder object (e.g., an EDP2 recorder object edp2Recorder) responsible for recording measurements received from calculators. Therefore, an Analyzer has to explicitly connect calculators to recorders using the Kieker Framework.

Figure 1.9 illustrates the running phase of an analysis. In this phase, the Analyzer

[1] In this example, we use probes of type “basic invoked”. These probes are explicitly invoked by the control flow of the analyzer (in contrast to reacting on an event). Chapter 5 provides more details on different probe types and their use-cases.

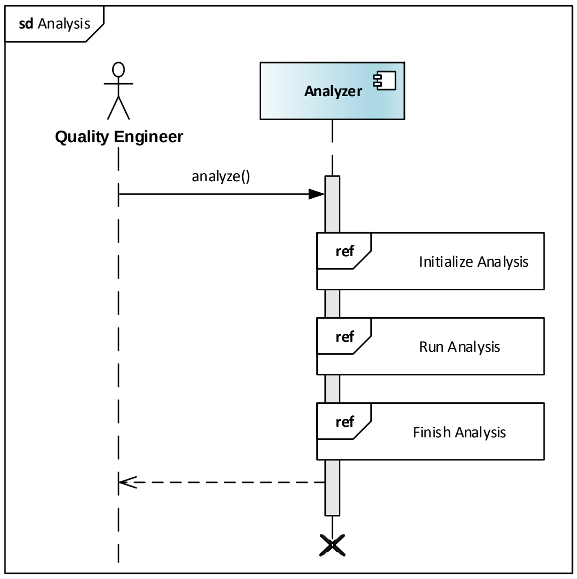


Figure 1.7: Sequence Diagram: Analysis

analyzes a system under study. As the figure shows, for example system entry calls can be analyzed. For instance, the callBob operation of the *Alice&Bob* system can be called. System entry calls can cause two events: the beginning of a call and the end of a call. To react on these events, the Analyzer provides the two call-back methods beginSystemEntryCallSimulation and endSystemEntryCallSimulation.

The beginSystemEntryCallSimulation method invokes takeMeasurement on the startProbe object to trigger the probe. Such a trigger forces the probe to take a point-in-time measurement (doMeasure method) and to inform its observers, i.e., registered calculators, that a new measurement is available (newMeasurementAvailable method). In the case of the start probe, the registered responseTimeCalculator object puts the received measurement in its memory because it has to wait for the point-in-time measurement of the end probe to calculate the overall response time.

The endSystemEntryCallSimulation method functions similarly but uses the stopProbe object instead of the startProbe object. The stopProbeObject also sends its measurement to the responseTimeCalculator object once triggered. The responseTimeCalculator object can now calculate the overall response time by using its calculate method. In particular, this method can calculate the response time based on the previously stored point-in-time measurement (for the starting time) and the recently received point-in-time measurement (for the ending time). After this calculation, the calculator informs its observers that a newly calculated measurement is available (newMeasurementAvailable method). For instance, the edp2Recorder object could be registered at the calculator such that it stores received data in a file (writeData method).

Figure 1.10 illustrates the finishing phase of an analysis. In this phase, no more

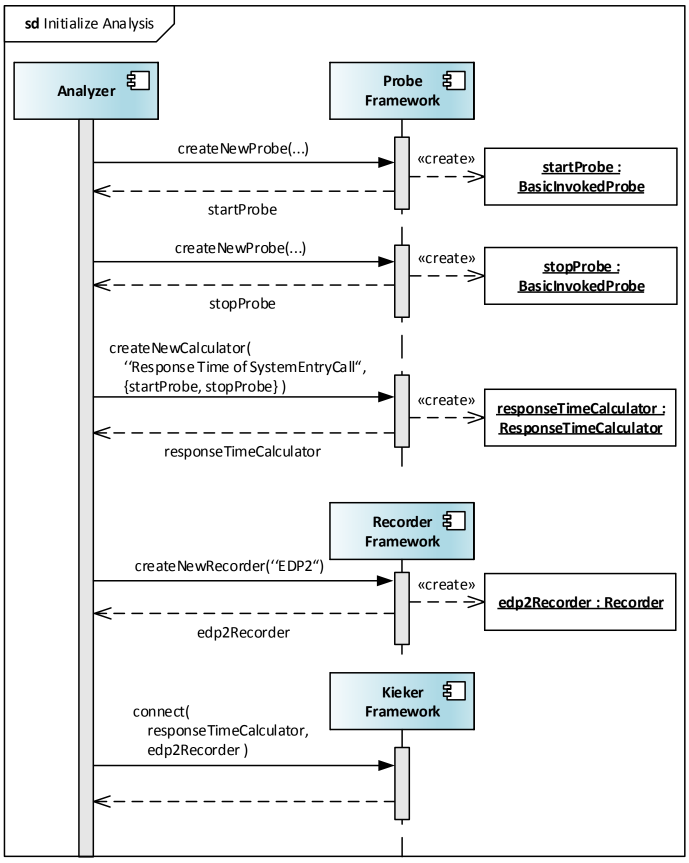

sd Initialize Analysis
Analyzer
Probe Framework
createNewProbe(...)
«create»
startProbe : BasicInvokedProbe
startProbe
createNewProbe(...)
«create»
stopProbe : BasicInvokedProbe
stopProbe
createNewCalculator("Response Time of SystemEntryCall", {startProbe, stopProbe})
«create»
responseTimeCalculator : ResponseTimeCalculator
responseTimeCalculator
Recorder Framework
createNewRecorder("EDP2")
«create»
edp2Recorder : Recorder
edp2Recorder
Kieker Framework
connect(responseTimeCalculator, edp2Recorder)

Figure 1.8: Sequence Diagram: Initialization of the Analysis

analyses are conducted and open input streams are closed. Therefore, the Analyzer invokes the finish method of Probe Framework that informs all calculators about the end of the analysis. In particular, all calculators inform registered observers about being unregistered (preUnregister method). For example, the edp2Recorder is informed about this event such that it can flush the recorded measurement data into its repository (e.g., a file or a database).

After the finishing phase, the analysis ends. All measurement data that was of interest should have been stored in the registered recorders or directly have been visualized by the live visualization component. In case the measurements have been stored by recorders, quality engineers can also investigate these measurements using the other visualization components of the UI Framework.

1.5 Document Structure

After giving a brief introduction to Palladio's Quality Analysis Lab, we next focus on concrete code examples illustrating how to use and extend it. We structure

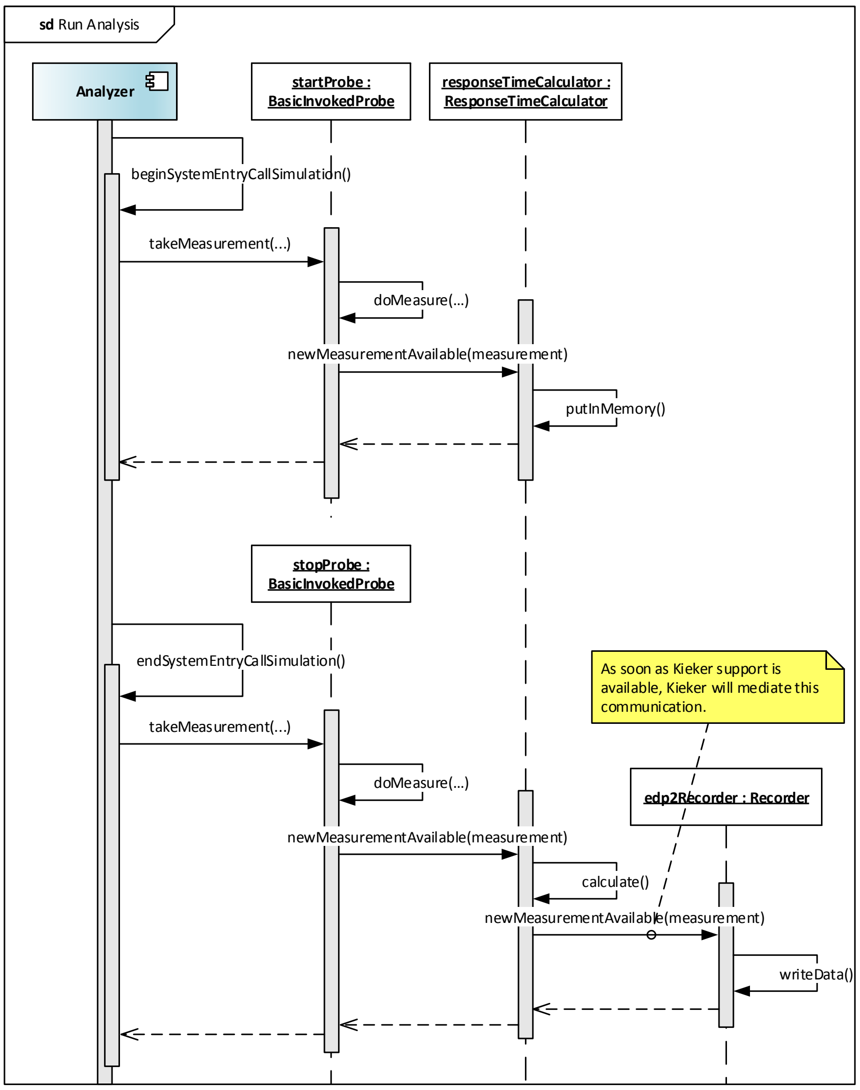

sd Run Analysis
Analyzer
startProbe : BasicInvokedProbe
responseTimeCalculator : ResponseTimeCalculator
beginSystemEntryCallSimulation()
takeMeasurement(...)
doMeasure(...)
newMeasurementAvailable(measurement)
putInMemory()
stopProbe : BasicInvokedProbe
endSystemEntryCallSimulation()
takeMeasurement(...)
doMeasure(...)
newMeasurementAvailable(measurement)
calculate()
As soon as Kieker support is available, Kieker will mediate this communication.
edp2Recorder : Recorder
newMeasurementAvailable(measurement)
writeData()

Figure 1.9: Sequence Diagram: Run of the Analysis

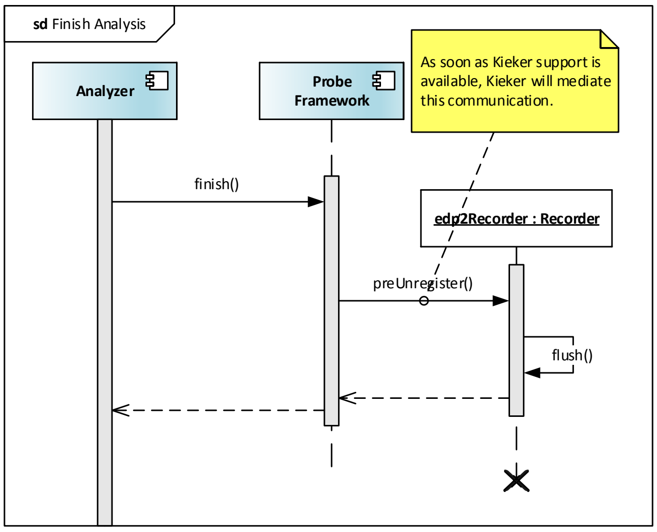


Figure 1.10: Sequence Diagram: Finishing the Analysis

our illustrations along the different components of the lab:

Chapter 2 covers the Metric Specification Framework. Here, we show how new metrics can be specified to be used by the lab.

Chapter 3 covers the architecture of the Measurement Framework. We commonly do not expect extension scenarios here.

Chapter 5 covers the Probe Framework. We describe the available library of probes, how it can be extended, and how probes can be used by analyzers.

Chapter 6 covers the Kieker Framework. We give a brief description of this framework and show how we use it within Palladio.

Chapter 7 covers the Recorder Framework. We briefly outline EDP2 here but commonly do not expect extension scenarios.

Chapter 10 covers the UI Framework. We describe how measurements recorded by EDP2 can be visualized and give an outlook on live visualization. We also describe the most typical extension scenario: adding a custom visualization (e.g., a new kind of graph).

Chapter 11 covers the Experiment Automation Framework. We describe the core concepts of the framework and show how custom analyzers can be added.

2 Metric Specification Framework

In this chapter, we describe the Metric Specification Framework of QuAL. The framework is build around a meta model for specifying metric descriptions (lower-left part of Fig. 1.4). This meta model allowed us to create a library of common metric descriptions. The meta model can also be used by Palladio developers who want to provide custom metric descriptions. Such metric descriptions themselves are tightly integrated in QuAL and occur all over this document (e.g., for typing and visualizing measurements).

This chapter covers the meta classes of the Metric Specification Framework metamodel (Sec. 2.1), our library of commonly used metric descriptions we specified as an instance of this metamodel (Sec. 2.2), and shows how Palladio developers can specify custom metric descriptions (Sec. 2.3).

