Master Thesis

January 27, 2015

Feedback Driven Development

Bringing Runtime Metrics to the Developer

Christian Bosshard

of Zürich, Schweiz (07-914-948)

supervised by

Prof. Dr. Harald C. Gall

Jürgen Cito, Dr. Philipp Leitner

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

Feedback Driven Development

Bringing Runtime Metrics to the Developer

Christian Bosshard

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Christian Bosshard, christian.bosshard@uzh.ch

Project period: 01.08.2014 - 29.01.2015

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

This Master Thesis is the final step on the way to my master's degree. For this reason, I would like to thank everybody who supported me in any modality during my studies over the past years. My sincere thanks goes to my family, my parents Regula and Markus, as well as my sister Eveline, for supporting me in so many ways throughout my whole life. I also want to thank my fellow students and especially the guys from the ICU board and staff for making our collaborative projects and exercises a pleasure and for many informative discussions and unforgotten moments during our university life.

I want to thank Prof. Dr. Harald C. Gall for giving me the opportunity to write my Master's Thesis at the software evolution and architecture lab at the University of Zurich. Special thanks go to the supervisors of this Thesis, Jürgen Cito and Dr. Philipp Leitner, for their valuable guidance and feedback during the entire process. Furthermore, I thank Linda Rudin for linguistically reviewing this thesis.

Abstract

Software Developers utilize various data sources to draw a more complete picture of the evolution of an application over its lifetime. What kind of data is being used by developers, and how they use this data, is constantly evolving. Techniques derived from empirical software engineering mostly focus on static information (e.g., change requests in issue trackers, test coverage and build information). However, as multifaceted as the development-time feedback discussed so far may be, it still only covers one part of the application evolution. So far, it rarely bothers with gathering data about the application at runtime while it is actually being used. This is the domain of performance engineering, a relatively decoupled separate research domain, which is mostly concerned with optimizing non-functional properties of applications without actually looking at development-time artifacts, such as source code or change requests. However, with the advent of Cloud Computing and DevOps, software development and operations activities are converging. Hence, feedback for empirical software engineering needs to be extended with runtime information. In the cloud, both development-time and runtime feedback are readily available. However, developers currently lack awareness of the potential data and tools provided in the cloud. In this thesis, an approach is developed that integrates feedback from operational data into the Integrated Development Environment (IDE). A prototypical implementation called PerformanceHat, which uses the gathered runtime data to automatically detect the two common software performance problems “hotspot method” and “critical loop”, is contributed. The obtained feedback is visualized in the existent Java editor inside the IDE. An exploratory study shows that the presented approach combines important characteristics to support the DevOps methodology that has not been covered in its entirety in previous research.

Zusammenfassung

Software Entwickler verwenden verschiedene Datenquellen, um ein umfassendes Bild der zu entwickelnden Applikation zu erhalten. Diese Datenquellen, sowie auch die Art, wie Entwickler die Daten verwenden, entwickeln sich stetig. Techniken aus der empirischen Software Entwicklung haben sich bisher stark auf die Bereitstellung von statischen Daten (z.B. Change Requests aus Issue Trackern, Build-Informationen und Testabdeckung) konzentriert. Dynamische Informationen, welche Aufschluss darüber geben, wie sich die Applikation zur Laufzeit verhält, werden bis anhin nur im Bereich Performance Engineering verwendet. Dieser Forschungsbereich ist relativ stark von der Software Entwicklung losgekoppelt und untersucht hauptsächlich die Optimierung von nicht-funktionalen Anforderungen, ohne dabei statische Artefakte, wie Quellcode oder Change Requests, einzubeziehen. Mit dem Aufkommen von Cloud Computing und DevOps konvergieren die Software Entwicklung und operative Tätigkeiten jedoch immer stärker. Somit sollte die empirische Software Entwicklung operationale Daten stärker miteinbeziehen. In der Cloud ist sowohl statisches als auch dynamisches Feedback leicht verfügbar. Die Bewusstheit der Entwickler über die Verfügbarkeit dieser Tools und Daten ist jedoch noch nicht zufriedenstellend. In dieser Arbeit wird ein Ansatz entwickelt, welcher Feedback von operationalen Daten in die Entwicklungsumgebung (IDE) integriert. Teil der Arbeit ist eine prototypische Implementierung namens PerformanceHat, welche operationale Daten verwendet, um die beiden gängigen Software Performance-Probleme “hotspot method” und “critical loop” zu ermitteln. Eine qualitative Evaluation zeigt, dass der vorgestellte Ansatz, anders als vorangehende Forschungsarbeit, die DevOps Methodologie in ihrer Gesamtheit unterstützt.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Cloud computing has experienced a tremendously high growth over the last few years, an increasing number of software providers offer their services *in the cloud*. There are a huge numbers of companies which build up big data centers to provide infrastructure and platforms *as a service*. Amazon EC2[1], Google App Engine[2], and Heroku[3] are some famous examples of such cloud services. On top of those infrastructural services, software vendors, such as Google, Facebook, or Dropbox, provide their services for end users as browser-based cloud applications. There is little software which needs to be installed on the users machine, most of the applications run in the cloud and are accessible over the web. This movement is not only recognizable in the area of private customer software, vendors of big enterprise applications are also moving their applications from on-premise usage models to the cloud in order to stay competitive by benefitting from the advantages cloud computing entails. Examples are SAP with the SAP HANA Cloud Platform[4], Microsoft with its Dynamics CRM[5], or Salesforce[6].

The growth of mobile computing strengthened the importance of cloud computing even more. There are different front-end devices (laptops, tablets, smartphones) that all connect with a relatively small client application to the actual application which runs in the cloud. Traditional desktop applications are disappearing more and more.

This new paradigm has raised various issues on how to design and implement the underlying hardware (networks and data centers), which is frequently discussed in research [Pal10]. On the other hand, there is a lot of impact on the lifecycle of software and thus on the discipline of Software Engineering and the daily work of software developers.

Traditionally, software developers use a number of tools to collect, display, and use different information during development. Issue trackers are used to fetch information about change requests or bugs in the existing source code. From Continuous Integration servers information about the build status, including testing results, is fetched and various analysis tools help to examine the static structure of source code artifacts (e.g., test coverage). Most of these tools are quite closely integrated into the development environment and the data is therefore quickly accessible for developers. Furthermore, there has been various research on improving the integration and visualization of data in those different systems to allow the developer to get a complete picture of the available information [BGG14a, BGG14b, Bir14, Hil14].

An aspect which is so far not covered by the above mentioned tooling, and, for that reason is completely missing in the typical software development process, is information about how

[1] http://aws.amazon.com/ec2/?nc2=h_ls
[2] https://cloud.google.com/appengine/docs
[3] https://www.heroku.com
[4] http://hcp.sap.com/index.html
[5] http://www.microsoft.com/de-ch/dynamics/crm.aspx
[6] http://www.salesforce.com

the developed application actually behaves at runtime in the production environment. The domain of performance engineering deals with such kind of data investigating the non-functional characteristics of an application, how it performs, and how users experience the system. There exist various tools that provide functionality to monitor those aspects of a software system, well-known examples in industry are New Relic[7] and Datadog[8]. Performance engineering is a very important area of the software evolution lifecycle, especially with the rise of cloud computing, since various research has shown that users have high expectations regarding the load times of web applications [Cit14, BKB00, Kin03].

Nevertheless, performance engineering has so far been an isolated domain relatively strictly separated from the software development processes. Software companies often have different teams for *development* and *operations*. These two teams have different tasks with different goals. While developers typically do not care about runtime aspects (non-functional aspects), performance engineers often do not look at the source-code or other development-time artifacts. This separation explains why, also in research, the two areas of software development and performance engineering are mostly treated in isolation.

But with the area of cloud computing, the *DevOps* approach has gained more and more traction. The overall goal of DevOps is to eliminate the described separation and instead closely integrate development and operations. This includes aligning the goals of the two teams but also building new tools that support this new process and allow a fast and automated exchange of data and feedback between development and operations.

1.1 Contribution

The goal of this thesis is to tackle the dichotomy described above from the perspective of a developer working on a cloud application. Foregoing research has shown that developers clearly observe that in cloud deployment there are more metrics available about the production environment than in traditional setups [CLFG14]. Nevertheless, the results of the same study have shown that developers are using their intuition before consulting any runtime data when working on typical software maintenance tasks (e.g., fixing an issue) [CLFG14]. Derived from the assumption that this contrariness is caused by the runtime data not being available to developers in an easy way [CLFG14], this thesis addresses the following two research questions:

- RQ1: "How can performance data from cloud infrastructures be integrated into software development environments?"
- RQ2: "How can this data be used to predict performance problems in advance?"

To answer these questions, a concept has been developed that describes how runtime data could be made better available for developers by combining it with the static code artifacts. The solution shows how performance metrics can be made available directly in developers environment and what feedback the developer could gain out of those numbers. Based on that theoretical concept, a prototypical application called *PerformanceHat* has been implemented as an extension for the Eclipse IDE[9]. The application is capable of collecting runtime data, analysing and aggregating it, and displaying it in a reasonable way in the Integrated Development Environment (IDE). In an exploratory study we compared our tool to other performance engineering tools and could show that our tool supports the DevOps approach better by combining both static and dynamic aspects of software systems.

[7] http://newrelic.com
[8] https://www.datadoghq.com
[9] http://www.eclipse.org

1.2 Thesis Outline

The remainder of this thesis is structured as follows:

- Chapter 2 discusses some basic terms and relevant topics. First, the terminologies *Cloud Computing*, *DevOps*, *Continuous Delivery*, and *Performance Engineering* are summarized. Then, an insight into common performance anti patterns is given. Finally, some statistical concepts which are applied in the approach of the thesis are discussed.
- In Chapter 3, Feedback-Driven Development is introduced as a solution to the problem of integrating runtime data into traditional development environments in order to support the DevOps approach. A concept is developed that specifies what an appropriate system should look like. In a second part, the architecture and implementation of PerformanceHat are illustrated.
- The evaluation in Chapter 4 is divided into two distinct parts. First, an exploratory study compares the implementation with other research prototypes as well as industrial tools from the domain of performance engineering. Second, a quantitative evaluation investigates some performance aspects of two different components of the implemented prototype.
- Chapter 5 summarizes a selection of preceding research work that is closely related to the topic of this thesis.
- Finally, Chapter 6 concludes the work, recaps the research questions, and discusses some possible future work to improve the presented concept as well as the implemented prototype.

Chapter 2

Background

The following chapter introduces basic terms and discusses important topics on which this thesis is based on. First, the terms Cloud Computing, DevOps, Continuos Delivery, and Performance Engineering are introduced and explained. Subsequently, important aspects of Peformance Engineering are discussed in more detail.

2.1 Introduction of Terminologies

2.1.1 Cloud Computing

Cloud computing is an omnipresent buzzword that stands for a huge technological trend in the landscape of information technology over the last years [Hil09]. However the topic has outstripped the scope of applications and technologies and various research is conducted investigating business-related issues and prospects of cloud computing [SMG11, CBWDR10]. From a technological point of view cloud computing is discussed very frequently in research [BYV+09, AFG+10, VRMCL08, MG11, WTK+08, AFG+09] as well as in literature [AI10, BBG10], but still no generally accepted definition has been established [VRMCL08, WTK+08]. The following section provides a summary of the most important technology-related aspects of cloud computing discussed in the sources mentioned above. In [WTK+08] Wang *et al.* propose the following definition of cloud computing:

> "A computing Cloud is a set of network enabled services, providing scalable, QoS guaranteed, normally personalized, inexpensive computing platforms on demand, which could be accessed in a simple and pervasive way." [WTK+08]

The definition given is supported by the National Institute of Standards and Technology[1] of the United States, which defines cloud computing as *a model for enabling on-demand network access to configurable computing resources* [MG11]. This covers a very important characteristic of cloud computing, which is elasticity: Resources have to be dynamically configurable and adjustable to variable loads (rapidly provisioning and release [MG11]) without service provider interaction and large manual effort for the consumer [VRMCL08]. Customized service level agreements (SLA) determine the quality of services promised by an infrastructure provider [VRMCL08]. Typically pay-as-you-go pricing models allow for the consumer to only pay the resources they actually use [VRMCL08]. Cloud services can be classified into three different layers depending on the level of abstraction they provide. Table 2.1 illustrates those layers:

[1] http://www.nist.gov

Layer	Description	Examples
Software as a Service (SaaS)	The consumer uses an application developed and hosted by the provider [MG11]. The consumer does not manage or control the underlying cloud infrastructure.	GMail[2], Salesforce[3]
Platform as a Service (PaaS)	The provider offers a platform (operating system with additional software components on top) tailored to a specific type of application that can be deployed on top of the provided stack [VRMCL08].	Google App Engine[4], Heroku[5]
Infrastructure as a Service (IaaS)	The provider offers a virtualized environment (OS). The consumer has control over the operating system and deploys its own software stack on top of it [VRMCL08, MG11].	Amazon EC2[6], RackSpace[7]

Table 2.1: Service models of cloud providers.

2.1.2 DevOps Approach

With the upgrowth of cloud computing, the term *DevOps* has become very popular. It is used diversely to describe the closer integration of development- and operations aspects of the software lifecycle in cloud setups. Similar to cloud computing, there is no clear definition of the term DevOps [Hü12]. Hüttermann explains the approach from the perspective of a developer as follows: *DevOps is a mix of patterns intended to improve collaboration between development (includes programmers, testers and quality assurance personnel) and operations (system- and database administrators as well as network technicians). DevOps addresses shared goals and incentives as well as shared processes and tools* [Hü12]. He states that conflicts between different groups are natural and finding shared goals and incentives is therefore often difficult or even impossible. The development team is typically interested in a frequent rollout of their changes to production (need for change), because their responsibility is to deliver new features and bug fixes, while the operations team wants to avoid making changes (fear of change) once the software is delivered, because their responsibility is to keep the performance stable [Hü12]. Hüttermann therefore suggests that the goals of the team should at least be aligned with one another, if no shared goal can be found. How this is achieved concretely by applying agile methodologies in both development and operations is described in [Hü12]. The second important aspect that Hütterman defines as part of the DevOps movement is the evolution of shared processes and tools: tools may no longer evolve independently targeting only the needs of one team. They should be designed to improve the communication between developers and operations and provide the required automation to stream feedback from production to development [Hü12]. According to Hüttermann DevOps enables an increase in the quality of software and a better alignment to individual requirements while being able to deliver the products in shorter time periods.

2.1.3 Continuous Delivery

Continuous Integration (CI) is a well-established principle in software engineering, targeting a fast integration of changes into version control systems and the automatic build and verification of

each integration such that errors, failing tests, or other violations are detected quickly [HF10]. The goal is to reduce the amount of integration problems and to allow rapid application development [HF10]. Continuous Delivery (CD) is an extension of Continuous Integration [HF10] that goes one step further by allowing to roll out single changes to productive systems very fast:

> "Continuous Delivery is a software development discipline where you build software in such a way that the software can be released to production at any time." Martin Fowler [Mar13a]

When applying Continuous Delivery it has to be ensured that every particular change is releasable [HF10]. A new version of the software can be released by simply clicking a button [Mar13a]. To ensure that the software will work in production, a *Continuous Delivery Pipeline*[8] as shown in Figure 2.1 is introduced.

