University of Kiel
Department of Computer Science
Software Engineering Group

Bachelor Thesis

Performance Monitoring for a Web-based Information System

Written by: Lars Kroll (`lkr@informatik.uni-kiel.de`)

June 29, 2011

Supervised by: Prof. Dr. Wilhelm Hasselbring (CAU Kiel)
Dipl.-Phys. Stefan Thiele (Thales)
Dipl.-Inf. Jan Waller (CAU Kiel)

Abstract

For the omni-present web-based information systems to be as unobtrusive as possible, when providing their services, it is necessary that they meet certain performance requirements. These could be, for example, Average Time to Answer (ASA) or Time Service Factor (TSF) metrics as specified in a Service Level Agreement (SLA). If a system has requirements on its performance, as many web-based information systems do, it will usually be necessary to verify compliance with these using performance testing. During performance tests, however, bottlenecks can only be identified when sufficient data is available. To reduce the overhead of gathering system metrics during performance testing, low-overhead monitoring frameworks like Kieker have been introduced.

In order to ensure compliance with performance requirements over the service life of a system, these monitoring frameworks can be used continuously. In that way they become a part of the software system. Therefore we argue that performance monitoring should be integrated into the development cycle.

In this thesis we evaluate four run-time monitoring approaches with regard to their usage for performance monitoring. *COMPAS JEEM*, Magpie, *the Rainbow Project* and *Kieker* are presented and analysed based on the effort of instrumentation, benefit of their data and the run-time overhead associated with using them.

Following the evaluation we present guidelines for the integration of continuous performance monitoring into software development. The guidelines are organised in a way that can be used for a large number of software development processes. They cover the most important decisions to be made while developing a continuously monitored software system. Additionally, we demonstrate and evaluate these guidelines on an example, by applying them to the Thales Product Information System (TPIS) using the Kieker framework.

Statutory Declaration

"I declare that I have developed and written the enclosed thesis entirely by myself and have not used sources or means without declaration in the text. Any thoughts or quotations which were inferred from these sources are clearly marked as such. This thesis was not submitted in the same or in a substantially similiar version, not even partially, to any other authority to achieve an academic grading and was not published elsewhere."

Kiel, June 29

Contents

1. Introduction

1.1. The Problem

In our modern, networked digital world, web-based information systems are omnipresent. Frequently we do not even realise we are using one. Often this 'obliviousness' on the user's part is exactly the goal of the system. This is especially true for mobile platforms, as Steve Jobs just recently put it in a nutshell: "It just works" [9].
For this unobtrusiveness to be possible it is crucial that such a system meets certain performance parameters. That is to say a user is likely to notice (and frown upon) a system with a very high response time. When assessing and later formulating a system's requirements such parameters are frequently considered. For example, this can happen in the scope of a Service Level Agreement (SLA), a contract defining the level of a service. When such parameters are formulated it will be necessary to design and implement a system which abides by them.
The problem is, that it is very hard to predict how a system will behave under load. Too many factors influence a software system's performance capabilities, among them the system's general architecture, the availability and quality of hardware resources and the network layout up to the end user. Furthermore, it is unrealistic to assume limitless resource availability. A web service provider is further limited in his ability to ensure performance requirements are met, in that he rarely has control over the user's environment, including the user's hardware, operating system and browser or whichever application is used to connect to the service. That means he can only tweak his system's performance on his side, but may have to account for deficiencies on the user's side.
It follows that, even with processes like Software Performance Engineering (SPE) [12], for example, the real performance capabilities (and limits) of a system emerge only under realistic load in a realistic environment. Which means in addition to verifying the software's functional requirements it is also necessary that we run tests to verify the non-functional requirements (NFRs), in this case performance tests. The very nature of performance tests though makes them impossible to be run with high-overhead profilers attached, as those would distort the results. But without detailed information on the processing of a request it is well-nigh impossible to locate bottlenecks. To monitor and record run-time behaviour of a software system, tools like Kieker [1] have been developed, which aim to induce only a very low overhead.

Still, testing a system's performance before bringing it 'live' might not suffice. The quality of the results, in terms of conclusiveness, of performance tests depends fundamentally on the assumptions about the load, which are reflected in the design of the test scenarios. To be more precise, if stress testing is not done in a production environment, which usually is not the case, then how can we be sure that actual users will behave in the way the test assumed they would? Or even if they did at first, their behaviour and usage patterns might change over time.

However, if we have a low-overhead monitoring solution available, we can just as well monitor the production system for its whole service life. This allows us to a) ensure compliance with SLAs at all times, b) detect trends in changing usage patterns, c) identify bottlenecks as they emerge and d) note and locate failures [13]. If therefore run-time performance monitoring is meant to accompany a software system for its service life, then monitoring becomes a part of the system.

Most software system development follows a systematic approach, called the software development life cycle (SDLC) or simply software development process. There are many models for such processes, probably the most well-known and easiest to understand among them the sequential waterfall model with requirements analysis, design, implementation, verification and maintenance phases. Because of these properties, and not because we think it is a recommendable process, we will use the waterfall's nomenclature in this thesis, when we speak about SDLCs.

Applying this process model to a software system, which needs continuous run-time monitoring, we believe it is necessary to integrate monitoring into the whole development cycle as well, from requirements analysis, during design and implementation to verification and maintenance.

In this thesis we evaluate four run-time monitoring approaches with regard to their usage for performance monitoring. *COMPAS JEEM*, Magpie, *the Rainbow Project* and *Kieker* are presented and analysed based on the effort of instrumentation, benefit of their data and the run-time overhead associated with using them.

Following the evaluation we present guidelines for the integration of continuous performance monitoring into software development. The guidelines are organised using the phases of the waterfall model. They cover the most important decisions to be made while developing a continuously monitored software system. Additionally, we demonstrate and evaluate these guidelines on an example, by applying them to the Thales Product Information System (TPIS) using the Kieker framework.

1.2. Structure of the Thesis

In the following section we cover basics for the thesis, explaining what we mean with *performance* and *monitoring*, as well as presenting the Thales Product Information System (TPIS). In section 3 we present four approaches to run-time performance monitoring, which have been chosen to be similar enough to be compared, yet different enough to illustrate monitoring possibilities. These are in order: *COMPAS JEEM*, *Magpie*, *Rainbow* and *Kieker*. After each approach is introduced, an evaluation based on effort of instrumentation, benefit and run-time overhead of the framework is described.

Section 4 explains the guidelines, we propose, for the integration of continuous run-time performance monitoring into the process of developing a software system. In the same section the reader will find examples, applying the guidelines to the TPIS.

In the last section we conclude the thesis, summarising our results, and proposing ideas for future work.

2. Foundations

2.1. Performance

To write about performance monitoring, it necessary to clarify what we mean, when we refer to *performance* and how we may measure it. The following definitions of performance, response time and throughput are based on [12; 7].
Performance is an indicator of the power (in the physical sense of work per time) of a software system. Thus, when we talk about a system's *high performance*, we mean that the system is able to process a large number of requests in a small amount of time.
When we measure performance, we usually do so using *response time* or *throughput*. Both concepts are closely related. While throughput measures the number of requests that can be processed in a given time frame, response time quantifies the required time to process a single request. That is the time it takes the software to generate a response to a given request.
For a whole software system we can measure both response time and throughput with different granularity: For response time we can measure anything from a single method call to, for example, the generation of a complete web page. For throughput we could look at the amount of database transactions the systems can handle or how many pages our system can serve in a given time frame.

In addition to these basic concepts we can identify two important aspects of performance: *responsiveness* and *scalability* [12]. Responsiveness describes a system's ability to meet its performance objectives. From a user's point of view it basically indicates how long a user has to wait in order to profit from the system's services. Scalability characterises the ability of a software system to take advantage of increased resources, when demand for its services rises.

All four aspects of performance need to be considered, when designing a system with performance requirements and its monitoring solution.

2.2. Monitoring

Apart from performance there are many other aspects of a software system, which can be monitored, and monitoring can take place on all layers of such a system. To give a few examples: On the hardware level we can monitor temperatures and power consumption of components. On the Operating System (OS) layer Central Processing Unit

(CPU) and memory utilisation, as well as network latency can be tracked. When surveying middleware we can keep an eye on Java Virtual Machine (JVM) status or make sure infrastructure components are available. Response times and throughput can be monitored from user interface level through application layer down to middleware. And finally on the user interface level we may observe the usage patterns of our customers, in order to improve user experience.
This thesis will mostly be concerned with run-time performance monitoring on middleware and application level. However, the Magpie approach in section 3.3 makes heavy use of OS level features.