2.1 Meta Classes of the Metric Specification Framework

In Fig. 2.1, we illustrate the meta classes of the Metric Specification Framework meta model.

MetricDescriptionRepository serves as the root node of the meta model. Such a repository contains the (unordered) set of metric descriptions to be specified.

A MetricDescription characterizes a metric, thus, being the core concept of the Metric Specification Framework meta model. MetricDescription inherits from Description such that metric descriptions can textually give the name and a description for the referred metric. Furthermore, we distinguish two types of metric descriptions: (1) BaseMetricDescription and (2) MetricSetDescription.

The first type (BaseMetricDescription) allows to specify typical characteristics of metrics. These characteristics are capture type, data type, and scale; each typed by a dedicated enumeration. Moreover, base metric descriptions can either be a TextualBaseMetricDescription or a NumericalBaseMetricDescription. The former (TextualBaseMetricDescription) is used for metrics that reference a (set of) string identifiers. For example, a "component names metric" could list the available components of a system ("Alice" and "Bob" in the case of the *Alice&Bob System*). The latter (NumericalBaseMetricDescription) is used for metrics that can be expressed as numbers and in a given unit. For example, response time is a metric that can be expressed as a real number and in seconds.

The second type (MetricSetDescription) composes an ordered list of subsumed metric descriptions using the composite pattern. Therefore, such metric set descriptions can contain basic metric descriptions as well as other metric set descriptions. A typical example for a metric set description is a "response time tuple"

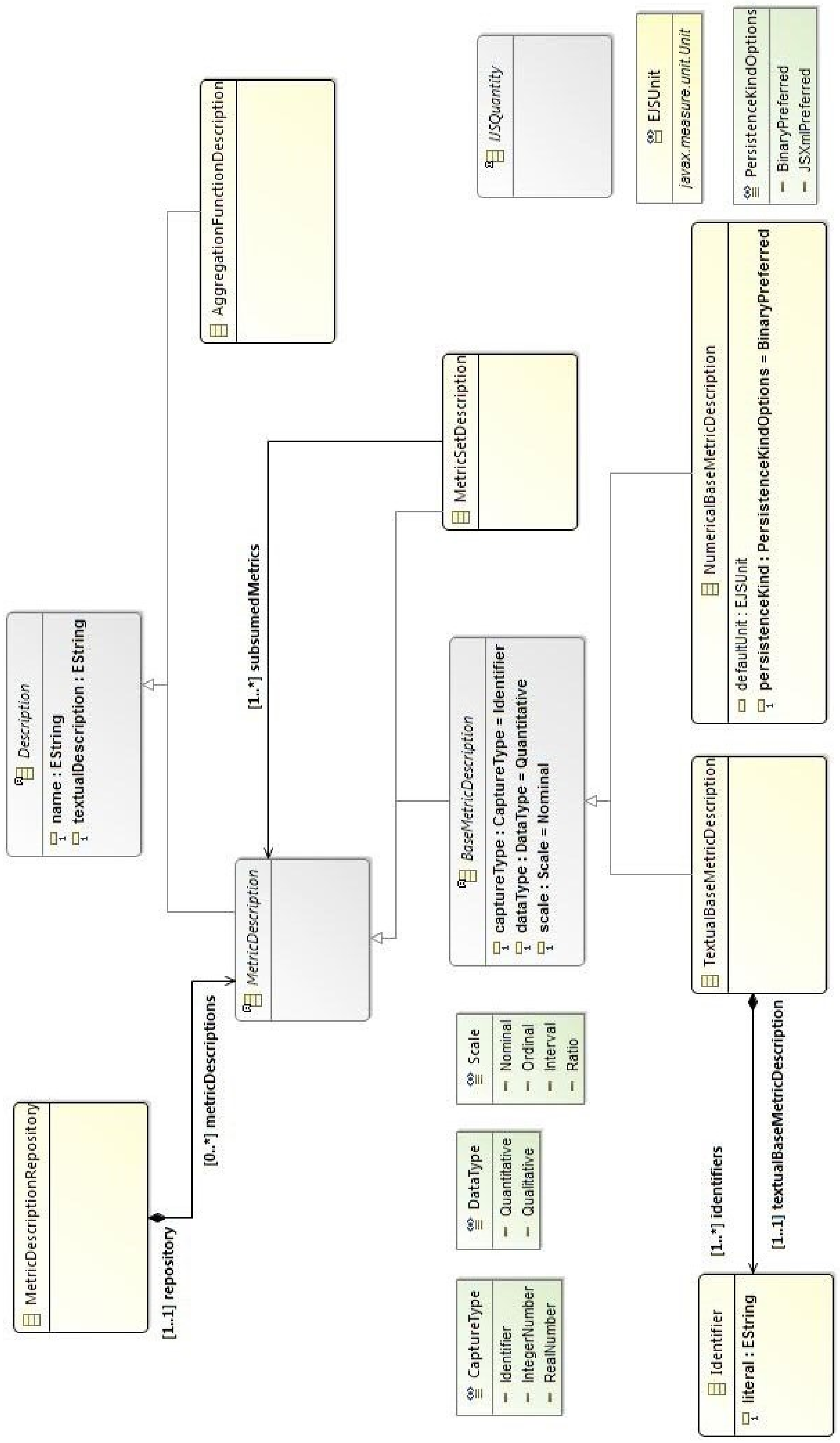


Figure 2.1: Meta Classes of the Metric Specification Framework

that specifies a "(point in time, response time)"-tuple, i.e., the end of an operation call plus the response time. For instance, such tuples are needed when response times are investigated over time like in Fig. 1.2.

For details about the illustrated meta classes, we directly refer to the Metric Specification Framework meta model. We specified this meta model in ECore and annotated the documentation directly[1].

2.2 Library of Common Metric Descriptions

Fig. 2.2 shows the tree editor for the Metric Specification Framework meta model with our library of common metric descriptions opened[2]. Our library of common metric descriptions includes the following metric descriptions:

- Classical performance metrics such as response time, throughput, and resource utilization. These metrics were used in Palladio even before we started developing QuAL; we simply integrated these into our library.

- Novel cloud computing metric descriptions for capacity, scalability, elasticity, and efficiency metrics such as user capacity, scalability range, number of SLO violations, and marginal cost. We added these metric descriptions during developing QuAL to illustrate its new capabilities. Moreover, we systematically derived these metric descriptions in the context of the CloudScale EU project[3].

Each metric description comes with a set of pre-configured characteristics that can be investigated via the properties view (see lower part of Fig. 2.2). For example, for the response time metric description, the "textual description" characteristic describes that "This measure represents the response time, e.g., to store the response time of operation calls.". Therefore, we directly refer to such characteristics for a full documentation of each metric description.

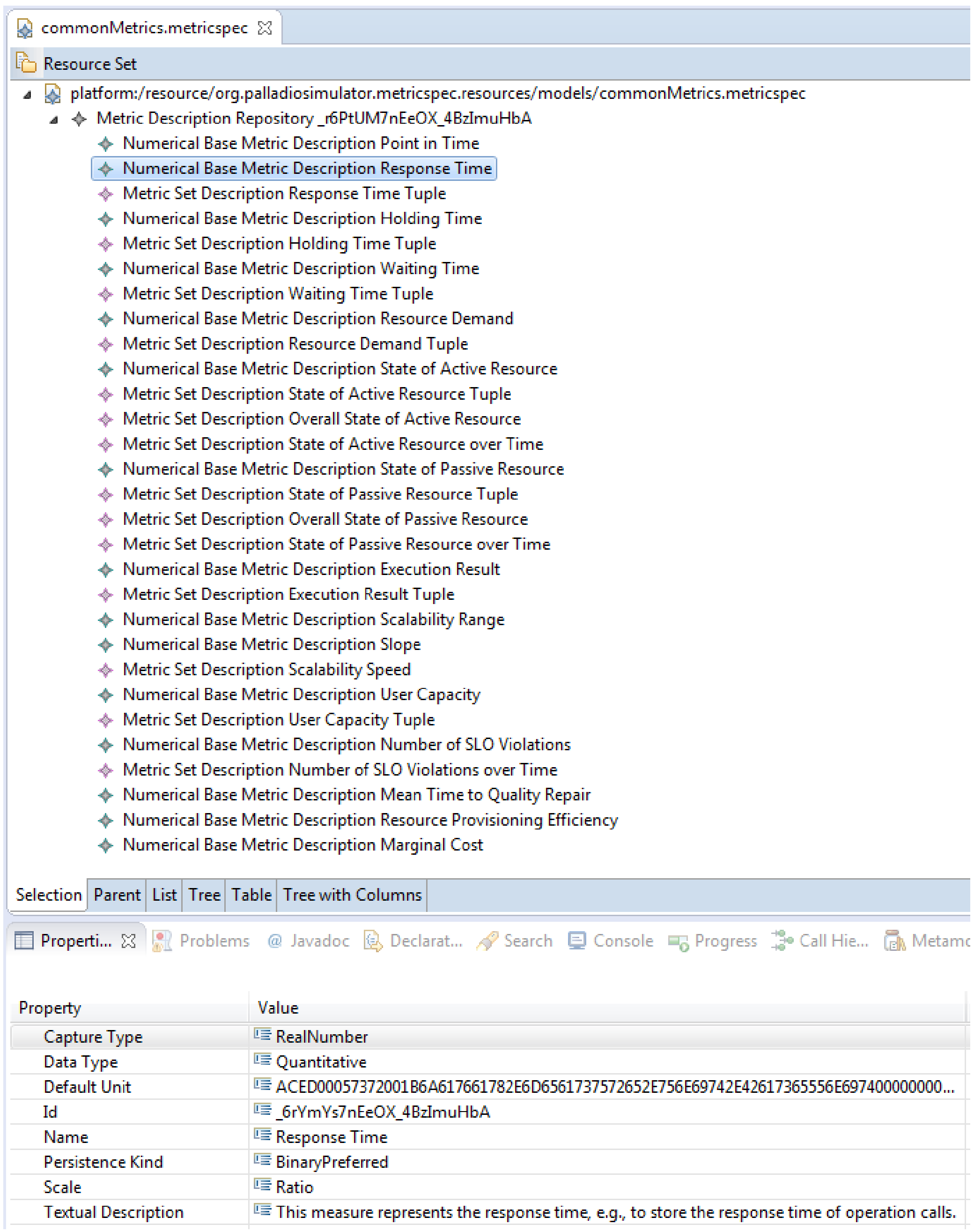
Palladio developers who want to use our library of metric descriptions within source code need to load the library's model file and to select the metric description of interest by referring to its identifier. To ease this task, we additionally provide public constants for each metric description that can directly be used within source code[4].

[1]Metric Specification Framework meta model: `https://svnserver.informatik.kit.edu/i43/svn/code/MetricSpecification/trunk/org.palladiosimulator.metricspec/model`

[2]Library of common metric descriptions: `https://svnserver.informatik.kit.edu/i43/svn/code/MetricSpecification/trunk/org.palladiosimulator.metricspec.resources/models/commonMetrics.metricspec`

[3]CloudScale metrics: `http://cloudscale.xlab.si/wiki/index.php/Glossary`

[4]Metric description constants: `https://svnserver.informatik.kit.edu/i43/svn/code/MetricSpecification/trunk/org.palladiosimulator.metricspec/src-man/org/palladiosimulator/metricspec/constants/MetricDescriptionConstants.java`



commonMetrics.metricspec

Resource Set

platform:/resource/org.palladiosimulator.metricspec.resources/models/commonMetrics.metricspec

Metric Description Repository _r6PtUM7nEeOX_4BzImuHbA

- Numerical Base Metric Description Point in Time
- Numerical Base Metric Description Response Time
- Metric Set Description Response Time Tuple
- Numerical Base Metric Description Holding Time
- Metric Set Description Holding Time Tuple
- Numerical Base Metric Description Waiting Time
- Metric Set Description Waiting Time Tuple
- Numerical Base Metric Description Resource Demand
- Metric Set Description Resource Demand Tuple
- Numerical Base Metric Description State of Active Resource
- Metric Set Description State of Active Resource Tuple
- Metric Set Description Overall State of Active Resource
- Metric Set Description State of Active Resource over Time
- Numerical Base Metric Description State of Passive Resource
- Metric Set Description State of Passive Resource Tuple
- Metric Set Description Overall State of Passive Resource
- Metric Set Description State of Passive Resource over Time
- Numerical Base Metric Description Execution Result
- Metric Set Description Execution Result Tuple
- Numerical Base Metric Description Scalability Range
- Numerical Base Metric Description Slope
- Metric Set Description Scalability Speed
- Numerical Base Metric Description User Capacity
- Metric Set Description User Capacity Tuple
- Numerical Base Metric Description Number of SLO Violations
- Metric Set Description Number of SLO Violations over Time
- Numerical Base Metric Description Mean Time to Quality Repair
- Numerical Base Metric Description Resource Provisioning Efficiency
- Numerical Base Metric Description Marginal Cost

Property	Value
Capture Type	RealNumber
Data Type	Quantitative
Default Unit	ACED00057372001B6A617661782E6D6561737572652E756E69742E42617365556E697400000000...
Id	_6rYmYs7nEeOX_4BzImuHbA
Name	Response Time
Persistence Kind	BinaryPreferred
Scale	Ratio
Textual Description	This measure represents the response time, e.g., to store the response time of operation calls.

