Live Execution Time Visualization of Component-based Software Systems

Master’s Thesis

Nelson Tavares de Sousa

October 31, 2016

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by:  Prof. Dr. Wilhelm Hasselbring
             M.Sc. Christian Wulf
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Abstract

With the increasing complexity of software, the development of component-based software system gains more attention. By encapsulating the functionality of certain parts of the system, the complexity can be contained. However, the performance of such software systems depends on the runtime usage and the architecture of the system. As a result, the demand on tools which facilitate the performance analysis and optimization grows continuously.

In this thesis we present an approach which allows to monitor the execution of component-based software systems and to visualize the results live at the runtime of the monitored system. To achieve this, the measured values are aggregated to derive the execution time of each component. Afterwards, by applying a filter, only certain time intervals are included in the visualization. One remaining step toward the visualization is the calculation of suitable values by applying different metrics, which is performed in a loop to continuously provide an updated visualization. This approach allows to optimize the system within the development process, as any delay between the execution and analysis is eliminated. One further contribution are two graphical notations which allow to visualize certain execution aspects.

To demonstrate the feasibility of our approach, we provide an implementation of it as an eclipse plug-in which can be used to analyze applications written in Java.
## Contents

1 Introduction .......................................................... 1
   1.1 Goals .................................................................. 2
      1.1.1 G1: Design of a Live Monitoring Visualization Concept for Component-based Software Systems ............... 2
      1.1.2 G2: Implementation of a Live Visualization Framework ................................................................. 2
      1.1.3 G3: Evaluation with TeeTime-based Applications .............................................................................. 2
   1.2 Document Structure .................................................. 3

2 Foundations and Technologies .................................. 5
   2.1 Performance Visualization of Software Systems .......... 5
   2.2 Monitoring with Kieker ............................................. 5
   2.3 Plug-In Development within Eclipse ....................... 6
   2.4 Development of Domain Specific Languages with Xtext ................................................................. 7
   2.5 Transient Visualization with Kieler Lightweight Diagrams ............................................................ 7
   2.6 Unified Modeling Language ...................................... 7
      2.6.1 Component Diagrams ........................................ 9
      2.6.2 Sequence Diagrams ......................................... 9
   2.7 The Pipe-and-Filter Framework TeeTime ................ 11

3 Requirements on a Live Visualization of Execution Times 13

4 An Approach for a Live Visualization Framework .... 15
   4.1 Process Flow for a Live Visualization of Execution Times .............................................................. 15
   4.2 Notations for a Visualization of Execution Times .............................................................................. 16
      4.2.1 Execution Behavior Diagram .................................. 16
      4.2.2 Timing Behavior Diagram ..................................... 17

5 Toward the Monitoring of Software Components .... 21
   5.1 Requirements .......................................................... 21
   5.2 Component Records .................................................. 23
   5.3 Component Probes ..................................................... 24

6 Architecture and Implementation of the Framework ... 29
   6.1 Architecture ............................................................. 29
      6.1.1 Monitoring Component ....................................... 29
      6.1.2 Rendering Component ....................................... 30
## Contents

6.1.3 Communication .................................................. 32  
6.2 Implementation .................................................. 32  
   6.2.1 Monitoring Framework ......................................... 32  
   6.2.2 Monitoring DSL ................................................ 33  
   6.2.3 Launch Manager .............................................. 37  
   6.2.4 Monitoring Log ............................................... 38  
   6.2.5 Record Manager .............................................. 41  
   6.2.6 Metric & Filter .............................................. 47  
   6.2.7 Visualization Framework ..................................... 50  

7 Evaluation ............................................................... 59  
   7.1 Goal 1: Feasibility of the Approach .......................... 59  
      7.1.1 Question: To which extent are the activities implemented? 59  
      7.1.2 Question: Is the approach applicable to a software system of larger scale? 72  
      7.1.3 Threats to Validity ........................................ 76  
   7.2 Goal 2: Performance Overhead of the Implementation .... 77  
      7.2.1 Question: To which extent does our framework affect the performance of the monitored code? 77  
      7.2.2 Threats to Validity ...................................... 79  

8 Related Work .......................................................... 83  

9 Conclusion ............................................................. 85  

Bibliography ............................................................. 87
With a great software architecture comes great complexity. As a consequence, developers tend to divide complex software systems into distinct components with reduced complexity. Such components encapsulate the system’s functionality and data, in order to achieve higher abstraction and modularity [Taylor et al. 2009]. Furthermore, any communication throughout the border of a component must be done through well-defined interfaces. By applying both properties, components become replaceable in their environment [Object Management Group 2007].

As already discussed by Kuperberg and Becker [2007], the performance of software components rely on various factors. For example, the architecture of the software system and the runtime usage of the application in question may, among other things, have a significant impact on the overall performance. Therefore, analyses on the component level need to be done to uncover performance issues caused by non-efficient components. On this level the developer is able to filter components with a negative impact on the overall performance and to concentrate on the optimization of those. Such approach may simplify the life of the modern homo programmatore ingenius.

Recent approaches, such as SLAStic [van Hoorn 2014] or ExplorViz [Fittkau 2015], show that research on software component monitoring is a current and important topic. For example, SLAStic aims in automatic architectural runtime reconfiguration. It monitors software components in order to optimize them for cloud-based systems, depending on their workload. ExplorViz monitors software landscapes to give administrators a visualization of it, and provides performance measurement tools and analyses of the landscape. Both concentrate on the operations of software systems.

In contrast, the contribution of this thesis is a framework which assists the developer in the development process of a component-based software system. Therefore, it runs as a plug-in within the Eclipse IDE, measures the execution time of each component and shows a live visualization of the results at the application’s runtime. By giving the developer analysis tools toward the performance of the architecture, such as hot-spot detection, they are able to detect bottlenecks and to perform optimizations on individual components.
1. Introduction

1.1 Goals

1.1.1 G1: Design of a Live Monitoring Visualization Concept for Component-based Software Systems

The major contribution of this thesis is to provide a concept which allows a live visualization of monitoring data. In particular, we aim to visualize the execution time of individual components. This includes the sole visualization of the architecture, as well as the additional data, such as median execution time. Additionally, we need to identify suitable monitoring methods and processing mechanisms, which finally make a live visualization possible.

Furthermore, the visualization must meet some requirements. For instance, the Unified Modeling Language (UML) [Object Management Group 2007] also includes a standard on how to illustrate software components and their communication channels. Those component diagrams include information and semantics which are not needed for our use case. As our visualization is intended to be used by the developer of the software system under analysis, we don’t need any of this additional information. Omitting this additional information results in a reduced and simpler notation.

To complete the whole concept, an API also needs to be provided to give the developer an access to the monitoring techniques which are required by the framework.

1.1.2 G2: Implementation of a Live Visualization Framework

In order to make use of the concept, its implementation will be the second goal of this thesis. As the resulting framework is intended for developers, it will be implemented as a plug-in for the Eclipse IDE. It will provide utilities to gain access to the framework, as well as the visualization within an Eclipse view.

Furthermore, this goal also covers questions regarding the software architecture of the framework.

1.1.3 G3: Evaluation with TeeTime-based Applications

In this step we aim to provide a feasibility evaluation by evaluating the implementation of the framework.

In order to show and test the functionality of the newly created framework, a variety of TeeTime-based applications are used. TeeTime allows to implement component-based applications and suits therefore perfectly this use case. The possibility to implement arbitrary architectures allows us to demonstrate the usage of the new framework for different scenarios. More specifically, some functionality needs to be covered, such as the visualization itself, or the performance impact of the monitoring.
1.2 Document Structure

This thesis is structured as follows. In Chapter 2 we introduce the foundations and technologies used in our work, including previous research on this topic and the frameworks which are used in our implementation. Chapter 3 depicts a set of requirements for a live visualization of execution times. Afterwards, an approach for the live visualization is presented in Chapter 4. We give an overview on our approach and discuss its individual steps. The concept we use to monitor the execution of components is presented in Chapter 5. In Chapter 6 we present our implementation of the approach. First, we discuss the underlying architecture of our implemented framework. Afterwards, we present the implementation aspects of the individual parts of it. Chapter 7 evaluates the feasibility of our approach and its performance. Related work made in this research domain is discussed in Chapter 8. Finally, Chapter 9 concludes our work by discussing the conclusion and potential future work.
Foundations and Technologies

This chapter gives a short insight into different foundations and technologies which are used in this thesis.

2.1 Performance Visualization of Software Systems

Gotel et al. [2008] already defined the first goal of visualization in software engineering as fixing and communicating software structures, as well as facilitating the development of such. As we see, the visualization of software also suits as a tool for debugging.

Ball and Eick [1996] showed how hot-spot detection and other code-level analysis techniques can be realized. Their hot-spot detection approach colorizes each line of code corresponding to its execution time, where a red color denotes a higher execution time than a blue one. Additionally, uncolored lines indicate dead code, for e.g. a comment or code which is never executed. By looking at this visualization, any developer is able to determine which code parts are suitable for optimizations.

By using this technique on other structures, we are able to show how other software parts perform. As this approach is not inherently limited to lines of code, other structures like packages/modules or even components benefit from such a visualization. Marwede et al. [2009] have also shown how to eliminate the propagation of the execution times of subsequent procedures. The time a procedure waits for its call on another procedure to return is eliminated, in order to gain the real execution time. On component-level, such propagations can also occur, which makes the usage of such techniques appropriate.

2.2 Monitoring with Kieker

van Hoorn et al. [2012] introduced with Kieker a framework which allows to monitor and analyze the runtime behavior of software systems. In particular, it allows to monitor an application’s performance. Due to Kieker’s extensibility it is possible to implement own analyses which run on the incoming monitoring data. Therefore, it allows every developer to modify its analysis part and adopt it to the developer’s needs. Figure 2.1 shows the individual components of Kieker. For instance, the Monitoring Probe and the Analysis Plugin may be changed to own implementations. Also, own Monitoring Records may be used to monitor the desired aspects.
2. Foundations and Technologies

By exploiting this extensibility we can make use of the Kieker framework for our needs. This thesis makes use of this and integrates Kieker into the monitoring part of the resulting framework. By writing an own analysis it is possible to analyze the measurement live at the application runtime.

Kieker uses aspect-oriented programming (AOP) to instrument program code. AOP enables a greater separation of concerns, by leveraging cross-cutting concerns to a different paradigm. Cross-cutting concerns are concerns which are present across the program’s structure, such as logging, or in particular monitoring. These concerns are separated from the business logic as aspects. AOP frameworks capture the crosscuts and automatically add these aspects to the executing system [Påhlsson 2002].

AspectJ is a framework which allows to use the AOP paradigm in a Java environment [Kiczales et al. 2001]. Therefore, it is used by the Kieker framework to instrument software systems.

2.3 Plug-In Development within Eclipse

Eclipse is an integrated development environment (IDE) [Eclipse Foundation 2016] which combines different development tools into one single application. It is intended as a Java development suite, whereas other programming languages are also supported. The Eclipse IDE provides a platform which can be enhanced with a variety of features by adding plug-ins to it.

One ability, which is used in this thesis, is to add views to running Eclipse instances. Views make part of the graphical user interface and provide special functionality. Some well-known default views in Eclipse include the Console, Editor and Project Explorer View. By using the Standard Widget Toolkit (SWT), also maintained by the Eclipse Foundation,
own Views can be implemented and added to the Eclipse user interface. This, and the fact that it is already used by a variety of developers makes it a suitable platform to provide new functionalities through plug-ins.

2.4 Development of Domain Specific Languages with Xtext

Xtext is a language development framework which allows to develop an own domain-specific language (DSL) [Behrens et al. 2008]. It provides tools to implement DSLs, such as parser, serializer, a scoping framework, and further more. All these are based upon Java and run as independent application.

The main point why Xtext came into question, is its integration into the Eclipse IDE. Beside the tools to describe an own DSL, it also provides an editor which can be embedded into Eclipse. This editor allows to seamlessly program in Eclipse with the new DSL, while also relying on other languages, such as Java.

2.5 Transient Visualization with Kieler Lightweight Diagrams

Some criteria need to be met to provide a live visualization. First, the generation of the image must be fast, which means it must be generated within a certain time interval which allows a live update of the visualization. Second, for our approach, the generated visualization needs to be interactive by means of modifications to the result after its creation, so that additional information can be displayed and modified at runtime.

With the Kieler Lightweight Diagrams framework (KLighD) [Schneider et al. 2013] we are able to meet these criteria. KLighD enables an automatic graph drawing based on meta-models. It receives model instances and transforms them in order to place and draw the resulting elements on a canvas. Figure 2.2 shows the graph of an Eclipse EMF Genmodel generated by KLighD. KLighD receives the genmodel instance and transforms it into a graph, by automatically calculating its layout and afterwards rendering the corresponding elements. Relations between the elements are emphasized by connecting them through edges.

Furthermore, the focus of KLighD lies on the reduction of time to diagram, as well as interactivity, which allows to manipulate the resulting graph. KLighD embeds well into the Eclipse Environment, as it provides an own view within the IDE.

2.6 Unified Modeling Language

In terms of collaborative development of software, a common communication platform is an essential prerequisite. To cope with this, the Unified Modeling Language (UML) was adopted, to give stakeholders tools for design and implementation of software systems. [Object Management Group 2007].
2. Foundations and Technologies

Figure 2.2. An example graph generated with KLighD

The UML specification consists of various elements which allow to illustrate different aspects of software systems. In the following we discuss two of those which are essential for this thesis.
2.6. Unified Modeling Language

2.6.1 Component Diagrams

As already mentioned, software systems can be divided into encapsulated parts, called components. UML holds notations for those components and groups of it.

In Figure 2.3 we see an example component diagram with two connected components, called ComponentA and ComponentB. By definition, components are depicted as rectangles, in which further information is collected. Components are marked with the keyword «component» and optionally with an component icon in the top right corner, in order to distinguish it from other UML elements [Object Management Group 2007].

Interfaces are divided into two categories, provided and required interfaces. Provided interfaces are implemented by the component and depicted as full circle, connected to the implementing component. Required interfaces make use of provided interfaces and are depicted as semicircle, connected to the corresponding component. Dependencies between both interface types show the wiring between different components, as shown in Figure 2.3 [Object Management Group 2007]. In our example, ComponentB provides an interface and is therefore connected to full circle denoting the interface. This interface is accessed by ComponentA which is shown by its connection to the semicircle. As the interfaces are connected, both the full circle and the semicircle are connected in the diagram.

2.6.2 Sequence Diagrams

We described how component diagrams represent internal structures of the software system in question. However, it does not reflect the interactions among them. To model interactions, the UML specification includes a set of notations which constitute these. The most common of those interaction diagrams are sequence diagrams. Sequence diagrams depict the communication between classes or components of a software system.

Figure 2.4 shows an example for a simple sequence diagram. The sequence diagram itself is surrounded by a frame with the keyword sd followed by the name of the diagram in the top left corner. Components are shown at the top as rectangles with dashed lifelines vertically below them and are denoted with a colon at the begin of their name. Along these lifelines are so called ExecutionSpecifications which indicate the control flow of the execution. The message passing between the different lifelines is indicated by arrows, showing the direction of the call. In our example, we see that ComponentA calls ComponentB and how the control flow is returned to ComponentA. Additionally, DurationConstraints and TimeConstraints may introduce relative timing constraints to the sequences. They specify...
2. Foundations and Technologies

![Sequence Diagram Example](image)

**Figure 2.4.** An example of a minimal sequence diagram

![Pipe-And-Filter Pattern](image)

**Figure 2.5.** Illustration of the Pipe-And-Filter pattern

Timing behavior by putting certain points of the execution into relation to observation points. For instance, the duration of a certain call may be limited, or a certain upper bounds in terms of its timing is guaranteed. However, the timing behavior cannot be derived from the notation of the lifeline, but only from additional constraints. Therefore, sequence diagrams are only qualified to depict a relative order of execution and are not intended to show absolute execution times [Object Management Group 2007].

Beyond this, UML specifies a variety of message types, as well as additional elements, such as loops or conditionals. As those are not covered by this thesis, we omit them at this point.
2.7 The Pipe-and-Filter Framework TeeTime

The Pipe-and-Filter pattern is an architectural style, where a system’s logic is separated into distinct components. Each component has a set of input and output ports, which receive incoming data or send processed data. Those components are encapsulated and independent from each other and perform transformations on the incoming data stream, therefore they are called Filter. Furthermore, they must not share any state between each other.

In order to enable the communication between filters, an additional entity is required. Since the data is given as a stream of data units, this entity is called Pipe. Pipes take the output of a filter and pass it to the following one, and may restrict this communication by certain constraint. For instance, the pipe may be typed and solely allows data which complies to a certain interface, or the amount of data may be bounded [Garlan and Shaw 1993].