2.3. Thales Product Information System (TPIS)

The Thales Product Information System (TPIS) is a web-based product information system that we develop for Thales Defence & Security Systems Deutschland GmbH (Thales). The system acts as a product catalogue and centralised product information hub for Thales' products. It is meant to increase the efficiency of product centric workflows by providing direct access to a plethora of product related data.

2.3.1. Background

Currently Thales has no common way to manage product data, like schematics, documentation, lists of parts and so on. Some of the data is stored digitally, some in paper form, and many things even in both forms and in multiple different systems. While Thales does have a system that manages products from a business point of view, that system does not suffice to hold the additional data required by engineers, project managers, quality assurance and other people intricately involved with the product.
The TPIS project is meant to provide them with an adaptable framework for storing, retrieving and thus sharing the necessary product-related data.

2.3.2. Domain Model

The core object of the TPIS is the *product*. Figure 1 shows an UML class diagram of the domain model. There are two types of products: *Self-produced parts* and *bought-in parts*. All products may be used as and consist of *components*. For most products there is a number of *variations* which are basically still the same product, but diverge

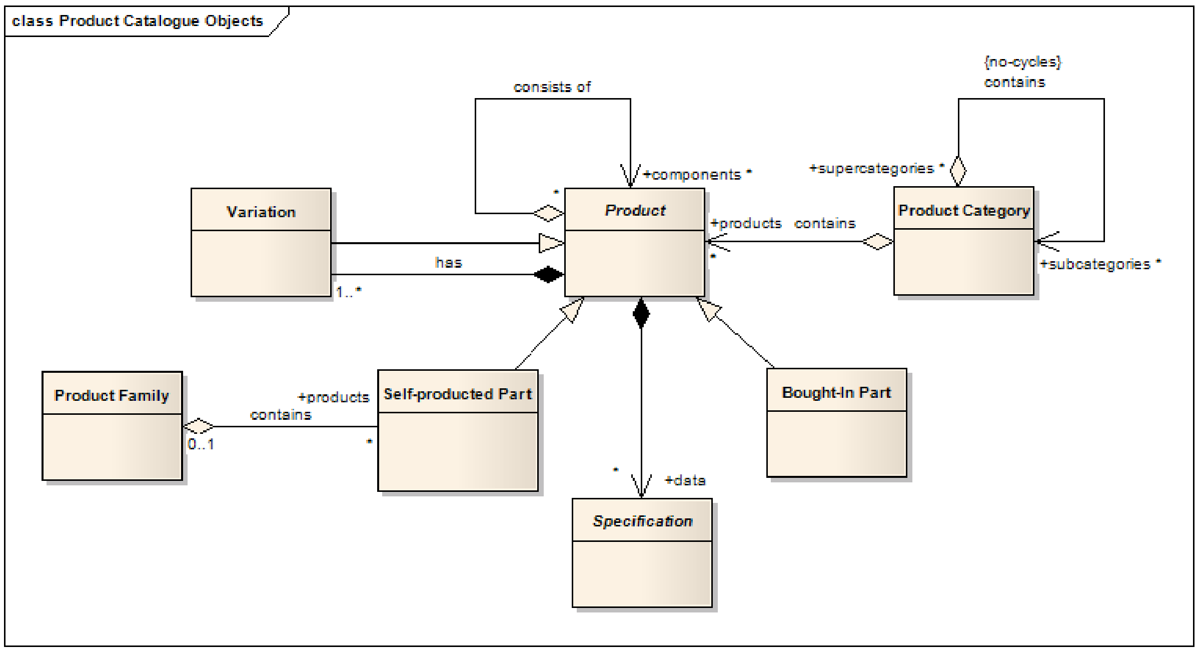

Figure 1: TPIS Product Catalogue Domain Model.

in superficial ways, like colour, weight or dimensions, for example. There always is a 'generic' variation representing the base product. Many products and their variations can be found in different *revisions* (not modeled in figure 1) in the system and thus have a revision history that can be followed.

All products are organised in *product categories*, but only self-produced parts may additionally be organised in *product families*. Products can be contained in multiple categories, as can subcategories. Product families do not have 'subfamilies' and a product may not belong to more than a single product family.

For every product there may be a (possibly large) amount of *specifications* stored by the system which may include:

- Meta-Data (e.g. dimensions, weight, comments, etc.)
- Replacement-Parts data (the parts itself are inferred from the components, but they might need meta-data such as the frequency at which they need to be replaced.)
- File attachments (images, schematics, documentation, etc.)

Metric	**Positive(+)/Negative(–) Factors**
Effort of instrumentation	+Automated instrumentation –Mixing monitoring code with business code –Constraints on system design
Benefit	+Large amount of (useful) data gathered +High quality (level of detail) of data gathered +Many possible usages for gathered data –Superfluous data gathered
Run-time overhead	+Low impact on response time –High resource utilisation

Table 1: Overview of the evaluation metrics.

3. Approaches to Performance Monitoring

In this section we present and evaluate four approaches to performance monitoring. However, the evaluation is not meant to yield a 'best approach', but to build a foundation for the necessary design decisions, when integrating performance monitoring into the SDLC.
The four approaches *COMPAS JEEM, Magpie, Rainbow* and *Kieker* have been chosen to be similar enough to be compared, yet different enough to illustrate monitoring possibilities. The basis for the evaluation are research papers and thesis' on the respective frameworks. As such scientific documentation for commercial frameworks like dynaTrace[1], JXInsight[2] or AppDynamics[3] is hardly accessible, these frameworks are not considered in this thesis. This however does not mean that they should not be considered, when pondering the framework question during system design.

3.1. Evaluation Criteria

The evaluation is based on three metrics, as table 1 shows: *Effort of instrumentation*, *benefit* and *run-time overhead*. Each of these is characterised by a number of positive and negative factors, which are motivated and explained in detail below.

[1]`http://www.dynatrace.com/en/`
[2]`http://www.jinspired.com/products/jxinsight/`
[3]`http://www.appdynamics.com`

Effort of instrumentation The first metric describes the *effort* it takes to use the approach for monitoring a new or existing application. As this is meant to be a broad evaluation and not detailed test, we will not use a quantitative approach to this (e.g. count lines of code (LOC) to be added while instrumenting). Instead we will describe how the application must be modified during instrumentation and what constraints apply for the system to be instrumented at all. Also of interest is, how much of the instrumentation work can generally be automated (for example in scripts), and how intrusive the instrumentation code is in the business code.
It is favourable for the approach if much of the instrumentation can be automated, while it should still possible to control the location of probes. Also good is if we don't have to alter the existing business code or perhaps even don't need to have it available. Additionally, we would like to be able to use the monitoring approach on as many different systems (e.g. languages, platforms, architectures, etc.) as possible.
An approach is penalised if it forces us to intertwine business code with monitoring logic, because this increases the expense of maintenance of the monitoring aspect and the business logic alike. We acknowledge that it is very hard to develop a monitoring solution that works with different programming languages, but if the framework focuses only on a single OS, allows only a certain fixed architecture or depends on special middleware, it will reflect negatively on our assessment. Furthermore, the expense of instrumenting the application should not be a significant part of the total work involved in implementing the system. Monitoring is meant to support our business goal and not to eclipse it.

Benefit The second metric we will use is the *benefit* of the instrumentation to the application. This characterises the amount and type of data gathered by the monitoring probes, and the possible applications of that data.
A good approach will likely be able to produce a large amount of different types of detailed information with a moderate number of monitoring probes. However, it should be possible to tailor the amount, type and level of detail of the gathered data specifically to our needs. It is favourable for the approach if it already provides the means to use the gathered data in different ways. As we are talking about monitoring this means providing analysis components, as well as supporting the integration of custom analysis solutions.
We want to avoid collecting data we don't need or want. It is also considered adverse

if the approach does collect very detailed data, but the data has few or no possible applications. Both types of data can be characterised as superfluous.