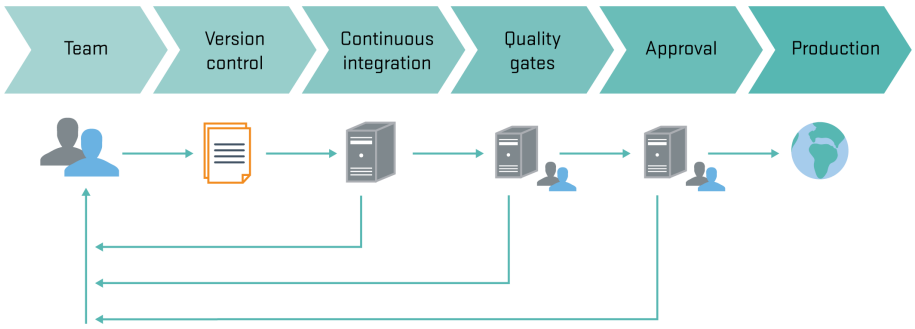


Figure 2.1: The Continuous Delivery Pipeline [Che].

The first step in the pipeline is the version control system (VCS) which holds all the code artifacts. The whole process is triggered as soon as someone contributes code to the VCS [Che]. Continuous Integration is started, which includes building the code artifacts and testing as well as verifying the built artifacts [Che]. If those steps are successful, further tests, like functional- or performance tests, are executed (quality gates) [Che]. Each intermediate step in the pipeline provides some form of feedback to the development team [Che, Mar13b]. Typically this is either a sucess message or a list of specific problems (e.g., failing tests). If all intermediate steps in the pipeline are passed successfully the software product is finally released to the productive environment [Che]. Making the whole pipeline work requires a close collaboration between development and operations [Mar13a] as suggested by the DevOps approach (see Section 2.1.2).

2.1.4 Performance Engineering

Performance Engineering is a sub-discipline of systems engineering [GNJ] covering all activities throughout the whole software development life cycle that are related to ensuring the non-functional (performance) requirements of a software system [WFP07]. It is a very complex domain, because the performance of an application is not only affected by the software itself, but rather by all its underlying layers in the execution environment such as the operating system, middleware, hardware, and the network connection [WFP07]. Performance is typically quantified by measuring the response time for the system's end users or counting the executed business

[8] sometimes also called *Deployment Pipeline* [Mar13b]

transactions per second [Man]. Satisfying the high user expectations [Cit14, BKB00, Kin03] can therefore be seen as the overall goal of all performance engineering activities.

Recapping the characteristics of cloud computing one could argue that application performance doesn't matter in cloud setups, because infinite resources and the ability to scale are available. This assumption has been proved to be completely wrong [Mur11]. In fact, performance engineering plays an absolutely crucial role in cloud computing [Mur11] and has even brought new challenges for performance engineers due to the heterogenous influence factors [ST13].

Although performance engineering is often recognized as an exclusively operational task, it is important that its activities start early in the software development lifecycle [GNJ]. In the requirements analysis phase of the system the non-functional requirements should be defined [WFP07]. Typical non-functional requirements are usability, maintainability, extensibility, scalability, reusability, security, and transportability [GNJ]. The defined requirements should then be considered during the design and implementation phases each time a decision is made: every aspect of the design and code can influence the performance of the system at runtime [WFP07]. During implementation performance tests are an important instrument to verify that the application is actually able to meet the defined non-functional requirements [WFP07]. Performance tests should be conducted on parts of the system as well as on the whole system. Additional load tests should inject artificial load onto the system to verify the system's performance in certain stress situations [GNJ]. As soon as a system goes live, the monitoring and instrumentation activities of performance engineering begins [GNJ]. The system is instrumented, measurements are collected, and as soon as performance issues are found, the root causes have to be identified and fixed. Often the source code of the application is optimized reactively [Man].

Summarizing the topic of performance engineering, we can state that the discipline deals with the application itself as well as with the underlying infrastructure (software stack and hardware). In cloud computing, the stack of hardware and virtualization layers is very complex and therefore the importance of an appropriate performance management increases even more [ST13]. The aspects of performance engineering that can be directly mapped to source code artifacts are of most interest for this thesis and therefore discussed in detail in the subsequent section.

2.2 Performance Anti Patterns

Section 2.1.4 briefly mentioned the importance of considering performance aspects while designing and implementing a software system. Those aspects have gained importance - especially in cloud computing [ST13]. Each architectural or technological decision potentially influences the performance of the application at runtime. The same applies to code-level decisions during implementation.

Various research has been conducted on examining big enterprise applications to find common patterns in source code that potentially result in performance problems at runtime. The architectural concept of patterns was introduced in 1977 by Alexander [AIS+77]. The book *Design patterns: elements of reusable object-oriented software* by Gamma *et al.* [GHJV94] made (design) patterns a famous concept in computer science and especially in software engineering. A design pattern is defined as a template that describes how to solve a commonly occurring problem [Mar00a]. It therefore provides a reusable solution that is applicable to many different situations [Mar00a]. As the counterpart of a pattern an anti pattern can be defined as a bad approach to solve a certain problem [CC]. More precisely an anti pattern typically describes a solution that *looks like a good idea*, but has potential negative consequences when being applied [CC]. Performance anti patterns finally comprise those software development anti patterns that potentially cause performance problems at runtime. The following Subsections describe well-known performance anti patterns described in previous research.

2.2.1 God Class

The *God Class* is one of the most familiar anti patterns in object oriented systems [Rie96]. Sometimes it is also referred to as *blob* [Bro98]. A god class has the main characteristic that it performs a lot of work and clearly violates the *Single Responsibility Principle* [Mar00b]. The work it does is often completely independent and therefore the cohesion of the class is very low. It interacts with many small classes that do not implement any logic but typically only hold data and provide accessor methods (i.e., getters and setters) to this data [SW00]. The god class operates on that data and pushes updates to the other classes. Class names containing `Controller` or `Manager` sometimes indicate god classes [SW00].

The example in Figure 2.2 shows a simplified class diagram of a typical god class: there is one big class (`GodClass`) taking over all the responsibilities while the other classes (`DataOne` and `DataTwo`) only hold the data. Furthermore, the Figure shows how the god class anti pattern can be eliminated by incrementally refactoring the system and moving the behavior closer to the data it operates on [DDN02].

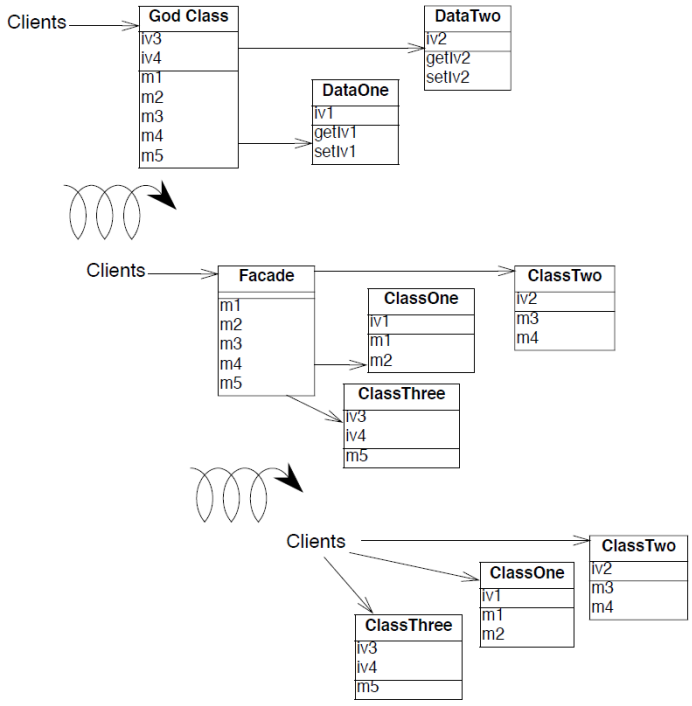


Figure 2.2: A sample god class and how to refactor it [DDN02].

God classes have many negative consequences on software systems: components containing god classes are poorly or even not at all maintainable and modifiable [SW00]. The reusability and extensibility is very low and also testing is very difficult [DDN02]. The main problem of a god class regarding performance is the increase of required message traffic [SW00]. A lot of traffic is required to transmit the data to the collaborators to operate on it and push the updated data back to the respective collaborators afterwards. In [SC93] Sharble and Cohen were able to verify this circumstance. They compared two object oriented designs, a data-driven- and a responsibility-driven design, by measuring the total number of messages that are required to

complete different scenarios. The data-driven design is mainly constructed by a god class while the responsibility-driven design is a refactored variation of the data-driven design. Figure 2.3 illustrates the dramatic impact of a god class on the number of required messages in the different scenarios[9] [SC93]. From that we can derive the time loss as shown in Equation 2.1.

$$T = M_{\Delta} \times O_M \tag{2.1}$$

where M_{Δ} is the number of additional messages used in the god class variation and O_M is the resulting overhead for one message [SW00]. Even for a very small messaging overhead the performance damages of a god class can become very high, as the large deltas in Figure 2.3 illustrate.

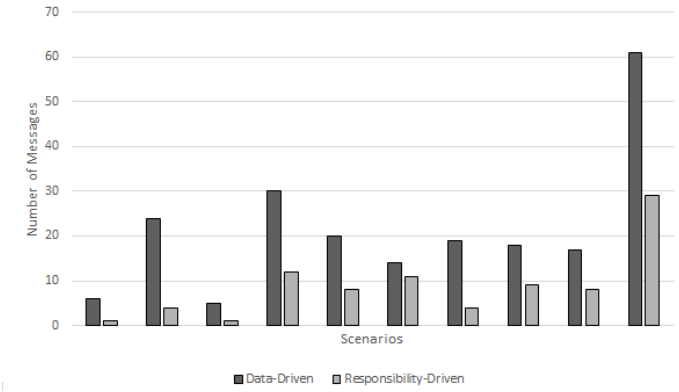


Figure 2.3: The impacts of a god class on the number of messages (data from [SC93]).

2.2.2 Excessive Dynamic Allocation

In object-oriented programming dynamic allocation denotes the principle of creating objects when they are first accessed and destroyed as soon as they are no longer needed [Smi01]. This can be a good approach in many situations. However, creating and destroying objects is expensive: the memory to contain an object and any objects it contains has to be allocated from the heap and the object's initialization code has to be executed [Smi01]. On object-destruction the required clean-up code must be executed and, in order to prevent memory leaks, the released memory has to be returned to the heap. The excessive dynamic allocation pattern addresses designs where many objects of the same class are created and destroyed unnecessarily during the application's life-cycle [Smi01]. In his book, *Object-oriented Design Heuristics*, Reil highlights this situation with a nice metaphor. He compares the approach of repeatedly creating and destroying objects of the same type with buying a piece of land and building up a whole gas station with all its components each time your car needs gasoline [Rie96]. When the gas tank of the (single) car has been filled, the gas station is destroyed and the land is sold. Equation 2.2 defines the time required for dynamic allocation.

$$T = N \times \sum_{depth} S \quad with \quad S = s_c + s_d \tag{2.2}$$

[9] The names of the scenarios are omitted because they are of no relevance

where N is the number of allocations, $depth$ is the number of (recursively) contained objects and S is the (service) time for creation (s_c) and destruction (s_d) of objects [SW00]. Figure 2.4 illustrates the overhead of excessive dynamic allocations for some typical values of the mentioned variables [SW00]. To eliminate dynamic allocation objects should, if possible, be reused. Shared objects can for example be collected in a central object *pool* and handed on to clients on-demand.

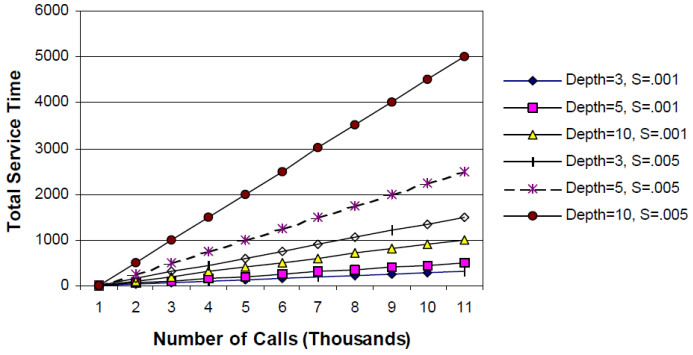


Figure 2.4: Overhead of excessive dynamic allocation [SW00].

2.2.3 Aggressive Loading of Entities

This anti pattern, similar to *excessive dynamic allocation*, deals with the costs of object creation, but more specifically focuses on deep recursive object structures, which typically apply to entity beans. The term entity bean is used in the Java world for objects that build up an in-memory representation of persistent data [PM04a]. The term (entity) bean is used in the following as a general term for such objects without restriction to the Java world. When a bean is instantiated, the data has to be loaded from the database, which can be a very time consuming task [PM04b]. Entity beans are typically strongly interconnected. Depending on the inter-bean connection graph, instantiating a single entity bean can lead to numerous other entity beans being loaded from the database [PM04b]. Given the fact that the majority of those interconnected objects are often not used, this is an expensive performance issue. The issue can be resolved using *lazy loading*, which means that references to other entity beans are not resolved until they are actually used [PM04b]. Upon the initialization of a bean only proxies [GHJV94] of the related beans are created.

2.2.4 Too Many Remote Calls

One of the most costly operation in a distributed system is the communication between different components. *Too many remote calls* addresses systems whose (bad) design results in a large communication overhead and thus in potential performance problems [PM04b, PM04a]. A typical example of such a design is shown in Listing 2.1. A remote call fetches some data items that are subsequently iterated and for each item another remote call is required to fetch any sub items. In cases where the remote call accesses a database this anti pattern is also known as the *N+1 select* or *N+1 query* problem [Acc, Docb], because fetching N items results in N+1 `select` operations on the database. The example in Listing 2.1 can be arbitrarily nested through iterating over sub items and conducting further remote calls. Remote database calls are often even more problematic than other remote calls, since additional costs for database connection establishment and query processing accrue in addition to the network delay [PM04a].

```
1 items = getRemoteItems()
2 for item in items:
3   subItems = getRemoteSubItems(item)
4   [...]
```

Listing 2.1: Simple example of *too many remote calls* (N+1 select problem).

Parsons and Murphy [PM04b, PM04a] extended the scope of *too many remote calls* and introduce the *circuit treasure hunt* anti pattern. The name originates from the analogy of a treasure hunt game, where one small hint helps to find the next one and so on until the treasure can be located [PM04b]. The only difference to the *too many remote calls* anti pattern is that it is not restricted to remote calls only, but to costly operations in general. They mention big response sets as an example: to actually get the data item of interest a client has to recursively invoke different operations on different intermediate objects. Sometimes intermediate objects even have to be created and/or destroyed, which results in additional costs (see Chapter 2.2.2 about *excessive dynamic allocation*).