This architectural style is spread among various domains. A well-known example is the pipe operator in the unix shell. By adding the “|” operator between two commands, the output of the first command is forwarded to the second. This enables a sequential processing of data in the shell.

Figure 2.5 shows an example of the Pipe-and-Filter pattern. It resembles the shell command

```
printf "Hello\nWorld\n" | grep Hello
```

which prints the lines Hello and World, followed by a filtering of the word Hello. The result is the output of the line Hello. The left rectangle in the figure represents a filter of the function `printf`, followed by the second filter `grep`, represented by the rectangle to the right. Both are connected by a pipe, indicated as arrow, which indicates the direction of the control flow.

TeeTime [Wulf and Tavares de Sousa 2016; Wulf et al. 2014] is a framework which supports the implementation and the execution of Pipe-and-Filter-based applications. Filters, which are called stage here, encapsulate the data and/or the functionality of the program and communicate through interfaces which are connected through pipes. Hence, the definition of stage overlaps with the definition of a software component.

A different aspect of TeeTime is the thread assignment within the resulting architecture. Stages in TeeTime can be either in an active or a passive state. Active stages will be executed in a dedicated thread by the framework, whereas passive stages are executed by their leading stages. The developer directly influences the thread assignment in the development process. Therefore, TeeTime enables various configuration possibilities with different thread assignment strategies.

For evaluation purposes, different scenarios need to be depicted. TeeTime with its variety of configuration possibilities allows to implement various scenarios for these purposes.
Chapter 3

Requirements on a Live Visualization of Execution Times

The development process of our visualization framework is proceeded by means of agile development methods. Requirements on the software system are analyzed at the beginning to avoid mistakes at later phases which cause high costs [Laplante and Neill 2004]. Paetsch et al. [2003] state a set of aspects concerning requirement engineering and divides them into functional and non-functional requirements. Functional requirements include functions which the final system should be able to perform. The definition of non-functional requirements however, is somewhat fuzzy. For our purpose, we rely on the definition introduced by Glinz [2007]. Non-functional requirements include attributes of or constraints on the final system, whereas we also summarize it as Quality Attributes. In the following, we elicit the requirements on our intended framework.

**Functional Requirements** In order to visualize execution times of component-based software systems, the resulting framework needs to provide a graphical user interface, as well as graphical notations to present the visualized information. As this graphical presentation needs to be live, the visualized information needs to be updated in frequent intervals. For a fluent perception we target to update the diagram every 40 milliseconds or less. Additionally, the presentation needs to be adaptable in a way that different aspects of the execution times can be emphasized. For instance, the visualization may emphasize the overall execution times or their mean values. Also, a time shift mode must be available. With this, the developer is able to observe past events in the execution of its system [Fittkau 2015].

The framework should assist the developer in the development process. Therefore, the framework needs to be embedded into a development environment. It also should provide functionality for the developer to define the components of the software system which need to be monitored.

Additionally, our approach needs to be minimal invasive to the monitored system. The developer must not be required to introduce any monitoring code into its software system under development to avoid an increased complexity.
3. Requirements on a Live Visualization of Execution Times

**Non-Functional Requirements (Quality Attributes)**  In order to provide a high maintainability of the framework, it will be designed and implemented with a high modularity in mind. This means, we will outsource parts into separate components, where appropriate. For instance, the metric, which eventually decides on the presentation of the monitoring information, should be encapsulated into a component. By doing so, the framework can be enhanced by simply swapping the metric to a different implementation.

Furthermore, the developer may not be familiar with the technologies and frameworks used in our implementation. Our framework must be usable without profound knowledge toward the used technologies, so that it is also accessible to developer of different fields.

The graphical notations also need to be defined in a way that developer understand their content with minimal effort.

For certain requirements, we are already able to give constraints. As the Eclipse IDE is extensible and allows contributions to the user interface, we determine to use it for our implementation. With the limitation set to this IDE, the set of usable frameworks is also reduced. Due to their strong integration into the Eclipse environment, we select the Xtext framework to develop a DSL and the KLighD framework to implement the visualization. Furthermore, the implementation of our framework requires the ability to monitor software systems in a specific way. As Kieker is adaptable to monitor arbitrary structures, we also specify it as our monitoring framework for the implementation.

Also, by relying on recent versions of the used frameworks, the maintainability of our framework is also increased, due to the available support by their developers. As a result, we commit ourselves to the versions 4.5 of the Eclipse IDE, version 0.11.0 of the KLighD framework, version 1.12 of Kieker, and version 2.9.2 of the Xtext framework.
An Approach for a Live Visualization Framework

With the requirements described in Chapter 3, we are able to design the approach for the visualization framework. In the following we analyze the required steps to provide a live visualization of execution times.

4.1 Process Flow for a Live Visualization of Execution Times

With UML’s activity diagram, we are able to draft and show the process flow we use for our approach. This type of diagram allows us to show the different actions and their relation toward each other, which we use to discuss our approach [Object Management Group 2007].

Figure 4.1 shows the intended process approach we use for a live visualization of execution times of component-based software systems. The whole process can be divided into the following four steps.

The first step includes the instrumentation of the software system in question. Before running any analysis or starting with the visualization, we need to weave monitoring code into the application in order to be able to measure its execution times.

![Figure 4.1. A UML activity diagram showing the process flow for a live visualization](image-url)
4. An Approach for a Live Visualization Framework

The following tasks are executed in an endless loop. First, the software system is started. As it is already instrumented, the execution times of each component are measured live and instantly available.

Afterwards, the resulting records need to be processed. Related records need to be matched together and associated to the corresponding component. Additionally, their values need to be processed in a way that an analysis of the raw execution times can be done. If this task is completed, further calculations can be done on those execution times. The results are filtered to just analyze data within a certain time interval. Finally, by applying a metric, we introduce a set of different presentation formats. After all these steps, we get aggregated and processed execution times for the next step.

The visualization is the last task in the loop. It uses the processed data and prepares them for a visualization. In the first iteration, the diagram is generated, else-wise it is updated upon the new incoming data.

These three steps need to run in a loop to always guarantee the presentation of current values. As mentioned in Chapter 3 a fast execution of this loop is essential for a fluent visualization. If the user terminates the plug-in, the loop is aborted, otherwise it runs endlessly.

4.2 Notations for a Visualization of Execution Times

One main question we need to discuss is how to display the measured values. As the developer should not be burdened with completely new concepts, we rely on known ones and adapt them to our needs. Therefore, we take on the component and sequence diagrams, given by the UML, and modify them so they fit our needs and purposes. In the following we introduce new diagrams, the execution behavior diagram and the timing behavior diagram, which focus on the visualization of execution times in different ways.

4.2.1 Execution Behavior Diagram

The execution behavior diagram is built upon UML’s component diagram. It removes some syntactic elements and adds the presentation of additional information of execution times.

In Figure 4.2 we see an example of an execution behavior diagram. In terms of syntax, we do also represent components with rectangles which include the name of the component. Additionally, the interfaces of each component are represented as smaller rectangles at its border.

In contrast to the component diagram, we do not require notations for different message types, so that we rely on a simple representation of the control flow between components. We do so by indicating the control flow with an arrow which connects the interfaces. The direction of the arrow itself resembles the direction of the control flow. In the example, both CompA and CompC send data to CompB. For our purpose we neglect any information on how the communication happens. Therefore we rely on this single notation. Furthermore,
4.2. Notations for a Visualization of Execution Times

Figure 4.2. An example of an execution behavior diagram - more saturated components need more execution time

the connection between the interfaces of components is optional. There are two reasons for this decision. First, we do not analyze the communication between components and therefore, do not need to emphasize this by any manner. Second, the monitored software system may compose of different sub-systems. This means, there may be components in the system we cannot reason about. As a result, we cannot model the communication of such a grey-boxed system and need to omit this information.

The main contribution of this diagram is the coloring of the components. Syntactically, components are colored differently to emphasize them. In our example, we colorize the components with different shades of red. However, the used color is not restricted to this and can be chosen arbitrarily. The information which is represented by this coloring is not strictly defined, but depends on the underlying metric. Figure 4.2 shows how a presentation of the overall execution times of different components may look like. Semantically, the more saturated a component is, the higher is its percentage on the overall execution time. With this information, we can see that CompB requires the most time for its execution, followed by CompA and CompC in last place.

4.2.2 Timing Behavior Diagram

The sequence diagram in the UML standard only depicts a relative ordering of timings. Even with time or duration constraints, it cannot be used to show the exact timing behavior of the system. By introducing the timing behavior diagram we are able to give a notation which shows absolute timings of software systems. It adopts well-known concepts from the sequence diagram notation, but modifies some of its elements.

Figure 4.3 shows an example of the timing behavior diagram which represents the same software system as in Figure 4.2. The base structure resembles the structure of a
An Approach for a Live Visualization Framework

sequence diagram. We see the components at the top, shown as rectangles, and their lifelines below them. Additionally, both diagrams show the relation between executions and their components. The concurrent execution of different components is also depicted by both diagram types. However, the timing behavior diagram differs from a sequence diagram by introducing a time scale on the left side. With this additional information, we are able to see the exact time when the execution of a component starts and ends.

In our example, we see how the three components are executed. CompA and CompB are executed three times, where every execution of CompB needs twice the time of an execution of CompA. More specifically, CompA takes about 2 seconds for every execution, as we can derive from the time scale.

If we take a look at the overall execution times of the components, we can see that CompB needs the most time, followed by CompA and CompC. This information equals the information of Figure 4.2, as both show the results of the same execution of the monitored
4.2. Notations for a Visualization of Execution Times

This notation strongly depends on the machine it runs on. As it shows concrete timings, the diagram differs on machines with varying performance, as well as after every execution due to non-deterministic behavior.

Nevertheless, with these two notations, we are able to visually analyze the execution times of component-based software systems. By applying different metrics we are also able to cover different aspects regarding the execution times. In the following, we do not only present an implementation of these notations, but also use them in this thesis for further explanations of examples.
Chapter 5

Toward the Monitoring of Software Components

A further major question we need to elicit, is how to monitor the timing behavior of individual components. We need to observe the beginning and ending of the execution of components. Furthermore, we also must be able to recognize an ongoing execution process in order to include it in the analysis.

5.1 Requirements

First, we take a closer look on how to achieve such monitoring of components. van Hoorn et al. [2009] introduced a technique for the reconstruction of execution traces. For this purpose, Kieker inserts a monitoring probe into the entry and exit points of relevant methods which are triggered upon the arrival of the control flow at those points. By logging the current system time, Kieker is able to derive the time needed for the execution of each method. We can adapt this approach to derive the execution times of components in a similar way. If the input and output interfaces of the component are known, we can insert probes into these methods and log the current system time.

However, if the monitored component is executed multiple times or the same component is executed concurrently, we are not able to correctly match the records of coherent entry and exit points. As we see in Figure 5.1, we would receive two records for the entry points and two records for the exit points of CompA. An assignment of records to each other is not feasible. Therefore, van Hoorn et al. [2009] introduced the traceId. Every execution flow is tagged with such a traceId to uniquely identify the control flow and to match associated records. As a result, in our example we are able to match the records by analyzing their traceId. The entry point of the green control flow is matched to the exit point on the same control flow. The same applies for the monitored points on the blue control flow.

Nonetheless, this approach still may lead to errors. As already stated by van Hoorn et al. [2009], if two events of the same type occur at the same time, or the clocks are not synchronized correctly, the ordering of the events cannot be reconstructed, as the system time cannot be used to derive the original order. For instance, depending on the accuracy of the system clock, two disjoint components can appear to be executed at the same time. If our record only provides the system’s time when the event happened, we cannot decide
5. Toward the Monitoring of Software Components

Figure 5.1. Sequence diagram showing the control flow between components

which entry point belongs to which exit point. Therefore, we need to expand the record format by more information.

We have multiple options to solve this problem. First, we can add an identifier to the record to link events to the triggering components. This allows us to directly link records to their corresponding component, even with concurrent executions or multiple components within the same control flow. This seems to be a perfectly suitable solution, if we don’t want to reconstruct the order of the executions of components. But, there are at least two reasons why the ordering is important for our use-case.

If we reconstruct the ordering of executions, we are eventually able to detect nested executions of components. This occurs if one component executes a second one without leaving its own boundaries. This is the case if a component is nested within another one or the developer misses to mark certain interfaces. One further feature can be implemented by reconstructing the order. Marwede et al. [2009] introduced an optimized algorithm for the visualization of timing behavior anomalies. By eliminating the propagation of the timing behavior, the resulting visualization depicts the execution timings of components, without the propagated time which the components waits for its successor.

In order to enable the reconstruction of traces van Hoorn et al. [2009] introduced the execution order index (eoi) and the execution stack size (ess). The first value gives any execution an absolute order by incrementing its value upon every new event. With the second value, we are able to determine the depth of the execution trace. The value is incremented upon
5.2. Component Records

the beginning of an execution and decremented after its exit. Like already mentioned, with
this ordering we are able to calculate the real execution time of a component by subtracting
the execution time of component with a higher \( eoi \). In our example, we see how \( \text{CompA} \)
executes \( \text{CompB} \) in the second execution flow. From its start to its end it takes \( r_1 \) time units,
whereas \( \text{CompB} \) takes \( r_2 \) time units. Now, as we know that \( \text{CompB} \) is executed by \( \text{CompA} \),
we can subtract \( r_2 \) from \( r_1 \) and get the real execution time of \( \text{CompA} \). This applies for any
arbitrary component and its successor. We define

\[
r_{C_i} = m_{C_i} - m_{C_{i+1}}
\]

as the real execution time of component \( C_i \) where \( r \) is the real execution time, \( m \) the mea-
sured execution time, and \( i \) the value of \( \text{ess} \). This equation does not apply for components
which are currently executed, as they do not have any successor.

5.2 Component Records

After eliciting the required data for the observation of execution times of components, we
move toward the design and implementation of corresponding records. Kieker does not
provide any record implementations which can be applied to our use case. However, the
\textit{OperationEvent} type provides most of the attributes we require. Therefore, we use it as base
type for our implementation.

Figure 5.2 shows the implementation of the \textit{AbstractOperationEvent} by Kieker. This
class inherits various attributes of super classes depicting certain aspects. For instance, the
\textit{ITraceRecord} is used to identify individual execution traces, therefore it has the \textit{traceId}
and the \textit{orderIndex} as attributes which were described in Section 5.1. The \textit{AbstractOperationRecord}
inherits the Attributes \textit{timestamp}, \textit{traceId}, \textit{orderIndex}, \textit{classSignature}, and \textit{operationSignature}
from the interfaces \textit{IEventRecord}, \textit{ITraceRecord}, \textit{IClassSignature}, and \textit{IOperationSignature}.
Kieker uses the \textit{AbstractOperationEvent} to implement three subclasses:

\textit{BeforeOperationEvent} Represents the data which is collected before the execution of the
monitored operation.

\textit{AfterOperationEvent} Represents the data which is collected after the execution of the
monitored operation.

\textit{AfterOperationFailedEvent} Represents the data which is collected after the execution of the
monitored operation is failed.

These records enable the monitoring of method executions of classes. A monitoring of the
execution of components is not feasible, as these records do not supply any information of
the executing component. In Section 5.1 we showed how such monitoring can be achieved,
by adding a further attribute which uniquely identifies the executing component.

Listing 5.1 shows our implementation of three new record types written with the \textit{Instrumenta-
tion Record Language} (IRL). The IRL allows to define own record types independent
5. Toward the Monitoring of Software Components

5. Toward the Monitoring of Software Components

Figure 5.2. Classes inherited by AbstractOperationEvent

from a specific language [Hasselbring et al. 2013; Jung and Wulf 2016]. Line 9 to 11 show the attributes we want to introduce to the record types. All new record types require the new componentId and therefore we can introduce this attribute to the records by creating a super type CompRecord which implements it. The records which require this attribute may inherit from CompRecord. As we exploit already existing record types, we implement the new ones by merging them with the new super type. For instance, to monitor the incoming interfaces of a component we introduce a BeforeCompOperationEvent, as we see in Line 15 of Listing 5.1, and let it inherit the classes BeforeOperationEvent and CompRecord. By doing so, the new class includes all attributes we need to monitor components. We repeat this for the AfterCompOperationEvent (line 19) and AfterCompOperationFailedEvent (line 23) and receive therefore a set of record types which enables us to monitor the executions of components.