Run-time overhead The third metric measures the *run-time overhead* incurred by using the approach to monitor an application. This overhead is characterised by response time increase through and resource utilisation by the monitoring framework.
Clearly it is favourable for a framework not to increase the response time of the monitored application at all and to use no system resources, i.e. CPU time, main memory, network capacity and hard disc (HD) space, except for the HD space required to store the monitoring logs (if it does so). While this surely is not a realistic goal, we will prefer approaches which are closer to reaching it over those with a larger run-time impact on the monitored software system.
It is not desirable for a framework to occupy a lot of system resources, because these resources will be unavailable for the monitored service. Especially we don't want the monitoring solution to occupy resources with features we chose not to use.

3.2. COMPAS JEEM

COMPAS Java End to End Monitoring [16] is an end-to-end run-time path tracing tool which is specifically designed to be non-intrusive. Though its primary goal is path tracing the tool collects resource and performance data for analysis as well.
The COMPAS monitoring framework was developed as a PhD project by Adrian Mos [15] and basically works by injecting a proxy layer in front of the application's Enterprise Java Beans (EJBs). The proxy layer contains logic which maintains the order of calls along the different user requests. In addition to its instrumentation core COMPAS is build with a pluggable architecture, which allows the addition of new functionality using so called Framework Extension Points (FEPs).
To allow for non-intrusive instrumentation, the framework was later expanded into COMPAS JEEM by Adrian Mos, Trevor Parsons and John Murphy [16]. The authors assume a three-tier Java 2 Enterprise Edition (J2EE) architecture, with *Web Tier*, *Business Tier* and *Enterprise Information System (EIS) Tier*, as shown in figure 2, for their approach. To insert probes at different tiers of the application, COMPAS JEEM adds *monitoring filters* and *EIS tier wrapper components* to the, in COMPAS already existing, EJB proxies, in order to monitor web and EIS tiers respectively. The monitoring

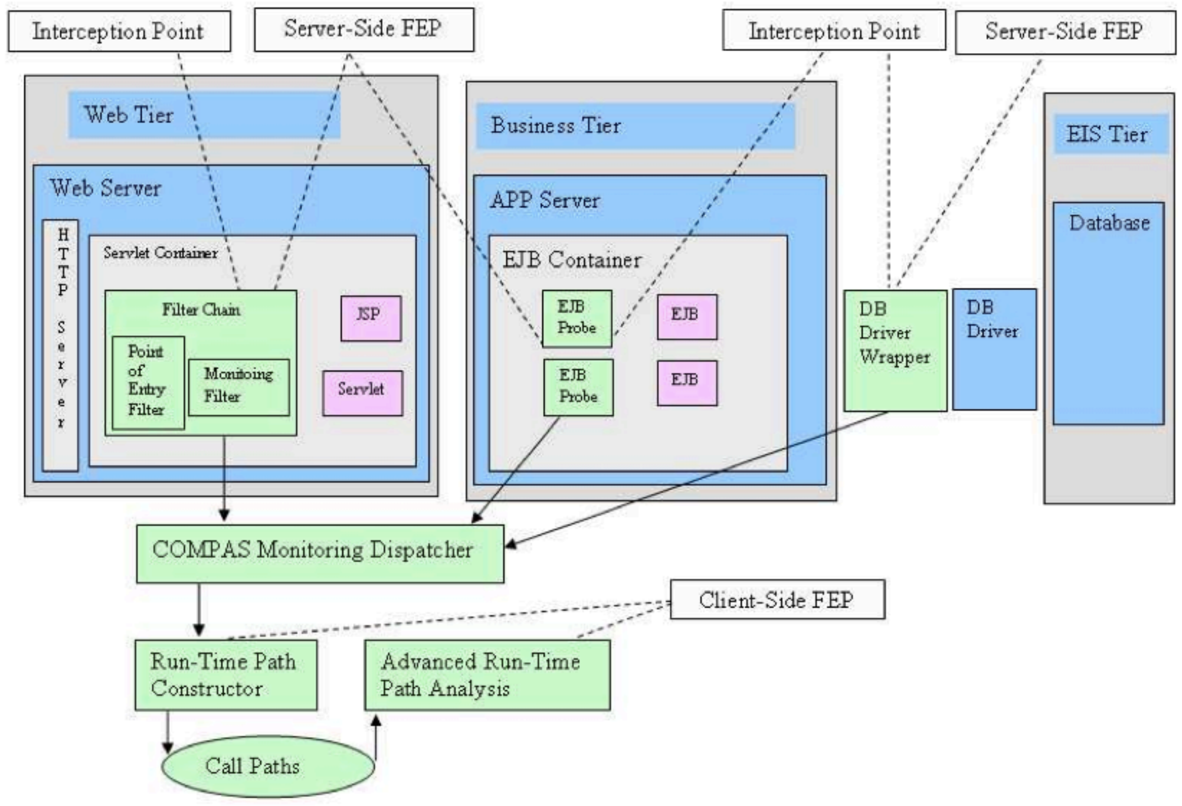

Figure 2: COMPAS JEEM architecture [16, fig. 3]

filters intercept servlet calls and provide point of entry information for traces trough the tiers. The EIS tier wrappers are actually database driver wrappers, and thus intercept database calls at a really low level. As figure 2 shows all types of monitoring probes provide their gathered data to the *COMPAS Monitoring Dispatcher*, from where it is available for analysis.
COMPAS JEEM uses the Extensible Markup Language (XML) deployment descriptors, which are mandated by the J2EE specification, to automatically generate and insert the proxy layers explained above into a packaged J2EE application. In this way the approach can even be used for components if no source code is available for them.

Our evaluation of COMPAS JEEM is based on information provided by Adrian Mos, Trevor Parsons and John Murphy themselves in [16].

Effort of instrumentation When it comes to instrumentation effort COMPAS JEEM really stands out by being able to automatically instrument a packaged J2EE application using a simple script. The same mechanism used for automated instrumentation, however, imposes a serious constraint on the type of applications that can be monitored with COMPAS JEEM: It can only instrument J2EE components in this way, as it relies on their XML deployment descriptors for meta-data. To monitor applications with different deployment mechanisms, the developer has to fall back on manual instrumentation. Furthermore, the control the developer has over the locations of the monitoring probes is very coarse if automated instrumentation is used.
As the COMPAS framework primarily uses wrappers, proxies and interceptor filters for monitoring, the intrusion of monitoring concerns into business code is minor. If the automated instrumentation mechanism is used, no code needs to be changed at all. If the developer wishes or needs to manually instrument the code, the changes are confined to setup code, i.e. the parts of source code which are tasked with creating and connecting classes and components. This extends to configuration files if part of the setup is based on such.

Benefit The data gathered by the COMPAS framework is very detailed and includes the type/tier of the entry (for example `URL`, `EJB` or `DB`), the component/class on which a method is called and the name of the method called as well as the execution time of the method. Additionally, COMPAS gathers user request IDs, sequence IDs and call depth to be able to reconstruct paths. Depending on the monitoring goal, we argue that the data is sometimes too detailed. For example, while it is surely of interest how long a prepared statement takes to execute on the database, it might well be of minor concern, how long every single `next` and `getString` call on a `ResultSet` takes.
As the examples in [16] show, the gathered data can be used to generate Unified Modeling Language (UML) sequence and component diagrams. However, the authors do not note if and in what way this is automated, or which tools were used. But as the gathered data contains component identification and response times, it is definitely sufficient to assess application performance and determine if SLAs are fulfilled, for example.

Run-time overhead The performance overhead induced by COMPAS JEEM has been measured by the authors to be very low using the PlantsByWebsphere [4] application as an example. They employed three machines for JMeter, WebSphere and logging COMPAS' data respectively (details can be found in [16]). The same load tests were run with and without COMPAS JEEM instrumentation and measured the averaged response times for 20, 50, 100, 150 and 200 concurrent users' load. In their experiment the authors were unable to show any measurable overhead for 20, 50 and 100 users. However, at 150 users the server approached its saturation point and average response times were 19% longer for the instrumented version. The results of the test with 200 users had to be discarded, as both application versions started throwing enormous amounts of exceptions at this level and the results thus became useless.