Figure 2.2: Library of Common Metric Descriptions

2.3 Custom Metric Descriptions

Palladio developers have two options to add custom metric descriptions:

1. extend our library of common metric descriptions, and
2. create a new library of metric descriptions.

Developers should prefer the first option if they anticipate that their custom metric descriptions are of common interest. For example, adding metric descriptions for common security or energy efficiency metrics would be a useful extension. Developers realize such an extension by adding new metric descriptions to our library[5] and by creating additional constants for these metric descriptions[6].

If the first option is infeasible, developers should follow the second option by creating their own library. In this case, they should proceed analogously to our approach of creating a library of common metric descriptions. The typical starting point is to create a new metric descriptions model using the tree editor we provide ("File - New - Other... - Example EMF Model Creation Wizards - MetricSpec Model - Choose 'Metric Description Repository' as Model Object"). Afterwards, a constants class should be created, similar to our class.

[5]Library of common metric descriptions: `https://svnserver.informatik.kit.edu/i43/svn/code/MetricSpecification/trunk/org.palladiosimulator.metricspec.resources/models/commonMetrics.metricspec`

[6]Metric description constants: `https://svnserver.informatik.kit.edu/i43/svn/code/MetricSpecification/trunk/org.palladiosimulator.metricspec/src-man/org/palladiosimulator/metricspec/constants/MetricDescriptionConstants.java`

3 Measurement Framework

In this chapter, we describe the Measurement Specification Framework of QuAL. The framework is build around the core concept of a measurement, i.e., an entity that includes a metric description, the concrete measuring point of a metric, and a list of measures taken (as illustrated in Fig. 1.4).

For metric descriptions, the framework makes use of the Metric Specification Framework (see Chap. 2). For measuring points, the framework makes use of the corresponding EDP2 functionality (see Chap. 7). For measures, the framework makes use of the JScience framework [5].

This chapter covers the core concepts of measure providers and measurements (Sec. 3.1), of measurement sources and corresponding listeners (Sec. 3.1), and shows how Palladio developers can specify custom measures if the ones from JScience are insufficient (Sec. 3.3).

3.1 Measure Providers and Measurements

Measure providers are entities that allow to receive lists of measures. Measurements are such entities as they include measure lists. Additionally, measurements include a metric description and the concrete measuring point where the measures where taken (these concepts are also illustrated in Fig. 1.4). Fig. 3.1 shows the packages, classes, and interfaces important for measure providers and measurements within the Measurement Specification Framework.

Measure providers are realized in the *.measurementframework.measureprovider package (highlighted in green). IMeasureProvider specifies the interface for measure-providing entities. This interface provides methods to receive measures - based an a given metric or completely as list or array. AbstractMeasureProvider abstractly implements this interface and additionally provides a convenience method to receive measurements that wrap corresponding measures based on a given metric description. MeasurementListMeasureProvider implements this abstract class for lists of measurements. Therefore, this class can also provide access to the list of subsumed measurements.

Measurements are realized in the *.measurementframework package (highlighted in yellow). The abstract class Measurement provides the core functionality of measurements. Measurement implements IMeasureProvider because measurements include a list of measures and IMetricEntity because measurements include a metric description. Moreover, Measurement provides a method to receive a concrete measurement based on a given metric description.

There are two kinds of measurements: basic measurements and tuple measurements. The former (BasicMeasurement class) are measurements with only a single measure; typed by given generic type parameters. The latter (TupleMeasurement class) are measurements that are realized as a list of subsumed measurements. Such tuple measurements can particularly realize the implementation of IMeasureProvider by delegating its method calls to an object of the MeasurementListMeasureProvider class.

3.2 Measurement Sources and Listeners

Measurement sources provide measurements from analyzers, e.g., by providing the values from probes. These measurement sources can have corresponding measurement source listeners that can be used to inform observers about newly arrived measurements. For example, calculators are such observers of measurement sources. We directly describe measurement sources and their listeners in the context of calculators in Sec. 5.1.

3.3 Custom Measures

Generally, we realize measures as instances of JScience [5] measures. However, in situations where no suitable JScience measure is available, Palladio developers have to provide their own custom measure that have to extend the JScience Measure class.

In the *.measurementframework.measure package, we provide a dedicated place for such custom measures. Currently, we only need one such custom measure: the IdentifierMeasure class realizes a measure for textual base metrics. The typical quantity to be used for this measure is Dimensionless with Unit.ONE. Palladio developers that need further measure can proceed analogously to our approach of specifying IdentifierMeasure.

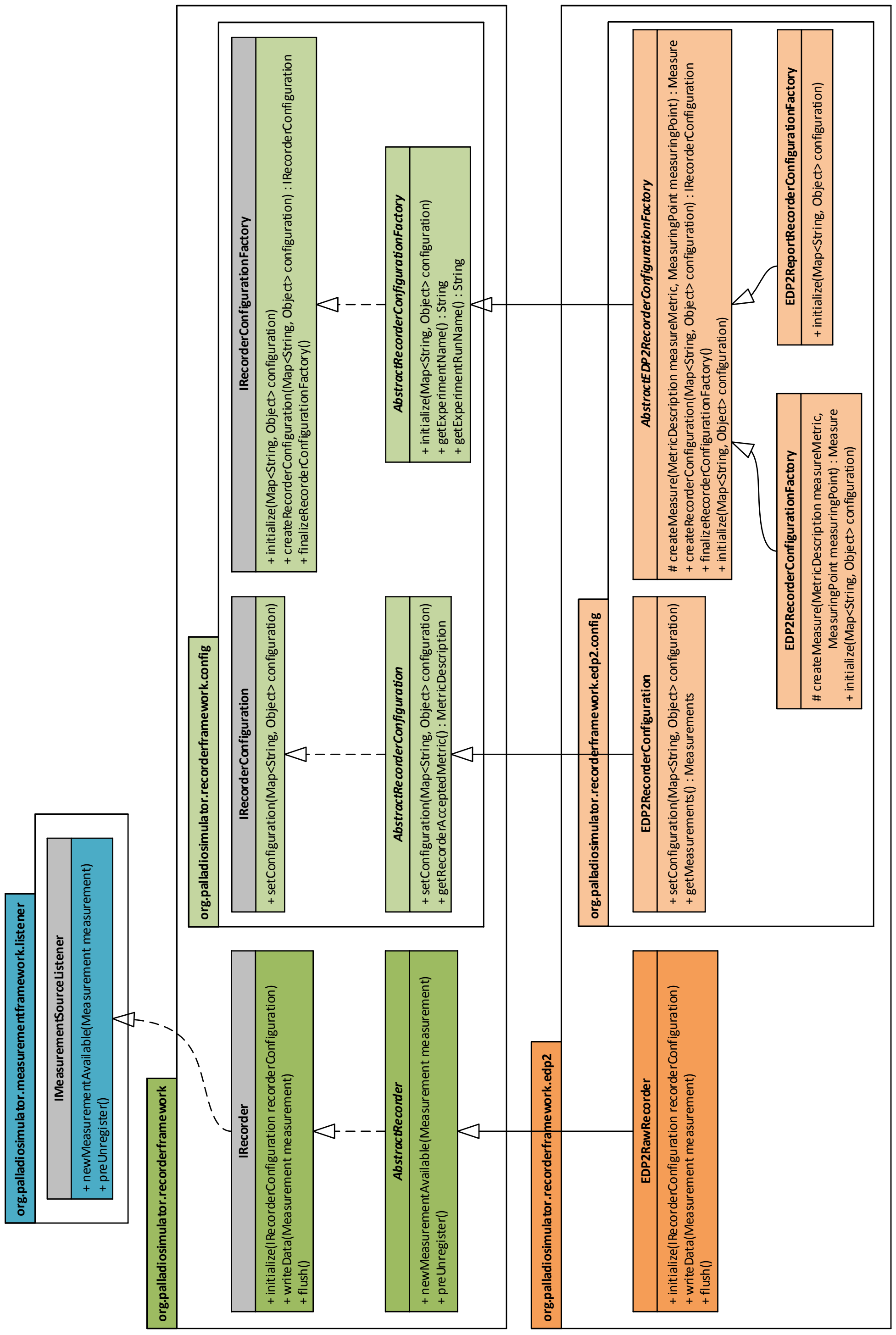

org.palladiosimulator.measurementframework.listener
IMeasurementSourceListener
+ newMeasurementAvailable(Measurement measurement)
+ preUnregister()
org.palladiosimulator.recorderframework
IRecorder
+ initialize(IRecorderConfiguration recorderConfiguration)
+ writeData(Measurement measurement)
+ flush()
AbstractRecorder
+ newMeasurementAvailable(Measurement measurement)
+ preUnregister()
org.palladiosimulator.recorderframework.config
IRecorderConfiguration
+ setConfiguration(Map<String, Object> configuration)
AbstractRecorderConfiguration
+ setConfiguration(Map<String, Object> configuration)
+ getRecorderAcceptedMetric() : MetricDescription
IRecorderConfigurationFactory
+ initialize(Map<String, Object> configuration)
+ createRecorderConfiguration(Map<String, Object> configuration) : IRecorderConfiguration
+ finalizeRecorderConfigurationFactory()
AbstractRecorderConfigurationFactory
+ initialize(Map<String, Object> configuration)
+ getExperimentName() : String
+ getExperimentRunName() : String
org.palladiosimulator.recorderframework.edp2
EDP2RawRecorder
+ initialize(IRecorderConfiguration recorderConfiguration)
+ writeData(Measurement measurement)
+ flush()
org.palladiosimulator.recorderframework.edp2.config
EDP2RecorderConfiguration
+ setConfiguration(Map<String, Object> configuration)
+ getMeasurements() : Measurements
AbstractEDP2RecorderConfigurationFactory
# createMeasure(MetricDescription measureMetric, MeasuringPoint measuringPoint) : Measure
+ createRecorderConfiguration(Map<String, Object> configuration) : IRecorderConfiguration
+ finalizeRecorderConfigurationFactory()
+ initialize(Map<String, Object> configuration)
EDP2RecorderConfigurationFactory
# createMeasure(MetricDescription measureMetric, MeasuringPoint measuringPoint) : Measure
+ initialize(Map<String, Object> configuration)
EDP2ReportRecorderConfigurationFactory
+ initialize(Map<String, Object> configuration)

Figure 3.1: Packages, Classes, and Interfaces important for Measure Providers and Measurements

4 Analyzer Framework

The Analyzer Framework of QuAL intends to provide common functionality to different analyzer implementations. For example, the framework provides the means to specify run configurations of analysis runs and a shared control interface for simulations. Example analyzer implementations that make use of this framework include Palladio simulators like SimuCom [2], SimuLizar [1], and EventSim [7] as well as the performance prototype provided by ProtoCom [2, 6].

In the context of QuAL, only the borders of the Analyzer Framework have been touched; we did not refactor it. However, we altered the framework where necessary in order to integrate other QuAL frameworks (e.g., Metric Specification Framework, Measurement Specification Framework, Probe Framework, and Experiment Automation Framework). A potential refactoring and design description may be provided in the future. Therefore, we advise Palladio developers interested in the Analyzer Framework refactoring and/or in existing/custom analyzers to investigate the projects belonging to de.uka.ipd.sdq.simulation and de.uka.ipd.sdq.simulation.abstractsimengine[1]. Also the analyzer implementations mentioned above are a good resource for getting familiar to the current Analyzer Framework.

[1]Analyzer Framework in the Palladio SVN: `https://svnserver.informatik.kit.edu/i43/svn/code/Palladio/Core/trunk/SimuCom`

5 Probe Framework

In this chapter, we describe the Probe Framework of QuAL. We describe the inheritance hierarchy of probes and calculators (Sec. 5.1), how probes and calculators can be used by analyzers (Sec. 5.2), and how custom probes and calculators can be added (Sec. 5.3).