5.3 Component Probes

Now that we have the data structure to log monitoring events related to the execution of components, we need mechanisms which perform the measurements. Kieker’s monitoring part provides so called Probes which collect measurements and eventually pass the resulting records to a monitoring log/stream writer [van Hoorn et al. 2012]. By
5.3. Component Probes

```java
package de.cau.se.ntd.kiekerEnhancements.records

import kieker.common.record.flow.trace.operation.AfterOperationEvent
import kieker.common.record.flow.trace.operation.AfterOperationFailedEvent
import kieker.common.record.flow.trace.operation.BeforeOperationEvent

@package 'Nelson Tavares de Sousa' @since '1.12'
template CompRecord {
    int componentId
    grouped by TraceMetadata.traceId long traceId = -1
    int orderIndex = -1
}

@package 'Nelson Tavares de Sousa' @since '1.12'
entity BeforeInCompOperationEvent
    extends BeforeOperationEvent : CompRecord {}

@package 'Nelson Tavares de Sousa' @since '1.12'
entity AfterOutCompOperationEvent
    extends AfterOperationEvent : CompRecord {}

@package 'Nelson Tavares de Sousa' @since '1.12'
entity AfterOutCompOperationFailedEvent
    extends AfterOperationFailedEvent : CompRecord {}
```

Listing 5.1. Component records written in the IRL

implementing own probes we are able to introduce and use our new record types, presented in Section 5.2. For the monitoring of method executions, Kieker uses the `kieker.monitoring.probe.aspectj.flow.operationExecution.AbstractAspect` aspect as default. This aspect is used to create concrete aspects which are woven around the desired method. The execution of the method is controlled by this aspect and suspended until the `BeforeOperationEvent` is created. After the execution of the monitored method, the concrete aspect logs an `AfterOperationEvent` record and eventually returns the control flow to the callee method.

This concept can be leveraged to implement aspects for the monitoring of components. In Figure 5.3 we see an example for the communication between two components. Both components have input ports which define their incoming interfaces. Additionally, `Component A` has an output port as outgoing interface, whereas `Component B`'s input port also depicts its outgoing interface. The tasks, which resemble the workload of a component, are performed in a `Task` class in `Component A` and within the input port of `Component B`. Both intervals, in which the tasks are executed, are marked as dashed regions. As we restrict
5. Toward the Monitoring of Software Components

Figure 5.3. Example of the communication between two components

the execution of the system to be linear, which means it proceeds without branches and reentries after passing through an outgoing interface, we assume that for every entry of the component, the control flow only exits the component once.

In order to measure the execution time of a component, we need to weave aspects into their interfaces so that the component’s tasks are enclosed by before- and after-measurements. There are multiple options where to insert the concrete aspects. Aspects which collect measurement in an incoming interface should collect the data before the execution of the corresponding method. This becomes obvious by looking at Figure 5.3. If the data is collected after the execution of ComponentA.InputPort.in(), observations toward the execution time cannot be done, as the task is not enclosed by measurements. Therefore, the measurements must be collected before the execution of the method.

For the outgoing interface in turn, there a two potential reference points we need to consider. As shown in Figure 5.4, concrete aspects can collect data before or after the execution of the corresponding method. The figure shows at which points the records are created by the probes for both approaches. For Component A in our example, either approach provides suitable data. We see that in both Figure 5.4a and 5.4b the workload is enclosed by the corresponding records. However, Component B shows a constraint we need to respect.

Component B does not have an independent outgoing interface. It receives data through the input port, performs tasks on it and returns eventually. Therefore, the input port also represents its outgoing interface, indicated by the method name io() in Figure 5.3. Figure 5.4a shows how the records are created if the aspects are inserted at the begin of the outgoing interface. As both aspects collect their measurements at the begin of io(), the records do not enclose the task performed by the component anymore. Especially, as both records are created at the same time, their timestamps show the same value. To avoid this, aspects must be inserted at the end of an outgoing interface, as in Figure 5.4b. The data is collected by the aspect, before io() returns. As a result, both measurements enclose the
task we want to monitor, hence providing reliable values to derive the execution time. If we apply this approach on both components, the execution time of Component B falls into weight for the execution time of Component A. However, as shown in Section 5.1, we are able to subtract this value to get the real execution time.

The same observations can be made on components with solely outgoing interfaces. For those components, the begin of the task needs to be marked as a hidden incoming interface. It is not used by any other component, but it is needed to mark the interface to perform measurements. Therefore, such components behave analogous to components with only incoming interfaces and need to be monitored the same way. As other options are not feasible, we rely on this approach for component monitoring.

As a result of these observations, we get two aspects called AbstractInCompAspect and AbstractOutCompAspect. AbstractInCompAspect is used to collect data from an incoming interface of a component, more specifically, it collects the data before the execution of the corresponding method. The collected data is represented as a BeforeCompOperationEvent record and forwarded to the monitoring log. The second aspect collects data after the execution of a method belonging to the outgoing interface of a component. Depending on the execution of the method, it forwards an AfterCompOperationEvent record or an AfterCompOperationFailedEvent record, if the method returns with an exception, to the monitoring log.
5. Toward the Monitoring of Software Components

Records are created at the begin of the method

Records are created at the end of the method

Figure 5.4: Communication examples showing different approaches toward the creation of AfterOutCompOperationRecords
Chapter 6

Architecture and Implementation of the Framework

In this chapter, the underlying architecture of our framework is described. First, in Section 6.1 we give an overview of the framework’s components and their contents. Afterwards, in Section 6.2, we take a closer look on the individual parts and their implementation.

6.1 Architecture

The architecture comprises of two major components, as seen in Figure 6.1. The framework is divided into a rendering and a monitoring part, both running independently from each other. The rendering component takes on the processing of the monitoring results and visualizes them. The monitoring component offers functionality to instrument code and monitors while running. Both use interfaces for the communication which are connected through a dedicated channel.

In particular, the logic of both components is separated for the sake of modularity. Therefore, the rendering component is able to run with other monitoring concepts as long as the interfaces are satisfied. However, we restrict to the design and implementation of one concept in the following sections.

6.1.1 Monitoring Component

Frameworks and components which enable a monitoring of code are included within the monitoring component. Its purpose is to provide functionality toward the monitoring itself and the instrumentation of the code. Furthermore, the measurements are provided by the interface to its environment.

A Monitoring Framework provides the monitoring functionality. This component takes only care of the measurements and needs input on which code needs to be monitored. Therefore, we need a component with which we are able to give the user the ability to define code parts which need to be monitored. This is solved by providing a language which is included in the Monitoring DSL component. The Launch Manager component serves as controller of the monitoring part. It uses the definition given by the Monitoring DSL in order to use the Monitoring Framework to instrument and launch the monitored code. Additionally,
6. Architecture and Implementation of the Framework

![Figure 6.1. Architecture of the live visualization framework](image)

![Figure 6.2. Closer view on the architecture of the monitoring component](image)

as seen in Chapter 5, we need an own data structure for the measurement results, as well as probes which perform the needed measurement. Those classes are included within the two components Component Records and Component Probes which enhance the used Monitoring Framework.

The Monitoring Framework is implemented with the Kieker framework we presented in Section 2.2, as it provides interface with which it can be adapted to our own needs. The adaptation is realized with the records and probes presented in Chapter 5, which are included in the Component Records and Component Probes components. The Monitoring DSL makes use of the Xtext framework presented in Section 2.4, to implement the language. Finally, the Launch Manager component uses APIs provided by Eclipse.

### 6.1.2 Rendering Component

The framework’s rendering component encloses various different nested components to create the live visualization of execution times. Figure 6.3 shows an overview of the component and its containing parts. The incoming records are read by the Reader and saved to a Monitoring Log. This component saves all incoming records and provides them for
analyses at any point in time. A Record Manager takes control over the visualization. It collects the records from the Monitoring Log and processes them to enable their visualization. By using classes of the component Metric & Filter a selection of relevant measurements and a further processing can occur. This functionality is intentionally encapsulated into a component, so different implementations can be used. Control over the Visualization Framework is also delegated to the Record Manager. The Visualization Framework provides functionality for the drawing of diagrams. It does so by using the API provided by the Graphical User Interface.

Some of the components are covered by third-party frameworks which we use. This includes the Graphical User Interface and the Visualization Framework. As the framework runs as an Eclipse Plug-In, we make use of the user interface provided by Eclipse itself which is based upon the SWT. The Visualization Framework in turn, consists of two parts. As mentioned in Chapter 4, we introduce two new notations for the visualization of execution times. For the execution behavior diagram we use the KLighD framework as it provides the functionality needed for such diagrams. The timing behavior diagram however, is not rendered with KLighD. For this diagram type, we implemented an SWT widget which provides the tools for drawing.

The remaining components do not make use of frameworks. For the Monitoring Log we use data structures provided by Java but modify them to fit our use case. The Record Manager is built from scratch, however Section 6.2.7 shows our approach on how to implement it based on the TeeTime framework. The Metric & Filter component provides interfaces so that it can be enhanced. Furthermore, it includes some default filter and metrics.

By interconnecting these components we provide a full chain from the incoming data to the diagram generation and therefore enable the visualization with this component.
6. Architecture and Implementation of the Framework

6.1.3 Communication

Concerning the communication between monitoring and rendering, we also need to keep modularity in mind. Both components need to be able to run independently from each other and be interchangeable. For instance, a different monitoring concepts may be used and therefore we need a generic communication channel. In particular, both components may rely on different systems and therefore we must neglect language-specific remote procedure calls, such as the *Java Remote Method Invocation*.

Furthermore, the communication between both components needs to be reliable. This means, any loss of records must be recoverable since missing records lead to an erroneous component execution trace and as a result to an incorrect visualization.

Taking these constraints into consideration, we may choose between different types of communication. To fulfill these constraints we can use a database, files on the file system or a *Transmission Control Protocol* (TCP) connection. However, the database approach requires an available database which in turn needs further configuration. By using files, the framework needs to repeatedly access the main memory which in most cases only allows a slow access. With TCP we are able to meet the requirements. It is commonly used, does not rely on a specific system, and is in particular reliable [Postel 1981]. Therefore, we use it for our implementation.

6.2 Implementation

The following sections show how the individual components of the framework are implemented.

6.2.1 Monitoring Framework

For the realization of the *Monitoring Framework* component we use Kieker. As already mentioned in Section 2.2, Kieker allows to use own implementations for certain components. We use the new records and probes presented in Chapter 5 to eventually monitor component executions with Kieker.

Three further aspects are needed to monitor code. The first is a properties file which is needed to configure Kieker. With this file, we can specify which monitoring data writer should be used. This is the *TCPWriter* in our case. Additionally, we can configure the *TCPWriter* to immediately send the records instead of holding them in a queue until it is full and then sent. As this configuration always remains the same, we provide it as a static file. Second is the specification of methods to be monitored. We need to tell Kieker which methods we want to monitor. As we use the AspectJ-based version of Kieker, we do so by providing an *aop.xml* file. This file is used by AspectJ to determine the points where aspects need to be woven into [Kieker Project 2015]. As this changes with the monitored code, we need to create a new file each time the monitored code changes. A solution to this
is presented in the following section. Lastly, we need to add Kieker itself to the JVM and provide it as Java agent upon the execution of the monitored code.

6.2.2 Monitoring DSL

In order for Kieker to perform the measurement, we need to specify all methods we want to monitor. For the AspectJ-based version of Kieker, this means we need to provide an `aop.xml` file which defines points where aspects need to be woven into. A simple solution is to burden the user with the creation of such file. However, this file can be derived from the structure of the components which the user wants to monitor. Therefore, we use a concept where the `aop.xml` file is created out of a model of the component structure which the user provides. As a result, AspectJ and the scheme of the `aop.xml` must not be introduced to the user. One further advantage of such an approach is its modularity. As we extract this concept into an individual component, we may change the environment at a later point without the need to reimplement the metamodel itself. For instance, a different monitoring framework could be used just by modifying the model transformation. The metamodel always remains the same.

Mernik et al. [2005] show the advantages and disadvantages of DSLs and appropriate use-cases. In particular, they uncover the role of a DSL as enabler of reuse. We leverage this role and use a DSL to introduce a notation for components and as system-frontend. By specifying a metamodel we create a notation with which components can be defined which need to be monitored. In addition, with this DSL the configuration of the framework is performed.

With Xtext, we can implement such notation by giving a concrete syntax and a mapping from a model to an in-memory representation [Behrens et al. 2008]. Xtext then generates all tools needed to begin with the development with this new DSL.

First, we need to specify the concrete syntax. This is provided to Xtext by giving a grammar written in Xtext’s own grammar language. However, before we start with the design of this grammar, we need to take a look at the requirements on our notation. The user must be able to define the components of its software system and their relations toward each other. Therefore, our metamodel needs to provide elements which resemble components and elements which resemble their communication paths. The latter is divided into two parts. We must give separate notations for interfaces of a component and their communication channels. If we neglect the interfaces, we are not able to determine the entry and exit points of a component which is required as shown in Chapter 5. Therefore, our notation consists of three entity elements: components, their interfaces and connections. As stated in Section 4.2, our framework also monitors grey-boxed systems. As a result, the user is not always able to fully specify the communication channels. Therefore, the element representing the connection must be optional.

By taking these aspects into consideration, the result is a metamodel in which components are composed of interfaces. Optional connections aggregate two interfaces. Listing 6.1 shows the implementation of this metamodel as an Xtext grammar.
6. Architecture and Implementation of the Framework

Lines 1-3 show the language declaration. In these lines, the name of the grammar is specified and a set of terminals is imported. Additionally, Line 3 instructs Xtext to generate an Ecore model for the specified grammar. Ecore is part of the Eclipse Modeling Framework (EMF) and provides a common standard for the implementation and usage of meta models [Steinberg et al. 2009]. By generating a model based on the EMF, we make it accessible to the
6.2. Implementation

Eclipse environment and are able use it with its API. Lines 5-8 show the model’s root, called Model. Model has two attributes components and connections. These are lists containing the corresponding elements. Xtext uses the extended Bachus-Naur Form to express terminals. Therefore, the cardinality of the components attribute is one or more and as connections are optional, the connections attribute can have an arbitrary cardinality. Next, we specify the Interface element in Lines 10-12. An interface can be either of an INCOMING or OUTGOING type. Interfaces, in our case, indicate the method which eventually implements the interface. Therefore, we need to specify the method by providing its qualified name and its signature. For the qualified name, we make use of Java’s concept for qualified names, but do not allow wildcards, as seen in Lines 20-22. With this name we are able to match the definition of the interface to the concrete method implementation in the monitored code. For the signature we need to specify the types of the method’s parameter. For this we make use of AspectJ’s notation. The types are declared after the method’s name in brackets and need to be in the same order as in the signature of the monitored method. Wildcards are allowed by AspectJ and indicated with the sequence ‘..’ [Kiczales et al. 2001]. Therefore, a parameter can be either a wildcard or a qualified name pointing to the concrete class, as defined in Line 25.

The definition of the Component element can be found in Lines 14-18. A Component has a name and is composed of at least one interface. For its name, we use the same definition of a qualified name as in Interfaces and therefore do not allow wildcards. Components do also have a unique identifier iden. The question mark indicates that this attribute is optional. This optionality is owed to the transformation of the model, as this attribute is overwritten in order to assign a truly unique identifier. The interfaces are enclosed by curly brackets in order to associate them with the correct Component.

Connection is the last element. Semantically spoken, a Connection aggregates two Interfaces to model the connection between the interfaces of two components. In Lines 28-30 we see that the attribute connector receives two elements in every Connection instance. Those elements are indicated by [Interface|QualifiedName], which syntactically means that we await a string which complies to the QualifiedName rule, but references to an Interface instance. The reference is established, if this value matches the name of an Instance. Both connected Interfaces are aggregated by an arrow in between, indicating the control flow.

With this grammar we are able to generate an editor, which opens and edits arch files, and are also able to provide a transformation from model to aop.xml. This file gives AspectJ instructions which aspects it should use and where to weave them into the code.

In the following we discuss the transformation to the aop.xml. As a special characteristic, we define multiple aspects for each component with this transformation. This is due to the fact, that each interface is mapped to its dedicated aspect.