3.3. Magpie

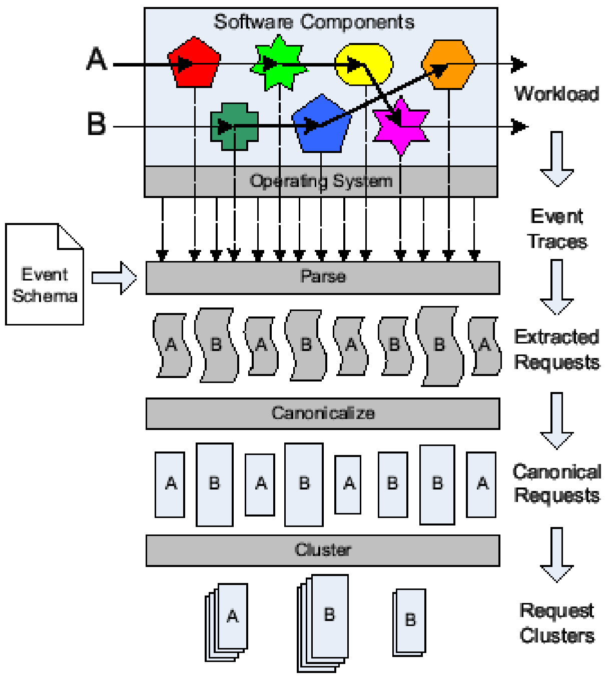

Figure 3: The Magpie toolchain [3; 4, p. 259].

Magpie [3; 4] is an online performance modelling infrastructure. Magpie maintains detailed logs of resource usage and combines them in real time to provide per-request

[4]`http://www.ibm.com/developerworks/websphere/library/samples/plantsby.html` – It is likely though that the authors used an older version than that one.

audit trails. It uses events and timestamps to trace in-kernel activity, Remote Procedure Calls (RPCs), system calls, and network communication, using Event Tracing for Windows [14], the .NET Profiling API [14], Detours [8] and `tcpdump`[5] respectively. To track requests over multiple physical hosts, Magpie uses synchronisation events on the Transmission Control Protocol (TCP) layer. The events are then combined by running the event parser for every machine used and subsequently connecting the trace on the basis of said synchronisation events.
The largest part of Magpie is actually comprised of its facilities for online and offline event parsing to create workload models. This can be seen in figure 3: Only the topmost layer of the Magpie toolchain is concerned with monitoring, while the other stages prepare the workload model for analysis. However, this thesis is focused on performance monitoring and the analysis of monitoring data would go beyond its intended scope.

Our evaluation of Magpie is based on the information in [4].

Effort Magpie makes extensive use of pre-existing Event Tracing for Windows (ETW) event providers, which are mostly low-level system libraries. While it may or may not be difficult to set these up, the effort of instrumenting the code itself is small. The authors do note, that they implemented their own event tracing, where none existed, but they are not very specific about those custom tracing probes. We assume though that the effort of implementing one's own event tracing is not negligible. It is also not clearly shown how intrusive their instrumentation is, but as most ETW they use is on the OS level and not the application level, it is likely that the separation of the business and monitoring concerns is quite good.
Because Magpie uses many low-level OS and .NET features, it can only be used for Windows .NET applications.

Benefits Magpie produces very detailed information, including not only application call traces and response times, but also detailed resource usage per request, in terms of CPU cycles as well as HD and network Input/Output (I/O). This multitude of information is stored in event logs though and the process of reconstructing request traces is not trivial, which accounts for the performance impact of the parser.
Because the collected data is so detailed, it can be used for a variety of purposes. As

[5]`http://www.tcpdump.org/`

was mentioned before, the goal of Magpie's toolchain is a workload model. In [4] this model is used for anomaly detection. But it could just as well be used to generate visualisations of the static or dynamic aspects of the system, for example. For more possible applications of Magpie's data compare section 4.1.

Run-time overhead Because Magpie depends on many low-level Windows features, a direct comparison between Magpie and the other approaches is somewhat difficult when it comes to overhead. The low-level features give Magpie a performance advantage on Windows, but as the other approaches' performance impact has mostly been measured on *nix systems (Unix, Linux, Solaris or similar) the comparison is weak.
Even so it can be noted that, according to a superficially described stress-test experiment in [4], the performance impact of Magpie's monitoring facilities is negligible. However, the performance overhead of the parser in online mode, i.e. running in parallel to the monitored application, and on the same machine is not, beause the parser reduces the computing power available to the application itself.

3.4. Rainbow

The Rainbow framework [5] is an architecture based self-adaption framework. It is meant to provide general and reusable infrastructures for self-adaption that can be tailored to particular system styles and quality objectives, and further customized to specific systems. We are, however, only interested in its monitoring facilities.
The Rainbow framework's monitoring facilities, like the rest of the system, are built upon a pluggable architecture approach, which makes the framework very flexible. Monitoring is done in two layers, which can be seen in the lower half of figure 4: Integrated into the target application and possibly other related system tools, are monitoring probes which collect data and make it available to so called *gauges* over the *ProbeBus*. The gauges then aggregate the provided monitoring data and interpret it, in order to finally decide which changes to the model state need to be made. Subsequently, they propagate these changes over the *GaugeBus* to the *ModelManager*.
The framework is written in Java, but due to its *Translation Infrastructure*, which "decouples and provides a bridge between the framework Architecture Layer and the platform and implementation of the System Layer"[5, p. 76], and its use of event middleware it is possible to use the framework for systems written in other languages or even heteroge-

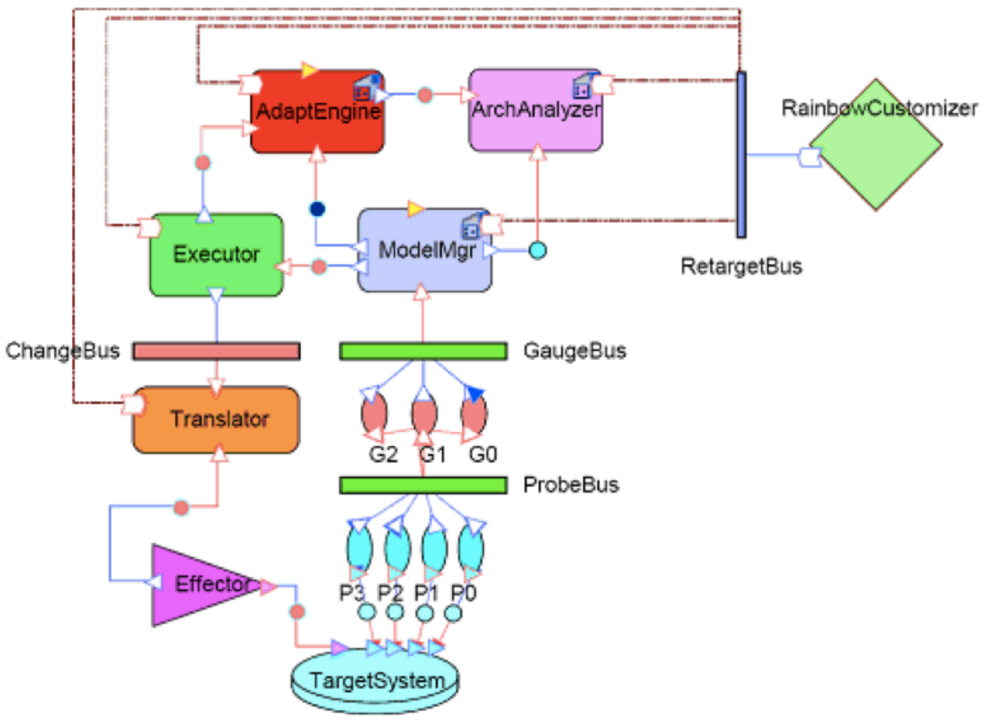

Figure 4: Rainbow architectural diagram [5, p. 69].

nous systems with different platforms and languages.

Our evaluation of the Rainbow framework is based on data from the original doctoral thesis by Shang-Wen Cheng [5].

Effort The Rainbow project is a very general framework and there are no obvious constraints, it places on the design of the target application. It is likely that many probes for a custom system need to be written by hand. But if the same technology is reused over multiple projects, the probes can be reused as well. In this case the intrusiveness of the instrumentation depends on the chosen design for the probes. Actually, it is even possible to use existing monitoring solutions to report to the gauges, as long as they can be adapted to adhere to the *Probe Protocol* on page 77 in [5].

Benefits The level of detail provided by the reported data depends on the number and types of deployed probes. It is possible to monitor anything from resource usage and

network latency to response time or number of active sessions. However, the system is not designed to reconstruct or work with complete traces. But adaptable as it is, this could likely be implemented as well.
It is the primary goal of the Rainbow project to provide run-time self-adaptation capabilities to the target system. This is not limited to run-time reconfiguration, though. Depending on the strategies defined, the Rainbow framework could also modify application internal variables like cache sizes for certain services or mitigate a Denial of Service (DoS) attack by automatically blacklisting certain Internet Protocol (IP) addresses, for example.