5.1 Probes and Calculators

In this section, we describe the inheritance hierarchy of probes and calculators. Furthermore, we list and describe the most common probes and calculators. For our descriptions, we refer to Fig. 5.1 that illustrates a structural viewpoint of Probe Framework using packages, classes, and interfaces.

Generally, we distinguish between **probes** (that take measurements from analyzers) and **measurement sources** (that provide measurements from analyzers, e.g., by providing the values from probes). Calculators are a special case of such measurement sources, specialized on probe processing (c.f., MeasurementSource class in package org.palladiosimulator.measurementframework.listener).

Both, probes and measurement sources, inherit from **MetricEntity**, a class of the Metric Specification Framework (package org.palladiosimulator.metricspec.metricentity). MetricEntity describes objects that are typed by a metric, thus, allowing to receive their metric description (via getMetricDescription(...)).

Moreover, probes and measurement sources implement the **IAbstractObservable** interface of Palladio Commons (package org.palladiosimulator.commons.designpatterns). The IAbstractObservable interface allows to specify observable objects according to the observer design pattern. Therefore, probes and measurement sources are such observable objects that can be observed by observer objects. Once probes or measurement sources have new measurements at hand, they inform registered observers about these new measurements. To do so, probes as well as measurements provide dedicated interfaces with call-back methods that observers have to implement. Such interfaces are typed by the generic type parameter T of IAbstractObservable. Probes bind the **IProbeListener** interface and measurement sources the **IMeasurementSourceListener** to T, respectively. Both of these interfaces provide call-back methods that pass the new measurement as a parameter (a ProbeMeasurement for the case of probes and a Measurement for the case of measurement sources).

Calculators are an example for such an observer because they implement the IProbeListener interface to observe one or many probes. On the other hand, they

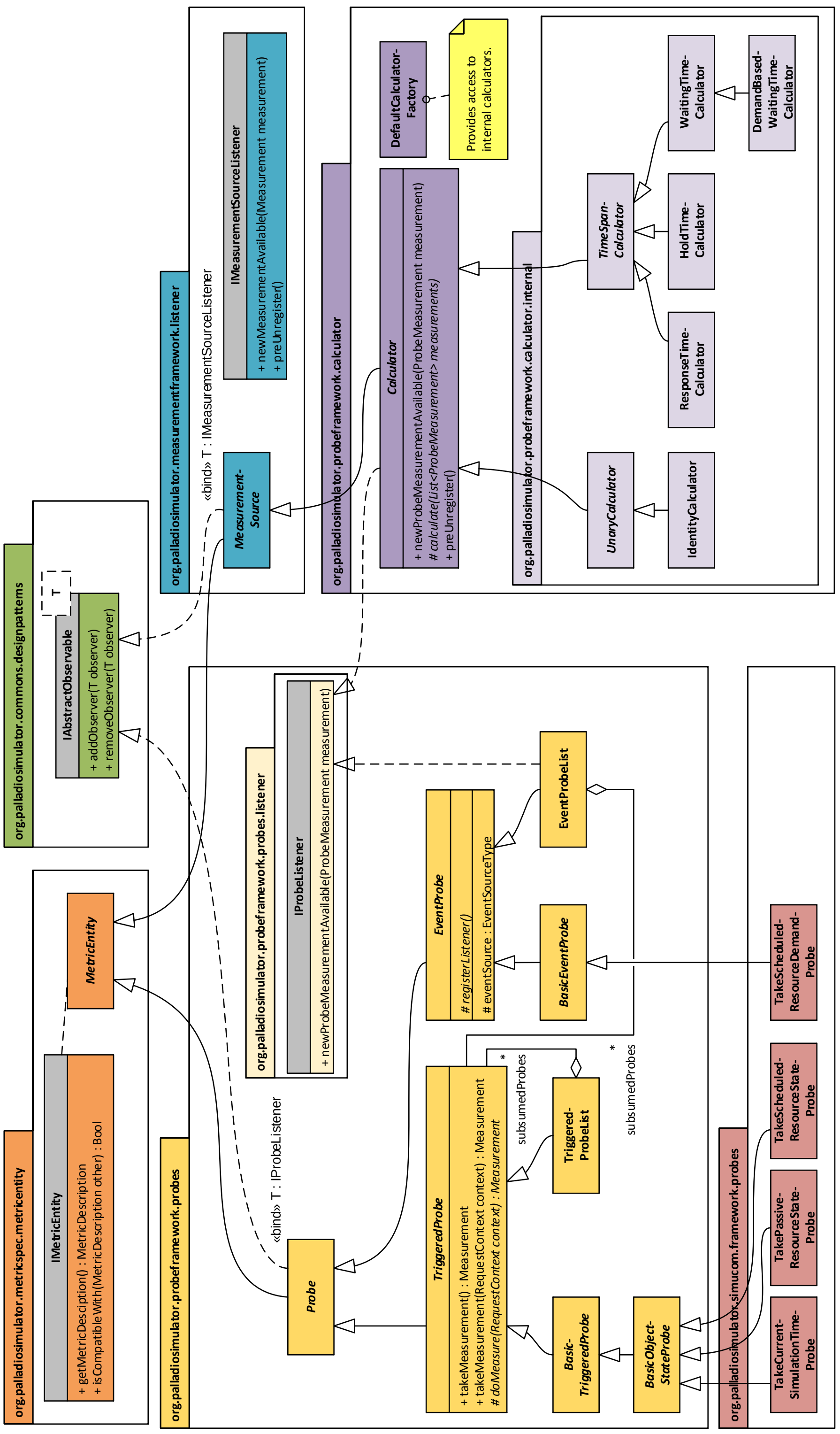

org.palladiosimulator.metricspec.metricentity
IMetricEntity
+ getMetricDescription() : MetricDescription
+ isCompatibleWith(MetricDescription other) : Bool
MetricEntity
org.palladiosimulator.commons.designpatterns
IAbstractObservable
T
+ addObserver(T observer)
+ removeObserver(T observer)
org.palladiosimulator.measurementframework.listener
«bind» T : IMeasurementSourceListener
Measurement-Source
IMeasurementSourceListener
+ newMeasurementAvailable(Measurement measurement)
+ preUnregister()
org.palladiosimulator.probeframework.calculator
Calculator
+ newProbeMeasurementAvailable(ProbeMeasurement measurement)
# calculate(List<ProbeMeasurement> measurements)
+ preUnregister()
DefaultCalculator-Factory
Provides access to internal calculators.
org.palladiosimulator.probeframework.calculator.internal
UnaryCalculator
IdentityCalculator
TimeSpan-Calculator
ResponseTime-Calculator
HoldTime-Calculator
WaitingTime-Calculator
DemandBased-WaitingTime-Calculator
org.palladiosimulator.probeframework.probes
«bind» T : IProbeListener
Probe
org.palladiosimulator.probeframework.probes.listener
IProbeListener
+ newProbeMeasurementAvailable(ProbeMeasurement measurement)
EventProbe
# registerListener()
# eventSource : EventSourceType
EventProbeList
BasicEventProbe
TriggeredProbe
+ takeMeasurement() : Measurement
+ takeMeasurement(RequestContext context) : Measurement
# doMeasure(RequestContext context) : Measurement
Triggered-ProbeList
subsumedProbes
*
Basic-TriggeredProbe
BasicObject-StateProbe
org.palladiosimulator.simucom.framework.probes
TakeCurrent-SimulationTime-Probe
TakePassive-ResourceState-Probe
TakeScheduled-ResourceState-Probe
TakeScheduled-ResourceDemand-Probe

Figure 5.1: Packages, Classes, and Interfaces of Probe Framework

also inherit from MeasurementSource, thus, making them observable objects on their own. For instance, they can provide newly calculated measurements to recorders (recorders implement the IMeasurementSourceListener interface and, thus, can act as observees of calculators).

Given these fundamental properties of probes and calculators, we discuss different types of probes in Sec. 5.1.1 and different types of calculators in Sec. 5.1.2.

5.1.1 Probe Types

The Probe Framework provides two types of probes: triggered probes (TriggeredProbe class) and event probes (EventProbe class).

Triggered Probes

Triggered probes measure as soon as their takeMeasurement method is invoked (with optional context parameter of type RequestContext, e.g., the thread that triggered the probe). This invocation triggers the abstract method doMeasure (with an optionally empty context). After doMeasure computed a new measurement, takeMeasurement further informs all registered observers about the newly available measurement.

The method doMeasure itself has to be provided by classes inheriting from the TriggeredProbe class. Currently, there are two types of such classes: basic triggered probes (BasicTriggeredProbe class) and triggered probe lists (TriggeredProbeList class).

Basic triggered probes (abstract class BasicTriggeredProbe) implement the doMeasure method by returning a BasicMeasurement, i.e., a measurement for a BaseMetricDescription. Therefore, they are constructed by passing an appropriate base metric description as a parameter that is used to construct basic measurements. For determining the measurement itself, they invoke the abstract method getBasicMeasure.

Currently, there is one class inheriting from BasicTriggeredProbe: the abstract class **BasicObjectStateProbe**. BasicObjectStateProbe is a probe that observes an object's state, thus, additionally maintaining an observeredStateObject member variable. However, it leaves the implementation of the abstract method getBasicMeasure to its subclasses that are currently part of the SimuCom Framework package org.palladiosimulator.simucom.framework.probes:

TakeCurrentSimulationTimeProbe Measures a point in time metric (in seconds) by requesting the current simulation time from the simulation controller (observed state object).

TakePassiveResourceStateProbe Measures a passive resource state metric (dimensionless) by calculating the difference between the capacity of the passive resource (observed state object) and its available resources.

TakeScheduledResourceStateProbe Measures a CPU state metric (dimensionless) by requesting the queue length from the scheduled CPU resource (observed state object). For example, the queue length can be used to calculate CPU utilization (the queue length gives the number of currently waiting jobs to be served by the CPU). Because scheduled resources can have many instances (e.g., a CPU can have many cores), a pointer to the concrete instance is used additionally.

Triggered probe lists (class TriggeredProbeList) group a list of subsumed, triggered probes. Therefore, triggered probe lists can implement doMeasure by invoking doMeasure on each of their subsumed probes and by returning a measurement tuple of measurement results from these probes.

Event Probes

Event probes measure as soon as an event is emitted for which they are registered. Therefore, they explicitly refer to an eventSource object. Subclasses have to realize the registration to this object by implementing the abstract method registerListener. Currently, there are two types of such subclasses: basic event probes (BasicEventProbe class) and event probe lists (EventProbeList class).

Basic event probes (abstract class BasicEventProbe) provide a generic notify method. This method allows subclasses to pass an event measurement to it when they receive an event. Subsequently, notify can generically inform registered probe listeners about a newly available probe measurement. However, BasicEventProbe leaves the implementation of the abstract method registerListener to its subclasses. The most common subclass is currently part of the SimuCom Framework package org.palladiosimulator.simucom.framework.probes:

TakeScheduledResourceDemandProbe Measures a resource demand metric (in seconds) by listening to demands to a scheduled resource (event source type). Therefore, it has to implement the IDemandListener interface and to register itself in the registerListener method to this scheduled resource. The measured demand is directly received by the demand call-back method of the listener.

Event probe lists (class EventProbeList) group a list of subsumed, *triggered* probes that are triggered as soon as an additional, dedicated *event* probe emits an event. EventProbeList can register itself to this event probe because it implements the IProbeListener interface. Once a new measurement is available (via the call-back method newProbeMeasurementAvailable), an event probe list creates a measurement tuple by creating a list with the new measurement plus the measurements from subsumed triggered probes. These measurements are received by invoking doMeasure on each subsumed probe (that is possible since only triggered probes are subsumed). Finally, registered listeners (e.g., calculators) are informed about the newly available measurement tuple.

5.1.2 Calculator Types

Calculators attach themselves to (a set of) probes to transform their measures to a measurement to be used further (e.g., for visualization or recording). Therefore, they specify which measures are of interest outside from analyzers.

The main super class of all calculators is the abstract class **Calculator**. The Probe Framework provides a factory class (DefaultCalculatorFactory) to create concrete calculator objects that inherit from this base class. These concrete calculators are collected in the internal package org.palladiosimulator.probeframework.calculator.internal (access without using the factory class is impossible in OSGI because classes of internal packages are not exported).

Calculator objects expect a list of probes to be measured before they start their calculation. For example, a response time calculator needs a measurement series of two probe measurements (one for start time and one for end time of an operation call). For such a measurement series, calculators maintain a memory to store the measurements received from observed probes. As soon as the last measurement sample arrives, calculators start their calculation by invoking the template method calculate (c.f., Calculator class).