Listing 6.2 shows an example of an aop.xml file we generated with our framework. With the weaver environment in Lines 2-4, AspectJ’s weaver can be configured. For instance, exclusions can be specified here, which remain always the same for our use case. In the aspects environment, beginning with Line 5, we give AspectJ its instructions. In this environment, concrete aspects and their pointcuts are defined. The term pointcut comes from the domain
6. Architecture and Implementation of the Framework

Listing 6.2. Example of an aop.xml file

```xml
<aspectj>
  <weaver>
    <exclude within="org.apache.commons.logging.."/>
  </weaver>
  <aspects>
    <concrete-aspect name="de.cau.se.ntd.AUniqueNameForAnAspect" extends="kieker.monitoring.probe.aspectj.flow.operationExecution.AbstractAspect">
      <pointcut name="monitoredOperation" expression="execution(* test.Test.execute())"/>
    </concrete-aspect>
  </aspects>
</aspectj>
```

of aspect-oriented programming and describes a set of join points, which in turn describes points in the control flow of a program [Kiczales et al. 2001]. Therefore, by giving the point cut a concrete value, we instruct AspectJ where the aspect should be woven into. The concrete-aspect itself is specified by a unique name and the abstract aspect it extends. In our example, we extend from `kieker.monitoring.probe.aspectj.flow.operationExecution.AbstractAspect` and give the concrete implementation the name `de.cau.se.ntd.AUniqueNameForAnAspect`. The aspect we extend from contains an abstract pointcut `monitoredOperation` which we need to provide. By giving its expression the value `execution(* test.Test.execute())`, we instruct AspectJ to weave this concrete aspect around the method declaration `test.Test.execute()` which may have an arbitrary return value, but no parameters.

In order to generate an `aop.xml` we need to provide a transformation of our model into this XML scheme. We see that this scheme has recurring elements, such as the pointcut expression, which we can map to a value in our model. Therefore, we generate a concrete-aspect element for each interface of a given model instance. As every concrete aspect needs a unique name, the generator holds an integer which is incremented upon every new concrete aspect. Its value is appended to a static string, which is `de.cau.se.ntd.FrameworkAspect` in our case that results in the aspect’s unique name. In addition, we add the component’s identifier to this unique name to link the aspect to the correct component. The abstract aspect we extend can be determined from the interface type. As mentioned in Section 5.3, we use two aspects depending on whether the monitored method represents an incoming or an outgoing interface. If the corresponding interface, given by a model instance, is of the type `INCOMING`, the generator uses the aspect `AbstractInCompAspect` to extend from. In case of an `OUTGOING` interface type, it uses the `AbstractOutCompAspect` aspect. The name of the pointcut element always remains the same. We only need to generate its expression. As our metamodel complies with AspectJ’s syntax in this point, we can copy the interface’s name and append it parameter to get the pointcut expression.
6.2. Implementation

6.2.3 Launch Manager

The Launch Manager represents a mediator between three participating components: the Monitoring DSL, the Monitoring Framework and Eclipse. Its goal is to give the user access to the framework’s functionality in a simple way.

Usually, a developer launches its code in Eclipse by using a Launch Configuration. It provides some configuration options, such as additional classpath entries, and eventually launches the execution of the code by applying these options. As this is a well-known concept within the Eclipse environment, we use it to provide our own implementation of a Launch Configuration. For this, Eclipse comes with an implementation for the execution of Java code. We use this implementation and introduce an enhancement which allows us to intercept the configuration process before the execution. The advantage of this approach is that its view remains the same as of the original one.

With the interception, we are able to perform tasks related to the configuration of the Monitoring Framework. As Kieker requires files within the classpath for its configuration, we introduce a further configuration step and create those files. For this, we hold a kieker.monitoring.properties file, which specifies the monitoring stream writer and configures it. This file is static, as we do not need to change it. Additionally, we need to supply Kieker itself, which we provide as a Java archive file. The last file is the aop.xml which we generate with our Monitoring DSL.

With the interception, we gain access to the classpath configuration and the arguments that are passed to Java’s virtual machine. Our own implementation of the Launch Configuration copies all three files into a temporary directory. As next step, the directory’s path is added to the classpath and Kieker’s Java archive file is passed as argument by specifying it as Java agent. After performing these tasks, the control flow is passed to the original Launch Configuration which eventually launches the execution.

With this approach, we do not burden the user with the configuration of Kieker, as we perform this task automatically in the background. By adding classpath entries programmatically, we are able to hide these aspects, which result in a less intrusive framework. The developer solely needs to provide a model of components which need to be monitored with a single arch file. In particular, with this approach, the developer does not need to add any monitoring code manually, create an aop.xml file which requires further knowledge about AspectJ, or configure Kieker.

Besides the configuration of Kieker, the Launch Manager also prepares the framework for the visualization of the results. It does so by instantiating the rendering part of the framework. This includes the views which provide the visualization, if they are already opened within Eclipse, and its data structures. In addition, the Reader also needs to be initialized, so the TCP connection can be established.
6. Architecture and Implementation of the Framework

6.2.4 Monitoring Log

The meaning of Monitoring Log here differs from Kieker’s definition. The Monitoring Log component, in combination with the Reader, receives the incoming monitoring record stream and saves it in a data structure, called ComponentRecordCache. This is implemented with a TeeTime-based configuration. In this configuration a TCPReaderStage receives all incoming records from the TCP connection and passes them one by one to an InstanceOfFilter stage. This stage checks if the class of the incoming element is an instance of our implemented record types. If this is confirmed, the record is passed on, otherwise it is omitted. The last stage is the SaveToCacheStage which eventually saves the record to the ComponentRecordCache.

The ComponentRecordCache needs to meet some requirements. First, it should be optimized for performance. The subsequent aggregation of records profits if the order equals the order of the incoming record stream, as the stream sends matching AfterCompOperationEvents after their corresponding BeforeCompOperationEvents. This prevents a reordering of records, thus leading to less operations within the aggregation. As the analysis iterates over the set of records multiple times, a data structure optimized for this case would be adequate. By saving the pointer of the records in adjacent regions of the memory, we can exploit the data locality which already leads to a better performance compared to an arbitrary selection of memory locations. This reduces the set of appropriate data structures to the Vector1 and ArrayList2 structures, both provided by the Java standard library, which save added elements in an array structure. Other structures, such as Set3 or LinkedList4 do not guarantee data locality. Vector and ArrayList do also maintain the order of the elements as they are added.

Second, the scenario in which the ComponentRecordCache is used requires a thread safe access to the data. This scenario is similar to the producer-consumer problem, but deviates from it. The Reader runs in its own thread while adding continuously records to the cache. On the other side, the available data is accessed by other threads which read from the data structure but do not remove elements from it. As concurrent modification of elements is avoided, we do not need to mutually exclude the threads. Therefore, we do not need synchronized access methods which in turn eliminates the Vector as suitable data structure, as preventing synchronized methods also contributes to an increased performance.

This leaves us with the ArrayList as an appropriate data structure. However, this type of list is not thread-safe, as its missing synchronization leads to stale data [Goetz and Peierls 2006]. As modifications of its variables are not propagated throughout all accessing threads, it may occur that threads do not notice newly added elements. Therefore, we need to modify the ArrayList in order to enable this propagation. For our case, we can leverage the volatile keyword, supplied by Java. The volatile keyword is a weaker form of synchronization. Modifications on variables marked with this keyword are always propagated to all threads.

---

1https://docs.oracle.com/javase/7/docs/api/java/util/Vector.html
2https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html
3https://docs.oracle.com/javase/7/docs/api/java/util/Set.html
4https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html
6.2. Implementation

[Goetz and Peierls 2006], so the visibility of the most recent value is guaranteed.

By combining the ArrayList with the volatile keyword on some of its attributes, we gain a data structure which suits the given scenario. We do so by marking the size attribute as volatile. This value indicates how many elements are stored within the array. Therefore, if we flag it as volatile, all accessing threads know the correct number of elements and, in particular, the position of the last element. Newly added elements are therefore always noticed. One further attribute we must mark as volatile is the underlying array, called elementData in the original implementation of ArrayList. The reason behind this, is the possible creation of a new array, leading to a new value of this attribute. This happens, if the required size of the array exceeds its current size. This usually occurs if a new element is added to the ArrayList while the underlying does not hold an empty memory field anymore. In this case, a new array is created, old values copied to it and then elementData is updated. If we do not mark it as volatile, a thread may not notice the switch to a new array instance, leading to an access to a faulty memory region.

The problem of potentially stale data also applies to the elements in the array itself. If we flag an array as volatile we only mark the reference to its location in the memory as volatile and not the references which are hold as elements. For instance, a thread modifies the second element of the adapted ArrayList. If a second thread now accesses to the second value, it may see the old value, leading to an error. However, this problem does not apply to our use case. The Reader only adds new elements and does not modify them. If a thread accesses the array to read the new element, it always reads the correct value. This is because the thread never accessed this array element earlier, thus not holding a copy of the value which could be obsolete. Furthermore, as this value is never modified, it never gets stale.

With this modification, we can adapt the ArrayList to be thread-safe concerning our use case. However, the ArrayList in Java also holds an Iterable. This Iterable is used for the foreach operator and provides an interface for the iteration over data collections. In the particular case of the ArrayList, its Iterator’s methods are fail-fast. This means, if a concurrent modification occurs after the instantiation of the Iterator, its methods fail with an error to avoid non-deterministic behavior. As we have a well defined scenario, we adapt the Iterator to not fail upon any external modification, as our scenario does not provoke faulty behavior.

Listing 6.3 shows the add method of the adapted ArrayList. Line 2 calls a method which checks if the current array provides sufficient empty fields. If this is not the case, a new array is allocated and the previous values copied. In Line 3, the new element is added to the array. As we recall, the fields elementData and size are marked as volatile. Therefore, it is guaranteed that the access in Line 3 is always correct. Finally, the value of size is incremented in Line 4. As this method is always accessed by the same thread, we do not need any mutual exclusion. Therefore, this adaptation suits our use case.

In Listing 6.4 we see how elements are collected by using the Iterator. This method is called, after a call of the method hasNext() which checks if the iterator has reached the

---

5https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html
6. Architecture and Implementation of the Framework

```java
public boolean add(CompRecord e) {
    ensureCapacityInternal(size + 1);
    elementData[size] = e;
    size++;
    return true;
}
```

**Listing 6.3.** The add method of the adapted `ArrayList`

```java
public CompRecord next() {
    int i = cursor;
    if (i >= size)
        throw new NoSuchElementException();
    Object[] elementData = RecordArrayList.this.elementData;
    if (i >= elementData.length)
        throw new ConcurrentModificationException();
    cursor = i + 1;
    return (CompRecord) elementData[lastRet = i];
}
```

**Listing 6.4.** The next method of the adapted `Iterator`

end of the list. Therefore, the if statement in Line 3 never triggers, as the `Iterator` is always accessed by the same thread and therefore, the cursor value is not modified after a call on `hasNext()`. In Line 5, a copy of the array is created. This value is only stale in the case of the creation of a new array, due to missing space. However, this fact may be neglected, as the obsolete array is not yet collected by the garbage collector and still holds correct values. As the check for a correct cursor value is already done at this point, the field we access in the array is also still available. Lines 6 and 7 check if elements are removed from the list while iterating over it, which is never done in our scenario. Finally, the value is collected and returned.

To even further facilitate the analysis in the next step, we optimize the ordering of data by grouping the incoming records by their `traceId`. For this, we use an `HashMap` where every `traceId` is mapped to the corresponding adapted `ArrayList` (which we call `RecordArrayList` from this point on). By using Java’s `HashMap` we rely on a well-known concept which also allows access to the values in constant time. In order to collect all `traceIds` the `HashMap` provides a method called `keySet()` which returns a set of all keys, or in our case `traceIds`. With this set, any thread is able to collect all `RecordArrayLists`. As the underlying field for this set is already set to `volatile`, stale values are avoided.

The result is shown in Figure 6.4. To the left we see the `HashMap` which holds the `RecordArrayLists`, which we see to the right. The resulting structure is called `Componen-
6.2. Implementation

Figure 6.4. Data structure of the `ComponentRecordCache`

Taking a closer look at the complexity of operations on the `ComponentRecordCache` we can see the advantages of this approach. Accessing a `RecordArrayList` happens in $O(1)$ time as the `HashMap` provides constant access times. As we iterate over the returned `RecordArrayList`, the access time to the individual elements is also constant, as we only need to calculate the offset toward the array’s start. Therefore, the overall access time is bounded by $O(1)$. To add records, the corresponding `RecordArrayList` must be collected first. Elements can be added to this list in $O(1)$ time, as the list holds a size value which is used to calculate the position of the first empty memory field. As a result, this operation is also bounded by $O(1)$. Other operations can be neglected, as we do not make use of them in our framework.

6.2.5 Record Manager

Analogous to the Launch Manager, the Record Manager mediates between the Monitoring Log, the Metric & Filter component, and the Visualization Framework. As we provide two different views, each displaying a different type of diagram, we need to implement two different types of data pre-processing. First, we discuss the implementation of the pre-processing for the execution behavior diagram.

Execution Behavior Diagram

As we recall, the execution behavior diagram provides a view on different aspects toward the execution behavior of the monitored code. For this, we need to implement a process which allows to analyze the measurements in order to emphasize those different aspects in the final visualization. To achieve this, it iterates over the data, given by the Monitoring Log, aggregates the data and calculates coloring values by applying metrics and filters in order to visualize those values.
6. Architecture and Implementation of the Framework

This process is divided into three steps. At first, the raw component records are aggregated. In Figure 6.5 we see a sequence diagram illustrating the single execution of a component. If we monitor this component with our framework, the Monitoring Log receives records which are collected at the points begin and end. Both records include the system time at the corresponding measurement point, therefore by subtracting the value of begin from the value of end we get the execution time \( t \) of the component. This procedure is applied on all record pairs resulting in a new data format which includes the component identifier and its execution time. Furthermore, unclosed traces can also be analyzed, by applying the current system time as measurement value. This is the case for currently executing components, as the execution does not reach its end point yet. Therefore, currently executing components are also included in the further analysis.

As next step, a filtering of those execution times is performed. This is needed, as we want to observe certain time intervals. For instance, the user may want to only visualize the execution behavior of the last ten seconds of the program’s execution. Therefore, we apply the filter, given by the Metric & Filter component, which checks if the execution happens within the given time window. If this is not the case, the execution data instance is dropped for the further analysis, as seen in Figure 6.6. We see two components of which the first need lesser time for its execution. The red region indicates the time window accepted by the filter. The result of this filtering process is shown to the right. Execution records which cross the border of the filtering window are reduced to their relevant part, leading to the cut execution data in the example. This step is intentionally put at the beginning of the process flow, as it reduces the relevant data and therefore reduces the amount of needed operations for the further analysis processes.

One further question is how to uncover the nested execution of components. In Section 5.1 we show how to calculate the real execution time which takes the execution time of nested components into account. At this point we need to consider how to implement such an approach. In order to uncover the nested execution, we need to exploit the \textit{traceId} which is provided by all record types we use. van Hoorn et al. [2009] show how they reconstruct
the execution traces by using a stack. We leverage this concept to also reconstruct the order in which the components are executed. For this, we hold a stack for every traceId. The order of the stack corresponds to the order of the execution of the components, meaning the element at the top is called by the underlying element. While iterating over all records, if the checked record’s type is BeforeOperationEvent we push this element to the stack in order to wait for its corresponding AfterOperationEvent. If such a record is received, we pop the top element of the stack which is the matching BeforeOperationEvent for this record. If the stack holds further elements, recently matched execution is a nested execution of the element at the stack’s top. Therefore, we may calculate the execution time of the matched records and use this value to determine the real execution time of its caller component.

One special case is the filtering of a nested execution, as seen in Figure 6.7. Component B is called by Component A. However, the time window used by the filter intersects the execution of Component B as well as the execution of Component A, especially at a point where it waits for Component B shown as dashed region in the figure. After the filtering process we get an execution time value for Component A of c and a value of a for the execution of Component B. If we now use the same approach to calculate Component A’s execution time, we see that it also works for this scenario. By calculating $c - a$, we get a value which equals to $b$ which in turn is the correct real execution time. Therefore, the approach is also reliable on such cases.

This reduced filtered set of execution time can now be analyzed. In our case, as we want to visualize the execution times by analyzing them in a specific way, we need a function which calculates the emphasis of each component. For this, we use a metric, provided by the Metric & Filter component, which takes on this task. We provide the set of filtered execution times, and the metric maps an emphasis value to every component in the range 0 to 100. A value of 0 means no emphasis on the mapped component, whereas a value of

![Figure 6.6. Timing behavior diagram showing execution records before and after the filtering process](image-url)
6. Architecture and Implementation of the Framework

Figure 6.7. Sequence diagram showing a filtered, nested execution of components and certain execution times

100 represents a maximal emphasis.

The final results of the metric are now passed to the Visualization Framework which applies them to the different elements of the diagram. The Record Manager also initializes the Visualization Framework by creating the initial diagram which only shows the structure of the components and no additional information.