There are different solutions to the *too many remote calls* and its related anti patterns depending on the situation. In the case of databases, one can, if possible, design the database schema itself in another way, such that typical requests are achievable with only one query [PM04b]. Denormalized database schemas often perform better than normalized designs. Of course, all scenarios (i.e., all typical database requests) should be considered, because optimizing for one scenario can have negative effects on others [PM04b]. An optimal tradeoff has to be found. Another way of optimizing the design to eliminate the discussed pattern is to introduce the *Adapter-* [GHJV94] or the *(Session) Facade* Pattern *gamma1994, alur2003* to align the (remote) interface to the client's needs.

2.2.5 Expensive (Nested) Loops

Studying over 100 different performance bugs in five real-world open-source projects, Jin *et al.* showed that 90% of all performance bugs involve loops and 50% involve nested loops [JSS+12]. In subsequent research Nistor *et al.* [NSML13] categorized those loop-related performance bugs into several categories. They detected two root causes for loop-related performance issues, one is redundant computation and the other is inefficient computation [PM04b]. Redundant computation means that the same computation is done on the same data - with the same result - in all or multiple cycles of the same loop (assuming the computation doesn't induce any side-effects). Considering nested loops Nistor *et al.* derived the following four problem categories:

- Redundancy in Outer Loop
- Redundancy in Inner Loop
- Inefficient Outer Loop
- Inefficient Inner Loop

Listing 2.2 shows a very simplified nested loop with a computation in the inner loop. Based on the data passed to the computation the result is potentially the same for all iterations of the inner or even all iterations of the outer loop. The first case is typically true if the computation operates only on data of the scope of the outer loop, the latter one can arise if the computation operates on the whole dataset of the outer loop (i.e., all items) or on data from outside the loops. In such simple cases as shown in Listing 2.2, one could argue that such redundant calculations should be easily detectable when looking at the source code, but most real-world examples are a

lot more complex. Typically, the loops are not directly nested, but some operation invocations are in between. Sometimes the pattern is put together by own code in combination with third-party libraries (whose code is not visible).

```
1 for outerItem in outerItems:
2   for innerItem in outerItem.getInnerItems:
3     result = computation(...)
```

Listing 2.2: Redundant computation in nested loops.

Figure 2.5 shows a real-world example of an outer loop redundancy in the *JFreeChart*[10] library that resulted in a severe performance issue, because it froze the chart display [PM04b]. The `drawVerticalItem`, called from `drawItem` in the outer loop, contains an inner loop that computes the maximum volume of all the items that are iterated in the outer loop. Of course, the maximum volume does not change, because the data set is the same for all invocations of `drawVerticalItem`. Therefore, the computation is redundant and the maximum volume could be computed only once and then be cached. This concept is called memoization [ABH03]. Resolving inefficient (inner/outer) loops sometimes works by simply moving costly calculations out of the loop [PM04b]. In some cases this does not work, because the calculation is based on data from loop iterations. In those cases a more efficient data structure has to be found to enable a more efficient algorithm [PM04b].

```
1  // Simplified from the XYPlot class in JFreeChart
2  public void render(...) {
3    for (int item = 0; item < itemCount; item++) { // Outer Loop
4      renderer.drawItem(...item...); // Calls drawVerticalItem
5    }
6  }
7  // Simplified from the CandlestickRenderer class in JFreeChart
8  public void drawVerticalItem(...) {
9    int maxVolume = 1;
10   for (int i = 0; i < maxCount; i++) { // Inner Loop
11     int thisVolume = highLowData.getVolumeValue(series, i).intValue();
12     if (thisVolume > maxVolume) {
13       maxVolume = thisVolume;
14     }
15   }
16   ... = maxVolume;
17 }
```

Figure 2.5: Example of a redundant outer loop [PM04b].

2.2.6 One Lane Bridge

A one lane bridge is a potential bottleneck in a traffic system, because the traffic can only traverse the bridge in one direction at a time, and multiple parallel lanes have to be merged into one, which results in a traffic jam and waiting times [SW00]. The analogous situation in software arises if at any point in time during an application's lifecycle only one or a few processes are able to continue doing their work [SW00]. All the other processes are blocked and have to wait. Unlike in the one lane bridge analogy, this is typically not a physical problem, but a design issue. A typical case of this anti pattern are resources, for example a database table or row, that are locked as long as one process accesses them [SW00]. Another example of this pattern arising is a non-multi-threaded process that is synchronously called by multiple other processes [SW00]. The calling processes are served one after another.

[10] http://www.jfree.org/jfreechart

The easiest solution for the *one lane bridge* anti pattern is to reduce the time for concurrent computations (i.e., the time to traverse the bridge). Figure 2.6 illustrates this solution by showing the results of a respective experiment of [SW00]: the residence time is the total time required to serve all waiting processes, the arrival rate is the number of arriving requests (to the shared resource) per second, and S (service time) is the time required to serve one process. The difference between the two curves illustrates that by reducing the service time even as little as one millisecond, the performance can be significantly improved.

In the case of database locks, the one lane bridge can be further improved by minimizing the locks by using row blocking (i.e., only blocking rows instead of the entire table), if updates do not operate on the whole table. Alternatively the database schema can, if possible, be adapted to reduce the number of tables that have to be locked for typical updates [SW00].

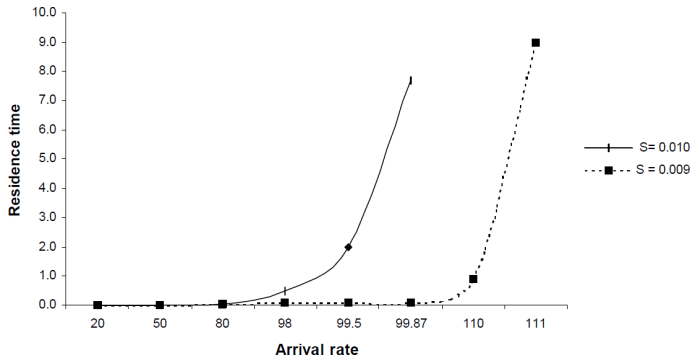


Figure 2.6: Impact of improving the service time of a one lane bridge [SW00].

2.2.7 Traffic Jam

Traffic Jam describes a situation where response times for the same operation widely diversify over time [Smi01]. The name of the anti pattern originates in the analogy of the stop-and-go behavior that often occurs on highways with high traffic volume [Smi01]. In software, the problem often occurs if there is a job backlog and jobs come in collectively: a lot of jobs come in more or less simultaneously while during other periods only a few or even no jobs come in [Smi01]. Often, the cause for this problem is the *one lane bridge* anti pattern (see Section 2.2.6). Other causes are covered in [Smi01].

Sometimes the *Traffic Jam* anti pattern is caused by user selections. In this case the solution to it is using the Flex-Time pattern: if a system allows its users to specify the time to generate a report (which is a costly operation), the selection options should consist of time intervals rather than specific times [Smi01]. Often the cause of multiple users choosing the same value is the usage of default values that are not changed by most of the users. Therefore, a solution can be to either not provide default values or vary their value randomly. If external factors result in a traffic jam, the online solution is to streamline the concerned operations as much as possible [Smi01].

2.2.8 The Ramp

This anti pattern describes a scalability problem where the processing time for a certain operation continuously increases over time [SW03]. The curve of the processing times on a timeline correspondingly looks like a ramp (see Figure 2.7). A typical cause for this problem is accruing data: requests operate on a data set and each request increases the data set. A typical example

is checking requests for duplicates, which requires to store all the requests and with each new request the list of requests to be checked gets longer [SW03]. Another example is covered by the *sisyphus database retrieval performance anti pattern* introduced by Dugan *et al.* [DGS02]. In this case a large data set is processed (e.g., a search on a table in a relational database) and a subset of the result is displayed to the user [DGS02]. Let us assume a web search displaying 20 results per page: requesting the first page requires searching the first 20 items, while the second page requires 40 items to be able to display items 21-40. The size of required result items of course increases with each subsequent request which results in very bad scalability.

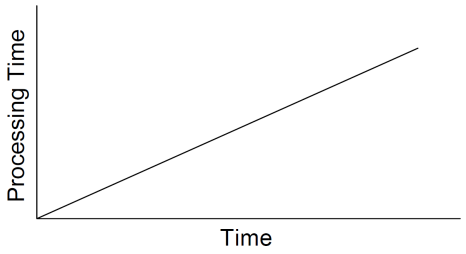


Figure 2.7: The ramp anti pattern [SW03].

The ramp anti pattern is typically not detected during testing, since test data sets usually do not contain enough items to determine a significant performance loss [SW03]. The solution to the Ramp anti pattern is to find a more appropriate (search) algorithm to operate on large data sets [SW03]. In some cases it might even make sense to implement self-adapting algorithms that are able to assimilate their processing logic according to the size of the data set [SW03]. Further potential solutions for the *sisyphus database retrieval performance anti pattern* are discussed in [DGS02].

2.2.9 Unbalanced Processing

When processes cannot use the available processors effectively, either due to processors being dedicated to other processes or due to single-threaded code, unbalanced processing occurs [SW03]. This is often the case in pipe and filter architectures, whereat long running filters block faster ones [SW03]. The solution to that problem is, on the one hand, to determine long running filters and, if possible, split them into multiple, parallelly executable, steps [SW03]. On the other hand, multiple short-running filters can be combined into one step [SW03]. This results in a more balanced workflow which in turn results in more efficient resource utilization. A second often occurring example of unbalanced processing is the usage of static routing algorithms to split up work items into multiple queues [SW03]. Due to the static characteristics the items can be distributed in an unbalanced way [SW03]. The solution to this problem, of course, is to change the algorithm to using dynamic information for the routing in order to distribute the items uniformly [SW03].

2.2.10 More is Less

More is less uses the analogy of trying to do too many things in parallel and then actually ending up doing nothing [SW03]. The same can occur in software systems, if too many processes try to run in parallel. This can result in the underlying system spending all its time doing paging and serving page faults rather than doing the actual work [SW03]. Although this might not be a

big issue in modern environments, the more is less anti pattern can also be related to bad system designs. Smith *et al.* determined the following causes:

- Creating too many database connections
- Allowing too many internet connections
- Creating too many pooled resources
- Allowing too many concurrent streams relative to the number of available processors

A possible solution to overcome those problems is to introduce a priority queue that can be filled with command objects [SW03]. The command objects are constructed using the *Command* pattern [GHJV94] and can perform various tasks. If this limited concurrency doesn't fit to the applications needs the system design has to be adopted to somehow limit the number of parallel streams [SW03].

2.2.11 Unnecessary Work

Unnecessary work (also known as *unnecessary processing* [SW03]) is as simple as it sounds: it addresses source code that simply does work that is not required. Although this might sound like a beginner's mistake, a study on open-source software showed that this anti pattern causes over 25% of all performance bugs [JSS+12], because in the context in which they were called, functions simply conducted unnecessary work [JSS+12]. A real-world example from a graphic program is the invocation of a draw()-method for multiple transparent figures [JSS+12]. Often, unnecessary processing occurs due to stale data being processed, even though the result is discarded afterwards [SW03]. Other cases involve unnecessary synchronization code, which results in processes waiting for other processes unnecessarily. This type of issue often occurs in server (e.g., Apache: 4 out of 15 investigated bugs [JSS+12]) and database applications (e.g., MySQL: 5 out of 26 investigated bugs [JSS+12]).

The elementary cases of unnecessary work can be eliminated by simply removing the respective operation invocations [SW03]. In more complex cases it sometimes helps to re-order steps in the execution to early detect stale data that does not have to be processed anymore [SW03].

2.3 Statistical Methods

In order to deal with data from performance engineering, a number of statistical models and methods can be used. This section describes three statistical concepts relevant for this thesis.

2.3.1 Time Series Data

A time series is a collection of observed values of a stochastic system ordered according to their occurrence in time [SS10]. While in a deterministic system, the future state of a system is only determined by the current state of the system, stochastic systems describe non-deterministic systems whose future state is not completely determined by its current state. This means, that there are always multiple possible transitions to future states. The aim of time series is to observe stochastic systems. If a time series is the collection of values $\{x_1, x_2, x_3, ..., x_n\}$, x_1 is the value observed at the first time point, x_2 is the value observed at the second point in time, and so on. Generally, x_t is the value observed at the point t in time [SS10].

The underlying system of the FDD approach is the monitored application, which clearly is a stochastic system and the monitored runtime data can be classified as time series data. Since

future states of stochastic systems are not determinable from past and current states, predictive analysis of the time series data is required in order to estimate how the performance of an application evolves.

2.3.2 Moving Average

The moving average (MA) is a statistical model that is used to smooth time series [BJR13]. Smoothing describes the process of filtering out random fluctuations from a time series that can arise due to some external phenomena, but are not relevant to the long-term trend of the time series [BJR13]. In software applications, such a fluctuation can occur due to temporary high network delays. To filter out the fluctuations from a time series, the MA model takes the time series as input and creates a new time series that is computed out of average values of subsets of the original time series [et12].

$$a_t = \frac{\sum_{i=0}^{w-1} x_{t-i}}{w} \tag{2.3}$$

For each time point t on the original time series X, the average of a pre-defined number of foregoing values $\{x_t - w, ..., x_t\}$ determines the value a_t of the new time series MA, where w is called the *window size* (see Equation 2.3). Figure 2.8 illustrates an example of a moving average: the red line, which is constructed of the MA values, is a smoothed trend of the observed values (black line).

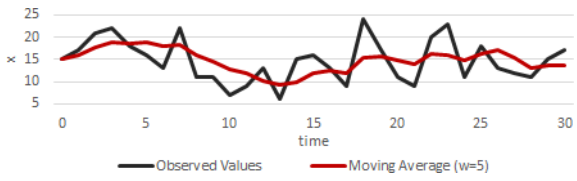


Figure 2.8: An example plot of a moving average with window size 5.

2.3.3 Change-Point Analysis

Change Point Analysis (CPA) is a statistical concept that enables investigating whether and at which point in time changes occur in a time series [Tay00]. There are multiple different approaches that allow to perform a CPA. In the following, we focus on an approach developed by Taylor in [Tay00]. It combines the two concepts of cumulative sum (CUSUM) and bootstrapping to detect changes. The approach is illustrated on the basis of an example: Figure 2.9 shows a time series plot containing the execution times of the last twenty executions of a procedure.

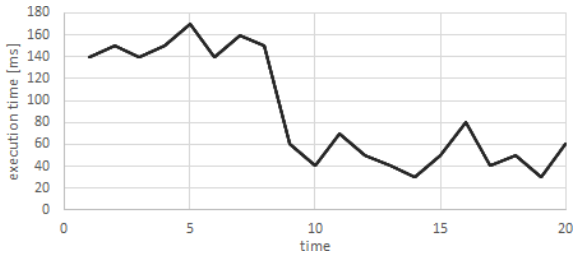


Figure 2.9: Execution times of a procedure.