Currently, there are two direct subclasses of Calculator: unary calculators (abstract class UnaryCalculator) and time span calculators (abstract class TimeSpanCalculator).

Unary Calculators

Unary calculators expect exactly one probe, thus, restricting calculators to exactly one observed probe. They leave the implementation of the template method calculate to their subclasses.

Currently, the only such subclass is the **IdentityCalculator** class. Identity calculators are unary calculators that directly let probe measurements pass through. For example, the current state of a “take passive resource state probe” may directly be passed through in order to determine its utilization.

Time Span Calculators

Time span calculators (abstract class TimeSpanCalculator) calculate a time span. These calculators expect two probes, each providing at least a point in time measurement. Subsequently, they calculate the time span by subtracting the point in time of the first probe from the point in time of the second probe. The result is a (start point in time, time span)-tuple, i.e., a measurement typed with a metric set.

Currently, three subclasses of TimeSpanCalculator exist:

ResponseTimeCalculator Calculates a time span representing the response time (as defined by the response time metric). It expects a probe giving the start and a probe giving the end point in time of an operation call. The final result is a (start point in time, response time)-tuple.

HoldTimeCalculator Calculates a time span representing the hold time (as defined by the hold time metric). It expects a probe giving the start of holding and a probe giving the end of holding, e.g., in a passive resource pool. The final result is a (start point in time, hold time)-tuple.

WaitingTimeCalculator Calculates a time span representing the waiting time (as defined by the waiting time metric). It expects a probe giving the start of waiting and a probe giving the end of waiting, e.g., at a passive resource pool. The final result is a (start point in time, waiting time)-tuple.

Besides this default waiting time calculation, the subclass **DemandBased-WaitingTimeCalculator** of the WaitingTimeCalculator class employs an alternative. Demand-based waiting time calculators can calculate the waiting time for resources in environments where the stop of the waiting period cannot be observed directly. Rather, the following values (respectively events) should be observable: *start* ("request for processing"-event), *stop* ("end of processing"-event; note that this is different from the waiting period stop), and *demand* (the demanded time). The waiting time results from calculating $(stop - start) - demand$. The final result is a (start point in time, waiting time)-tuple.

5.2 Using Probes and Calculators

In this section, we exemplify the usage of probes and calculators. For this exemplification, we use a simple probe (a simple specialization of a triggered basic object state probe) that takes the time of a simulation in Sec. 5.2.1. Subsequently, we use two of these probes to construct and use a response time calculator in Sec. 5.2.2. Finally, we point to further usage examples that can directly be found within the Probe Framework in Sec. 5.2.3.

5.2.1 Using an "Example Take Current Time Probe"

In Listing 9.1, we show how an example probe for taking the current time of a simulation can be used. The probe is instantiated by registering it to the simulation (line 5). Afterwards, an example request context is created that provides context information about the measurement to be taken (line 6). Finally, the simulation runs for 100 seconds (line 8) and then the probe measurement is triggered (line 9).

```
1 ExampleTakeCurrentTimeProbe probe;
2 RequestContext context;
3 ProbeMeasurement probeMeasurement;
4
5 probe = new ExampleTakeCurrentTimeProbe(simContext);
6 requestContext = new RequestContext("example probing");
7
8 simContext.setSimulatedTime(100d);
9 probeMeasurement = probe.takeMeasurement(requestContext);
```

Listing 5.1: Using an “example current time probe”

5.2.2 Using a “Response Time Calculator”

In Listing 5.2, we show how an example calculator for calculating the response time of an example operation call (myOperationCall) can be used. The calculator is instantiated using the DefaultCalculatorFactory (line 11). For this instantiation, two “example current time probes” (see previous section) are used: one for the start of myOperationCall, one for the end of myOperationCall. We also register a simple observer to this calculator that directly outputs the measurement (line 19). Afterwards, we start the simulation at time 0 seconds and trigger the start probe (lines 26/27). Subsequently, we run the simulation for 100 seconds and trigger the end probe (lines 29/30). Because that was the last registered probe of the response time calculator, the output “Response Time: 100 seconds” should have been created.

```
 1  ICalculatorFactory calculatorFactory;
 2  RequestContext requestContext;
 3  ExampleTakeCurrentTimeProbe startProbe;
 4  ExampleTakeCurrentTimeProbe endProbe;
 5  Calculator rtCalculator;
 6
 7  calculatorFactory = new DefaultCalculatorFactory();
 8  requestContext = new RequestContext("myOperationCall");
 9  startProbe = new ExampleTakeCurrentTimeProbe(simContext);
10  endProbe = new ExampleTakeCurrentTimeProbe(simContext);
11  rtCalculator = calculatorFactory.buildResponseTimeCalculator(
12                  "Example Response Time Calculation",
13                  Arrays.asList(
14                    (Probe) startProbe,
15                    (Probe) endProbe
16                  )
17               );
18
19  rtCalculator.addObserver(new IMeasurementSourceListener() {
20          @Override
21          public void newMeasurementAvailable(Measurement measurement)
                 {
22                  System.out.println("Response Time: "+measurement);
23          }
24  }
25
26  simContext.setSimulatedTime(0.0d);
27  startProbe.takeMeasurement(requestContext);
28
29  simContext.setSimulatedTime(100.d);
30  endProbe.takeMeasurement(requestContext);
31
32  // Output is "Response Time: 100 seconds"
```

Listing 5.2: Using a "response time calculator"

5.2.3 Further Examples on Using Probes and Calculators

We provide further usage examples of probes and calculators within the Probe Framework. We provide these examples in the form of test cases in the package org.palladiosimulator.probeframework.tests. Their documentation is directly available within the code (JavaDoc).

5.3 Custom Probes and Calculators

In this section, we exemplify the creation of custom probes and calculators. For this exemplification, we describe how to create the previously used "example current time probe" in Sec. 5.3.1. In the Probe Framework, we provide several further examples for probe creation. In particular, we provide several examples of calculator creation. Therefore, we just point to these examples in Sec. 5.3.2; customization should be straight-forward.

5.3.1 Creating an "Example Take Current Time Probe"

In Listing 5.3, we show how to create the previously used "example current time probe". This creation only requires one constructor (ExampleTakeCurrentTimeProbe) and one method (getBasicMeasure) to be implemented. Because this probe is a basic object state probe, we simply have to specify which object holds the state to measure and what is the metric we want to measure. The state object is the simulation and the metric is a point in time metric as specified in the constructor (line 9). In getBasicMeasure, we simply have to receive the simulation time from the state object (i.e., the simulation) to determine the measurement (line 14).

5.3.2 Further Examples on Creating Probes and Calculators

We provide further implementation examples of probes and calculators within the Probe Framework. Examples for probe implementations are in the package org.palladiosimulator.probeframework.probes.example. Examples for calculator implementations are in the package org.palladiosimulator.probeframework.calculator.internal. Their documentation is directly available within the code (JavaDoc).

```
public class ExampleTakeCurrentTimeProbe
        extends BasicObjectStateProbe<
                        SimpleSimulationContext,
                        Double,
                        Duration
                > {

    public ExampleTakeCurrentTimeProbe(SimpleSimulationContext
        simulationContext) {
        super(simulationContext, MetricDescriptionConstants.
            POINT_IN_TIME_METRIC);
    }

    @Override
    protected Measure<Double, Duration> getBasicMeasure(
        RequestContext measurementContext) {
        return Measure.valueOf(getStateObject().getSimulatedTime(),
            SI.SECOND);
    }

}
```

Listing 5.3: Using a “response time calculator”

6 Kieker Framework

In this chapter, we discuss the Kieker Framework [4] integration of QuAL. The Kieker Framework allows Palladio developers to setup pipes and filters chains for measurements.

For example, measurements travel from calculators (measurement sources) to recorders (measurement sinks). On the path from source to sink, such measurements can be filtered, e.g., if only measurements that break a certain service level objective are of interest. Such a scenario can heavily foster from Kieker because it provides the means to specify such filters and pipe them from one to the other. For instance, Kieker comes with direct support for JMS-based transfer of data through such chains. Note that Kieker realizes a push-based pipes and filters chain, i.e., elements are pushed through the pipeline, not pulled. Therefore, a filter explicitly sends measurements to its output ports once it has a measurement available. In contrast, the visualization components of QuAL's UI Framework are pull-based - they request the data from the respective filter to get it (c.f., Chap. 10).

Unfortunately, we have not realized the Kieker integration yet - we so far limited ourselves to plan its integration here, in this document. The overview in Fig. 1.5 already reflects our integration ideas and, thus, does not show the actual connection between calculators and recorders/live visualization. In the current state, we directly connect calculators to recorders. This has the consequence that we do not support any filters or JMS-based delivery of measurements. Moreover, we also do not support live visualization yet. The integration of Kieker is one of the very next steps planned for future releases of QuAL; including a full developer documentation.

7 Recorder Framework

This chapter covers the Recorder Framework of QuAL. Palladio developers can use this framework to specify custom recorders for their measurements. Currently, two of such recorders exist: “Sensor Framework” and “Experiment Data Persistency & Presentation (EDP2)”. Sensor Framework was the first recorder to be realized and is now, with the advent of the EDP2 recorder, deprecated. Therefore, this chapter exemplifies the EDP2 recorder only.

In Fig. 7.1, we illustrate the relation between the generic classes of the Recorder Framework and their use by the EDP2 recorder. The upper part shows recorders and recorder configurations as the main concepts of the Recorder Framework (Sec. 7.1). The lower part exemplifies the specification of concrete recorders by means of EDP2 recorder classes inheriting from Recorder Framework classes (Sec. 7.2). Similarly to this EDP2 recorder, Palladio developers can create custom recorders and enable these by dedicated extension points (Sec. 7.3).

7.1 Recorders and Recorder Configurations

Recorders (Sec. 7.1.1) and recorder configurations (Sec. 7.1.2) are the main concepts of the Recorder Framework. In the upper part of Fig. 7.1, we visualize corresponding classes for these concepts.

7.1.1 Recorders

Recorders are realized by an interface IRecorder and an abstract implementation AbstractRecorder of the interface. Concrete recorders have to inherit from this abstract recorder class.

IRecorder reflects the typical workflow when using recorders: (1) initialize the recorder using a recorder configuration (initialize method), (2) write a given measurement to the recorder (writeData method), and (3) flush all measurements to the recorder for final storage (flush method).

Moreover, IRecorder inherits from IMeasurementSourceListener of the Measurement Framework (c.f., Sec. 3.2) such that recorders can be added as observers to measurement source, e.g., to calculators. AbstractRecorder directly implements this additional interface by delegating its methods to the corresponding IRecorder methods (when new measurements are available, they are directly written and when being unregistered, the recorder is flushed first).