To communicate with the Visualization Framework we test two different concepts of which one turns out as less efficient. The first, and less efficient concept, is a push approach, as shown in Figure 6.8. The figure depicts the control flow of this approach and includes participating structures and threads as components. We use a thread, called PushThread, to frequently push the current values to the diagram. For this, the thread runs endlessly in the background, even after the termination of the monitored code, executing the whole process flow and applying the results to the diagram. This approach needs some considerations.

Figure 6.8. The push approach of the communication with the Visualization Framework
related to synchronization, as any modification of the user interface must be done by the UI Thread. The Graphical User Interface provides an API which allows access to this thread, by adding tasks to a queue which are eventually processed by it. To avoid an unresponsive user interface, these tasks should be of smaller extent to not burden the UI Thread with heavy processing. Therefore, we perform the whole pre-processing in a separate thread and only apply the final values in the UI Thread. As the first thread waits for the UI Thread to finish its task, we do not need to consider any special synchronization of the data, as concurrent modification of the data is impossible, thus mutual exclusion is assured. However, the API does not give any time constraint on when the task is executed, in particular it executes the tasks in the order as they are added to the queue. This leads to an enhanced waiting time of the pre-processing thread as it must also wait for the previous tasks in the queue to be finished.

The fact that we use an additional thread to the UI’s one introduces one further problem. As the user should be able to change the metric or modify the filtered time interval, we need to ensure the visibility of changes of these components. The problem relies on the fact that all actions performed by the user on the user interface are executed in the UI Thread, therefore a communication between threads is performed. For instance, this is the case if the user selects a different metric in a drop-down box. Synchronizing the involved methods is a bit exaggerated, as this is not required in this scenario. As the access pattern is similar to the one’s of the ComponentRecordCache, presented in Section 6.2.4, we may also here rely on the usage of the volatile keyword. As the UI Thread solely updates the pointer referring to the metric or filter, we simply need to ensure that this update is propagated to the pre-processing thread, which is exactly what the volatile keyword does. However, the disadvantage of higher access times plays a large role in the pre-processing mechanism, as those values are frequently accessed. In particular, the filter is accessed once for every pair of matching BeforeCompOperationEvent and AfterCompOperationEvent. Moreover, if the filter or metric is changed within a loop iteration, the final results can get inconsistent as parts of the incoming records are processed differently from each other. To avoid this behavior, the value of the fields marked as volatile can be copied at the begin of each iteration. This eliminates accesses to the main memory and ensures a consistent behavior as the copied value remains the same through the whole iteration. Especially, in the case of the filter, we reduce the amount of volatile accesses from roughly $\frac{n^2}{2}$ to one for each iteration, where $n$ is the amount of BeforeCompOperationEvent and AfterCompOperationEvent records.

This approach turns out to be not efficient, resulting in a non-fluent updating of the diagram. We target to update the diagram at least 40 milliseconds after the last update which results in 25 frames per second. With a small scaled test program, consisting of two components, the average execution time of this approach already outnumbers this value. Additionally, this pre-processing thread is permanently running and therefore consuming processing time which may be used by the monitored code otherwise. We see, the push approach turns out to be not suitable.

The second concept is a pull approach. Figure 6.9 depicts the control flow of this ap-
6. Architecture and Implementation of the Framework

This approach also uses two threads, a notification thread and the UI Thread. The notification thread runs as long the monitored code is executed and notifies the Visualization Framework every 30 milliseconds to update the diagram. Between two notifications the thread is put to sleep, thus not consuming any processing power. The Visualization Framework updates the diagram upon an incoming notification or UI event, such as mouse clicks or resize events. In order to update, the framework uses the UI Thread to redraw the diagram. For this, the UI Thread executes the pre-processing chain, including the final update on the diagram. Hence the name of this approach, as the values are pulled by the Visualization Framework itself. This approach comes with various advantages. Any synchronization mechanism can be dropped, as all values are accessed by only one thread, the UI Thread, therefore making any volatile flag obsolete. Furthermore, as the whole process is instantly executed, rather than added as a task to a queue, any waiting time on the thread is omitted. With the same test program, this approach executes every iteration in less than one millisecond in average. One further major advantage is the lack of a resource consuming thread permanently executing in the background.

Comparing both approaches, it turns out to be obvious that the pull approach should be preferred.

Timing Behavior Diagram

The pre-processing of data for our timing behavior diagram differs from the execution behavior diagram. As the diagram shows the executions of components throughout the whole execution of the software system, we do not need any filtering for this diagram type. Above this, the diagram only shows the raw data without any focus on certain aspect, which makes a metric obsolete. Therefore, only the matching of BeforeCompOperationEvents to their AfterCompOperationEvents is required and equals to the approach above. We also use a stack here, but derive one further information from its data. The top element is the most recently added, meaning it represents the current executing component. This information can be used to indicate the current position of the thread.

In summary, the pre-processing consists of the iteration over the data coming from
6.2. Implementation

the Monitoring Log, followed by the aggregation in order to get corresponding pairs of records and the final transfer of those pairs to the Visualization Framework. This process is run in a loop as long as the monitored code is executed. Thus, the Visualization Framework is always updated with the most current data. If the monitored code stops, the thread is also terminated, as the Visualization Framework holds the previous data, thus not requiring any update for the visualization.

6.2.6 Metric & Filter

As stated in Chapter 3, we want to be able to change the semantics of the calculated visualization, as the user may want to emphasize different aspect. Furthermore, a closer examination of certain time intervals may be desired. Therefore we need to enable the possibility to provide own metrics and filter. We need to elicit the exact functionality which both components provide, and adapt the defined interface to comply with it. Above this, we implemented examples which make use of both interfaces, which we also present here.

The Interval Filter Interface

Instances which implement this interface receive data structures which represent the execution time of a component, called Duration, and check if it lies in a certain time window. Furthermore, interval filter should be able to remove certain components from any further analysis. In Section 6.2.5 we also set a further requirement on its functionality. If the execution of a component intersects the time window, the filter should return the relevant part of the Duration. As the Duration includes the exact time when the corresponding execution started and ended, we also need to provide the filter with the times when the monitored code started its execution and the current time.

The interface for the implementation of filters needs to provide these functions. However, we need to add further functionality to our interface. The user interface provides an option to alter the filter. The time window may be moved or even altered in its size. Therefore, we
6. Architecture and Implementation of the Framework

Figure 6.11. Time scale showing a filter interval in red

need methods which allow to set this properties and read them in order to visualize them. Furthermore, filters may also be static, thus not modifiable.

Figure 6.10 shows the final implementation of the interface. To the left side we see the IIntervalFilter interface which defines a set of methods. The first method, called filterDuration, provides the interface for the main logic. This method is called by the framework to filter a single Duration instance. Therefore, this method is called by providing the Durations of the observed execution and the application itself. Additionally, the executed component is also provided to enable a filtering by components. Its return value is again a Duration providing the values which should be used for further analysis.

The following methods are related to the user interface. To configure the filter, we use values relative to certain points, as shown in Figure 6.11. The offset represents the temporal distance of the beginning of the filtering time window toward the current point, or the point at which the program finished. Our example is showing the time in milliseconds. Therefore, in the example the filter’s offset is 500 milliseconds. The interval value represents the size of the time window we use to filter. As we can take from our example, its interval value is 2250 milliseconds. The getter and setter methods in IIntervalFilter are used by the framework to modify these values. The modifications toward the time interval are however only applied to the execution behavior diagram and do not modify the timing behavior diagram.

With the last two methods the filter instance can tell the framework which of both values can be changed. The method isModifiable should return true if the filter instance allows the modification of its interval value, allowing to increase or decrease the size of the time window. The isFixed method in turn indicates if the offset value can be changed. If this is not the case, the whole time window may be moved, thus it is not fixed.

We provide two implementations of this interface to show the viability of this interface. The first implementation is a static filter, called DefaultIntervalFilter, which only lets Duration instances pass which represent the last \( n \) seconds of the program’s execution, where the value of \( n \) can be passed to the filter as parameter of its constructor. The second implementation is a fully dynamic filter, called AdaptableIntervalFilter. This filter can be fully modified through the user interface and can thus be used to observe every time interval of the program’s execution.
6.2. Implementation

The Metric Interface

Metric instances analyze the filtered set of Durations and give every component a value on how much the component’s element in the diagram should be colored. This gives components different emphasis toward different aspects. It is the task of the metric to depict an aspect and adapt the colorings in order to visualize it. This can be, but is not limited to the overall execution time of each component or the mean value of execution times of a component.

However, as a metric needs to compare all components and in particular their execution times, the framework must provide the whole set of Durations to the metric instance. The metric instance must return a pair which puts the observed component in relation to its calculated value. This value must be an integer and may be in the range of zero to 100, resembling a percentage value.

The resulting interface is shown in Figure 6.10 as IMetric. Metrics which implement this interface must provide an computeHotness method. The name of this method stems from the concept of hot spot detection, where hotter spots indicate a region of higher resource consumption. The method receives a list of ComponentExecutionRecord, which wraps a Duration and its corresponding Component. The return type is a Map which can be used in Java to put two different elements into relation.

For the interaction with the user, we need to diverge from the approach used by IIntervalFilter. This stems from the fact that an IMetric is not modified at its run time. We do not provide interfaces for any modification, but need an interface which allows to create and identify IMetric instances as the user interface allows to change the underlying metric. By exploiting the Abstract Factory pattern, we have a possibility to obtain instances of a concrete IMetric implementation [Ellis et al. 2007]. The framework simply calls a factory method and does not need to care about the instantiation of it. This also allows singleton implementations which carry a state throughout their lifetime.

In Figure 6.10 we also see the definition of the IMetricFactory, which allows to implement concrete factories. As the user interface strongly cooperates with this interface, it must provide a getName and a instanceofThisMetric method. The first method identifies the name of the given metric which is eventually shown to the user, by returning the name a string value. The second method is used by the framework to check if the current metric instance was created by this factory. We use this to give the user a feedback on which metric is currently selected. For this, we pass the current metric instance to the concrete factory, which checks if it is the corresponding metric type. If this is the case, true is returned, otherwise false is returned. The factory method which provides the framework with new instances is called getMetricInstance. The method returns an IMetric instance upon its call.

To provide a set of metrics, we implement three different version. An OverallExecutionMetric sums the execution times for every component up and calculates afterwards the component’s hotness by applying the function which maps values linearly from 0 for non-executing components, up to 100 for the component with the most summed up execution time. A further metric is the MeanExecutionMetric. It identifies the mean value
6. Architecture and Implementation of the Framework

Figure 6.12. Timing behavior diagram of two components

of every component’s execution times. A hotness value of 100 means the highest mean execution time, a value of 0 no execution. The remaining components are mapped linearly between both values. The third metric is the NormalizedMeanExecutionMetric. The difference between this and the MeanExecutionMetric is the normalization toward the component with the lowest mean execution time, therefore assigning it the hotness value 0. The value 100 still shows the highest mean execution time and the remaining value are also mapped from 0 to 100 linearly.

Figure 6.12 shows two components. Component A is executed three times, whereas Component B is only executed once. However, Component B needs twice the execution time. Suppose we observe the whole execution of the program, thus not filtering out any component execution. If we now analyze the execution times by applying the OverallExecutionMetric the execution times of Component A form the maximal value concerning the sum of execution times, therefore Component A’s hotness value is 100. Component B’s execution times sum up to a value equally to \( \frac{2}{3} \) of Component A’s, therefore it becomes 67 assigned as hotness value. The MeanExecutionMetric assigns Component B the value 100 and Component A the value 50 as it execution occur in half the time. Component B maintains a hotness value of 100 if we apply the NormalizedMeanMetric, however, as Component A’s mean execution time is the lowest of all monitored components, it is assigned with a hotness value of 0.

6.2.7 Visualization Framework

The Visualization Framework component provides both views for our diagrams. It receives the pre-processed data from the Record Manager and forwards it to the corresponding SWT widgets. As we provide two different diagram types, the plotting of both diagrams also differs significantly.
6.2. Implementation

The execution behavior diagram view showing the result of the OverallExecutionMetric

**Figure 6.13.** The execution behavior diagram view showing the result of the OverallExecutionMetric

**The Execution Behavior Diagram View**

The view for the execution behavior diagram consists of two parts. As we see in Figure 6.13, the top part of the view consists of the actual diagram. At the bottom, a time scale provides access to the filter where its time interval is presented as a red region. It also represents the lifetime of the monitored application, beginning with 0 and counting the running time in milliseconds.

Upon the start of the execution of the monitored code, the Visualization Framework initializes the view. In this particular case, the actual diagram is created by using the KLighD framework. In order to use it, we need to provide a synthesis class which transforms instances of the metamodel of our Monitoring DSL into a model representing the diagram elements. KLighD uses a concept of connected nodes, whereas nodes can have any arbitrary shape. We transform each component into a rectangle shaped node and add ports to it. Those ports are derived from the component’s interfaces, thus every port represents one specific interface. Optionally, we connect all ports which are also connected by the incoming model. KLighD automatically creates the diagram which is drag- and scalable. After the transformation process, a map is created linking each component to its corresponding diagram node.
6. Architecture and Implementation of the Framework

The time scale is created manually. We implement an own SWT widget and place it below the diagram in the same view. A PaintListener allows to draw within the provided canvas which we use to draw the different elements. All elements are positioned toward a reference point, which is the point of the last execution of the monitored application, for instance 6500 milliseconds in Figure 6.13. At first, the time interval of the filter is drawn, by collecting its bounds from the Record Manager and placing a red rectangle within these bounds. For this, we use the methods presented in Section 6.2.6. After this, the actual scale is drawn. This order must be respected, as otherwise the scale would be hidden behind the filter’s rectangle. For the scale, vertical lines are plotted in certain intervals, whereas every 500 milliseconds this line is emphasized by applying an increased height and placing the corresponding time value above.

One requirement is the ability to observe certain passed time intervals. Therefore, we need a mechanism which allows to shift the observed time span to different points. We achieve this by adding a drag and drop interaction to the time scale and the filter window. A MouseListener, provided by the SWT framework, allows to react on actions performed by the user. If the user clicks on an element, the listener checks which element was clicked and moves this elements upon every mouse moving. This is done by shifting the reference point or updating the filter, depending on what was clicked.

The coloring of the diagram elements is performed by modifying the corresponding node. KLightD holds a so called KRendering for every element of its generated diagram. The KRendering is a part of a metamodel which is KLightD-specific and allows annotations such as background color. The metamodel is based upon the EMF, which provides a mechanism to monitor changes on model classes. Therefore, if the background color of a KRendering is changed, a listener is called which eventually handles the coloring of the element. This means, our framework just needs to update this attribute depending on the values it receives from the Record Manager.

Above these technical aspect, we consider some qualitative aspects with the target to optimize this view for the human sensation. Fechner [1965] discovered that the just noticeable difference of stimuli depends on the initial stimuli, leading to the observation that the relative difference between two stimuli must be constant. Therefore, stimuli of a higher magnitude must change by a greater amount to be noted by a human. This observation is called Weber’s law. We may consider this effect when it comes to the coloring of the diagram elements, to aid the user to differentiate better the values. For this, we need a function which calculates the color value of an element by taking its hotness value into account. A simple approach is to use a linear function which interpolates the hotness value to a value range of 0 to 255. However, exactly this approach should be avoided as differences toward higher magnitudes of hotness would not be easily differentiated. The preferable approach would be to use a quadratic function as base, in order to achieve greater differences in the color value toward higher magnitudes. Figure 6.14 shows a plot of the function

$$c(h) = \frac{h^2}{100} + \frac{255}{100}$$ (6.1)
6.2. Implementation

to the right side. \( c(h) \) is the calculated color value for a hotness value of \( h \). We see in the figure, that with this function, an increase of the hotness value of 0 to 20 would lead to an increase of the color value by 10.2. On the other side, if we increase the hotness value from 80 to 100, the difference of the color value would be 91.8. Figure 6.14 also shows a comparison between this function and the linear function

\[
l(h) = \frac{255}{100} \cdot h
\]

where \( h \) also depicts the incoming hotness value which is located to the left side in the figure. This function solely upscales the hotness value and does in particular not modify the characteristic line. Below both functions some examples are illustrated which show how each function applies to a red color. Both depict the results of the hotness values 0, 20, 80, and 100. Especially, the differences between the values 80 and 100 are of interest. The perceived color difference of the linear function is lower compared to the colors calculated by the quadratic function. Therefore, the quadratic function is able to provide a characteristic line which better satisfies the human sensation as observed by Fechner [1965]. This function is not an optimal solution, rather than an approximation toward it.