Looking at the plot one can see a change after the time point 8, where the execution time decreases significantly for a longer time. To detect changes algorithmicly, the CUSUM chart of the time series values $\{x_0, x_1, ..., x_n\}$ is calculated as described in the following. First, the average of all the observed values is determined according to Equation 2.4.

$$\bar{x} = \frac{x_1 + x_2 + ... + x_n}{n} \tag{2.4}$$

Now, the cumulative sums $\{s_0, s_1, ..., s_n\}$ can be calculated as shown in Equation 2.5. For each time point, the difference between the value of the current time point and the average value is added to the CUSUM of the preceding time point. For the first CUSUM s_1, the preceding CUSUM is declared to be zero ($s_0 = 0$).

$$s_i = s_{i-1} + (x_i - \bar{x}) \tag{2.5}$$

In the time series example from Figure 2.9, the average value $\bar{x}$ is 90, and the CUSUM values are therefore calculated like as follow:

$$\begin{aligned}
&s_0 = 0 \\
&s_1 = 0 + (140 - 90) = 50 \\
&s_2 = 50 + (150 - 90) = 110 \\
&... \\
&s_{20} = 30 + (60 - 90) = 0
\end{aligned}$$

Due to the fact that the cumulative sums are calculated from the differences between the values and the average, they always sum to zero. That is why the last CUSUM value is always zero (s_{20} in the example). Figure 2.10 shows the CUSUM chart of the example. The CUSUM chart can be interpreted as follows: As long as the curve goes upwards, the values of the underlying time series tend to be above average. Analogously, a segment where the curve goes downward indicates values below average. If the chart follows a more or less constant value, this indicates underlying values which are close to the average. The interesting places in a CUSUM chart are the points where the curve suddenly changes its slope. Those are the points where a change is assumed in the time series. In Figure 2.10, a turn is recognizable after time point 8, which coincides with our manual observations made in Figure 2.9.

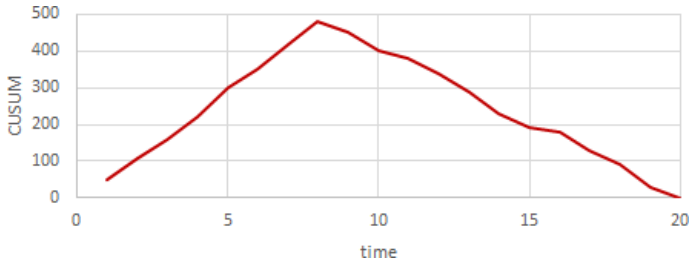


Figure 2.10: The CUSUM plot of the time series in Figure 2.9.

Calculation of Confidence Level

The interpretation of the CUSUM chart in Figure 2.10 is relatively clear. However, there is no guarantee that a turn of the CUSUM chart always indicates a change point in the time series. There can be CUSUM charts with less obvious turns that can be interpreted differently. To make reliable statements about change points, their confidence level can be calculated as described in the following. First, s_{diff} is defined to be the difference of the maximum and the minimum value of $\{s_1, s_2, ..., s_n\}$ (see Equation 2.6).

$$s_{diff} = s_{max} - s_{min} \quad , \quad where \quad s_{max} = \max_{i=0,...,n} s_i \quad and \quad s_{min} = \min_{i=0,...,n} s_i \tag{2.6}$$

Now, the actual bootstrapping analysis is conducted. A bootstrap sample is created randomly reordering the original values $\{x_1, x_2, ..., x_n\}$ from the time series and then calculating the cumulative sums $\{s^0{}_1, s^0{}_2, ..., s^0{}_n\}$ and from that the difference of the maximum and minimum $s^0{}_{diff}$ is determined. The bootstrap sample creation is repeated, each time with another random order of the original values. Figure 2.11 visualizes the original CUSUM chart together with three different bootstrap samples. The differences $s^j{}_{diff}$ of the particular bootstraps is determined by the distance between the topmost and the lowermost point of the respective curve.

To calculate the confidence level, the difference s_{diff} is compared to the difference calculated for each bootstrap $s^j{}_{diff}$ and it is checked whether the s_{diff} is larger than $s^j{}_{diff}$. If N is the number of conducted bootstrap samples and X is the number of bootstrap samples for which $s_{diff} > s^j{}_{diff}$, then the confidence level CL of the change point is defined as the proportion of X and N (see Equation 2.7).

$$CL = 100\frac{X}{N} \tag{2.7}$$

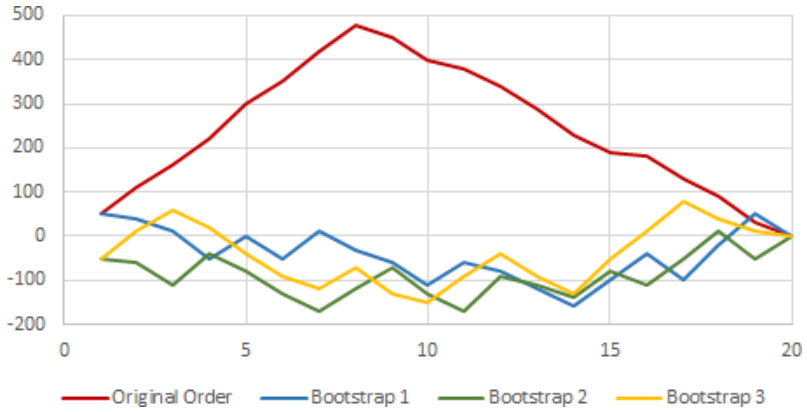


Figure 2.11: Bootstrap samples of the CUSUM from Figure 2.10.

Table 2.2 contains the values from the bootstrapping samples from Figure 2.11. All of the three samples exhibit a smaller difference than the original time series, which means that the confidence level is 100%. However, three samples is a very small number for this kind of analysis. The higher the number of created samples is chosen, the more precise the calculated confidence level. In order to receive a reliable confidence level, the number of samples should at least be 1000. For the example presented here more precise bootstrapping analyses have been conducted with 1000, 10000, and even 100000 samples, which all resulted in a confidence level of 100%. Therefore, the detected change point in this example can be treated as significant. Typically, a confidence level of 90%-95% is required to classify a change point as significant [Tay00].

Sample	$s^j{}_{max}$	$s^j{}_{min}$	$s^j{}_{diff}$	$s_{diff} > s^j{}_{diff}$
Original Order	480	0	480	-
Bootstrap 1	50	-160	210	yes
Bootstrap 2	10	-170	180	yes
Bootstrap 3	80	-150	230	yes

Table 2.2: Results of the boorstrapping analysis.

As already mentioned in the beginning of this section, there are other methods to conduct a Change Point Analysis than the one presented here. Deeper knowledge about the CPA can be found in [CG11].

Chapter 3

Bringing Runtime Metrics to the Developer

This chapter describes the approach of Feedback-Driven Development (FDD) [BFKB+14]. First, the underlying problems is analyzed based on previous research and the main goals of Feedback-Driven Development are derived. A conceptual approach providing a possible solution is then presented and discussed in detail. It is shown how the approach solves the elicited problem and consequently targets the derived goals. Finally, the architecture of a respective system implemented as part of this thesis is illustrated - followed by some essential implementation details.

3.1 Problem Analysis

Modern, high-level software platforms and their corresponding programming languages provide a valuable abstraction to developers by hiding system- and hardware-related aspects. With frameworks on top, the level of abstraction is further increased and developers can completely focus on implementing business value rather than dealing with complex technological problems. This trend has, of course, positive impacts on the productivity - it allows to deliver features faster. Nevertheless, the downside of this evolution is that developers often lose the awareness and knowhow of the technological layers behind the high-level abstraction:

> "Modern advances in software technologies have allowed developers to concentrate less on issues such as performance and resource management, and instead developers have been able to spend more time developing the functionality of their applications. An example of this can be seen in modern languages (Java, C#) that provide garbage collection facilities, freeing developers from the task of having to manage memory, which had typically been a complex and time consuming exercise. [..] **A downside of this advance in software technologies is that developers become less familiar with the mechanics of the underlying system, and as a result, can make decisions during development that have an adverse effect on the system.**" [Par07]

While high-level technologies enable developers to implement features faster, non-functional aspects are often not covered sufficiently. The above-mentioned quotation of Parsons nicely illustrates that developers generally care insufficiently about their application's performance. They tend to make decisions during development without having the complete understanding of how the underlying system of the application behaves [Par07] and often do not know the implications of a particular design on the runtime system. Performance problems are regarded as issues that have to be solved solely on lower abstraction layers. This leads to bad software quality,

non-functional requirements being broken, and thus problems at runtime, such as performance issues.

This problem is further strengthened by the lack of tools providing runtime information during development. IDE's provide a huge amount of functionality, but often only static aspects are considered. A lot of static source code analysis tools, such as *Checkstyle*[1], *PMD*[2], or *FindBugs*[3], are directly integrated into the IDE. Additional information is provided from external tools like issue trackers, version control systems, and Continuous Integration servers. But they all have in common that they focus on static aspects and lack runtime information. While various research has been done integrating those different systems, filtering and visualizing the combined data sets [BGG14a,BGG14b,Bir14,Hil14], only little research has been done in integrating runtime data into the developers environment. Most of the available tools that monitor runtime systems, such as New Relic[4] or Datadog[5], are relatively isolated tools that are not treated during development. A study of Cito *et al.*, based on qualitative interviews and a quantitative survey, has shown that developers are indeed aware of the available runtime metrics [CLFG14]. Nevertheless, the vast majority stated that they use their intuition during development, rather than consolidate those metrics, even if they are working on performance issues [CLFG14]. Although this has not been proven so far, it can be assumed that one of the main reasons for this dichotomy is that the data is not enough easily available [CLFG14].

Summarizing the discussed problems it can be said that modern, high-level software technologies, even though they improve the productivity of developers, have negative impacts on the developers awareness of non-functional aspects, because those are often tightly related to lower level technologies. The available tools typically used by developers mainly focus on static source code artifacts and mostly exclude runtime aspects. Together those circumstances result in isolating developers from runtime and especially performance aspects which is completely contradictory to the DevOps approach (see 2.1.2) which is recommended when developing cloud-based applications.

3.2 Feedback-Driven Development

As current tools lack the support of runtime data and developers do not seem to be willing to consolidate other tools to get information about the runtime behavior of the developed applications, a new approach is required to increase the developers awareness of the available metrics and overcome issues related to the lack of runtime-related knowledge.

> "If the developer won't go to the metrics, the metrics must go to the developer."[6]

The quotation above nicely puts the obvious solution in a nutshell: in order to enforce a developer to use operational data, it has to be closely integrated into his usual development environment, which is precisely the target of Feedback-Driven Development (FDD), an approach introduced in [BFKB+14] to align the engineering methods and tools for the development of cloud-based applications.

Figure 3.1 provides a simplified illustration of the suggested principle, where at the red parts constitute the elements of Feedback-Driven Development. An application is developed in an IDE and then deployed to any cloud platform. The deployment itself, whether and how Continuous

[1] http://checkstyle.sourceforge.net
[2] http://pmd.sourceforge.net
[3] http://findbugs.sourceforge.net
[4] http://newrelic.com
[5] https://www.datadoghq.com
[6] https://speakerdeck.com/citostyle/the-developers-devops-mountain?slide=11

Integration is applied, is of course very important to the lifecycle of cloud-based applications, but is not in the scope of Feedback-Driven Development and therefore not discussed in more detail in this thesis. From the time when the application is released on a productive server, Feedback-Driven Development comes into effect: the running application is continuously monitored and metrics, such as the execution time of a particular method, the CPU- or memory usage, are gathered and stored. The huge data set that accrues thereof is filtered such that only the data which is relevant for specific development activities can be sent back into the IDE of the developer. This runtime feedback can either contain raw data, aggregated values (e.g., average execution time), or any suggestions derived from that data.

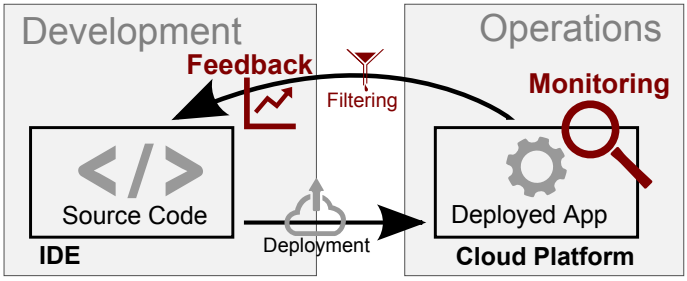


Figure 3.1: The principle of Feedback-Driven Development.

3.2.1 A Practical Scenario

This section briefly describes a contrived practical scenario to further illustrate the purpose of Feedback-Driven Development. Let's assume a development team of a software provider working on an enterprise application for a retailer of spare parts. The core module of the application is a customer relationship manamgenent (CRM) system to manage the large amount of the retailer's customers of different types (local garages, car dealers, private clients). The CRM system is already well-engineered, only maintenance work is conducted. A second module provides a web shop solution that is integrated into the CRM system and is still under active development. One big issue with the system are customer record duplicates: due to the fact that customers can be registered over different front- and backend systems, it very frequently occurs that duplicates are registered. To overcome this problem, there is a service in the CRM that allows to detect duplicates automatically. This service is executed each night and each morning an administrative backend provides a list of duplicates that should be merged manually. Observations have shown that most of the duplicate records are introduced through online registrations of customers in the web shop that are already registered in the CRM due to a previous contact with the retailer. Therefore, the development team is charged with introducing a validation in the registration process of the online shop that checks whether a respective customer record already exists. The developer taking care of this issue knows about the job triggered each night to detect duplicates and consults the respective code to see whether he can reuse some functionality. Indeed he finds the service that parses the whole data set for duplicates and re-uses it in his implementation. As he finishes the task, he integrates the code into the system and the feature goes live with the next release at the end of the month. After releasing this new version, problems with the registration process suddenly arise. A lot of visitors of the web shop report very long waiting times when creating a new account. Due to the high frequency of account-creation negative impacts to the overall performance of the web shop are recognizable. While analyzing the issue it turns out that the problem is the newly introduced duplication-check. The reused service from the nightly-job

has a relatively long execution time. So far, this has not been detected as a problem, because the service was not frequently executed and no end-users were concerned. The problem could also not be detected in the testing phase, because the data sets for the tests were too small to detect any performance impacts. The development team rolls back the feature as fast as possible and creates an issue to find another solution to the problem.

Feedback-Driven Development tooling would have been capable of detecting the issue right when the new code was introduced, because runtime information would have been available that covers the execution time of methods and the frequency of method execution. Therefore, the developer could have been warned directly in the IDE when she invoked the long running duplication detection service in the frequently executed customer creation procedure. This illustrates how developers can profit from Feedback-Driven Development through shorter feedback loops. Problems can be identified very early and are not rolled out to productive systems. Thereby costs and time can be saved and end-users of the system are not affected.

3.2.2 Goals

The overall goal of Feedback-Driven Development is to provide the necessary techniques to overcome the problems discussed in Section 3.1 and to enable a DevOps (see Chapter 2.1.2) way of development. This includes a lot of subordinated goals discussed in the following. Operational data shall be made available to developers in an easy way that does not require additional (manual) steps to access it. This is achieved by integrating the data directly into the developers IDE, which enables taking decisions during development activities based on concrete metrics instead of assumptions. Consequently, decisions become more comprehensible and logical and therefore hard to refute, because there's much less space for interpretation [Kla12]. The decision-making process becomes much more data-driven (objective facts) and less emotional (subjective opinions) [Kla12]. Through using the measurable metrics to compare various versions of an application over time, the impact on software quality and application performance can be recognized and even predicted. This is achieved through predicting the future state of an application based on the measured metrics from the past, which is another important goal of Feedback-Driven Development.