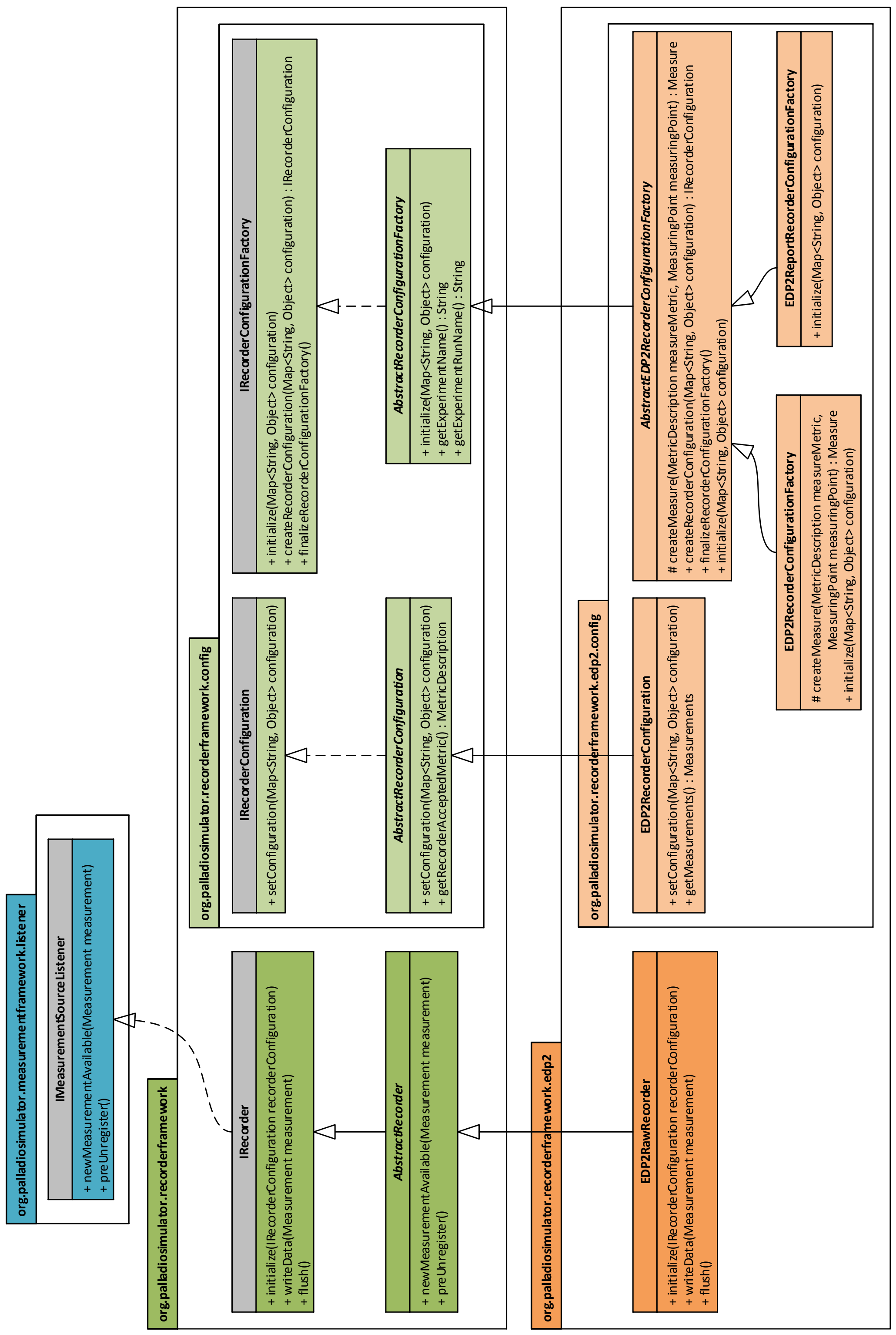

org.palladiosimulator.measurementframework.listener
IMeasurementSourceListener
+ newMeasurementAvailable(Measurement measurement)
+ preUnregister()
org.palladiosimulator.recorderframework
IRecorder
+ initialize(IRecorderConfiguration recorderConfiguration)
+ writeData(Measurement measurement)
+ flush()
AbstractRecorder
+ newMeasurementAvailable(Measurement measurement)
+ preUnregister()
org.palladiosimulator.recorderframework.config
IRecorderConfiguration
+ setConfiguration(Map<String, Object> configuration)
AbstractRecorderConfiguration
+ setConfiguration(Map<String, Object> configuration)
+ getRecorderAcceptedMetric() : MetricDescription
IRecorderConfigurationFactory
+ initialize(Map<String, Object> configuration)
+ createRecorderConfiguration(Map<String, Object> configuration) : IRecorderConfiguration
+ finalizeRecorderConfigurationFactory()
AbstractRecorderConfigurationFactory
+ initialize(Map<String, Object> configuration)
+ getExperimentName() : String
+ getExperimentRunName() : String
org.palladiosimulator.recorderframework.edp2
EDP2RawRecorder
+ initialize(IRecorderConfiguration recorderConfiguration)
+ writeData(Measurement measurement)
+ flush()
org.palladiosimulator.recorderframework.edp2.config
EDP2RecorderConfiguration
+ setConfiguration(Map<String, Object> configuration)
+ getMeasurements() : Measurements
AbstractEDP2RecorderConfigurationFactory
# createMeasure(MetricDescription measureMetric, MeasuringPoint measuringPoint) : Measure
+ createRecorderConfiguration(Map<String, Object> configuration) : IRecorderConfiguration
+ finalizeRecorderConfigurationFactory()
+ initialize(Map<String, Object> configuration)
EDP2RecorderConfigurationFactory
# createMeasure(MetricDescription measureMetric, MeasuringPoint measuringPoint) : Measure
+ initialize(Map<String, Object> configuration)
EDP2ReportRecorderConfigurationFactory
+ initialize(Map<String, Object> configuration)

Figure 7.1: The EDP2 Recorder uses Recorders and Recorder Configurations from the Recorder Framework

7.1.2 Recorder Configurations

As stated in the previous section, recorders are configured using a recorder configuration. Such recorder configurations are typed by the IRecorderConfiguration interface that provides the setConfiguration method to initialize configurations based on key-value maps. AbstractRecorderConfiguration provides an abstract implementation of IRecorderConfiguration, along with the convenience method getRecorderAcceptedMetric that returns the particular metric description a recorder object is configured with. Concrete recorder configurations have to inherit from this abstract configuration class.

The creation of recorder configurations is realized via the factory method software design pattern. Accordingly, we the interface IRecorderConfigurationFactory allows to create new recorder configuration objects (createRecorderConfiguration). Before creation of a configuration, the factory has to be initialized using a given key-value map (initialize). Before destruction of a configuration, the factory has to be finalized (finalizeRecorderConfigurationFactory).

The abstract class AbstractRecorderConfigurationFactory implements initialize as well as provides convenience methods to get the experiment name and the experiment run name based on the a previous initialization. Concrete recorder configuration factories have to inherit from AbstractRecorderConfigurationFactory.

7.2 Example: the Experiment Data Persistency & Presentation (EDP2) Recorder

The Experiment Data Persistency & Presentation (EDP2) Recorder realizes a concrete recorder as well as associated configurations and factories by extending the Recorder Framework. We visualize corresponding classes in the lower part of Fig. 7.1.

EDP2RawRecorder subclasses AbstractRecorder by providing EDP2-specific implementations for IRecorder methods. Similarly, EDP2RecorderConfiguration provides an EDP2-specific implementation for recorder configurations.

EDP2 supports two conceptually different types of recorder configuration factories: a basic factory for single experiment runs as well as a report factory for measurements over several experiment runs. Therefore, we provide AbstractEDP2RecorderConfigurationFactory as an abstract subclass of AbstractRecorderConfigurationFactory that provides common functionality for both EDP2 factories. Based on this abstract superclass, EDP2RecorderConfigurationFactory implements the factory for basic factories and EDP2ReportRecorderConfigurationFactory the factory for report factories.

7.3 Custom Recorders and Extension Points

Palladio developers who want to provide custom recorder implementations should proceed analogously to our EDP2 recorder implementation (Sec. 7.2). That is, they should provide subclasses for AbstractRecorder, AbstractRecorderConfiguration, and AbstractRecorderConfigurationFactory.

Moreover, these custom recorders have to be enabled using the dedicated extension point provided by the Recorder Framework[1]. The extension point particularly requests implementations for AbstractRecorder and AbstractRecorderConfigurationFactory. The EDP2 recorder (Sec. 7.2) exemplifies an extension of this extension point.

[1]The plug-in ID and extension point ID are both org.palladiosimulator.recorderframework.

8 Simulation Instrumentation using Monitor Repository Models

As has been outlined in the previous chapters, QuAL allows for flexible collection of metric measurements in Palladio analyses. By design, the simulator SimuCom and the analytical evaluations have collected and persisted measurements for a fixed set of measurement types, like the aggregate CPU utilization, or total response time of Usage Scenarios. While this does not require effort for specifying desired measured metrics, it comes at the disadvantage of having to collect all measurements for the predefined types. The Monitor Repository model was introduced to allow for a flexible definition of measurements to be collected and instrumented in the simulation. It builds upon the EDP2 Measuring Point Model of QuAL.

Currently, only SimuLizar supports the flexible specification of simulation monitoring points using the Monitor Repository model.

8.1 Monitor Repository Model

Figure 8.1 depicts the central elements of the Monitor Repository Model. A Monitor Repository contains a set of *Monitors*. Each Monitor specifies an instrumented Measuring Point. The Measuring Points are instances of EDP2's Measuring Point model. A Measuring Point represents the part of the system that is to be mon-

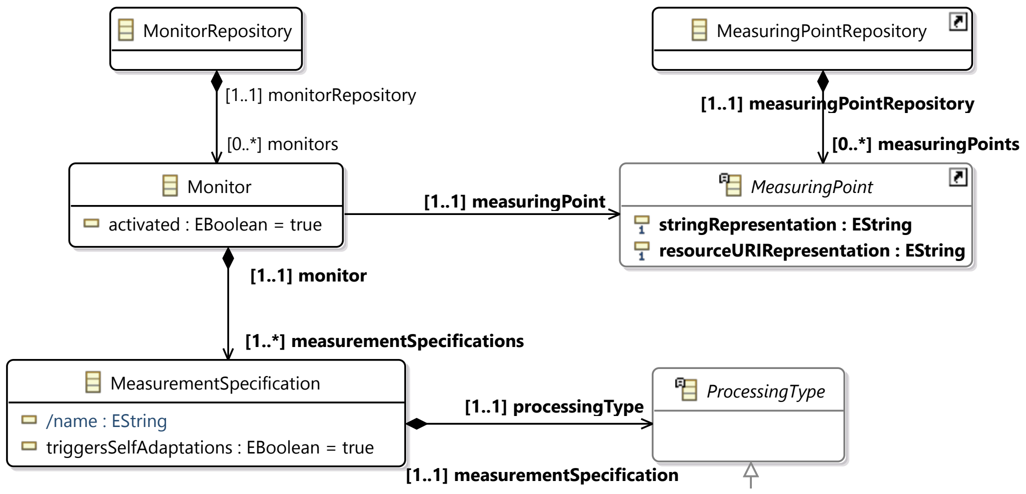


Figure 8.1: Overview of Monitor Repository Model

itored. Measuring Points can be grouped in a *MeasuringPointRepository*. Every Monitor can be *activated* or deactivated. Per Monitor, there exists a *MeasurementSpecification* that describes the collected Measurement. In case of analysis tools like SimuLizar that support self-adaptations, *triggersSelfAdaptations* determines whether measurements should be passed as input to self-adaptation mechanisms. If it is deactivated, measurements can be stored in the Recorder Framework. In cases where it does not make sense to persist the aggregate metrics, a realization of the measurement instrumentation defined in a Monitor might not store the measurements in the Recorder Framework.

Every *Measurement Specification* has a *Processing Type* that defines how measurements are to be collected and propagated from the Probe Framework to connected analyses, such as SimuLizar's self-adaptation mechanisms. Figure 8.2 provides an overview of supported Processing Specifications. *FeedThrough* defines that a measurement recorded at the Monitor should directly be passed through without further processing.

Aggregation subsumes Processing Types that aggregate measurements recorded at Monitors. The definition of aggregation allows a flexible processing of results similar to Kieker's Pipes and Filters. *TimeDriven* are a Processing Type suitable for measurement specifications that demand a sliding window, e.g., before being recorded, but are not based on statistical aggregation. It extends Aggregation. Examples for such measurement specifications are the window-based utilization calculation or the power and energy calculation based thereof. *windowLength* defines the length of the window, *windowIncrement* the increment with which the interval is moved forward.

MeasurementDrivenAggregation specifies that a set of measurements should be aggregated every *numberOfMeasurements*. *VariableSizeAggregation* aggregates all measurements from the set of measurements restricted by a retrospection interval of the length *retrospectionlength*. Measurements that are older than the current measurement's measuring time minus *retrospectionLength* are discarded. *FixedSizeAggregation* aggregates the last *numberOfMeasurements* since the current measurement, independent of the time at which they were measured.

Every Aggregation is processed using an extensible and exchangeable *StatisticalCharacterization*. As reference, the Monitor Repository model already supports Geometric and Arithmetic Mean as well as Median. Additional Statistical Characterizations can be supported by extending the model using Ecore's *Child Creation Extender*[1] model feature.

8.2 Extending the Monitor Repository Model

The Monitor Repository model has been built to be extensible by design. This section discusses the two main extension scenarios, namely adding support for new aggregate functions and new aggregation processing types.

[1]http://ed-merks.blogspot.de/2008/01/creating-children-you-didnt-know.html

8.2.1 Adding Statistical Characterizations

As explained in the previous section, StatisticalCharacterization can be extended to support further aggregation functions. To implement a custom aggregator, a subclass of *StatisticalCharacterization* needs to be realized in an extension model.

```
return new org.palladiosimulator.monitorrepository.
    statisticalcharacterization.GeometricMeanAggregator(
    expectedDataMetric);
```

Listing 8.1: Body value of the GeometricMean's *getAggregator()* implementation

StatisticalCharacterization defines the *getAggregator()* function. An implementing subclass needs to override this method. In the model, this is achieved by adding a *GenModel* annotation to the method with the *key* set to the method body. The *value* of the annotation should create an instance of the actual implementation of the *StatisticalCharacterizationAggregator*. Listing 8.1 shows the *getAggregator()* method body of GeometricMean.

expectedDataMetric is the metric characterizing the resulting aggregate.