One further approach is to reduce the possible values the element’s color may take. The Equation 6.1 allows the color to take any value from 0 to 255 in steps of one. However, we can summarize certain value ranges to one single value, reducing the granularity. By combining this with approach with the Equation 6.1, we can further aid the user to differentiate between values. Figure 6.15 shows an approach with six steps on the right side.
6. Architecture and Implementation of the Framework

![Diagram showing a plot of the stepped coloring function](image)

**Figure 6.15.** Diagram showing a plot of the stepped coloring function

The advantage of this is the higher difference between possible color values. This comes with the cost of a reduced resolution. For this example, we are not able to show more than six different values. Additionally, the user is not able to differentiate between hotness values which lie on the same step range. With Java, the realization of these steps can be done with four further operations.

\[ s_k(h) = \left( \frac{c(h)}{255} \right) \cdot \frac{255}{k} \quad (6.3) \]

Equation 6.3 shows how we calculate stepped color values. \( k \) is the number of steps the function should create. Furthermore, the division operator depicts the integer division here. This operator allows us to achieve the steps without heavy calculation. The reason behind this is the fact that results of Java’s integer division operation are rounded toward zero\(^6\). Therefore, this division scales the color value down to a value between 0 and \( k \). By applying the multiplication after this, we rescale the value to the range 0 to 255. As the first value is an integer, we summarize all \( h \) values which are scaled down to the same value in the first step.

Additionally, the user should also be provided with some tools, to adapt the visualization for its needs. In Figure 6.13 we see a set of icons in the top right corner. The two rightmost icons are used to minimize and maximize the view. In addition to this, we have three further icons, beginning with the arrange button on the left. This is provided by KLighD and rearranges the diagram in order to make most out of the view’s size. Next, is the metric button, depicted with an icon which resembles an equation. Upon an action, this opens a dropdown menu in which the user is shown a set of metrics and in particular

---

\(^6\)https://docs.oracle.com/javase/specs/jls/se7/html/jls-15.html#jls-15.17.2
6.2. Implementation

which one is currently used. By selecting an element of this list, the corresponding factory is called in order to create a new metric instance which is passed to the Record Manager for further use. The remaining icon, an arrow pointing downwards, opens a list with secondary menu elements. This list contains elements which are not frequently used and therefore may be hidden from the primary view. Our contribution to this list is a menu to select the color with which the elements are colored. As the user may use a metric which shows cold spots instead of hot ones, a blue color may be a better choice. The colors are presented with a dropdown menu showing four default colors and a menu entry to open a color choose dialog. For the selection of a color, we use the strategy pattern and provide a color interface called IMenuColorItem. It defines three methods which are used by the framework to calculate the final color, called getRed, getGreen, and getBlue. Each method receives an integer depicting the color value between 0 and 255, describing the color’s intensity. Based on this value, the implementing class can calculate the final color and return its RGB values. The framework provides four default colors, including red, green, blue, and gray. However, the user can pick any arbitrary color through a dialog. As this dialog returns the RGB values of the picked color, we are able to use these values in an anonymous class which provides the selected color to the framework.

Timing Behavior Diagram

The second view is used to display the timing behavior diagram presented in Chapter 4. The view uses the Canvas widget provided by the SWT framework to draw the final diagram. For this, a set of elements is hold by the view, which should be displayed. This set is updated by the Record Manager as long as the monitored program is running. After its termination, the set remains the same, therefore no updates are needed after this point.

The drawing of the diagram is performed in separate steps. At first, the time scale is plotted in a similar way to the execution behavior diagram view. The main difference is its orientation, which is now vertical instead of horizontal. Additionally, we use a different reference point for this diagram. All elements are positioned relative to the monitored program’s start, therefore toward the 0 milliseconds point. For a further explanation of the time scale plotting, we refer to the execution behavior diagram view.

After the time scale is plotted, the components and their timelines are drawn. To allow arbitrary names for each component, its size needs to be adapted dynamically depending on the size of its name. Therefore, we use a method provided by SWT’s API which returns the size of a string. With this information, the components can be finally placed and the position of their timelines can be calculated. As we use 0 milliseconds as reference point and know how many milliseconds are presented by a pixel, we are able to place rectangles denoting the individual execution time information for each component. In contrast to the execution behavior diagram, we use a data structure called Sequence here. Furthermore, a dedicated data structure reinforces the encapsulation as dependencies between this view and the execution behavior diagram view are avoided. A Sequence represents the individual executions of a component with their begin and end time, but also provides
6. Architecture and Implementation of the Framework

Figure 6.16. The timing behavior diagram view of an executing program

information toward the executing component and above this if the Sequence is currently executed. Therefore, we iterate over the received data, calculate each rectangle’s position, and eventually plot the rectangle. The information provided by a Sequence allows us to emphasize sequences which are currently executed, as this diagram is drawn live while the monitored program is executed. For this, we check if the corresponding Sequence is currently executed and fill the rectangle with a different color. In our case, we use a light green color indicating the current execution of the component. Additionally, as the diagram grows in size, the needed space for its visualization may exceed the size of the view. To deal with this, the view itself is scrollable, which we achieve by shifting the reference point depending on the scroll event. As all elements are placed relative to this point, they are also repositioned. Figure 6.16 shows the final view showing a the live execution of a component-based program. This consists of two components which are executed by two threads. By looking at the diagram we can see that CompA already was executed once and is currently executed a second time. The same applies to CompB.

For this view, the user should also be provided with tools to facilitate the view’s handling. By looking at the diagram, exact execution times can only be estimated. To improve this, the exact data of each Sequence can be visualized. Therefore, we use a
6.2. Implementation

MouseListener to check if a Sequence is clicked and if so, to visualize the data next to the Sequence itself.

The diagram is drawn from top to bottom, meaning the most recent data is shown at the bottom. As the diagram may exceed the size of the view, the recent data would be hidden virtually below the view. Therefore, if the user wants to see recent data, a continuously scrolling is needed. To aid the user, we provide a function which always shifts the diagram to the bottom. This is achieved by pushing the scrollbar to the end point before the draw process is started, as the draw process itself calculates the offset depending on the position of the scrollbar. This function can be activated or deactivated with the leftmost button in the top right corner of the view.

One last function reduces the need to scroll to identify the component related to an observed timeline. If the diagram exceeds the size of the view, components may be hidden. Therefore, the user needs to scroll to the top to identify certain components. This is solved by fixing the corresponding rectangles to the top of the view. However, this requires a reordering of the drawing process, as due to the used painter’s algorithm, the timelines would overlap the rectangles. Therefore, the components need to be virtually placed first and drawn as final step. This way, the components are drawn over other elements, therefore not overlapped by them anymore. To activate this function, the view provides a button at the top right corner, depicted with a pin icon.
After the implementation of the framework, we need to evaluate and demonstrate its purpose. As mentioned in Section 1.1, the main target of this thesis is to realize a live visualization for developers to measure and analyze code in their development process. Therefore, in Section 7.1 we show that the final framework fulfills these claims.

Beside this, performance aspects should also be examined. As our framework interferes with the monitored code, disadvantages concerning the performance of the code may occur. As a result, we analyze in Section 7.2 the influence which the framework takes on the execution of the code.

In the following we take a closer look on both aspects and use the Goal-Question-Metric (GQM) paradigm by Basili and Rombach [1988] as framework for our investigation. Questions toward the goals, which we aim with this evaluation, are defined. Metrics eventually quantify the results we achieve to give concrete values which can be compared and discussed.

7.1 Goal 1: Feasibility of the Approach

Our first goal is to evaluate our approach’s feasibility. In Chapter 4 to 6, we presented a variety of concepts which the framework is composed of. In this section we show the correct behavior of these concepts one by one and thus demonstrate the final working result.

For this purpose, we first examine and discuss individual concepts in Section 7.1.1. Afterwards in Section 7.1.2 we show the feasibility of the approach by applying an example software system which resembles possible systems which may be used for our framework.

7.1.1 Question: To which extent are the activities implemented?

Figure 7.1 shows the intended approach presented in Chapter 4. In order to provide a live visualization of the monitoring data, the implemented framework must proceed in this way. Therefore, the implementation of every activity is essential for the framework’s correct behavior.

As a result, we need to examine the contents of each activity and to show their status toward their implementation.
7. Evaluation

![Activity Diagram](image1.png)

**Figure 7.1.** An activity diagram showing the process flow for a live visualization

![Component Diagram](image2.png)

**Figure 7.2.** A component diagram of a fictional encryption plug-in for a file hosting system

**Metric: Activity Instrumentation is implemented/not implemented**

The *Instrumentation* activity summarizes all processes which enable the monitoring of the code. In particular, with the use of the Kieker framework, this means the supply of an *aop.xml* file. For our approach, as we do not want to burden the user with it, we provide a DSL which translates into this file. Section 6.2.2 describes how we implement this functionality in our approach.

In order to show this functionality, we create a simple example which we instrument by using our provided DSL. In Figure 7.2 we see an example of a plug-in which allows to encrypt files on a private file hosting service. The functionality of the plug-in is encapsulated into three components, called *FileHandler*, *FileEncrypter*, and *FSWriter* which all lie in the Java package called *fileHost.plugins.encrypt*. The first component is responsible for file handling and exposes a method called *open* as a provided interface to the environment. This method accepts data of the type *File* and reads it. After the processing within the component, the method *send* is called with a parameter of the type *FileWrapper*. This method calls the second component *FileEncrypter*, which performs the actual encryption by accessing it through its *accept* method. *FSWriter* eventually saves the file. This is performed upon a call of its method *save* with a parameter of the type *EncryptedFile* which it receives from the method *saveFile* of the component *FileEncrypter*.

If a developer wants to observe the behavior of this program with our framework, he needs to provide an *arch* file describing the program’s structure. Listing 7.1 shows the corresponding *arch* file for the given example. The target of the *Instrumentation* activity is to instrument the corresponding code by using this file. The process presented in Section 6.2.2
7.1. Goal 1: Feasibility of the Approach

```java
FileHandler {
    in fileHost.plugins.encrypt.FileHandler.open(fileHost.plugins.encrypt.data.File)
    out fileHost.plugins.encrypt.FileHandler.send(fileHost.plugins.encrypt.data.FileWrapper)
}

FileEncrypter {
    in fileHost.plugins.encrypt.FileEncrypter.accept(fileHost.plugins.encrypt.data.FileWrapper)
    out fileHost.plugins.encrypt.FileEncrypter.saveFile(fileHost.plugins.encrypt.data.EncryptedFile)
}

FSWriter {
    in fileHost.plugins.encrypt.FSWriter.save(fileHost.plugins.encrypt.data.EncryptedFile)
}
```

Listing 7.1. `arch` file describing the components of the file host example in Figure 7.2

...yields in the `aop.xml` as presented by Listing 7.2.

This result contains the instructions where Kieker, or to be more specific AspectJ, needs to weave code into. We see how our translation gives a correct XML file. For instance, the name of the concrete aspects must be unique. As stated in Section 6.2.2 this is done by using an integer which is incremented upon the creation of each concrete aspect. In Listing 7.2 we see this in lines 12, 15, 18, 21, and 24, where after each occurrence of the sequence `FrameworkAspect` a unique integer is appended.

We also discussed, how the linking between the concrete aspect and the defined component is realized. In this example, the components have the identifier 1, 2, and 3, linking to the `FileHandler`, `FileEncrypter`, and the `FSWriter`. We also find these integers in the resulting `aop.xml`. The names for each concrete aspect end with the sequence `Id` followed by the corresponding integer. Comparing these with the original `arch` file reveals a correct assignment of aspects to components.

The individual pointcuts for each concrete aspect also needs to be provided. As the definition of interfaces in our grammar is a subset of the needed information for the expression definition in AspectJ, the generator wraps the interface definition in an `execution` statement and allows arbitrary return types. For instance, Line 13 in Listing 7.2 defines the pointcut for the incoming interface of the `FileHandler` component. We see, that our transformation directly maps the interface to the pointcut expression. However, the return type of the method is assigned with a wildcard. The reason lies on Java’s method signature...
Listing 7.2. Transformed aop.xml file based on Listing 7.1
7.1. Goal 1: Feasibility of the Approach

specification\(^1\). If two methods equal in their name and parameter types, a definition with two different return types leads to a compile-time error, as both methods have override-equivalent signatures. Therefore, we do not need to constraint the pointcut by adding a return type and as a result spare this redundant information to the user.

As a last point, we need to ensure that incoming interfaces are distinguished from outgoing interfaces. Lines 12 and 15 of the resulting aop.xml show the concrete aspects for the FileHandler component. In this case, both concrete aspects extend from different abstract aspects which are chosen by the generator, depending on the interface type. The first, representing the incoming interface, extends from AbstractInCompAspect whereas the second extends from AbstractOutCompAspect which should be used for outgoing interfaces.

Lines 4 to 10 show static code which is always added to instruct AspectJ which packages should not be considered for weaving. In our case, we exclude any packages related to logging framework. As the actual instrumentation is handed to Kieker, we refer to [Kieker Project 2015] and [van Hoorn et al. 2012] for a more profound discussion toward this topic.

**Results and Discussion**  As we see, our implementation of the Monitoring DSL and in particular the transformation to an aop.xml provides the needed functionality to fulfill the Instrumentation activity. By creating this file, Kieker is eventually able to instrument the components as defined by the user. Nevertheless, this solution lacks in some points. As we see in our approach, the transformation does not create complementary aspects for the FSWriter component, due to its missing outgoing interface. By definition the component does only have one incoming interface, leading to a missing measurement point at the end of its execution. To solve this, we need to deviate from the strict definition of incoming and outgoing interfaces and implicitly define the incoming interface as outgoing interface as well. This is already feasible, if the user explicitly defines the interface as both types. We dissociate from an automatic solution at this point, as this needs some further investigation. For instance, a component with only one incoming interface could automatically define this interface as outgoing as well. However, this solution is not trivial for multiple interfaces, especially if the number of incoming interfaces differs from the number of outgoing interfaces, as a determination of the implicit interface cannot be done anymore. Therefore, our implementation does not provide such mechanism.

Considering these aspects we consider the Instrumentation activity as implemented.

**Metric: Activity Measure is implemented/not implemented**

The Measure activity is executed at the runtime of the instrumented code. Essential for this activity is the collection of measurements at certain points which is performed by the Kieker framework. However, in Chapter 5 we show how this framework is enhanced so we can use it for the monitoring of components. Therefore, our approach relies on own

\(^1\)http://docs.oracle.com/javase/specs/jls/se8/html/jls-8.html#jls-8.4.4
7. Evaluation

Figure 7.3. A sequence diagram showing the process flow for a live visualization

implementations of monitoring probes. For the correct execution of this activity we need to analyze if the implemented probes behave correctly.

For the demonstration of our implementation, we recall the example in Figure 7.2 of the previous section. This example can also be used to show the behavior of our implemented probes. Therefore, we model a control flow sequence and implement it as an execution example. The sequence diagram in Figure 7.3 shows a possible execution flow of the plug-in. As already stated in the previous section, we also need to declare the `save` method of the `FSWriter` component as an outgoing interface. However, if we apply this modification to the `aop.xml` file presented in Listing 7.2 we are able to measure the times of $t_a$, $t_b$, and $t_c$.

The entry probe is located at the beginning of the incoming interface method, and the exit probe on the end of the outgoing interface probe, as presented in Chapter 5. In the case of Figure 7.3 the corresponding probes are located on both ends of the arrows. Therefore, the values can be determined as the subtraction of the first value of the latter one.

In order to show the measurement mechanism we provide, we implement this scenario and emulate its execution. This is done by defining specific time intervals where the executing thread is put to sleep. At each point where workload needs to be simulated, we perform such sleep. In our example, the `FileHandler`'s workload is simulated in the method `execute` which is called by the `open` method. The `FileEncrypter` performs its task within the method `accept`, where we therefore insert a sleep call. As the `FSWriter` only consists of one single method, it also performs its task within this `save` method. This is also the method, which puts the thread to sleep. The values we use are 500 milliseconds for `FileHandler`, 1000 milliseconds for `FileEncrypter`, and 300 milliseconds for `FSWriter`. The advantage of this approach lies in the predictable execution times of the components.

For the demonstration, we need reference values. As we do not have any absolute measurement values we can use as reference, we need to measure both points
7.1. Goal 1: Feasibility of the Approach

and calculate the raw execution time. We may use the defined sleep times as refer-
ence values, but these do not include side effects like the scheduling or any delays
upon the method call. Therefore, we use Kieker and its already implemented moni-
toring probes to measure the execution times to gain reference values for each compo-
ment. To get reference values as close as possible to the correct values, we use Kieker’s
kieker.monitoring.probe.aspectj.flow.operationExecution.AbstractAspect aspect, as our own im-
plementations base on it. However, this aspect creates two records for every method it
monitors: one record before and one after the execution. Therefore, this approach with
Kieker does not completely resemble our implementation but suits to generate reference
values. We use the same aop.xml but adapt it to use the different aspect and use milliseconds
as resolution for the records, as also done by our framework. The collected records are
stored in a human-readable csv file.