Achieving all the mentioned goals enables development teams to shorten their development and innovation cycles, which in turn increases the agility of a team, because they can react to changes faster, which is important for the success of small as well as large projects [Ado06]. Through integrating the operational data into the development environment, a shared tooling for development and operations aspects is provided. This enables the evolution of DevOps processes along the software development lifecycle [Hü12].

3.2.3 FDD and Continuous Delivery

In Section 2.1.3 Continuous Delivery has been introduced as an extension of Continuous Integration that allows to release single changes very fast by going through several subsequent steps in the so called Deployment Pipeline [Mar13a]. An important aspect of Continuous Delivery is that developers retrieve feedback from each step in the pipeline, which is illustrated in Figure 2.1. Taking into account Feedback-Driven Development, the Continuous Delivery Pipeline can be extended with an additional feedback loop, providing feedback from the runtime system, which is the last step in the Continuous Delivery Pipeline. Figure 3.2 shows an adapted version of the pipeline from [Che] with Feedback-Driven-Development being integrated.

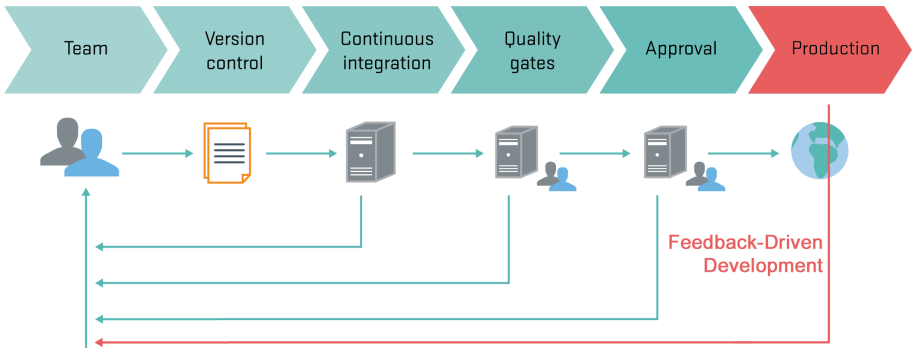


Figure 3.2: The Continuous Delivery Pipeline extended with FDD (adapted from [Che]).

According to the modified Continuous Delivery Pipeline illustrated in Figure 3.2 we can look at Feedback-Driven Development as a major extension of Continuous Delivery by providing an additional feedback loop that takes into account operational data. It effectively integrates the production system into the pipeline.

3.3 Introducing a Concept for FDD

After introducing the basic idea of Feedback-Driven Development and positioning it inside the software development lifecycle, this section describes a concept of what a system that provides FDD tooling looks like.

3.3.1 Characteristics of an FDD System

From the problem analyis (see Section 3.1) and the discussed goals (see Section 3.2.2) we can derive some basic characteristics that a system implementing Feedback-Driven Development has to exhibit in order to provide the discussed benefits. We can also look at those characteristics as the high-level requirements for such a system:

- **IDE-Integration:** One of the discussed goals of FDD is to bring the metrics closer to the developer in order to solve the problem that developers do not consolidate the available runtime data [CLFG14]. The logical way of targeting this issue is to integrate the provided runtime feedback directly into the IDE, because this is where developers typically are during their daily work. If developers would have to switch the tool in order to receive the runtime feedback, we can assume that they would make less use of it. Therefore, this characteristic is absolutely crucial in order to achieve the goals of FDD.

- **Filtering and Aggregation:** When monitoring the runtime behavior of an application, a huge amount of data can be collected, metrics of different types can be measured. It makes sense to monitor comprehensively and to store all gathered data. But it doesn't make sense to directly expose the gathered data to the developer. Rather, the data should be filtered and metrics should be aggregated in order to not flood developers with information.

- **Mapping to Source Code:** The feedback that developers receive from operational data has to be directly mappable to concrete source code artifacts. Feedback that is not allocatable to specific places in source code is of less value to the developers.

- **Minimal setup costs:** The activities required to make an existing project FDD-ready should be held as minimal as possible. The installation of the tooling for an individual developer needs to be easy and fast. Existing software projects should be FDD-ready without major adoptions. Apart from minimal configurations no additional changes should be required.

3.3.2 From Runtime Data to Meaningful Feedback

To meet the requirement of filtering and aggregation we suggest an FDD system working very similarly to a data-driven decision support system [Pow02], which is a special type of decision support system (DSS) that takes a large amount of data as input [Pow02] for its analysis. The output of a DSS is transformed data from which decisions can be generated [Pow02]. Figure 3.3 illustrates this process for an FDD system. The input for the analysis is the gathered runtime data of the application. In a first step, this large amount of data is filtered and aggregated according to specific use cases resulting in a smaller data set. In a second phase this smaller data set is combined with the corresponding static code artifacts. The analysis of the combined data set then results in the transformed output data in form of *advices*.

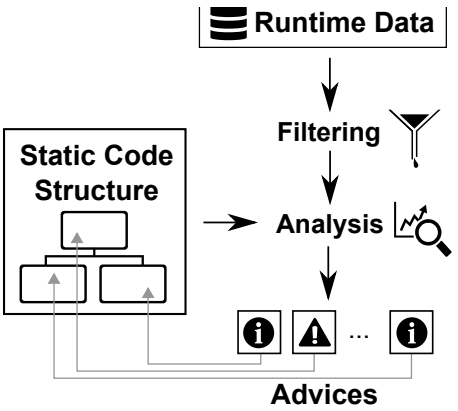


Figure 3.3: From raw data to Advices.

Advices

For the output of the system visualized in Figure 3.3 we introduce the term *advice*. An advice is one particular outcome of the filter, aggregation, and anaylsis steps of an FDD system. An advice has the following two basic properties:

- **Based on operational data:** An advice is always elicited from the analysis of runtime data. As shown in Figure 3.3, the analysis can be supported by static code analysis. Nevertheless, outcomes elicited from pure static code analyis are not called advices. Those results do not require an FDD system and are therefore not in the scope of our approach.

- **Referable to code artifacts:** Each advice is precisely allocatable to a particluar element in the source code (visualized by the gray arrows in Figure 3.3). This element can be of arbitrary granularity: it can be a class, a method, a code block (e.g., a for-loop), or even a single line of code (e.g., a method invocation). Advices do not comprise architectural aspects that refer to whole modules, layers, or a system rather than one particular code-level element.

Taxonomy of Advices

The definition of an advice is kept very generic. Despite its two basic properties, it does not provide any restriction about the information it gives to the developer. This makes advices adaptable to different environments with different technologies (e.g., Java Platform, .NET Framework) as well as different programming paradigms (e.g., object-oriented programming, functional programming). Nevertheless, advices can be classified into two basic characteristics, which are *informative* and *warning*. For each of these generic advices types, a number of concrete sub-types exists depending on the particular programming paradigm. Figure 3.4 introduces an overview of the advice taxonomy proposed for an object-oriented system. Each individual type is discussed in the following paragraphs, including mockups visualizing how the advices can be displayed in an IDE.

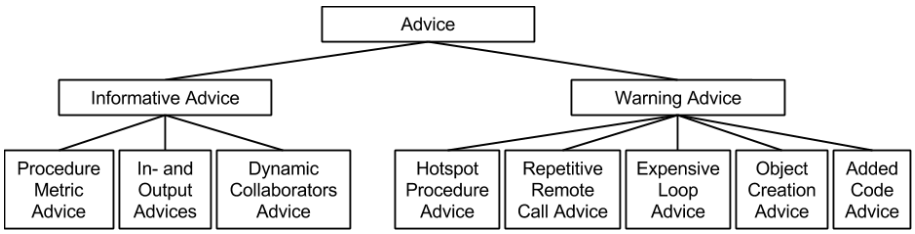


Figure 3.4: Taxonomy of advices for an object-oriented system.

Informative advices. Informative advices provide the developer with any runtime information about a source code element that can be used during any kind of development activity. The purpose of informative advices is to extend the static (design-time) information that is typically available in IDE's with operational data. Informative advices basically cover the same information that debugging tools provide, but with the essential differences that the data is gathered from the productive system and over a long time period instead of one specific execution. Figure 3.5 visualizes possible informative advices and how they can be displayed in an IDE. A possible type of advice is the *average method execution time*. This can be of interest to the developer in many different situations, for example to compare different algorithms solving a particular task. Another example of useful runtime information is a list of the dynamic collaborators of a particular class or method (see Figure 3.5). IDE's typically provide functionality to find all the places in the source code where a method or a class is referenced. This is only partially useful, because due to polymorphism, the accurate type used at runtime cannot be determined without operational data. Displaying the dynamic callers and callees of a method is therefore a useful extension of the existing IDE feature. On class level, this information can be especially useful if dependency injection [Mar04] is used, in order to find the concrete implementation that is injected. We suggest the following informative advices:

- **Procedure**[7] **Metric Advices**: Advices may give information about the average execution time of a procedure and the average CPU usage required for the execution. The aggregated average values can be complemented with more detailed information visualizing the two metrics on timelines to illustrate any possible trends.
- **Input and Output Advices** In static typed prgoramming languages the signature of a procedure specifies the types of the input parameter as well as the return value (output). Advices

[7] The term *procedure* is used here as generic term for any kind of sub-routines, such as functions, methods, or constructors.

extend this information through providing information about the values of those types at runtime. Gaining knowledge about the actual input and output of a procedure at runtime allows a developer to improve the source code accordingly. In case of numeric types, the average value can be of interest. For textual parameters, the average length may be relevant. In case of polymorphic types, the frequency of particular concrete types can be interesting. For value objects [Fow12] it can be interesting if the same value is returned very frequently. In this case, a possible improvement may be to cache the value instead of recalculating and regenerating it each time.

- **Dynamic Collaborator Advices:** As described above, advices can provide information about the dynamic collaborators in polymorphic systems. Two advices comprise the dynamic collaborators of a procedure: One contains its callers and the second its callees, which both has been proven as valuable pieces of information for developers [RGN07, BRB10]. In the case of a class the dynamic collaborators correlate to the dependencies of the class. These dependencies are typically injected at runtime through setters, a constructor, or a dependency injection framework. Therefore, the concrete types of the dependencies are not known at design-time. Dynamic collaborator advices instead are capable of giving the developer information about the concrete collaborators of a class.

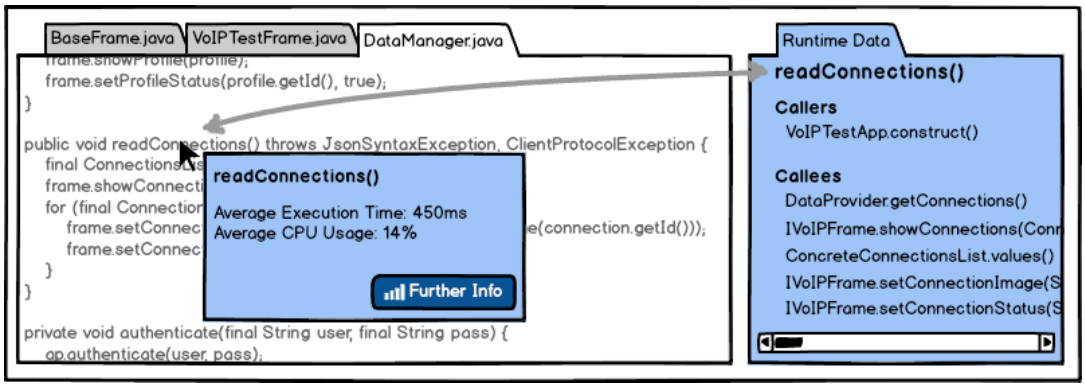


Figure 3.5: Mockup visualizing a part of an IDE displaying informative advices.

Figure 3.5 furthermore illustrates the UI elements we suggest in order to display the advices in the IDE. The first possibility is a hover appearing directly in the source code editor. Most IDE's already use this principle for their own hovers. Therefore, we suggest to make the hover appear on performing a keyboard shortcut after hovering the respective element in the source code. Hovers are useful to display advices that summarize a large amount of data and possibly provide a link to a separate view providing more detailed advice. Taking the example of the *average method execution time* from the mockup in Figure 3.5, the linked view could provide a plot visualizing the execution times of the method on a timeline.

Figure 3.5 also illustrates a second possibility to display informative advices by using an external view beside the source code editor. This is useful for advices displaying comprehensive information about a large amount of runtime data. Such views should always be synchronized with the editor, which means that the information displayed in the view is always related to the source code element in the editor where the cursor is located.

Warning advices. A warning advice points to source code elements that are assumed to be critical for the performance of the application. A simple example is a long-running method that is executed very frequently. Warning advices provide a meaningful summary of the operational data that hypothesizes the performance issues and, optionally, suggests a solution to resolve it. Warning advices therefore help developers to resolve existing performance problems or to even detect them in advance. Figure 3.6 illustrates our suggestion of how to display warning advices in the IDE: the corresponding location in the source code is somehow marked to indicate a warning at this position. This concept is known from other warnings in IDE's, such as compiler errors. Hovering the warning shall open a hover providing detailed information about the advice. Furthermore, we suggest an additional view outside the source code editor that summarizes all the warning advices and allows to navigate to the appropriate source code artifact. This concept is also well known from regular IDE warnings. Using regular IDE concepts prohibits developers from learning new things. Suggested warning advices are:

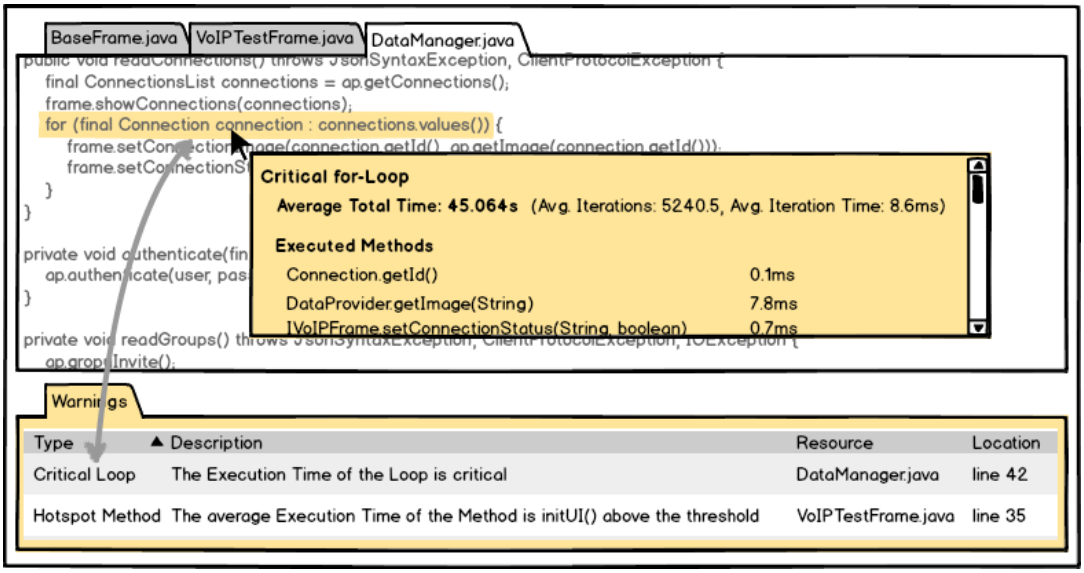


Figure 3.6: Mockup visualizing a part of an IDE displaying warning advices.