Run-time overhead Cheng does not report any data about the performance overhead incurred by simple monitoring, as opposed to running the whole adaption framework. In fact, according to page 110 of [5], he used the application version that was only monitored as control group in his test. The monitoring overhead probably depends primarily on the monitoring facilities used and it is not clear, what he used for his test. Cheng notes though that the local delegate of the framework's core requires between two and ten percent of the CPU on every node. However, the probes on nodes with tight resources could instead use the delegates of neighbouring nodes, according to Chen.

3.5. Kieker

Kieker [23] is a monitoring and analysis framework for Java applications. It can be used for monitoring software run-time behavior, like internal performance or (distributed) trace data. Kieker consists of two major components [6]: `Kieker.Monitoring` is tasked with gathering monitoring data and logging it, whereas `Kieker.Analysis` analyses monitoring data. Between these components resides the *Monitoring Log*, which connects them.

Kieker allows for a number of different types of monitoring probes. Among them are Aspect Oriented Programming (AOP) based probes as well as Enterprise Java (JavaEE) Servlet interceptors. Additionally, it is possible to write new probe implementations for different middleware. All probes submit *Monitoring Records* to the `MonitoringController`. From there the data is written into the Monitoring Log using one of several available *Monitoring Log Writers* as configured during deployment. At the time of writing Kieker comes with support for synchronous and asynchronous writers

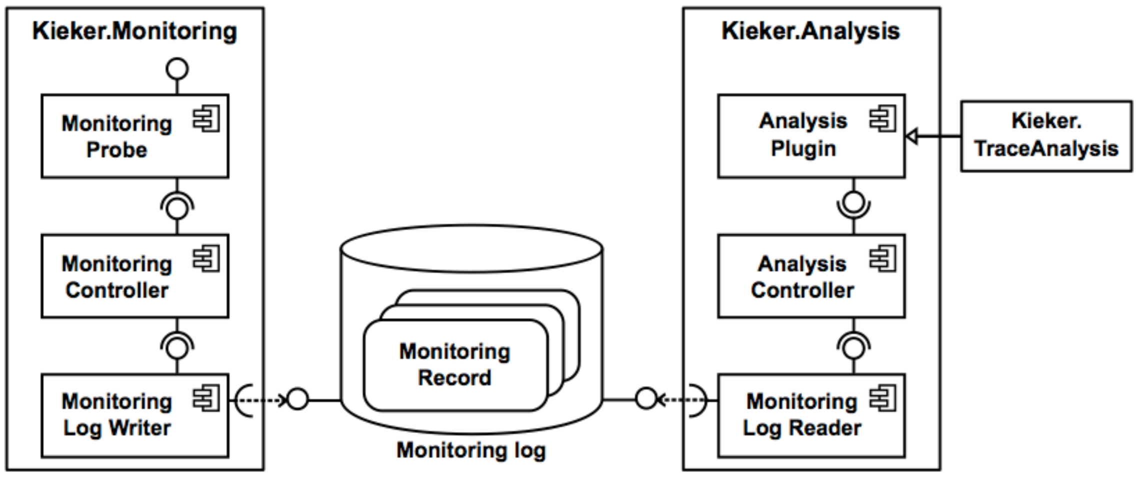

Figure 5: Kieker architecture overview [6, p. 3]

that log data either to the filesystem or into a database, as well as an asynchronous Java Message Service (JMS) writer. As the name implies the asynchronous writers decouple the I/O operations from the control flow of the instrumented application[6].
On the analysis side of Kieker corresponding *Monitoring Log Readers* retrieve the monitoring data from the Monitoring Log and *Analysis Plugins* interpret the data. This process is implemented in an easily extensible producer-consumer scheme. Included in the Kieker distribution is the `Kieker.TraceAnalysis` tool, an Analysis Plugin, which can be used to reconstruct and visualise architectural models of the monitored systems. Some possible visualisations are UML sequence diagrams and dependency graphs.

Our evaluation of Kieker is based on the publication by van Horn et al. [23] and the Kieker User Guide [6].

Effort Depending on the desired type and granularity of the monitoring data, it can be anything from very easy to quite time-intensive to instrument an application for Kieker monitoring. The easiest, least intrusive way to instrument, for example, a web application is using AOP based probes and servlet interceptors. This requires only the configuration of the weaver, i.e. `aop.xml` for AspectJ[20] Load-Time Weaving (LTW), and the addition of the filter in `web.xml`. More elaborate monitoring requirements though might introduce the necessity to implement custom monitoring probes or even

custom monitoring records. As the example above demonstrates, the instrumentation of business code for Kieker is not very intrusive. A few entries into config files and a few method annotations are usually needed, at most.
At this time, however, only Java applications can be monitored with the Kieker framework. As stated in the kieker user guide[6], an expansion to .NET is planned.

Benefit Kieker allows for the collection different types of data using different monitoring records. The example above considers only *Operation Execution Records*, which collect data about the execution time of a method and allow for trace reconstruction. They can also be used to trace calls over different execution environments. Other monitoring records are used to log resource consumption or network latencies, for example. The data collected by Kieker can be used to reconstruct and visualise static and dynamic aspects of a software system. In addition to the sequence diagrams and dependency graphs mentioned above, Kieker's TraceAnalysis tool can also generate *Call Trees* on deployment and assembly level [6].
Furthermore, N. Marwede et al. present an approach for automatic failure diagnosis [13] based on data provided by Kieker. Kieker can also be used as the monitoring part of the SLAstic self-adaption framework [22].

Run-time overhead In [23] the authors show that Kieker's monitoring overhead is very low and increases with the number of probes in a linear fashion. To test this they used a single Java class with two monitored methods, one of them a recursive method with variable depth, instrumented using AspectJ AOP based probes. The test was run on a powerful server machine with Solaris 10 and Java 64-Bit Server VM. Both monitored methods were tested in separate experiments with four test runs each. The test runs had different configurations measuring different sources for overhead incrementally. In the experiment roughly under $2.5\mu s$ mean overhead in response time of the non-recursive method was encountered, when monitored and logged with the asynchronous filesystem writer.

3.6. Conclusion

During the evaluation it has turned out that, while all the presented tools serve a similar purpose, the means by and the degree to which they achieve their goal show substantial differences. The gathering of monitoring data for example is done in many different ways, using proxies, AOP based instrumentation or low-level OS features like ETW. The collected data differs as well, providing execution times, resource usage, network latencies or call trace information amongst others. However, all tools share their desire to impose only a small overhead on the monitored application with their monitoring facilities (the analysis components usually have higher demands).

In the next section, we will present our guidelines for the integration of continuous performance monitoring into the SDLC and evaluate them at the example of the TPIS. We will motivate our choice of framework for that system here:
TPIS does not conform to the J2EE standard and has no EJBs or XML deployment descriptor. Therefore COMPAS JEEM's greatest advantage, the automated instrumentation, is not available for it. Furthermore, the very detailed database monitoring seems inappropriate, especially as MyBatis[2] is used for object relational mapping (ORM). Because TPIS is written in Java and supposed to run on a linux machine Magpie is no option. Rainbow would be possible, but it is focused on self-adaption, which is not a goal of TPIS' monitoring. Additionally, Rainbow is too large a framework when we would just use such a small part of it.
Kieker's monitoring facilities however align very well with our goals. The standard *Operation Execution Record* provides data for call tracing and response time monitoring, which is sufficient to monitor usage and basic performance. Furthermore AOP based probes add little additional expense, because AOP is already being used for security features and transaction management. Therefore we are using Kieker in the Thales Product Information System (TPIS).

Phase	Actions
Requirements analysis	Define monitoring goals.
	Define acceptable margins for monitoring overhead.
Design	Decide on a monitoring framework.
	Select probes.
	Choose probe locations.
	Define methods and thus intrusiveness of the instrumentation.
	Decide on physical location of monitoring log.
Implementation	Implement missing probes if necessary.
	Instrument code.
Verification	Test instrumentation.
	Include data from instrumentation in other tests.
Maintenance	Use monitoring data according to defined monitoring goals.
	Adapt implementation to achieve monitoring goals (if necessary)

Table 2: Performance monitoring and the software development process - summary.