8.3 Adding Custom Processing Types

Beyond custom aggregation function, Monitor Repository also supports the introduction of custom *Processing Types*. This allows for an implementation of custom filters, e.g., to support a pipeline-based processing of results. This section discusses an extension of the existing Processing Types by a generic *Map* function. The added Processing Type supports the specification of a mapping, i.e., some sort of transformation function that shall be applied to every measurement. The source code for the plugin can be found in the Palladio SVN[2].

Figure 8.3 depicts a model overview of the Map function extension. *Map* applies the mapping function of a set *Mapper* to an input *Measuring Value*. The mapping function hereby is a Java UnaryOperator. An example Mapping function realized as part of the model is the *Exponential Smoothing* function, that is, e.g., used by the Linux kernel to calculate CPU utilization metrics. The following listing contains the body of Exponential Smoothing's *getMappingFunction()*.

```
return (input) -> {
            org.palladiosimulator.metricspec.
                NumericalBaseMetricDescription expectedMetric = (org
                .palladiosimulator.metricspec.
                NumericalBaseMetricDescription) getMap()
                     .getOutputMetricDescription();
            javax.measure.unit.Unit<Quantity> unit = expectedMetric.
                getDefaultUnit();
            double value = input.getMeasureForMetric(expectedMetric)
                .doubleValue(unit);
```

[2]https://svnserver.informatik.kit.edu/i43/svn/code/MonitorRepository/trunk/org.palladiosimulator.monitorrepository.map/

```
            Measure<?, javax.measure.quantity.Duration> time = input.getMeasureForMetric(
                    org.palladiosimulator.metricspec.constants.MetricDescriptionConstants.POINT_IN_TIME_METRIC);
            double oldValue = getSmoothedValue() == null ? 0d : getSmoothedValue().doubleValue(unit);
            setSmoothedValue(
                    Measure.valueOf(oldValue * (1 - getSmoothingFactor()) + value * getSmoothingFactor(), unit));
            return new org.palladiosimulator.measurementframework.TupleMeasurement(
                    (org.palladiosimulator.metricspec.MetricSetDescription) input.getMetricDesciption(), time,
                    getSmoothedValue());
        };
```

Listing 8.2: Body value of the ExponentialSmoothing's *getMappingFunction()*

The smoothing function applies a exponentially weighted moving average to a measurement input. It outputs the smoothed value for the point in time of the last incoming measurements.

Since the base Monitor Repository model does not contain *Map* functions, the analysis also needs to be extended to support registering Mapping functions. Section 9.1 explains how extensions to the monitoring and measurement logic can be realized for SimuLizar.

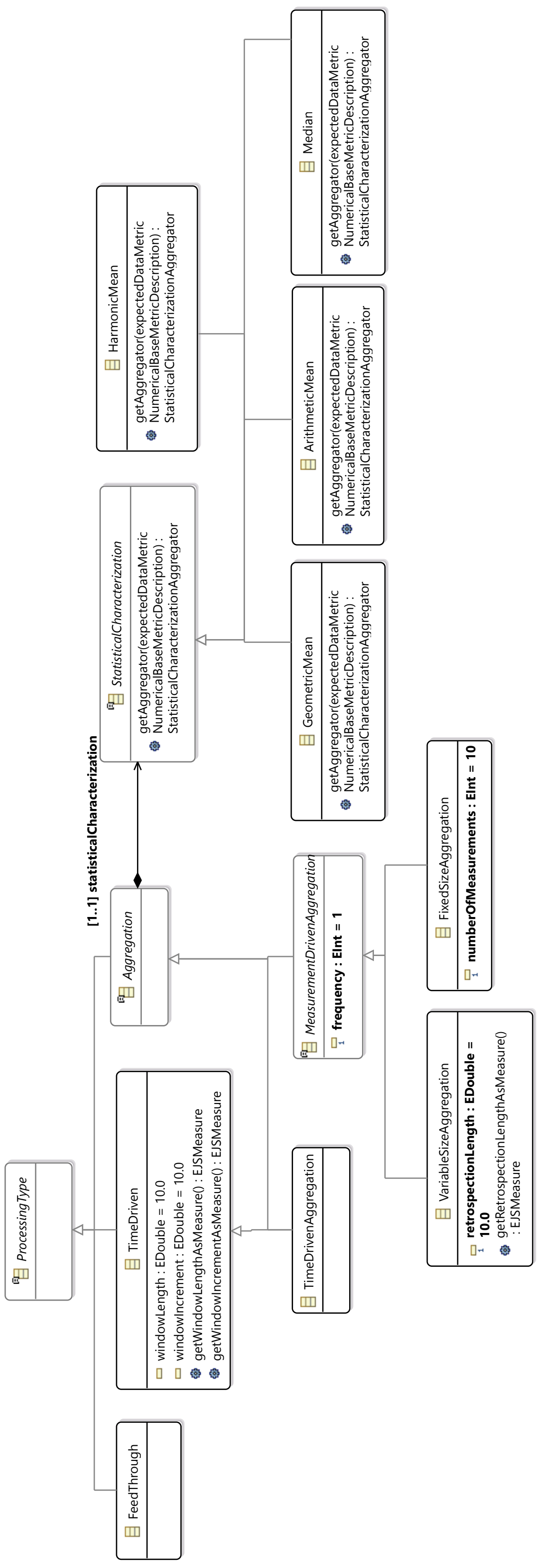

ProcessingType
FeedThrough
TimeDriven
windowLength : EDouble = 10.0
windowIncrement : EDouble = 10.0
getWindowLengthAsMeasure() : EJSMeasure
getWindowIncrementAsMeasure() : EJSMeasure
TimeDrivenAggregation
Aggregation
[1..1] statisticalCharacterization
StatisticalCharacterization
getAggregator(expectedDataMetric : NumericalBaseMetricDescription) : StatisticalCharacterizationAggregator
HarmonicMean
getAggregator(expectedDataMetric : NumericalBaseMetricDescription) : StatisticalCharacterizationAggregator
GeometricMean
getAggregator(expectedDataMetric : NumericalBaseMetricDescription) : StatisticalCharacterizationAggregator
ArithmeticMean
getAggregator(expectedDataMetric : NumericalBaseMetricDescription) : StatisticalCharacterizationAggregator
Median
getAggregator(expectedDataMetric : NumericalBaseMetricDescription) : StatisticalCharacterizationAggregator
MeasurementDrivenAggregation
frequency : EInt = 1
FixedSizeAggregation
numberOfMeasurements : EInt = 10
VariableSizeAggregation
retrospectionLength : EDouble = 10.0
getRetrospectionLengthAsMeasure() : EJSMeasure

Figure 8.2: Processing Types supported by Monitor Repository

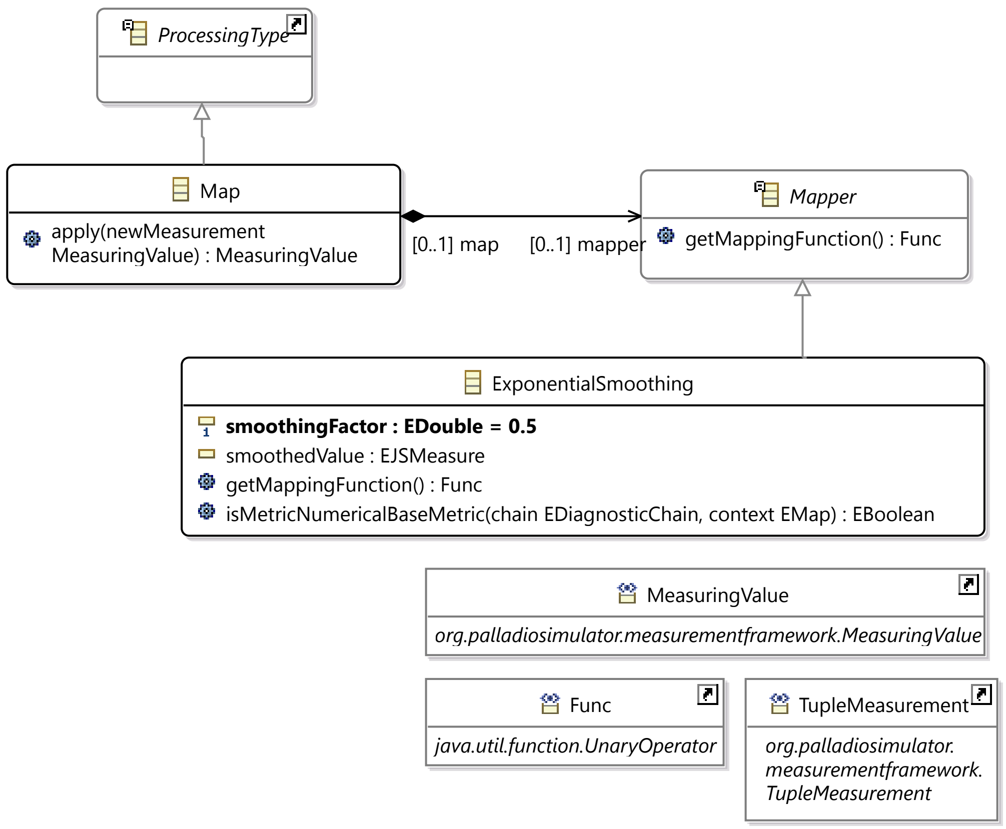

ProcessingType
Map
apply(newMeasurement MeasuringValue) : MeasuringValue
[0..1] map
[0..1] mapper
Mapper
getMappingFunction() : Func
ExponentialSmoothing
smoothingFactor : EDouble = 0.5
smoothedValue : EJSMeasure
getMappingFunction() : Func
isMetricNumericalBaseMetric(chain EDiagnosticChain, context EMap) : EBoolean
MeasuringValue
org.palladiosimulator.measurementframework.MeasuringValue
Func
java.util.function.UnaryOperator
TupleMeasurement
org.palladiosimulator.measurementframework.TupleMeasurement

Figure 8.3: Map Function extension

9 Extensible Monitoring in SimuLizar

Initially, SimuLizar had supported a set of predefined measurement types such as response times, or aggregate CPU utilization. In order to support the extension of SimuLizar to additional metric measurements such as power consumption or the Mapping functions discussed in Section 8.3, a set of extension points have been added. An extension may register itself to the extension point via Eclipse's extension point mechanism[1]. This chapter provides an overview of extension points for monitoring and measurement processing in SimuLizar.

9.1 Extending Measurements recorded by the Probe Framework

The extension point *org.palladiosimulator.simulizar.interpreter.listener.probeframework* can be used by SimuLizar extensions to record additional measurements by means of a *ProbeFrameworkListener*. Any extending plugin must extend the abstract class *AbstractRecordingProbeFrameworkListenerDecorator*. Before starting a simulation run, SimuLizar registers its main *ProbeFrameworkListener* with the extension by calling

```
setProbeFrameworkListener(final AbstractProbeFrameworkListener
   probeFrameworkListener)}
```

Once the SimuLizar analysis setup has set the decorated listener, the analysis setup calls *registerMeasurements()* to register any measurements contributed by the SimuLizar extension.

An example extension that contributes measurements to SimuLizar via the Probe Framework extension point is *org.palladiosimulator.simulizar.aggregation*. Aggregation is responsible for handling the correct registration of measurements collected for *FixedSizeAggregation* and *VariableSizeAggregation* of the Monitor Repository model. For an explanation of the aggregation functions refer to Section 8.1. The method is responsible for registering the aggregation calculators with the Probe framework. Aggregation measurements are only set up when *isTriggersSelfAdaptations* has been set for the Monitor of the aggregator. The aggregation measurements are propagated to the Palladio Runtime Measurement that SimuLizar exposes to reconfiguration mechanisms.

[1]`http://www.vogella.com/tutorials/EclipseExtensionPoint/article.html`

The SimuLizar Power extension *org.palladiosimulator.simulizar.power*[2] is another example of an extension that leverages the Probe Framework extension point. Unlike the Aggregation extension point, it also registers the collected Power and Energy metric measurements to be propagated to the Recorder framework.

The custom Aggregation Processing Type discussed in Section 8.3 is also added to SimuLizar's measurement infrastructure via the Probe Framework extension point. The *MonitorRepositoryMapProbeFrameworkListenerDecorator* of *org.palladiosimulator.simulizar.monitorrepository.map* adds Probe Framework calculators that apply the Mapping function to the measurements collected at the Measuring Point of the aggregating Monitor.