- **Hotspot Procedure Advices**: A hotspot procedure is one that is potentially critical to the performance of the application. The simplest case to define a procedure as a hotspot is if its execution time is above a pre-defined threshold. Developers can take care of such methods and try to improve their performance. Other criteria of high interest are the frequency of execution of particular procedures [BRB10, RHV+09a] or the percentage of executions per user request [RGN07]. This helps to identify the methods that are critical to performance due to their frequent execution. A short, but frequently executed method may have more impact on the overall performance than one that exceeds a threshold but is executed only a few times.

- **Repetitive Remote Calls Advices:** In distributed systems remote calls are required to communicate and pass data between different nodes. As discussed in Section 2.2.4, remote calls are typically expensive calls and hence relevant to performance. A repetitive remote call advice indicates multiple remote calls that are often executed after each other. The assumption is that those calls can be combined into one call by refactoring the system. Hence, this

kind of advice targets the detection of the *too many remote calls* and the *N+1 select problem* anti patterns (see Section 2.2.4).

- **Expensive Loop Advices:** This type of advices covers the exposure of runtime data on (nested) loops that are very expensive (i.e., long execution time) and therefore critical to the performance. The time required for one iteration, the number of iterations as well as the total execution time can be of interest to the developer. Furthermore, the advice can contain informative advices about the methods executed in the loop body. Expensive loop advices help the developer to detect the anti pattern of the same name discussed in 2.2.5.
- **Object Creation Advices:** Section 2.2.2 illustrates how expensive the creation of objects can be and explains the *excessive dynamic allocation* anti pattern addressing the repetitive creation and destruction of similar objects. Object creation advices indicate locations in the source code where this anti pattern arises by comparing the objects created at runtime. Furthermore the detection of the *aggressive loading of entities* anti pattern is possible through analyzing which kind of procedures and fields of a method are used how frequently. This helps developers to improve the code through introducing lazy loading for particular parts of objects.
- **Added Code Advices:** Even the smallest code changes can have significant impacts to the performance of an application [NNH+14]. Added code advices target those code changes that have a high probability of resulting in such performance impacts. The scenario described in Section 3.2.1 is a good example of such a code change. A respective warning can be shown in the IDE as soon as a developer has introduced such a code change.

In summary, we can state that informative advices extend the information available to a developer helping to find a concrete design for source code artifacts of different granularities (e.g., classes, procedures), while warning advices help to find weaknesses and possible performance problems in the chosen design.

3.3.3 Application of Statistical Methods

An interesting aspect of the FDD approach is the possibility to provide the developer with predictive analyses about the performance of a system using the available runtime data. This section describes how the statistical methods introduced in Section 2.3 can be applied for this purpose.

Predictive Analysis with the Moving Average Model

We propose to use the moving average (MA) model (see Section 2.3.2) as a base for predictive analyses of an application as part of an FDD system. The MA can be used to estimate the future values of the execution time of a method, its CPU utilization, or memory usage. Let us assume a procedure m with measured execution times $\{et_{m_1}, ..., et_{m_n}\}$. Derived from Equation 2.3, we can predict the execution time of m as given in Equation 3.1

$$et_{m_p} = \frac{\sum_{i=0}^{w-1} et_{m_{t-i}}}{w} + \epsilon \tag{3.1}$$

The result et_{m_p} is the estimated execution time of m with a statistical error ϵ. This gives us a simple model for the performance of an application based on the measured operational data in

the past. The predicted values of CPU utilization and memory usage of procedures can be defined analogously.

For many performance problems, procedures are a reasonable level of granularity for the measurement of operational data (see Section 2.2). However, as discussed in Section 2.2.5, performance problems are often related to loops. We therefore further propose to make predictive analyses of the execution time of loops. To do so, we first have to define the execution time of a loop. Taking the example loop of Listing 3.1, the execution time of an iteration is the sum of the execution times of the methods `a()`, `b()`, and `c()`. Multiplying the sum with the number of executions, which in this case is determined by the size of the collection returned by `getItems()`, gives the execution time of the loop. The execution time required to check the termination condition of the loop is negligible.

```
1 for (Object item : getItems()) {
2   a(b(item));
3   c(item);
4 }
```

Listing 3.1: A simple loop example (Java).

Generally, the total loop execution time et_l of a loop l can be defined as the sum of the execution times $\{et_{m_1}, ..., et_{m_n}\}$ of all procedures $\{m_1, ..., m_n\}$ executed in the loop-body multiplied by the number of iterations i of the loop (see Equation 3.2).

$$et_l = \sum_{a=1}^{n-1} et_{m_a} \times i \tag{3.2}$$

We previously proposed to use the moving average to predict the execution time of a loop (see Equation 3.1). If the loop execution time is defined as shown in Equation 3.2, we can multiply the sum of the moving averages of all the methods inside the loop with the predicted number of iterations to predict the execution time of the loop. To predict the number of iterations, the moving average of the iterations can be calculated analogously. Integrating these moving averages into Equation 3.2 results in Equation 3.3. The first sigma sign sums up the first bracket, which calculates the MA for each method. This is multiplied with the second bracket, which calculates the MA of the number of iterations. w_m and w_t are the window sizes of the moving averages of the method execution times and the number of iterations respectively. As for every predictive analysis, there is a statistical error ϵ.

$$et_{l_p} = \sum_{a=1}^{n-1} \left(\frac{\sum_{b=0}^{w_m-1} et_{m_{a_b}}}{w_m} \right) \times \left(\frac{\sum_{c=0}^{w_i-1} i_c}{w_i} \right) + \epsilon \tag{3.3}$$

The introduced predictions for method- and loop execution times allows an FDD system to make reasonable assumptions about the evolution of the performance of an application. The developer can be kept up to date with respective information in form of advices.

Using Change Point Analysis in FDD

As illustrated in Section 2.3.3, Change Point Analysis can be used to detect significant changes in the evolution of the performance of procedures. We propose to use this information in FDD and combine it with release information in order to automatically detect releases that introduce or resolve major performance problems. If newly arising performance problems can be mapped to code releases, the developer is able to rapidly find the responsible code change containing the source of the problem. This is especially valuable if Continuous Delivery is applied (see Section 2.1.3) and the FDD system is integrated into the Continuous Delivery process (see Section 3.2.3). The idea of using Change Point Analysis to conduct analyis in performance management has already be proven by Cito *et al.* in [CSLD14].

3.3.4 Proposed Conceptual System Architecture

Based on the aspects discussed so far we propose an architecture for an FDD system that meets the characteristics described in Section 3.3.1 with the purpose of achieving the goals mentioned in Section 3.2.2. The focus lays on illustrating the necessary components and their interaction required by each system to implement the FDD approach. The conceptual architecture is visualized in Figure 3.7. It contains three different types of elements. The *Main Components* are the actual building blocks which establish the FDD system. The *Environmental Components* encompass the (existing) environment the FDD system is embedded into. Finally, the *Target Application* corresponds to any application that is developed and deployed in the illustrated environment and shall benefit from the capabilities of the FDD system.

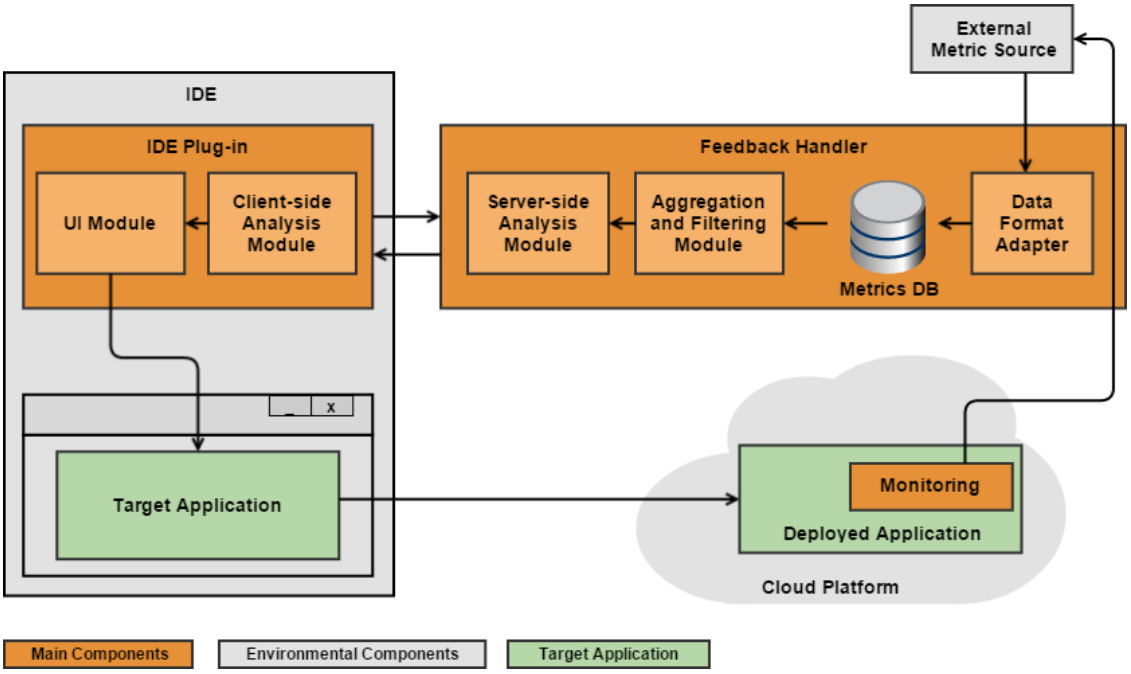


Figure 3.7: The proposed architecture for an FDD system.

Environmental Components and Target Application

The following list describes the characteristics of the environmental components and the Target Application. The main components are described individually in more detail in the subsequent sections.

- **IDE:** The Integrated Development Environment is the starting point of the FDD process. It is the place, where the developer works on the source code of a target application. The integration of an FDD system into the IDE is a crucial requirement.
- **Cloud Platform:** This is the platform where the application is deployed to run in production. It can be an arbitrary platform of any provider and is totally independent of the components of the FDD system.
- **Target Application:** The Target Application represents the application the developers are working on while using the FDD system. It is developed in the IDE and deployed onto the cloud platform where it is used in production. The exact deployment process, including Continuous Integration, is not relevant for the architecture and therefore not further visualized.
- **External Metric Source:** This can be any external source that handles incoming runtime data and calculates metrics. Examples of such providers are New Relic[8] or Kieker[9]. The metric source can also be part of the Feedback Handler component itself if no external source is used.

Monitoring Component

The Monitoring Component is responsible for gathering the runtime data. It is somehow injected into the program flow of the Target Application. How this is achieved depends on the concrete technologies of the Target Application. For machine language, the injection typically has to be done during compilation. For interpreted languages there are other possibilities such as bytecode modifications. Conceptually, there are no restrictions on how to conduct the injection, the only condition that has to be obeyed is that the developer doesn't have to take care of the injection. Furthermore, the source code or even the design of the Target Application may not be influenced by the Monitoring Component.

The Monitoring Component is able to track the program flow during execution and measure data such as the execution time and the CPU usage. The exact extent of the gathered data depends on the types of advices (see Chapter 3.3.2) that are supported. At least the execution time of the executed program elements (typically procedures) should be measured, which is the basis for most of the proposed advices. The Monitoring Component regularly sends the collected data to a metric source, which can either be an external provider (as shown in Figure 3.7) or an internal component of the Feedback Handler. We propose to provide an Application Programming Interface (API) on the Feedback Handler to send metrics in both cases. The Feedback Handler then either treats the data itself or redirects it to an external source. This decouples the Monitoring Component from the concrete API's of the External Metric Source providers and hence makes them exchangeable more easily.

[8] http://newrelic.com
[9] http://kieker-monitoring.net

Feedback Handler Component

The Feedback Handler is the core component of the FDD system. Together with the IDE plug-in it spans up a client-server architecture where the Feedback Handler acts as server and processes requests from the IDE plug-in. The Feedback Handler receives the runtime data either directly from the Monitoring Component or from an External Metric Source provider. The data fetched from the external metric sources will most likely exhibit different formats. The Feedback Handler itself has its own representation of the runtime data and should be as independent as possible from the third party formats. A Data Format Adapter therefore takes the responsibility of converting the different formats from the metric source providers into the common one known and shared by the other components in the FDD system. After converting the data into the common format, it is stored in the metrics DB, which is also part of the Feedback Handler. The metrics DB is the data source for the Aggregation and Filtering and the Server-side Analysis Module. Those two modules operate on the data from the Metrics DB in order to serve client requests from the IDE plug-in. They provide the implementation for the concept described in Section 3.3.2.

IDE Plug-in Component

The IDE plug-in is the third main component of our proposed FDD system architecture. It extends an existing IDE (e.g., Eclipse[10], IntelliJ[11], or Cloud9[12]) with the proposed FDD features. It gets the partially analyzed runtime data from the Feedback Handler. The Client-side Analysis Module combines this data with the available static data from the source code artifacts. The outcome of this process is a collection of advices (see Section 3.3.2). The UI module gets this collection of advices and is responsible for visualizing them in an appropriate manner inside the IDE (see Figures 3.5 and 3.6). This closes the feedback loop by bringing back the runtime aspects to the Target Application and consequently to the developer. The concrete visualization of the advices should be assimilated to the concepts and *look and feel* of respective IDE and should be tightly integrated into the existing views, such as source code editors.

3.3.5 Summary of the Approach

From the practical scenario illustrated in Section 3.2.1 and the goals discussed in Section 3.2.2, the main characteristics an FDD system have been derived. To get from raw operational data to valuable feedback for the developer, the concept of advices has been introduced. It has been shown how advices are elicited from runtime data and how they are linked with the static source code elements in the IDE. Advices have been further classified into informative and warning advices and for both types a concrete mockup has been shown giving a suggestion on how to visualize the respective information in the IDE. The statistic method of the moving average has been treated and it has been shown how it can be used to predict the future performance of applications on procedural level. The Change Point Analysis has been suggested to automatically detect significant performance variations and it has been proposed to link these information to the release history of the application in order to detect problematic releases. Finally, a conceptual architecture for an FDD system has been illustrated providing an overview of the environment and a description of the main components.

[10] http://www.eclipse.org
[11] https://www.jetbrains.com/idea
[12] https://c9.io

3.4 Implementing an FDD system

This section describes the implementation of PerformanceHat, a concrete prototype of an FDD system that has been developed in order to verify the concept discussed so far. While the conceptual approach has been introduced independently of any concrete technologies, technological decisions have to be made in order to implement such a system. Due to the fact that an FDD system is integrated very closely into a development environment, it is not possible to implement a generic application for multiple platforms, since the tight integration is one of the basic characteristics of an FDD system (see Section 3.3.1). PerformanceHat therefore is established for the Java platform and the Eclipse IDE. Nevertheless, the architecture has been designed to be adaptable to other platforms. Modules that are independent of the concrete environment have been designed to be reusable if the application is extended to support other platforms. This section first illustrates the concrete system architecture, describes the particular system components and finally explains some important implementation details.