In order to receive the raw measurements collected by our framework, a temporary
modification on the framework needs to be done. We instruct the writer of the monitoring
part to also save the records in a human-readable csv file.

We run both frameworks once each to receive comparable results. As this is only a
course comparison and not intended to show the performance, this approach meets our
requirements. Both approaches return files containing the raw record data, which is not
aggregated and only shows the time each method was reached. Above this, Kieker also
provides redundant data we may ignore. Eventually we need to subtract the entry point
timings from the exit point timings in order to get the execution time.

<table>
<thead>
<tr>
<th>Component</th>
<th>Kieker</th>
<th>Our Framework</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileHandler (t_a)</td>
<td>1815</td>
<td>1812</td>
</tr>
<tr>
<td>FileEncrypter (t_b)</td>
<td>1310</td>
<td>1308</td>
</tr>
<tr>
<td>FSWriter (t_c)</td>
<td>304</td>
<td>308</td>
</tr>
</tbody>
</table>

The results are shown in Table 7.1. Each row represents the execution times of a
component. The name of the corresponding component is shown to the left. The left
column shows the execution times we measured and calculated for the execution with
Kieker. To its right are the execution times measured with our framework. What seems to
be surprising is the fact that all components, but FSWriter need more time than expected.
As mentioned, the result give the raw execution times. This means in particular, it also
includes the time needed to wait for a subsequent component to finish. For instance, if
we subtract the times the FSWriter needs for its execution from the results of FileEncrypter,
we receive values of 1006 milliseconds, and 1000 milliseconds. For our example these are
the correct times. The same applies for the FileHandler where we subtract FileEncrypter’s
execution times. Due to the rule of transitivity, the execution times of FSWriter are already
included. We receive therefore, 505 milliseconds for the execution with Kieker and 504
milliseconds for the execution with our framework. This also nearly equals our pre-defined
7. Evaluation

execution times. A perfect match would also be unusual due to two facts. First, any introduced overhead may take influence on the performance of the execution. As we introduce overhead through the monitoring probes, this applies here. Secondly, the values we hand over to the sleep method only give a lower bound on how long the thread must sleep. Depending on the scheduler and the underlying operation system, the awaken of the thread may be delay an uncertain amount of time.

Results and Discussion We see how we are able to measure the execution times of components by using our implementation of own monitoring probes. We place these implemented probes on the entry and exit point of the monitored component and are then able to calculate the correct execution time out of the collected monitoring data. However, it is not possible to measure the real execution time which does not include any waiting time. This is owed to constraints which force a certain placement of the probes, as discussed in Chapter 5.

Nevertheless, as our implementation is able to measure almost perfect execution times, we consider the Measure activity as implemented.

Metric: Activity Aggregate is implemented/not implemented

The Aggregate activity includes the analysis of the incoming records so that values toward the visualization are processed. Its input are the collected records of the previous activity and the output are visualization values which emphasize the importance of a component toward the execution times.

In our implementation, as presented in Section 6.2, the activity is performed by a Record Manager which matches records to calculate the execution times, a Filter which filters certain time intervals, and a Metric which analyzes the execution times and calculates the values required for the visualization. Therefore, we need to demonstrate the correct behavior of these three parts in order to show the correct implementation of this activity.

First, we take a closer look on the Record Manager. In Section 6.2.5 we show that this component consists of two different processes, one for each final view, due to the different needs of the views. As the data process of the timing behavior diagram is a subset of the data process of the execution behavior diagram, we limit our observation to the last one. Both approaches must match related records, whereas the execution behavior diagram needs further filtering and calculation.

First, we show the behavior of the record matching procedure. The approach toward the record matching is shown Chapter 5, whereas certain implementation aspects are discussed in Section 6.2.5. Depending on the traceId the Record Manager is able to match related records by using a stack. In its basic implementation, the Record Manager is able to determine the raw execution times of components, which equal to our manually calculated values in the previous subsection. Furthermore, we presented a technique to determine the real execution times of components. To show the working implementation of both techniques, we use again our file hosting plug-in example.
7.1. Goal 1: Feasibility of the Approach

In Section 7.1.1 we used the example shown in Figure 7.3 to show how our probes are used to derive the raw execution times of each component. We now use the same example and let the Record Manager derive the actual execution times. By default our implementation derives the real execution times, which represents the time needed for execution without any waiting times. However, we are able to temporary deactivate this feature in our implementation to show the results as raw execution times. We use the same values, monitor the example software system, and write the results to the console.

<table>
<thead>
<tr>
<th>Component</th>
<th>Raw Execution Time</th>
<th>Real Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileHandler</td>
<td>1812</td>
<td>506</td>
</tr>
<tr>
<td>FileEncrypter</td>
<td>1306</td>
<td>1001</td>
</tr>
<tr>
<td>FWriter</td>
<td>305</td>
<td>305</td>
</tr>
</tbody>
</table>

Table 7.2 shows the values our implementation derived from the incoming records. The left column shows the values for the derived raw execution times. We see that these values equal the manually calculated values in Section 7.1.1 to a certain accuracy. The results do not fully equal the manually calculated values, as these values are collected on a different execution of the framework, hence resulting in different timings. However, these values equal the values we expected and are therefore correct.

After this, we reactivate the calculation of the real execution times. The Record Manager subtracts the execution times of components of the execution times of calling components. The values in the right column of Table 7.2 show the values our framework derived for the same execution of the monitored program. The results equal the expected values, we would calculate manually. For instance, if we subtract the raw execution time of the FileEncrypter from the raw execution time of the FileHandler, we get 1812ms \(-\) 1306ms \(=\) 506ms. For the FWriter we also get a value which equals the one derived by our framework.

As a result, we see that our implementation of the Record Manager matches the records correctly and is also able to derive the correct execution times of each component. However, one major question which arises from our observation is whether or not our Record Manager behaves correctly in a multi-threaded environment. As the matching is based on the individual traceIds we should expect a correct behavior. To evaluate this, we simulate our example by implementing a simplified version with TeeTime. We create three individual stages, each one resembling a component of our example, and execute all three concurrently.

Figure 7.4 shows the new execution flow of the parallelized version of our example software system. The sequence diagram is simplified and only shows the communication of the components. The difference to the sequential version lies in the waiting times of each component. By using asynchronous calls, any waiting for subsequent components is eliminated. A propagation of execution times does not take place anymore. Therefore, the raw execution times our framework measures for this scenario should not include propagated execution times and equal the values of the real execution times. Furthermore,
7. Evaluation

Table 7.3. Derived execution times of the parallelized example system in milliseconds

<table>
<thead>
<tr>
<th>Component</th>
<th>Raw Execution Time</th>
<th>Real Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileHandler</td>
<td>507</td>
<td>507</td>
</tr>
<tr>
<td>FileEncrypter</td>
<td>1005</td>
<td>1005</td>
</tr>
<tr>
<td>FSWriter</td>
<td>303</td>
<td>303</td>
</tr>
</tbody>
</table>

Table 7.3 shows the results, which meet our expectations. The derived values equal throughout the different calculation methods and do also meet the execution times of our software system. Therefore, our implementation does behave correct in multi-threaded scenarios. As the process of the timing behavior diagram relies on the same matching mechanism, but without any execution time calculation, we also call its implementation correct.

The next step within the Aggregate activity is the filtering of execution times. In order for the user to observe certain time intervals, a Filter is needed to only pass on relevant data, as discussed in Section 6.2.6. Its input are the execution times, annotated with the corresponding component and the exact time the execution was performed. The output is the same set, reduced by the executions which fall outside the filter interval.

To show this, we pick a component of the execution and observe how the filter behaves on the component’s execution values. In our example, the FileEncrypter is executed between...
7.1. Goal 1: Feasibility of the Approach

3444 ms and 4449 ms after the program is started. As we are able to shift the filter interval, we may now evaluate it against a set of scenarios. These include a full inclusion of the execution time instance, a full exclusion and two intersecting scenarios where the filter should return parts of the execution. Therefore, we have four different test scenarios and four expected values the filter should return.

We configure the filter to use a filter interval of 1500 milliseconds and shift this time window corresponding to our test cases. To evaluate the full exclusion, we place the filter window at 1900 milliseconds after the program start, hence only passing values between 1900 and 3400 milliseconds. As the observed execution time instance begins at 3444 milliseconds, the filter should not pass it through in this case. The other cases are evaluated in the same way. More specific, the partial filtering should only return the values which lie in the interval filter. For instance, if the filter interval is set to 4000 – 5500 ms, the filter should return 449 milliseconds which represents the execution instance in its part of 4000 to 4449 milliseconds. In order to gain access to the filter’s results, we print them temporarily to the console.

Table 7.4. Filtered execution times of FileEncrypter in ms

<table>
<thead>
<tr>
<th>Filter Interval</th>
<th>Filtered Execution Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1900-3400</td>
<td>0</td>
</tr>
<tr>
<td>3000-4500</td>
<td>1005</td>
</tr>
<tr>
<td>2500-4000</td>
<td>556</td>
</tr>
<tr>
<td>4000-5500</td>
<td>449</td>
</tr>
</tbody>
</table>

Table 7.4 shows the collected results. As we see, the results equal the expected values. The first line shows the case of a full exclusion, followed by a full inclusion. The third and fourth line show a case where the filter interval overlaps with the interval of the execution, whereas the third line shows the case of an overlapping at the execution’s beginning and the fourth line an overlapping at the end. It should be mentioned, that the filter which we use for our investigation here is the AdaptableIntervalFilter which is fully adaptable. In Section 6.2.6 we also present the DefaultIntervalFilterImplementation which almost equals the shown filter, but does not allow any adaptions on the filter interval. As the actual filtering is done in the same way, we do not show this implementation at this point.

The last step in the Aggregate activity is the execution of a metric. This is used to calculate the values for the visualization in order to emphasize certain components. In Section 6.2.6 we presented the approach and a set of concrete implementations which need to be evaluated for correct behavior.

To demonstrate the functionality of our metric implementations we again use the plug-in example. We use the TeeTime-based implementation which was used to demonstrate the filter’s functionality and modify is slightly. We extend the execution flow shown in Figure 7.4 by executing the FileHandler component twice. However, the second execution
does not invoke the subsequent components, hence the control flow stops right after the execution. The remaining components stay untouched. While executing this extended program, we print the interim results, calculated by each metric, to the console.

Our framework implements three different metrics. The first metric, called MeanExecutionMetric gives every component a hotness value depending on how much time an execution needs in average. The component with the highest value is assigned with the value 100, the remaining values are linearly interpolated between this value and 0, indicating no execution at all. For our example, the mean execution time values of the components equal the values of a single execution. For instance, FileHandler is executed twice with an execution time of 500 milliseconds, therefore the mean value is also 500 milliseconds. As the mean value of FileEncrypter is 1000 milliseconds, and FSWriter’s mean value is 300ms, we are able to give expected values the metric should return. The FileEncrypter needs the most average execution time of all monitored components, therefore it is assigned the value 100. As the FileHandler requires only half the amount of time, we expect its value to be 50. The FSWriter executes in 30% of the time the FileEncrypter needs, therefore we expect its hotness value to be 30.

The second metric is the NormalizedMeanMetric. Its behavior is similar to the MeanExecutionMetric, however a hotness value of 0 represents the component with the lowest mean execution time. Therefore, we expect the FileEncrypter to maintain the hotness value 100, but a hotness value of 0 for the FSWriter component. By interpolating the value of the FileHandler, we determine a value of 29 for it.

The third metric we implemented is the OverallExecutionMetric. It emphasizes components depending on the overall time they were executed. Similar to the MeanExecutionMetric the component with the highest execution time is assigned with a hotness value of 100, whereas 0 indicates no execution of the component. As the FileHandler executes twice with an execution time of 500 milliseconds, its overall execution time is 1000 milliseconds, which is the same value of the FileEncrypter. As this is the maximum value, both are expected to be assigned with a hotness value of 100. This leaves the FSWriter with a hotness value of 30, as its overall execution time of 300 milliseconds equals to 30% of the values of the other two components.

Table 7.5. Hotness values of each component

<table>
<thead>
<tr>
<th>Component</th>
<th>Mean</th>
<th>Normalized Mean</th>
<th>Overall</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileHandler</td>
<td>49</td>
<td>29</td>
<td>100</td>
</tr>
<tr>
<td>FileEncrypter</td>
<td>100</td>
<td>100</td>
<td>98</td>
</tr>
<tr>
<td>FSWriter</td>
<td>30</td>
<td>0</td>
<td>28</td>
</tr>
</tbody>
</table>

Table 7.5 shows the results of all three metric implementations. First, we concentrate on the column denoted with Mean which shows the results of the MeanExecutionMetric. The results we see deviate slightly from our expected values. These values are not false, yet the slight difference is a result of the rounding performed by Java’s division operator.
and the measurement results, which also are slightly higher than the expected values. The same applies to the other metrics. Their values also deviate from the expected results, but this deviation is owed to the same effects. Nevertheless, with these aspects in mind, these results show our expected values.

Results and Discussion We have shown how the results of the Aggregate activity are matched, filtered, and eventually used to calculate hotness values. Our implementation of the matching mechanism turns out to behave correctly. However, its values deviate from the expected due to the noise introduced by the measurements. As we are not able to determine the significance of this noise, we are not able to eliminate this measurement errors and therefore this effect is propagated through all subsequent processes. The implementation of our filter is also shown and behaves correctly. Also our implementations of the metric interface behave correctly, but introduce a further error toward the results, as the division also performs a rounding toward zero. However, as the value is rounded to the next integer, we are able to give an upper bound for the expected error of 1%.

As all of its sub-process are proven to be implemented correctly, we also consider this activity as implemented.

Metric: Activity Visualize is implemented/not implemented

The Visualize activity includes the application of the hotness values to the elements of the diagram within the execution behavior diagram view, as well as the application of the executions of components to the timing behavior diagram view.

Execution Behavior Diagram As discussed in Section 6.2.7, before applying the values to the execution behavior diagram, the color values are determined based on the component’s hotness value. For this, we use a quadratic function to calculate the value. If we use the values of Table 7.5 calculated by the OverallExecutionMetric, we are able to manually calculate the expected color values. For this we apply the equation \( c(h) = \frac{h^2}{100} \times \frac{255}{100} \) we presented in Section 6.2.7 on the hotness values, and receive color values of 255, 244, and 21 for the hotness values of 100, 98, and 29. The results calculated by the framework only deviate for the last result, which is again caused by errors introduced by the rounding of the division operator. However, if we apply these values on an execution behavior diagram, we expect both components FileHandler and FileEncrypter to have colors of similar intensity. The FSWriter is expected to be low saturated.

Figure 7.5a shows the final execution behavior diagram on which those values are applied. As expected, both components have a similar color, indicating a similar overall execution time of those components. The higher saturation compared to FSWriter shows the user the higher impact of both components on the overall execution time. One further aspect we also want to show here is the ability to select different colors, we also presented
7. Evaluation

in Section 6.2.7. Both figures in Figure 7.5 show the same results, whereas Figure 7.5b does not make use of the red color, but uses the color gray.

**Timing Behavior Diagram** For the visualization of the timing behavior diagram no additional calculation needs to be done. To get an image we may expect to see, we visualize the execution of the plug-in with it. As the timing behavior diagram is based on the sequence diagram, we may expect a diagram similar to the sequence diagram shown in Figure 7.4. However, `FileHandler` was executed twice, therefore two separate sequences should appear on the corresponding timeline. Additionally, the timing behavior diagram show absolute timings, hence the sequences of the different components should differ in size.

The result is shown in Figure 7.6 and meets our expectations. The diagram shows the individual executions of the components in a correct way. We see how the `FileEncrypter` is executed right after the `FileHandler` is executed for this first time. Furthermore, the diagram also shows how the `FileHandler` is executed twice and more specifically, the second execution is performed instantly after the first. Additionally, as the sizes of the sequences indicate the time needed for the execution, the user sees how the execution of both `FileHandler` executions take roughly the same amount of time as the execution of the `FileEncrypter`, which we already claimed in the previous subsection. One further functionality seen in the figure is the selection of specific sequences. The sequence of the `FileEncrypter` components is selected in this example, hence details toward this sequence are also shown in the view, which also equal to the values we collected in the previous subsections.