4. Integrating Performance Monitoring into Software Development

We will now present our approach to integrate run-time performance monitoring into the SDLC. Because there is no such thing as a single 'software development process', we try to give a general guideline for integration that is agnostic of the actual process used. As we have already motivated in the introduction, we will use the phases of the waterfall model to organise our approach. These phases are again *requirements analysis, design, implementation, verification* and *maintenance*, in order. They can be iterated or broken up in the actual process but to some extend they will always be part of software development.

The presented guidelines concern themselves primarily with the collection of monitoring data. Its analysis and application is a very wide field and, while we will give some examples, detailing analysis concerns goes beyond the scope of this thesis.

The summary of the proposed development process for monitored software is presented in table 2 and the next sections explain every action in detail and give examples from the TPIS project presented in section 2.3.

4.1. Requirements Analysis

The requirements analysis phase is the place to define the scope of monitoring. Here we should address the questions

1) 'Do we want monitoring of our software system?',
2) 'Why do we want monitoring?' and
3) 'How much overhead are we willing to accept to gather monitoring data?'.

While the first question is basically a simple 'yes' or 'no' question, it is by no means trivial. Indeed it is very likely that we need to answer question number two first. That is, if we do not have a monitoring goal then it is likely that we do not need monitoring. Most people will not collect data just for the sake of collecting data.
To answer the second question, we need to know what we can possibly do with monitoring data. There are quite a number possible goals of monitoring, a few of which are listed below:

(Distributed) Performance Debugging [3] When we collect detailed performance data, the most obvious application is to use the data to identify bugs or bottlenecks in the system. This is even more of interest when dealing with distributed systems, as distributed traces allow a unique view of the dynamic aspects of the systems.

Capacity Planning [3] With detailed resource usage and trace data it is possible to create workload models for a software system. These models can then be used to predict system performance.

Tracking Workload Level Shifts [3] Over time the behaviour and usage patterns of the application's users might change. This change is rarely sudden and can be anticipated by comparing differences in the system's workload at different times. Of course, it is necessary to differentiate between real trends and normal fluctuation. While Barham et al. [3] propose that even some types of DoS attacks can be detected in this way, Marwede et al. present an approach for doing so [13].

(Automated) Component Failure Diagnosis Sometimes, software or hardware components fail in non-obvious ways, leading to unaccountable behaviour of the system. Given monitoring data with sufficient detail, the problem can be pinpointed and remedied [3]. In [13] an architecture-centric approach is shown to automate failure diagnosis for distributed large-scale systems.

'Bayesian Watchdogs' [3] It is possible, when a probabilistic model of normal request behaviour is available, to form an estimate about how likely a certain request that moves through the system is. If such a request is deemed unlikely and thus suspicious, it can be redirected into a safe sandboxed area.

Monitoring SLAs [3] The monitoring data acquired at service level can be used to verify that the provided services comply with their respective SLAs. As Barham et al. mention, we could not only measure our own compliance, but also that of services we use ourselves.

Self-Adaptive Software Systems Self-managed or autonomic systems are those systems which are able to adjust their operation in reaction to changing demands, components and workloads as well as failures [10]. In the *observation phase*[17] of every self-adaptive software system there is a monitoring component, which gathers the data for analysis.

If we have decided that we wish to pursue one or more of the monitoring goals presented above, we have effectively answered question one and two.
It remains to decide how much overhead we are willing to accept, regarding application performance and size, and how much storage space we are willing to sacrifice for the monitoring logs. In most cases the choice of monitoring goals will already set a lower bound on acceptable overhead, and it is all that is left to demarcate a range, in which the software designer may choose appropriate tools and means to reach the selected monitoring goals.
It is probably not a good idea to set hard limits here, although this behaviour is usually recommended for the documentation of requirements. However, for monitoring the size of the logs is directly proportional to the frequency of usage and limiting log size means either discarding valuable monitoring information or limiting the usage of the application. Neither of which is desirable. Furthermore, the induced overhead depends on the type and number of instrumentation points as well as many deployment and implementation parameters. A hard limit may be difficult to adhere to.

Example. In the TPIS Software Requirements Specification (SRS) the following monitoring requirements were documented (section 2.1.9).

> "The system shall support continuous run-time performance monitoring of (most of) its components. Monitoring of the software system has two goals:
>
> 1) The monitoring shall support debugging and the detection and localisation of performance bottlenecks.
> 2) Usage patterns shall be collected to survey acceptance of the tool and identify possible optimisation points.
>
> To achieve these goals a small performance overhead will be acceptable as well as the storage of a few gigabytes of monitoring logs."

In our experience, the guidelines fit very well to this phase. It is straight-forward to ask a client the three questions, which govern the decisions of this phase, and provide the possible monitoring goals as examples to stimulate a discussion. However, while concepts like monitoring of SLAs is usually quite easy to grasp for a client, more advanced concepts like 'bayesian watchdogs', for example, may pose quite a challenge, depending on the client's background.

4.2. Design

After we have laid out the basic parameters and goals for monitoring, we have to make a number of important decisions for monitoring while designing our system.

Monitoring Framework The monitoring framework provides the tools for monitoring. These include different types of monitoring probes, different models for storing monitoring logs and often basic support for the analysis of monitoring data.
In section 3 we have presented four approaches to monitoring with corresponding frameworks, but there are many more frameworks available, commercial as well as open source. As frameworks come with different constraints, the choice is heavily influenced by design decisions about implementation language(s) and deployment platform(s).

Selection of Monitoring Probes A monitoring probe is like a sensor collecting and possibly pre-processing data from within the software system [21]. The collected data varies from probe type to probe type and may include response times, trace information, resource usage or latencies, for example.
The choice of probes is dependent on the chosen monitoring framework as well as driven

by the data necessary to achieve the monitoring goal. Additionally, it can sometimes be inevitable to implement custom probes in order to achieve the monitoring goal, if the framework supports this.

Location and Quantity of Monitoring Probes [21] Another important and often difficult decision concerns the number and exact position of the selected monitoring probes. It is clear that more monitoring probes result in increased monitoring overhead and in larger monitoring logs. Thus, it is necessary to find a compromise between the level of detail of monitoring data and the aforementioned overhead. In order to find that working middle-ground, it will often be advantageous to identify the key locations concerning the monitoring goal inside the code. Furthermore, different usage scenarios of the application should be considered, as most of the time an equally distributed coverage of the system with monitoring probes is desirable [21].

Method/Intrusiveness of Instrumentation As pointed out by van Hoorn et al. [21], the method of integrating monitoring logic into the business logic has a major impact on the maintainability of application-level monitoring. In addition to making source code harder to read, changes in the monitoring approach are harder to perform if the monitoring code is mixed with the source code of the business logic.
Clearly, monitoring is an aspect as defined by Kiczales et al. [11]. Therefore one possible and extremely suitable method of integrating monitoring probes into the business code is AOP[11]. In section 3 multiple instrumentation techniques have been mentioned while presenting the approaches, and that is not an exhaustive list. Possible techniques include AspectJ[20] and Spring[18] AOP based probes, proxy classes, Java Servlet API filters and ETW[14].

Physical Location of the Monitoring Log Usually, the collected monitoring data is written to a so-called monitoring log [21]. Analysis of the data can be done *online*, i.e. while the system is actively writing to the log, or *offline*, after an old log has been copied to an archive location, for example. Depending on the timeliness of the analysis required by the monitoring goal, the monitoring log can be a simple file, a database or a messaging middleware. It can even be a custom aggregation component, as is the case for the Rainbow project (cmp. section 3.4). Important factors to be included in the decision about the location of the monitoring log are the amount of monitoring data

generated, the required timeliness of its analysis and possibly restricted access rights or policies [21].
Local file system storage is fast, because no network communication is required and there is no Database Management System (DBMS) overhead. However, if it is to be used in a distributed setting, a common network filesystem for all components would be necessary, which again includes the drawback of network communication. Using a single database for monitoring record storage allows for centralised analysis support, such as convenient queries, but comes at the cost of performance. The overhead of messaging middleware depends on the framework used and is difficult to assess generally.
It is important to note though that monitoring data should not be written into the log synchronously with the executing business logic, as that may have a considerable impact on application performance [21].

Model-Driven Instrumentation In addition to making the necessary decisions presented above, it will often be necessary to document them. In [21] van Hoorn et al. propose Model-driven Software Development (MDSD)[19] as a means to do both. They state that MDSD provides a convenient means for lifting the abstraction level for the instrumentation from source code to architecture or business level. It is further proposed to annotate static or dynamic models with monitoring information in order to integrate its transformation into the platform-specific instrumentation into the MDSD process. The annotation could be done through tagging, i.e. adding the instrumentation to the functional part of the model, and decoration, that is the definition of a dedicated monitoring model which refers to the architectural entities captured in external models.