3.4.1 Architecture

Figure 3.8 visualizes the architectural components of PerformanceHat, derived from the conceptual architecture treated in Section 3.3.4. While the conceptual architecture visualized the system landscape, the system architecture focuses on the developed components. The system is designed to support target applications written in Java and the IDE component is built on top of the Eclipse IDE. The system components themselves are all implemented in Java, too. The following list provides a short overview of the components and their functionality. A detailed discussion about the most important implementation details follows in the subsequent sections.

- **Common Component:** The Common component, which was not part of the conceptual architecture, contains all the modules that are shared amongst the other components. It is not deployed as standalone component, but rather as part of all the other components. It contains the shared runtime model the system uses to deal with operational data, basic functionality for the communication between the components, a number of data transfer object (DTO), basic error handling capabilities, and some utility classes.

- **Monitoring Component:** The Monitoring component is designed to be easily integrated into target applications[13] and is kept as thin as possible. It consists of a Data Gathering module that is responsible for instrumenting the source code of the target application in order to collect runtime data and a Data Transmission module that takes over the responsibility of sending the gathered data to the Feedback Handler component.

- **Feedback Handler Component:** The Feedback Handler is implemented as a web application based on the Spring Framework[14]. Most of its modules have already been described in Section 3.3.4, because they are tightly related to the concept. The Feedback Handler provides a RESTful HTTP API that enables the communication with the Monitoring component and the Eclipse plug-in.

- **Eclipse Plug-in:** The IDE component of the FDD system is implemented as a plug-in for the Eclipse IDE. The plug-in is based upon the Eclipse core components and the Eclipse JDT[15]. The plug-in contains a REST-Client that communicates with the Feedback Handler to receive runtime data. A *Resources Extension* module is designed to deal with Eclipse resources

[13] The target application is the application that makes use of the FDD system, see Section 3.3.4
[14] http://projects.spring.io/spring-framework
[15] https://eclipse.org/jdt

(see Section 3.8.1). The *Feedback Builder* is integrated into Eclipse's builder mechanism (see Section 3.8.3) in order to distribute the elicited advices which are visualized by the *Marker and Hover* module (see Section 3.8.5).

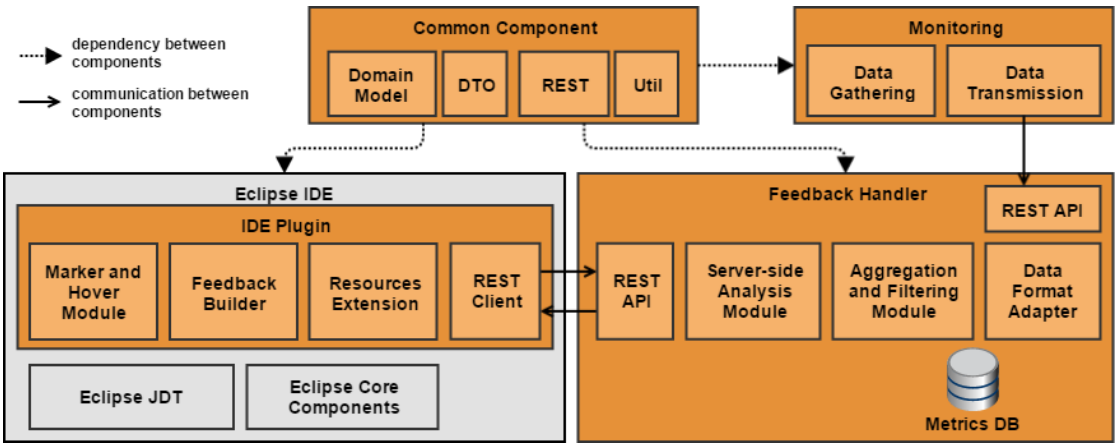


Figure 3.8: Architecture of the FDD system.

3.5 Common Component

The Common component contains all the code that is shared amongst the other three components in order to avoid duplicate source code along multiple components. It therefore contains miscellaneous modules and has no inherent structure. The main modules of the Common component contain a shared runtime data model and data transfer objects. The runtime data model is a representation of the gathered runtime data. It constitutes the domain of the FDD system and is explicitly discussed in Section 3.5.1. The data transfer object (DTO) module contains a collection of plain old Java object (POJO)'s[16] used to transfer domain objects as well as other data between the three other components. DTO's are therefore a serializable, often aggregated, representation of the domain objects. The data transmission between the components is achieved through RESTful HTTP API's. For this purpose the Common component provides an abstract base implementation of a REST client that is based on the Web module of the Spring Framework[17]. Additional classes from the Common components provide functionality to create HTTP headers and request bodies to construct HTTP requests. Corresponding generic error handling for the REST communication is also provided by the Common component. A single exception type is introduced that exhibits an error type property allowing to explicitly specify different error types related to the HTTP transmission. This allows the client component of the communication to handle different types of errors by catching only one single exception type. Other concrete exception types, especially those from the underlying frameworks, are decoupled from the client.

Finally, the Common component includes a utility module containing different classes that provide a common functionality for various issues such as the dealing with date/time data types and strings or the creation of meaningful hash codes.

[16] Plain Old Java Object: an *ordinary* Java object independent of any frameworks, libraries, or conventions. See http://www.martinfowler.com/bliki/POJO.html.
[17] http://projects.spring.io/spring-framework

3.5.1 Runtime Data Model

Operational data comprise all the dynamic data gathered through application monitoring. It is a very complex domain that can be regarded from different perspectives upon different levels of granularity. In order to deal with operational data, the components of PerformanceHat require a shared view of how the world of operational data looks like. This view is provided by the *Runtime Data Model*, a domain model for operational data. Figure 3.9 visualizes its entities.

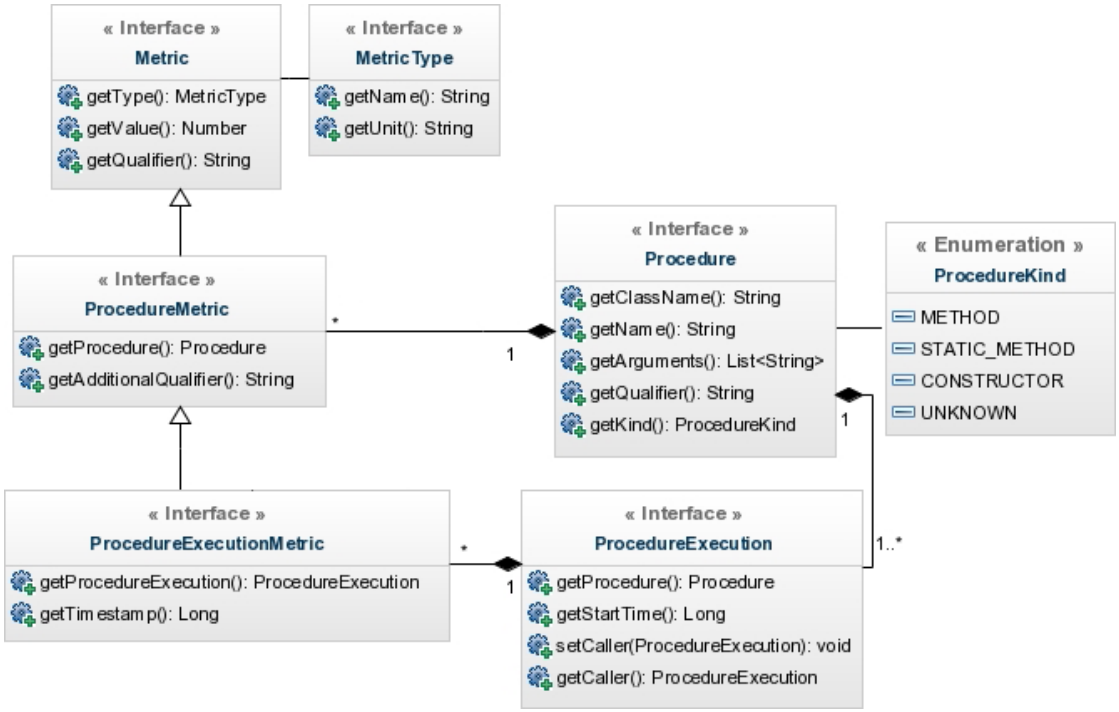


Figure 3.9: The domain model of the FDD system.

The runtime data model can be divided into two main parts, where the first part, composing the right part of the diagram, covers the call trace. The call trace is the tree of executed Java procedures. `Procedure` is the super-type of all sub-constructs of a class. In Java, these are methods and constructors. A `Procedure` is uniquely identifiable through its qualifier property (`getQualifier()`) that is compounded out of its main properties which are the name, its list of parameters, and the qualified name of the class the procedure belongs to. Additionally, the procedure has an attribute `ProcedureKind` specifying the type of procedure. In Java, this is either *method*, *static method*, or *constructor*.

While the `Procedure` only focuses on the static part of a procedure, the `ProcedureExecution` is introduced to cover the dynamic part that actually happens at runtime. A `ProcedureExecution` represents the execution of its associated `Procedure` at any point in time. It contains a start time and a caller, which refers to the `ProcedureExecution` that has called the current one. This structure is sufficient to model the whole execution trace of an application at runtime.

The second part of the domain model covers metrics. The `Metric` interface, which is the central entity of this part, represents any metric measured at any point in time. Instead of inheriting the `Metric` interface for all possible kinds of metrics (e.g., execution time, CPU usage),

a `MetricType` interface is introduced that specifies the type of a `Metric`. A `MetricType` is composed of a name and the proper unit (e.g., milliseconds). Furthermore, a metric has attributes containing its measured value and a qualifier that allows to store additional context of the metric. The `Metric` interface is completely independent of the call trace part of the model. To combine metrics with the call trace, the `ProcedureMetric` and `ProcedureExecutionMetric` types are introduced. The `ProcedureMetric` is a specialization of a generic metric which is directly related to a `Procedure`, for example the average CPU usage the procedure requires. The base qualifier of a `ProcedureMetric` metric is consequently identical to the qualifier of the related *Procedure*. To further distinguish between different metrics of the same type related to the same *Procedure*, an additional qualifier is introduced that is appended to the base qualifier. A `ProcedureExecutionMetric` is a continuative specialication of the `ProcedureMetric` that is analogously related to a `ProcedureExecution`. The execution time of a `ProcedureExecution` is an example of such a metric.

This design provides a lot of flexibility. The basic `Metric` interface is kept very generic and allows the representation of any kind of possible runtime metrics, which makes the FDD system flexible and adaptable to concrete types of target applications. Despite the metrics that are common in each application, domain-specific metrics can be introduced per target application. For example, in case of a voice over IP or a video streaming application, it could be valuable to introduce metrics such as *jitter* to measure some Quality of Service (QoS) aspects. Beside the high flexibility of the `Metric` interface, its specialized sub-types provide a tight integration into the call trace part of the model, which increases the cohesion of the model and makes the association of metrics to source code constructs very easy.

Implementation of the Model

Besides interfaces visualized in Figure 3.9, the Common component also provides an implementation layer of the domain model. Abstract classes provide basic implementations of all the domain entities and a second layer provides concrete default implementations. Figure 3.10 shows the implementation layers on the example of `ProcedureExecution` (the other types are omitted for improved clarity). The other components can either use the abstract base implementations or the default implementations to implement their own extension classes on top of it. The DTO class in Figure 3.10 (`ProcedureExecutionDto`) for example expands the default implementation with serialization meta-information, while the database-specific implementation (`DbProcedureExecutionImpl`) adds some information about the database mapping. The domain model itself, including the default implementation, is therefore completely independent of any concrete framework or library, those aspects are introduced in the subclasses.

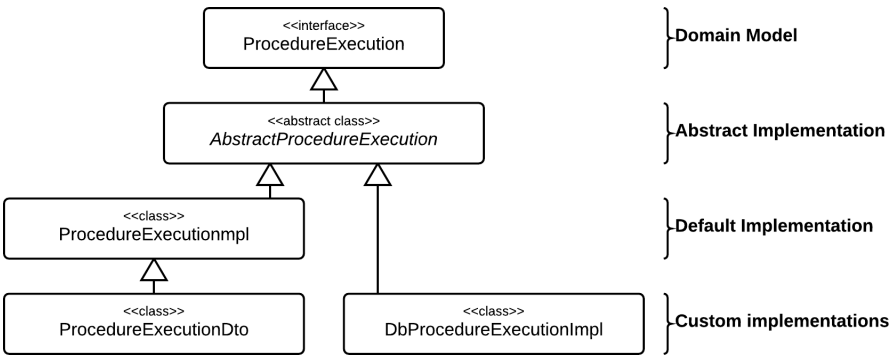


Figure 3.10: The layers of the domain model entities.

3.6 Monitoring Component

This section describes the most important implementation aspects of the Monitoring component, which constitutes the part of the FDD system that is capable of monitoring the target application during runtime.

3.6.1 Requirements

For being able to monitor the target application, the Monitoring component has to be injected into the target application's source code somehow. This process has to be undertaken each time the code of the target application is rebuilt. In order to meet an FDD systems basic requirement of minimal setup costs (see Section 3.3.1), the initial costs of integrating the Monitoring component into the target application have to be as low as possible. Furthermore, the developer should not be involved in the recurring step of injecting the Monitoring component after she did the initial setup. These are the most important requirements.

Another important requirement is to keep the monitoring overhead as low as possible. The monitoring overhead is the additional time that is required to execute the target application due to the execution of the Monitoring components code. Depending on the granularity of the source code elements that are monitored, the Monitoring component can compose a significant percentage of the execution time and therefore has to be implemented very efficiently.

3.6.2 Instrumenting Java Applications

In order to inject the code of the Monitoring component, the target application's source code has to be somehow instrumented. When working with Java, there are two possibilities to achieve that: Java bytecode instrumentation and Aspect-Oriented Programming (AOP). Bytecode instrumentation is a technique to modify the interpreted bytecode of virtual environments [BHM07]. Since JDK 1.5, Java provides the Java bytecode instrumentation API that allows to augment the bytecode of a class using a Java agent[18] as soon as the class is loaded by the classloader of the Java Virtual Machine (JVM) [BHM07].

Aspect-Oriented Programming is a paradigm that targets the modularization of crosscutting concerns in software design [KH01]. An aspect is a crosscutting concern that is executed at specific points in the application execution (join points). AspectJ[19] is a Java library that has become the de facto standard for AOP in Java applications. Using AspectJ, the aspects can either be weaved into an application during compilation, as post-compilation action or at load-time of the class[20]. The load-time weaving is based on bytecode instrumentation and therefore requires a Java agent to be executed.