**Results and Discussion** We have shown the correct behavior of both views. In particular, the color values of the execution behavior diagram are correctly determined and the coloring of the corresponding elements is also done correctly. The timing behavior diagram shows the executions as sequences. Our implementation correctly places the sequences depending on their absolute timings. However, both approaches also have their limitations. The execution behavior diagram only shows the additional information in one dimension. For instance, we are not able to show in addition to the mean value the number of executions of the component. The timing behavior diagram is not able to show concurrent executions of a component, as sequence are simply overlapped. Furthermore, the time intervals in which a component waits for the execution of a subsequent component to terminate is also not shown.

**7.1.2 Question: Is the approach applicable to a software system of larger scale?**

In the previous question we show how the different functions of our framework work. However, as the framework is intended to be used with arbitrary software systems, we
7.1. Goal 1: Feasibility of the Approach

(a) Red colored execution behavior diagram

(b) Grey colored execution behavior diagram

Figure 7.5. Differently colored execution behavior diagrams
7. Evaluation

Figure 7.6. Timing behavior diagram of the plug-in example

need to show that it works with systems of larger scale. For this, we take on the example of file hosting plug-in and enlarge it to give an example of larger scale.

Figure 7.7 shows the architecture of our example. It consists of ten different components and is designed to show certain aspects of our framework. With these components we are able to simulate the upload of files and the access to a directory. As shown in Figure 7.8 we model certain control flows. A server is accessed twice to upload files at first, and to read a directory afterwards. Parts of the upload are performed concurrently, as the encryption may be executed in the background. However, upon the second access to the server the \textit{HTTPServer} must wait for all files to be read until it responds to the client. With this scenarios we are able to show how our framework processes nested executions and how the results look in a multi-threaded environment.

The different execution times of the individual components are selected in a way so that the metrics show different results. We achieve this by giving components which are not often executed a higher execution time. We do so, by simulating workloads by putting the thread a certain amount of time to sleep as the control flow reaches the corresponding component. For instance, the components \textit{FileEncrypter}, \textit{FSWriter}, \textit{FileReader},
7.1. Goal 1: Feasibility of the Approach

Figure 7.7. Example showing parts of a file hosting system

and FileDecrypter are executed multiple times, simulating their execution upon multiple files. Therefore, we assign these components low execution times compared to the remaining components, so they have a low mean execution time, but yet a high overall execution time. To be more precise, the components FileEncrypter and FSWriter are executed ten times in our example, whereas FileReader and FileDecrypter are executed 15 times. Vice versa, the remaining components get a higher execution time, but are only executed once or twice, leading to a lower overall execution time. In the following, we take a closer look on the final views after the monitored software system is terminated. Therefore, both views show the result of the entire execution.

Figure 7.9 on Page 81 shows the final timing behavior diagram. As we monitor all components, the diagram shows the individual sequences of each component. In particular, we see the execution flows we defined in Figure 7.8. Although the timing behavior diagram does not show the communication paths between the components, we see for instance, how the HTTPServer calls the UploadManager and PluginManager. At the end of PluginManager’s execution, the FileHandler is executed in a separate thread and the PluginManager instantly returns. As we are not able to see the caller-callee-relationship of two components, it is possible that a user may mistakenly understand that the HTTPServer executes the FileHandler on its second execution. Nevertheless, with our knowledge of the software system, we are able to see how the HTTPServer waits for the DirReader upon its second execution instead. Also, the multiple executions of the four components are shown here. The different sizes of the sequences also resemble the different execution times. As the four multiple executed components show sequences of smaller size, we are able to confirm their lower execution times compared to the remaining components.

Both diagrams in Figure 7.10 show execution behavior diagrams for the same execution of the monitored system, but with different applied metrics. Figure 7.10a shows the diagram with the MeanExecutionMetric. It therefore shows the mean execution times each component
7. Evaluation

required. A more saturated color indicates a higher mean execution time. We see the four components, we assigned with a low execution time, with a lower saturated color. The remaining components, with a higher execution time, are more saturated, as we expected. One exception are both the HTTPServer and the AccessControl, as both have a very low execution time and are only executed twice.

In Figure 7.10b we switch to the OverallExecutionMetric which emphasizes components depending on their overall execution time. This diagram differs significantly from the first. As we mentioned, we assign low execution times to the four component which are frequently executed. However, due to this frequent execution of those components, their overall execution time results in a high value, hence the higher saturation. In particular, the higher saturation of the FileDecrypter is owed to the fact it is executed 15 times compared to the ten times which the FileEncrypter is called.

Results and Discussion We see that our approach is also able to run with a software system of larger scale, consisting of ten components. Even with 55 executions at large, resulting in 110 records, the visualization is performed correctly. Furthermore, we are able to use this example and scale the amount of executions by running the system multiple times. Even with 8 iterations, and therefore 440 executions, or rather 880 records, the visualization has shown correct results and still remained fluent on a machine with a 2.6 Ghz Intel Core i5 4278U and 8GB of RAM. Nevertheless, such large scale system uncovers even more the limitations of this approach. With the higher amount of components and their executions, the timing behavior diagram gets more complex as the communication paths are not clear.

7.1.3 Threats to Validity

Every evaluation’s validity is subject to certain threats we need to disclose.

Internal Validity

The conclusions we made with each evaluation rely on the example we used to demonstrate the functionality. As this system only resembles a real world system and the execution times are generated by delaying the executing thread, guarantees of a correct measurement with our framework cannot be made. The measurements may depend on external factors beyond our knowledge. However, in order to prevent false conclusions, we used the Kieker framework to measure reference values which then are compared to the measurement results of our framework.

External Validity

To demonstrate the functionality of our framework, we used an example which covers different aspects our framework must support. In particular, we predefined the behavior of
7.2. Goal 2: Performance Overhead of the Implementation

the software system to enable the demonstration of the individual functions. Therefore, the results only show particular scenarios defined by our example. Different software architectures could lead to a different result.

The evaluation was performed on a single machine, which may cause different results, depending on the used machine. In particular, the results of this evaluation depend on our framework configuration.

7.2 Goal 2: Performance Overhead of the Implementation

As our implementation interferes with the execution of the monitored code, we need to analyze its impact to give users estimates on how the performance of their code varies. The interference may be caused by the instrumentation, as additional code is introduced to the software system causing overhead [Waller and Hasselbring 2013], or by the visualization itself, as threads execute in the background collecting the data, and therefore consuming resources.

7.2.1 Question: To which extent does our framework affect the performance of the monitored code?

With the final framework, we have five different scenarios of its usage we need to consider.

*No Framework (NF)* The software system executes without any instrumentation.

*With Framework (F)* In this scenario, we instrument the code by using our framework, but do not visualize the results.

*With Framework and Timing Behavior Diagram (Ft)* This equals scenario F combined with the visualization of the timing behavior diagram.

*With Framework and Execution Behavior Diagram (Fe)* In this case we show the execution behavior diagram while executing the monitored software system.

*With Framework and both Diagrams (Fte)* This scenario makes use of both visualization tools to show the execution of the monitored system.

The main target of this question is to compare these scenarios to derive a possible impact caused by our framework.

**Metric: Measurement of execution times of different scenarios**

For a comparison we need concrete values, which we get by measuring the execution time of the software system in each scenario. This macro benchmark is performed on a 2.6 Ghz Intel Core i5 4278U with two cores and four threads, and 8GB RAM which runs on macOS 10.12. Furthermore, we use the Java SE Runtime Environment in its version 1.8.0_31-b13.
7. Evaluation

As benchmark, we use a TeeTime-based application. The version of TeeTime used in this case is 3.0-SNAPSHOT built on 8 September 2016. The application generates a MD5 hash value for a given integer which is forwarded to a second stage that uses a brute force method to determine the original integer. This is done by calculating the hash value of a variable which is incremented after every calculation. As our original value is a positive integer, this guarantees the termination of the application.

We use this application as benchmark due to its CPU-intensive task. The example of a file hosting system we used earlier does not suite as benchmark, as the processor is mostly in idle and does therefore not require resources in the meanwhile. As these resources are available to the UI, the interferences are expected to be low.

This application hashes five integers and terminates afterwards. To increase the amount of operations, we execute this application multiple times. Furthermore, we also need a warm-up phase, as the Java Virtual Machine (JVM) performs various optimizations upon the start up of a Java application. Therefore, we execute the application eight times without measuring the execution time to let the JVM perform these optimizations. After this, we measure the time the application needs to execute further 8 times. Additionally, with every JVM execution, different strategies toward the optimization are selected. As a result, we need to invoke the JVM multiple time to gain comparable results. In summary, we execute the benchmark four times with different JVM invocations, run the application eight times as warm up phase, and eventually measure the execution time of further eight executions. This is done for every scenario and the average values for each scenario is determined, to gain comparable values.

Table 7.6. Measurement results of the benchmark in milliseconds

<table>
<thead>
<tr>
<th>Scenario</th>
<th>NF</th>
<th>F</th>
<th>Fe</th>
<th>Ft</th>
<th>Fet</th>
</tr>
</thead>
<tbody>
<tr>
<td>max</td>
<td>24622</td>
<td>24764</td>
<td>26297</td>
<td>26797</td>
<td>26720</td>
</tr>
<tr>
<td>min</td>
<td>24034</td>
<td>23960</td>
<td>24208</td>
<td>26126</td>
<td>26058</td>
</tr>
<tr>
<td>mean</td>
<td>24271.25</td>
<td>24352.25</td>
<td>24674.5</td>
<td>26502.75</td>
<td>26319.5</td>
</tr>
</tbody>
</table>

Table 7.6 shows the results we measured. For each scenario we see its maximum, minimum, and mean execution time.

Results and Discussion Considering the results we see a significant overhead caused by our framework if the visualization is used. Comparing the instrumented code (F) with the non-instrumented (NF), we see the execution time of the code increases by 0.3%. However, if diagrams are plotted using the collected measurements, the overhead increases. By using the execution behavior diagram, the execution time increases 1.8% compared to scenario NF. The timing behavior diagram causes an increase of 9.3%. As of our observation, this may be caused by the plotting of the diagram itself, as the drawing area linearly increases with the execution time, consuming more time to plot the diagram, whereas the drawing area of the execution behavior diagram does not change in size.
7.2. Goal 2: Performance Overhead of the Implementation

Considering these observation, the monitored software system undergoes measurable interferences which cause a higher execution time. User of our framework must therefore take this aspect into consideration when monitoring their code with our framework.

7.2.2 Threats to Validity

Internal Validity

The conclusion we made are based upon a certain set of measurements. These may deviate from the actual behavior of the software system we used. Furthermore, the measurements may be corrupted by external factors. To reduce these effects we performed multiple executions of the benchmark and determined the average values.

External Validity

The results of our measurements depend on the software system we used. As this example does not represent all use cases, the result may differ for different software system and use cases.

Furthermore, the measurements are performed on a single machine. On systems with different resources, the results may vary. In particular, the results also depend on the used JVM and OS.
7. Evaluation

Figure 7.8. Control flows executed with the example file hosting system
7.2. Goal 2: Performance Overhead of the Implementation

Figure 7.9. Timing behavior diagram of the executed plug-in
7. Evaluation

(a) Execution behavior diagram showing the mean execution times

(b) Execution behavior diagram showing the overall execution times

Figure 7.10. Execution behavior diagrams of the large scale example
In the domain of component-based software systems, a lot of research is done toward the performance optimization of software systems. For instance, van Hoorn [2014] introduced SLastic, a framework to reconfigure component-based software systems at runtime, depending on their workload. However, this approach does not deal with the root cause of poor performance. For this, developers need to be provided with reliable tools to monitor and analyze the performance of component-based software systems. Therefore, our approach aims toward the improvement of software systems during the development process.

Moving toward the visualization of component-based software systems and their performance, different approaches were developed. ExlopViz is a framework which gives information of the software system to the administrator by visualizing them [Fittkau 2015] to promote the comprehension of the software system. This framework is able to visualize the underlying architecture and the interactions among them during its runtime. Main focus of this framework however, is on the operation of software systems. A developer can derive bottlenecks in its software system from ExlopViz’s visualization, however this is not intended and does not integrate well into the development process. By integrating our framework directly into the development environment of the developer, changes on the software system can be detected before deployment and therefore fixes can be applied faster.

Nevertheless, approaches with a main focus on the development process, were presented. Heath and Finger [2003] focus on the visualization of message passing with their framework Paragraph. This framework visualizes certain aspects toward message passing by showing how messages are passed between the processors. This could be used to monitor the communication. Our approach on the other hand focuses on the performance of software systems themselves. With Paragraph, the developer is not able to give any statements on the performance of single components.

Other approaches, such as presented by Nutt et al. [1995], use offline analysis to such performance data. Their framework is able to show a variety of diagrams, but uses analyses on files which include trace information of the monitored software system. Similar analysis can be done by using Kieker’s analysis component. Our framework visualizes the data live at runtime and frequently updates the view depending on the recent execution data of the monitored software system. This again allows faster responses to bottlenecks.
8. Related Work

Ball and Eick [1996] presented an approach to show hot-spots of software systems. The execution of the system is monitored to determine the time needed to execute certain code lines. These values are used afterwards to color the corresponding lines. Red colored lines indicate higher execution times compared to blue lines. Our implementation takes on this approach and leverages it to components. This enables the usage of this concept on software systems of larger scale, as many lines of code are aggregated into a single component.
Chapter 9

Conclusion

This thesis introduces an approach on the live visualization of execution times of component-based software systems. Individual components of a software system are monitored, so each execution of the component is measured. These results are then aggregated to gain the individual execution times, filtered, and eventually prepared for a visualization. The visualization itself gives the user the ability to uncover certain performance aspects without heavy analysis of monitoring results.

Contributions were also made by introducing two notations toward the visualization of performance aspects. We introduced the timing behavior diagram which shows the execution of each component on timelines. With this notation the invocations of each component can be quantified and furthermore absolute timings when the invocation occurred can be given. The execution behavior diagram follows a different approach. It resembles a component diagram, as given by the UML, but reduces it to a certain set of elements and adds a further layer of information. The user is able to determine the software structure and the additional information layer adds an emphasis on certain components depending on their execution behavior. The semantic of the emphasis is not determined, therefore different aspects can be implemented.

Based on this approach, we implemented a live visualization framework in Eclipse. With this implementation, we introduced a DSL which is used to specify the component structure of a software system. Based on this DSL an automatic instrumentation of the software system can be performed by our framework. The eventual instrumentation and the collection of measurement results is performed by the Kieker framework. However, we have shown in Chapter 5 how to instrument components to derive their execution time which leads to an own implementation of records and probes which are provided to the Kieker framework. By providing interfaces to filter these execution time data and to calculate hotness values with a metric, we provide an adaptable implementation to visualize these values. With filter, certain time intervals can be observed, whereas metrics do emphasize different aspects of the performance. The final diagrams are eventually shown in a dedicated view within the Eclipse IDE.

In our evaluation we have shown the correct functionality of different parts of our implementation. In addition, we also uncovered certain aspects which cannot be covered by our approach. In summary, our implementation provides the functionality to visualize the performance of component-based software system live at their runtime. Furthermore, the implementation also works with software systems of larger scale, as shown in Section 7.1.2.
9. Conclusion

However, the monitoring of software systems comes with its cost. In our implementation we were able to detect a higher execution time of the monitored software system. Our benchmark in Section 7.1.2 shows how the framework impacts on the execution time of the monitored system in different scenarios.

Regarding future work, the approach should be extended in a way that different execution paths within a component should be possible. By now, every entry of a component must be closed by exactly one exit. A removal of this one-to-one-relationship between records would also allow different interface configurations. For instance, this would allow a component which distributes incoming element through its multiple outputs.

Beside this, another significant work would be a user study. With our notations we only propose a visualization of performance aspects which should aid a user toward debugging. A study should evaluate this statement and prove or disapprove whether this approach actually improves the debugging process.

Furthermore, certain aspects which are not covered by our approach should also be taken into consideration. Our approach only covers a flat hierarchy of components. Nested components are not intended to be monitored with our approach, but could be in future work. The connections between the components are not subject to any analysis, yet our DSL allows to model them which could be used in a future approach. Furthermore, the timing behavior diagrams lacks the functionality to show the communication paths between components. In combination with the model given by the DSL, this should be further investigated. Lastly, as the framework was implemented with a high modularity in mind, its functionality should be topic of further research. Different notations could be introduced or different program structures could be monitored. Also, different host languages could be monitored.
Bibliography


[Fittkau 2015] F. Fittkau. Live Trace Visualization for System and Program Comprehension in Large Software Landscapes. Doktorarbeit/PhD. Faculty of Engineering, Kiel University, Dec. 2015. (Cited on pages 1, 13, and 83)


Bibliography