Example In the TPIS Software Design Documentation (SDD) monitoring was already integrated into the architecture as seen in figure 6 , which is described in details in there, and the following monitoring requirements were documented (section 2.3.5).

> "To locate performance bottlenecks and collect usage data Kieker *Monitoring Probes* will be placed at the entrance points (communication interfaces) of every layer. In case of presentation this means at servlet container boundaries **and** SpringMVC controllers, as the user's browser can not easily be probed. In the storage tier only the DAOs will be instrumented as the performance of the *Attachment Manager* can and must be measured by other

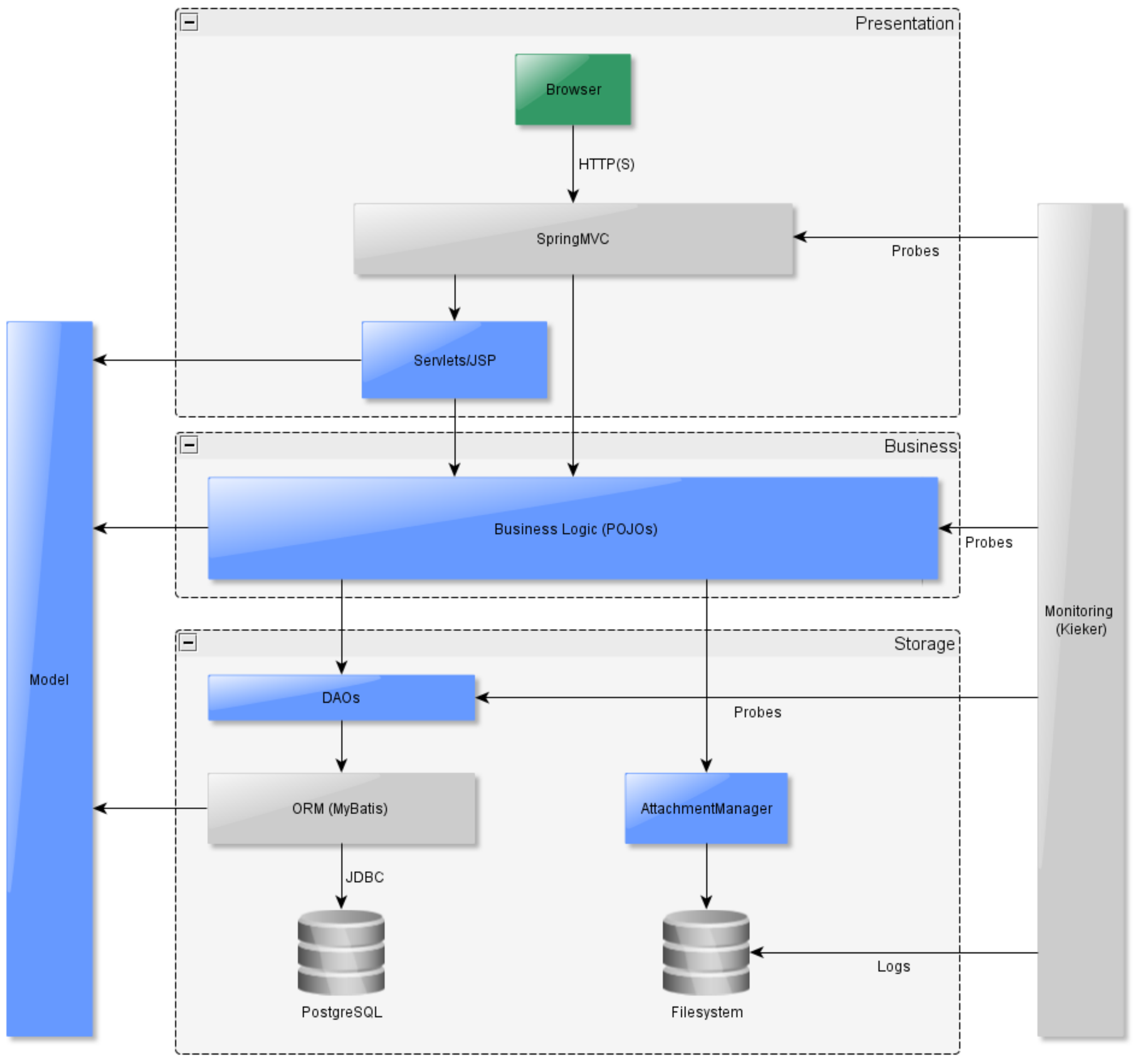

Figure 6: TPIS architecture overview.

means (because it is a separate non-Java component).
As instrumentation method AOP and *Servlet Filter* based probes will be used. The *Monitoring Log* will be stored in the filesystem, in Tomcat's `tmp` directory and copied to an archive location during nightly backups for offline analysis. The collected data will be response time and trace data, using Kieker's record type `OperationExecutionRecord`."

The guidelines stated above also fit well into the software design process. The designer needs to account for the contents of every paragraph mentioned above and make the proposed decisions. He should do this while keeping his overall architectural goal in mind and at the same time consult the framework's manual to discover possible integration points. The choice of framework is usually straight-forward when the monitoring goals and much of the architecture are already known, as can be seen in our motivation at the end of section 3.6.

4.3. Implementation

The implementation of monitoring aspects is very much unique to the chosen monitoring framework (or custom solution) and the monitored application, and few things can generally be said about it. However, the instrumentation should happen as early as possible during implementation, because in most cases the monitoring data can already be used for debugging purposes.
Should custom probes be necessary, these should also be developed while writing the code which they will be used to instrument (if they instrument code which is developed as part of the project). This procedure allows for quick adaption if complications should arise and changes become necessary.

Example As TPIS is using Kieker for monitoring, a Kieker-specific AspectJ AOP based instrumentation is shown. Listing 1 shows an AspectJ configuration file which defines that methods of classes in packages below `com.thalesgroup.tpis` which are annoted with `@OperationExecutionMonitoringProbe` are to be monitored. For debugging purposes it also instructs AspectJ to print weaving information to the log. In listing 2 an excerpt from the class `OrganisationService`, whose `getCategoryProducts` method matches the pointcut defined in listing 1, is shown.

In TPIS we have implemented monitoring at the earliest possible stage, namely while building and configuring our application to pass a simple test call through all layers (as seen in figure 6). We were then able to use the monitoring data, that is its existence or lack thereof, to assess whether the LTW configuration worked or not. To be able to rule out Kieker misconfiguration as a cause for failure, we had previously written monitoring code manually to test the Kieker configuration.

As the TPIS project has not yet progressed beyond its implementation stage, we are unable to provide further examples or evaluate our guidelines for the following phases.

Listing 1: TPIS aop.xml example.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN" "http://www.eclipse.org/
    aspectj/dtd/aspectj.dtd">
<aspectj>
 <weaver options="-verbose -showWeaveInfo">
  <!-- instrument the all tpis packages -->
  <include within="com.thalesgroup.tpis..*" />
 </weaver>
 <aspects>
  <aspect name="kieker.monitoring.probe.aspectJ.operationExecution.
      OperationExecutionAspectAnnotationServlet"/>
 </aspects>
</aspectj>
```

4.4. Verification

During testing it is important to verify that monitoring works as intended and all probes are being deployed and integrated correctly. This is especially important when weaving with AspectJ, as it can easily happen that defined point-cuts do not match all desired locations, for example. With the exception of testing custom probes, however, unit testing does not work well for monitoring, which depends on too much context to work properly. Thus, it is likely that most testing of the monitoring capabilities will occur during integration test, when the system is properly deployed.
As already noted in the last section, it is advantageous if monitoring works as early as possible, because we can use the resulting monitoring data during integration tests to identify and locate bugs.

Listing 2: TPIS OrganisationService excerpt method annotation example.