9.2 Dynamic Extension of Measurements with a Model Observer

SimuLizar supports the analysis of self-adaptive software systems. It is not possible to register all measurements in a self-adaptive system prior to the execution of the analysis. If, for example, a scale-out occurs in a horizontally scalable application, one might want to collect response time measurements for the newly launched component instance. The Probe Framework Measurement extension discussed in Section 9.1 is well suited for registering additional measurements during the setup of an analysis run. This works well for additional measurements that are known at the time a SimuLizar analysis is started. However, it does not support the dynamic registration of additional measurements. Dynamic registration and deregistration of metric measurement collection in SimuLizar can be realized via the *org.palladiosimulator.simulizar.modelobserver* extension point.

Extensions registering to the extension point have to implement the IModelObserver interface:

```java
public interface IModelObserver {

    public void initialize(final AbstractSimuLizarRuntimeState
        runtimeState);

    public void unregister();

}
```

Listing 9.1: IModelObserver interface used to register a SimuLizar model observer

The abstract class *AbstractModelObserver* extends the IModelObserver interface and implements the most commonly used model observing code. It is recommended to extend this class when implementing dynamic measurement extensions. An AbstractModelObserver listens to changes on a specific set of models analyzed by SimuLizar. It is notified whenever changes of a predefined subset occur (see

[2]https://svnserver.informatik.kit.edu/i43/svn/code/Palladio/Incubation/SimuLizar/trunk/org.palladiosimulator.simulizar.power/

EMF's *org.eclipse.emf.common.notify.Notification*). In order to react to a specific change event, such as `Notification.ADD`, one has to override the respective method of AbstractModelObserver. For `Notification.ADD`, this method is `void add(final Notification notification)`.

An example for dynamic registration of Probes in SimuLizar is the Response Time Monitor[3]. The Response Time Monitor allows to register additional response time measurements with the Probe framework during a SimuLizar analysis run. The Response Time Monitor listens to changes on the Monitor Repository model. If a new response time Monitor is added to the Repository as the result of an adaptation, the Response Time Monitor registers a response time Recorder with the Recorder framework. The extension also deregisters Recorders if a response time Monitor is removed from the model.

[3] `https://github.com/cactos/Cactos-Prediction/tree/master/org.palladiosimulator.simulizar.responsetimemonitor`

10 UI Framework

The UI Framework allows to create visualizations of recorder measurements. These visualizations can be XY plots and histograms like shown in Fig. 1.2 and Fig. 1.3; but also pie charts or arbitrary custom visualizations are possible. The UI Framework comes as a part of EDP2 and directly supports a generic visualization of measurements stored within EDP2 recorders.

Currently, we do not provide further documentation for the UI Framework and directly refer to the code available within the org.palladiosimulator.edp2.visualization plug-in.

11 Experiment Automation Framework

Experiments allow to conduct a series of analyzer runs and to aggregate recorded measurements to new measurements (so-called experiment reports). For example, an experiment can repeat an analyzer run for ten times. The variance over these runs can be reported, indicating the statistical significance of these analyzer runs. Another example is to conduct the same analysis using different analyzers and to report on the analysis time per analyzer.

Palladio's "Experiment Automation Framework" adds support for such experiments. Experiment Automation Framework allows to specify analyzer configurations, to run analyzers, and to store experiment reports based on investigating analyzer measurements within recorders. In particular, the framework provides extension points for adding custom analyzers and for hooking-in the normal Experiment Automation workflow.

In this chapter, we first overview the Experiment Automation workflow (Sec. 11.1). Second, we detail the two extension scenarios to Experiment Automation mentioned above: adding support for custom analyzers (Sec. 11.2) and hooking-in to the normal Experiment Automation workflow (Sec. 11.3).

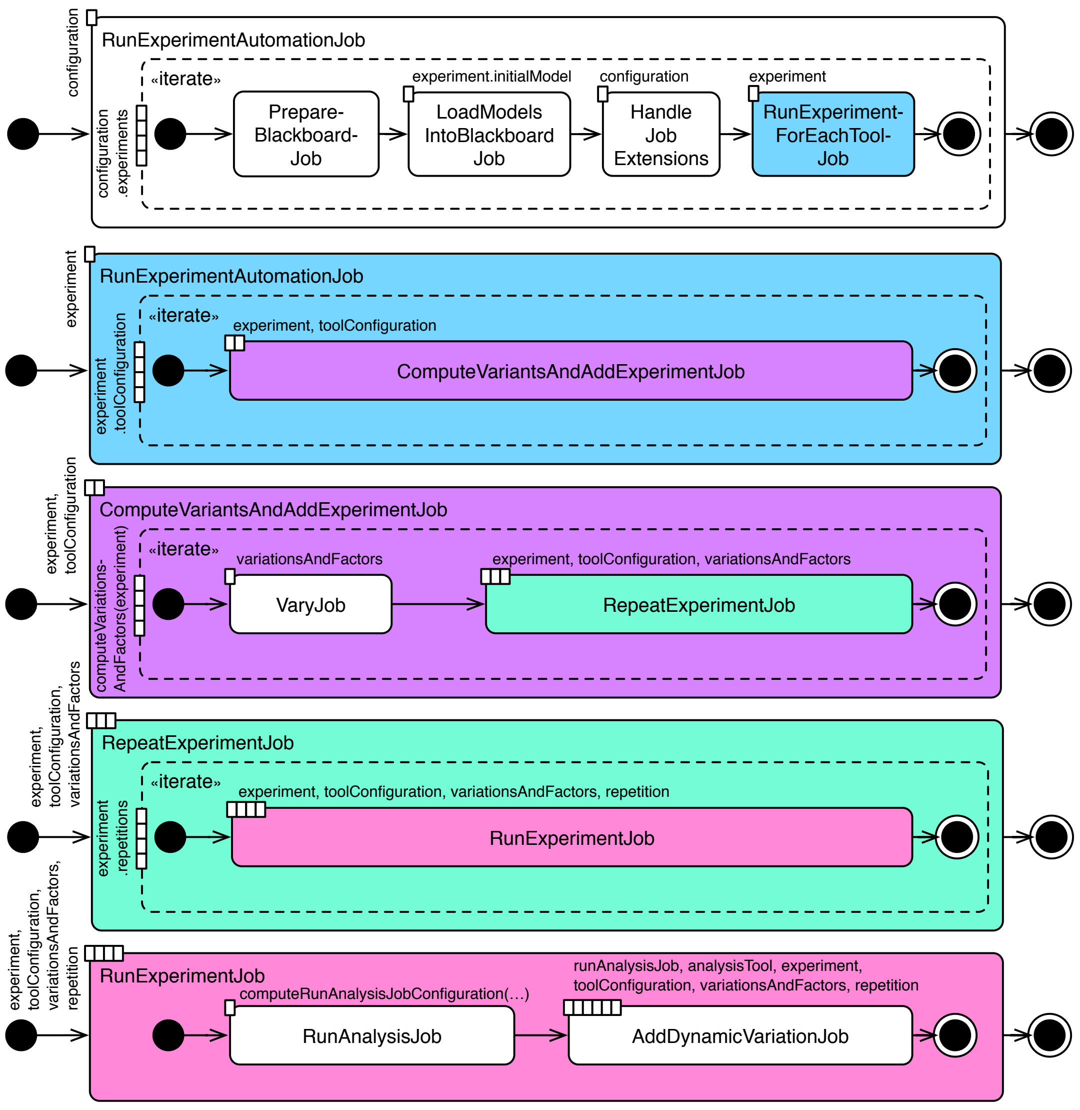
11.1 The Experiment Automation Workflow

Figure 11.1 overviews the Experiment Automation Workflow. Each action of the activity diagrams is realized as a dedicated job of Palladio's Workflow Engine (the "Handle Job Extensions" action is an exception; it is an extension point for adding custom jobs). Colors are used for linking calls to sequential jobs and associated activity diagrams that illustrate the inner jobs of these. White actions illustrate non-sequential jobs and therefore have no inner jobs.

11.2 Adding Support for Custom Analyzers

Custom analyzers need to:

- extend the org.palladiosimulator.experimentautomation.application.tooladapter extension point and
- provide a metamodel with a metaclass that extends AbstractSimulationConfiguration; a metaclass of the metamodel that comes with the Experiment Automation Framework. Instances of the new metamodel can then be used for configuring Experiment Automation runs.



configuration
RunExperimentAutomationJob
«iterate»
configuration .experiments
Prepare-Blackboard-Job
experiment.initialModel
LoadModels IntoBlackboard Job
configuration
Handle Job Extensions
experiment
RunExperiment-ForEachTool-Job
experiment
RunExperimentAutomationJob
«iterate»
experiment .toolConfiguration
experiment, toolConfiguration
ComputeVariantsAndAddExperimentJob
experiment, toolConfiguration
ComputeVariantsAndAddExperimentJob
«iterate»
computeVariations-AndFactors(experiment)
variationsAndFactors
VaryJob
experiment, toolConfiguration, variationsAndFactors
RepeatExperimentJob
experiment, toolConfiguration, variationsAndFactors
RepeatExperimentJob
«iterate»
experiment .repetitions
experiment, toolConfiguration, variationsAndFactors, repetition
RunExperimentJob
experiment, toolConfiguration, variationsAndFactors, repetition
RunExperimentJob
computeRunAnalysisJobConfiguration(...)
RunAnalysisJob
runAnalysisJob, analysisTool, experiment, toolConfiguration, variationsAndFactors, repetition
AddDynamicVariationJob

Figure 11.1: Overview of the Experiment Automation Workflow. Each action of the activity diagrams is realized as a dedicated job of Palladio's Workflow Engine (the "Handle Job Extensions" action is an exception; it is an extension point for adding custom jobs). Colors are used for linking calls to sequential jobs and associated activity diagrams that illustrate the inner jobs of these. White actions illustrate non-sequential jobs and therefore have no inner jobs.

Example for currently existing Analyzers are the following:

- SimuCom: org.palladiosimulator.experimentautomation.application.tooladapter.simucom
- SimuLizar: org.palladiosimulator.experimentautomation.application.tooladapter.simulizar

Please refer to these examples if you want to add your own Analyzer.
Note that the configuration of Analyzers within experiment models is currently tricky because it needs to be done directly within the XMI file. The generated tree editor for experiment models does not yet support editing of subclasses of AbstractSimulationConfiguration. Therefore, instances of these subclasses have to be copied to the appropriate place in the XMI file. Please have a look at the XMI representation of any example experiment model to learn about the configuration of custom Analyzers.

11.3 Hooking-in to the Experiment Automation Workflow

Custom jobs that wat to hook-in to the Experiment Automation Workflow need to:

- extend the de.uka.ipd.sdq.workflow.job extension point by referencing the workflow.extension.experimentautomation.before.experimentrun workflow ID as a hook-in point.

This extension is realized by the "Handle Job Extensions" action in Fig. 11.1. Refer to org.scaledl.architecturaltemplates.completion.jobs as an example for such an extension.

Bibliography

[1] Matthias Becker, Steffen Becker, and Joachim Meyer. "SimuLizar: Design-Time Modelling and Performance Analysis of Self-Adaptive Systems." In: *Proceedings of Software Engineering 2013 (SE2013), Aachen*. 2013.

[2] Steffen Becker. "Coupled model transformations for QoS enabled component-based software design." http://d-nb.info/989923983. PhD thesis. Carl von Ossietzky University of Oldenburg, 2008, pp. 1–262. ISBN: 978-3-86644-271-9. URL: `http://docserver.bis.uni-oldenburg.de/publikationen/dissertation/2008/beccou08/beccou08.html`.

[3] Steffen Becker, Heiko Koziolek, and Ralf Reussner. "The Palladio component model for model-driven performance prediction." In: *Journal of Systems and Software* 82.1 (Jan. 2009), pp. 3–22. ISSN: 0164-1212. DOI: `10.1016/j.jss.2008.03.066`. URL: `http://www.sciencedirect.com/science/article/pii/S0164121208001015` (visited on 11/07/2012).

[4] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. "Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis." In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. Boston, Massachusetts, USA, April 22–25, 2012: ACM, Apr. 2012, pp. 247–248. ISBN: 978-1-4503-1202-8.

[5] Jean-Marie Dautelle. *JScience*. 2014. URL: `http://jscience.org/`.

[6] Sebastian Lehrig and Thomas Zolynski. "Performance Prototyping with ProtoCom in a Virtualised Environment: A Case Study." In: *Proceedings to Palladio Days 2011, 17-18 November 2011, FZI Forschungszentrum Informatik, Karlsruhe, Germany*. 2011.

[7] Philipp Merkle and Jörg Henß. "EventSim – An Event-driven Palladio Software Architecture Simulator." In: *Proceedings to Palladio Days 2011, 17-18 November 2011, FZI Forschungszentrum Informatik, Karlsruhe, Germany*. 2011.