```java
package com.thalesgroup.tpis.service;

import kieker.monitoring.annotation.OperationExecutionMonitoringProbe;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

@Service("organisationService")
public class OrganisationService {

	@Autowired
	private OrganisationDAO organisationDAO;

	@OperationExecutionMonitoringProbe
	public List<Product> getCategoryProducts(int id) {
		return organisationDAO.getCategoryProducts(id);
	}
}
```

4.5. Maintenance

When the system is in production use, the efforts to integrate monitoring finally pay off, and the collected data can be used to the ends defined when choosing the monitoring goal.

But as the requirements for a software system are often changing over time, so may the monitoring goals. In addition to keeping the monitoring solution in line with changes to the business code, it might also become necessary to change monitoring strategy and add or remove probes, change the location of the monitoring log or revise other previous decisions. The concerns and propositions presented above should be kept in mind when adapting the monitoring solution to altered requirements.

5. Conclusions

In this thesis we presented and evaluated four approaches to run-time performance monitoring: *COMPAS JEEM*, *Magpie*, *Rainbow* and *Kieker*. While these approaches show considerable differences, there are also some constants among them. For example, the common use of some kind of monitoring probes to gather monitoring data. Also notable is that none of these frameworks is a pure performance monitoring solution. They all serve different purposes as well and are able to collect different types of data.
We were, however, not able to include any commercial frameworks into our analysis. A similar evaluation approach for some of these could be the topic of a future project, as a means to reach a better foundation for the framework decision. In order to be able to accurately compare different solutions, it would also be desirable to test them under similar conditions, as far as their inherent differences allow for.

Furthermore, we presented a set of guidelines for the integration of continuous run-time performance monitoring into the process of developing a software system. To do that we collected a list of concerns, we believe need to be addressed. This list is not exhaustive but it is a starting point. It turned out to be difficult to make good recommendations for the implementation, verification and maintenance phases. One of the reasons for this is the fact that these three phases are very much dependent on the system to be developed and the chosen monitoring framework as well as the monitoring goals. Additionally, Thales Product Information System (TPIS) had not grown beyond implementation at the time of writing, so needed experience was still missing. Perhaps a later project can address these points in a more detailed way.
The experience collected in the TPIS project, however, suggests that at least for requirements analysis and design phases the guidelines provide a sound and easily applicable basis. It still remains to be seen though if they fit different SDLCs as well as the waterfall model used in this thesis and the V-model used for the TPIS project.

References

[1] Kieker, 2011. URL `http://kieker.sourceforge.net`.

[2] MyBatis, March 2011. URL `http://www.mybatis.org/`. Version 3.

[3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online modelling and performance-aware systems. In M. B. Jones, editor, *HotOS*, pages 85–90. USENIX, 2003. ISBN 1-931971-17-X.

[4] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *OSDI*, pages 259–272, 2004.

[5] S.-W. Cheng. *Rainbow: Cost-Effective Software Architecture-Based Self-Adaptation*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 2008.

[6] N. Ehmke, A. van Hoorn, and R. Jung. *Kieker 1.2 User Guide*. Christian Albrechts University of Kiel, September 2008.

[7] E. Gyamarti and P. Stråkendal. Software performance prediction using spe. Master's thesis, Blekinge Institute of Technology, 372 25 Ronneby Sweden, June 2002.

[8] G. Hunt and D. Brubacher. Detours: binary interception of win32 functions. In *Proceedings of the 3rd conference on USENIX Windows NT Symposium - Volume 3*, pages 14–14, Berkeley, CA, USA, 1999. USENIX Association. URL `http://portal.acm.org/citation.cfm?id=1268427.1268441`.

[9] S. Jobs. What is Apple iCloud? WWDC Presentation, June 2011.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, pages 41–50, Januar 2003.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *LNCS*, pages 220–242. Springer, 1997.

[12] C. Lloyd G. Williams. *Performance solutions : a practical guide to creating responsive, scalable software*. Addison-Wesley, 2002.

[13] N. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis support in distributed large-scale software systems based on timing behavior anomaly correlation. In A. Winter, R. Ferenc, and J. Knodel, editors, *CSMR*, pages 47–58. IEEE, 2009.

[14] Microsoft Corp. ETW Tracing for .Net, 2011. URL `http://msdn.microsoft.com/en-us/library/ms751538.aspx`.

[15] A. Mos. *A Framework for Adaptive Monitoring and Performance Management of Component-Based Enterprise Applications.* PhD thesis, Dublin City University, 2004.

[16] T. Parsons, A. Mos, and J. Murphy. Non-intrusive end to end run-time path tracing for J2EE systems. In *IEE Proceedings-Software*, 2006.

[17] M. Rohr, S. Giesecke, W. Hasselbring, M. Hiel, W.-J. van den Heuvel, and H. Weigland. A classification scheme for self-adaptation research. In *Proceedings of the International Conference on Self-Organization and Autonomous Systems In Computing and Communications*, SOAS'2006, page 5, September 2006.

[18] SpringSource. Spring Framework, February 2011. URL `http://www.springsource.org/about`. Version 3.1.

[19] T. Stahl and M. Völter. *Model-Driven Software Development: Technology, Engineering, Management.* Wiley, June 2006.

[20] The Eclipse Foundation. AspectJ, 2011. URL `http://www.eclipse.org/aspectj/`. Last visited June 27, 2011.

[21] A. van Hoorn, W. Hasselbring, and M. Rohr. Engineering and continuously operating self-adaptive software systems: Required design decisions. In G. Engels, R. Reussner, C. Momm, and S. Sauer, editors, *Proceedings of the 1st Workshop des GI-Arbeitskreises Langlebige Softwaresysteme (L2S2): "Design for Future – Langlebige Softwaresysteme"*, volume 537, pages 52–63. CEUR, October 2009. URL `http://ceur-ws.org/Vol-537/D4F2009_Paper05.pdf`.

[22] A. van Hoorn, M. Rohr, A. Gul, and W. Hasselbring. An adaptation framework enabling resource-efficient operation of software systems. In *Proceedings of the*

Warm Up Workshop for ACM/IEEE ICSE 2010, WUP '09, pages 41–44, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-565-9. doi: http://doi.acm.org/10.1145/1527033.1527047. URL `http://doi.acm.org/10.1145/1527033.1527047`.

[23] A. van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. Continuous monitoring of software services: Design and application of the kieker framework. Technical report, University of Kiel, November 2009.

Acronyms

AOP Aspect Oriented Programming. 19–22, 27, 30

ASA Average Time to Answer. 2

CPU Central Processing Unit. 7, 12, 16, 19

DBMS Database Management System. 28

DoS Denial of Service. 19, 24

EIS Enterprise Information System. 12, 13

EJB Enterprise Java Bean. 12, 22

ETW Event Tracing for Windows. 16, 22, 27

FEP Framework Extension Point. 12

HD hard disc. 12, 16

I/O Input/Output. 16

IP Internet Protocol. 19

J2EE Java 2 Enterprise Edition. 12–14, 22

JavaEE Enterprise Java. 19

JMS Java Message Service. 20

JVM Java Virtual Machine. 8

LOC lines of code. 11

LTW Load-Time Weaving. 20, 31

MDSD Model-driven Software Development. 28

NFR non-functional requirement. 4

ORM object relational mapping. 22

OS Operating System. 7, 8, 11, 16, 22

PDF Portable Document Format. 39

RPC Remote Procedure Call. 16

SDD Software Design Documentation. 28, 39

SDLC software development life cycle. 5, 10, 22, 23, 33

SLA Service Level Agreement. 2, 4, 5, 14, 25, 26

SPE Software Performance Engineering. 4

SRS Software Requirements Specification. 25, 39

TCP Transmission Control Protocol. 16

Thales Thales Defence & Security Systems Deutschland GmbH. 8

TPIS Thales Product Information System. 2, 5, 6, 8, 22, 23, 25, 28, 30, 31, 33, 39

TSF Time Service Factor. 2

UML Unified Modeling Language. 14, 20

XML Extensible Markup Language. 13, 14, 22

A. Attachments

One CD containing

- this thesis as Portable Document Format (PDF) document labeled `Bachelorthesis.Lars.Kroll.pdf`,
- the folder `LatexSources` containing the LaTeX sources of this thesis including the BibTeX library used,
- a folder named `TPIS` containing
  - the TPIS SRS as PDF document labeled `TPISSoftwareRequirementsSpecification.pdf`,
  - the TPIS SDD as PDF document labeled `TPISSoftwareDesignDocumentation.pdf`
  - a folder `Source` containing the source code of the TPIS prototype,
  - a folder `Tomcat` containing a prepared Tomcat installation with the TPIS prototype deployed,
  - a `README` text-file explaining how to build the TPIS prototype and how to use the provided Tomcat installation.