When designing the Monitoring component, both alternatives, Java bytecode instrumentation and Aspect-Oriented Programming (AOP) with AspectJ, were considered. Early prototypes were implemented for both of them in order to become more familiar with the technologies and be able to verify the advantages and disadvantages. While the main advantage of AOP and AspectJ is its easy and higher-level API, bytecode instrumentation enables more flexibility. AOP is restricted to the bounds of methods and fields, which means that aspects can only be weaved before, after, or instead of a method invocation or a field access. With AspectJ, it is not possible to, for example, weave an aspect before or after a loop execution[21]. Bytecode instrumentation on the

[18] http://www.javabeat.net/introduction-to-java-agents

[19] https://eclipse.org/aspectj

[20] https://eclipse.org/aspectj/doc/next/devguide/ltw.html

[21] The issue of a loop pointcut in AspectJ is also discussed in the web: http://dev.eclipse.org/mhonarc/lists/aspectj-users/msg00929.html

other hand allows to add code anywhere in the bytecode. However, working on bytecode level to find source code constructs such as loops is very complex and requires deep understanding of Java bytecode. There are various discussions on the web providing partial approaches for such issues[22], but no reliable approach could be found. Bytecode instrumentation libraries, such as ASM[23], Serp[24], or Apache Commons BCEL[25], hide such bytecode-related issues from the developer by providing a higher-level API allowing to instrument Java source code statements, but exhibit similar limitations as AOP. The big advantage of AspectJ over bytecode instrumentation is the ability of compile- or post-compile weaving. This makes the code instrumentation a part of the build-process, no effort is required at runtime and most importantly there is no need to start a Java agent. This is a considerable benefit, because starting a Java agent couples the FDD system with the runtime environment the target application is embedded into, because a Java agent can only be started on the startup of the Java Virtual Machine (JVM). If the target application is a standalone application, this problem is reduced to considering starting the Java agent together with the application. The only impact, therefore, is a slight increase of the setups costs to start the Java agent, which would be tolerable according to the requirements (see Section 3.6.1). However, FDD is typically used in cloud computing environments where standalone applications are rarely the case. Assuming a Java web server, the JVM is started together with the server rather than on application deployments. Hence, the start of the Java agent is decoupled from the application startup and depending on the level of access of the used cloud service it can be difficult to define hooks on the server startup. Therefore, load-time instrumentation breaks the requirements of low setup costs (see Section 3.6.1) and using AOP with AspectJ is the logical consequence. Since minimizing the monitoring overhead has also been identified as a requirement, a second major benefit of this choice is the elimination of additional monitoring overhead caused by load-time instrumentation.

3.6.3 Setting up the Target Application

As discussed in the previous section, the compile-time weaving alternative of AspectJ is used to inject the code of the Monitoring component into the target application. This reduces the setup costs to a minimum. Being a Maven[26] project is the only precondition a target application has to meet in order to integrate the Monitoring component. The Monitoring component is delivered as a Maven dependency, too. Hence, the code can simply be integrated as managed dependency in the target application's Maven project object model (POM). Additionaly the *build* has to be filled with some configuration that integrates the weaving into the compilation step using the AspectJ Maven plug-in[27]. The only additional thing that is required, in order to enable the communication with the Feedback Handler, is a simple configuration file (config.properties) containing the *ID* and the *Authentication Token* of the target application. This information is received when registering the target application at the Feedback Handler. The *PerformanceHat User Guide*, which can be found in Appendix C of this thesis, provides a step-by-step instruction of how to set up a target application including concrete examples of the required configurations.

[22] There is various discussion in the web about the problem of detecting loop constructs in Java bytecode: http://cory.li/bytecode-hacking, http://stackoverflow.com/questions/6792305/identify-loops-in-java-byte-code
[23] http://asm.ow2.org
[24] http://serp.sourceforge.net
[25] http://commons.apache.org/proper/commons-bcel
[26] http://maven.apache.org
[27] http://mojo.codehaus.org/aspectj-maven-plugin

3.6.4 Monitoring Aspects

In order to monitor the source code of the target application with AOP, an aspect has to be defined that allows to add monitoring code before and after each method and constructor invocation. Listing 3.2 shows the respective code. AspectJ provides an API that allows to specify aspects in pure Java, using annotations. Using the *@Pointcut* annotation, pointcuts[28] for all method calls (see *allMethodCalls*) as well as all constructor calls (see *allConstructorCalls*) are specified. An Around-advice[29] is defined (see *aroundProcedureCalls*) that executes the actual monitoring code. The regular expression that specifies at which time the advice is executed is composed of a disjunction of the two mentioned pointcuts for the method- and the constructor invocations associated with an additional pointcut that excludes all invocations of methods inside the monitoring package. This prohibits methods of the Monitoring component themselves from being woven with the advice code, which would result in an infinite recursion.

```
1  @Aspect
2  public class MonitoringAspect {
3
4    @Pointcut("call(* *.*(..))")
5    public void allMethodCalls() {}
6
7    @Pointcut("call(*.new(..))")
8    public void allConstructorCalls() {}
9
10   @Pointcut("within(eu.cloudwave.wp5.monitoring..*)")
11   public void allMonitoringPackages() {}
12
13   @Around("(allMethodCalls() || allConstructorCalls()) &&
         !allMonitoringPackages()")
14   public Object aroundProcedureCalls(final ProceedingJoinPoint joinPoint) {
15     return TracingHandler.of().execute(joinPoint);
16   }
17 }
```

Listing 3.2: The Java class defining the aspect that is used to monitor the target application (JavaDoc comments are omitted).

3.6.5 Join Point Handlers

In order to collect the runtime data, join point handlers are introduced. A join point handler is a class that executes the additional monitoring of the code at a join point. Each join point handler extends the abstract implementation `AbstractAroundJoinPointHandlerTemplate` shown in Listing 3.3. This class operates as a template for handlers of procedure[30] call join points by specifying the execution flow and providing hooks to execute specific code before and after the execution of the procedure itself. In line 3, the `execute()` method first creates an object of type `ProcedureCallJoinPoint` which is a decorator extending the `ProceedingJoinPoint` with domain-related functionality. This decorator is passed to the `before`-hook that is executed by concrete subclasses. Then, the actual procedure of the join point is called and, together with the `ProcedureCallJoinPoint`, its result is passed to the `after`-hook which is again executed by

[28] A pointcut is a group of join points, whereas a join point is any point in the execution of the program flow where an advice (code) is injected.

[29] An @Around advice acutally replaces the code of the join point with the advice. It has therefore been taken care to execute the actual join point inside the advice.

[30] We use *procedures* as superset of methods and constructors

subclasses. This design makes it easy to define new join point handlers if needed. The current join point handlers already gather extensive runtime data: the whole execution trace is recorded and for each procedure in the trace, the execution time and its CPU usage is measured. Furthermore, information about the size of collections (i.e., objects whose type inherits from the `Collection` interface of the Java standard library) is gathered. This information is sufficient to fill the domain model discussed in Section 3.5.1 with data. The gathered data is buffered locally and sent to the Feedback Handler as soon as a call trace is finished.

```
1  public abstract class AbstractAroundJoinPointHandlerTemplate {
2    public final Object execute(final ProceedingJoinPoint joinPoint) {
3      final ProcedureCallJoinPoint procedureCallJoinPoint = new
           ProcedureCallJoinPoint(joinPoint);
4      before(procedureCallJoinPoint);
5      try {
6        final Object result = joinPoint.proceed();
7        after(procedureCallJoinPoint, result);
8        return result;
9      }
10     catch (final Throwable e) {
11       e.printStackTrace();
12     }
13     return null;
14   }
15
16   protected abstract void before(ProcedureCallJoinPoint joinPoint);
17
18   protected abstract void after(ProcedureCallJoinPoint joinPoint, Object
         result);
19 }
```

Listing 3.3: The `AbstractAroundJoinPointHandlerTemplate` class specifying a template for all procedure call join point handlers (JavaDoc comments are omitted).

3.7 Feedback Handler Component

The Feedback Handler component constitutes the server-side of the FDD architecture. It is implemented as a web application based on the Spring Framework and can be deployed on any Java webserver. To communicate with the other components, it provides a RESTful HTTP API. It receives the runtime data from the Monitoring component and from external metric sources and is responsible for processing and storing this data. The Feedback Handler is capable of aggregating and filtering the stored data to expose it to the client (i.e., the IDE Plug-in).

3.7.1 Handling incoming Data

As suggested in Section 3.3.4, the Monitoring component always sends the operational data to the Feedback Handler rather than directly to external metric source providers. The Feedback Handler then has the possibility to pass the data on to an external metric source that processes and aggregates the data. The current implementation of the Feedback Handler directly stores the data in its own local storage by default. There is an experimental implementation of the data format adapter that allows to pass the data on to either NewRelic[31] or the CloudWave infrastructure[32].

[31] New Relic is an industrial software monitoring service: http://newrelic.com
[32] For more information about the CloudWave project see Section 3.9

3.7.2 Data Storage

The operational data one received either directly from the Monitoring component or from any external metric source is stored on the Feedback Handler in a mongoDB[33] database. Several reasons argue for the usage of this database system. One important aspect is high flexibility, because the domain model is designed to be very flexible and extendable (see Section 3.5.1). The current implementation is a research prototype and there is a high probability that the underlying domain model will change as further research is pursued. Therefore, NoSQL data stores are preferred over relational databases due to their schemaless nature that allows the data to evolve without having to make expensive schema changes [SF12]. This gives the developer high flexibility and increases the productivity when changes are made to the database [SF12]. When dealing with operational data, scalability is an important aspect, because the data expeditiously increases over time. NoSQL databases are suitable to handle large-scale data [SF12], which is another benefit over relational databases.

The selection of mongoDB over other NoSQL technologies is due to its suitability for time series data. Time series data can be defined as an *ordered sequence of observations* [Mad07], which operational data for the most part consists of. With its aggregation framework, mongoDB provides proper support for dealing with time series data [Mih14]. Having Spring as the underlying framework of the Feedback Handler, a secondary benefit of choosing mongoDB is its integration into the Spring framework. The Spring Data MongoDB[34] project, amongst other things, provides support for mapping domain model entities to a mongoDB database, templates for common database operations, and a Java API for the mongo aggregation framework.

3.7.3 Filtering and Aggregation Techniques

As described in the previous section, mongoDB has an extensive aggregation framework. Therefore, it makes sense to do a lot of filtering and aggregating directly on the database. The mongoDB aggregation framework is modeled as a data processing pipeline [Doca]. A simple example of such a pipeline from the official mongoDB documentation is shown in Figure 3.11.

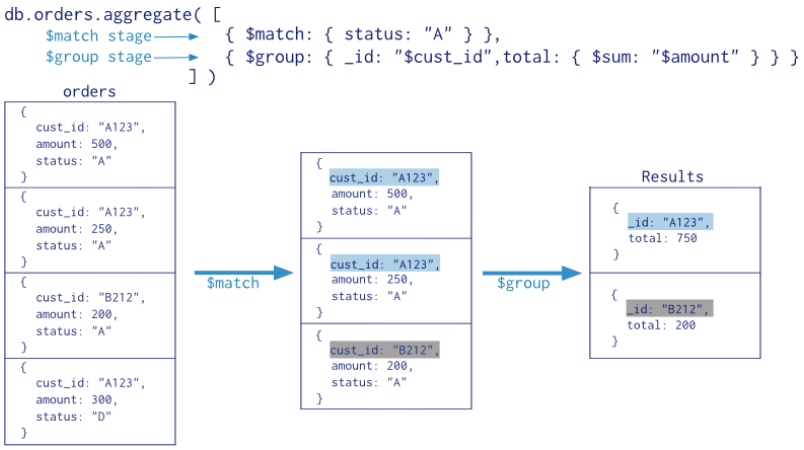


Figure 3.11: The mongoDB aggregation pipeline [Doca].

[33] http://www.mongodb.org
[34] http://projects.spring.io/spring-data-mongodb

An input collection is filtered by subsequently passing a number of pipeline stages, whereas the result of a stage is the input of the next stage. In the example in Figure 3.11, a list of orders is first filtered with a *match* criteria that checks for a given type and in a second operation grouped by the customer id. With the `$sum` statement in the query, the total amount for each customer is calculated. As true for any data in mongoDB, the result sets are always JSON nodes. The aggregation query, which is constructed from an array of pipeline stages, is itself written in JSON.

The described aggregation framework is used by the Feedback Handler to filter and aggregate the operational data. Rather than directly creating queries in the Java code, the Spring Data MongoDB project is used, which provides a Java API for those operations. Listing 3.4 shows an example: the method `aggregateProcedureMetrics` is capable of computing the average value of the metrics with the given type (e.g., execution time, CPU usage) for all procedures of the given application. To do so, it creates an aggregation using the fluent Java API of Spring Data MongoDB as described in the following. First, a *match* operation is composed that filters the metrics according to their type and the application they belong to (line 3). Second, a *group* operation groups the filtered metrics based on the procedure they belong to (line 4). With the appended `avg` method, it is specified that the average of each group shall be calculated. Finally, a *sort* operation is created that sorts the remaining data in descending order according to their `averageValue` attribute (line 5). Out of these three pipeline operations, an aggregation is created (line 6), which is then passed to the mongo template together with the name of the targeted database table and the output type (line 7). The output type (`ProcedureMetricAggregation`) is a POJO that contains fields and accessor methods for the attributes of the aggregation result items. The mongo template, which is a simple template class that provides methods to operate on a mongoDB database, is responsible for executing the query on the database and return the result. It is provided as a Spring Bean and can therefore be injected with Spring's dependency injection mechanism. Listing 3.5 shows the respective aggregation query that is executed in the background with the code of Listing 3.4.

```
1 public AggregationResults<ProcedureMetricAggregation>
      aggregateProcedureMetrics(final Application app, final MetricType type) {
2     final MatchOperation matchOperation = match(new Criteria("application.$id")
3         .is(application.getId()).and("type").is(type.toString()));
4     final GroupOperation groupOperation =
          group(fields).avg("value").as("averageValue");
5     final SortOperation sortOperation = sort(Sort.Direction.DESC,
          "averageValue");
6     final Aggregation executionTimeAggregationSpec =
          newAggregation(matchOperation, groupOperation, sortOperation);
7     return mongoTemplate.aggregate(aggregation, DbTableNames.METRICS,
          ProcedureMetricAggregation.class);
8 }
```

Listing 3.4: An aggregation pipeline in the Feedback Handler code.

```
1 db.metrics.aggregate([
2   { $match : { "application.$id" : ObjectId("<application-id>"), type :
        "<METRIC_TYPE>" } },
3   { $group : { _id : { procedure : "$procedure"}, averageValue: { $avg:
        "$value" } } },
4   { $sort : { "averageValue" : -1 } }
5 ]);
```

Listing 3.5: The corresponding aggregation query of Listing 3.4.

3.7.4 REST API

To expose runtime data to the Eclipse plug-in (or theoretically any other client), the Feedback Handler provides a RESTful HTTP API. The API is designed on top of the Spring Web MVC framework[35], which is part of the Spring Core Framework. Spring Web MVC allows to specify controller methods using Java annotations. A controller method is mapped to its URL with the `@RequestMapping` annotation, defining the relative URL as a String. Listing 3.6 shows an example of such a method called `avgExecTime`, which returns the average execution time of a procedure. The procedure is given through a number of request parameters, which are passed to the method as parameters using the `@RequestParam` annotation. The method operates on the data and returns the result, which builds the HTTP response body sent to the client. The `handleUnauthorized` method in line 9, defined in an abstract base controller class, checks the validity of the authentication parameters of the client. The authentication parameters consist of an application ID and an authentication token and are passed as HTTP headers with each request. If the authentication is successful, `handleUnauthorized` does nothing and the application returns to the invoking method. Otherwise, an exception is raised that is sent back to the client.

```
1  @RequestMapping(Urls.ANALYIS__AVG_EXEC_TIME)
2  @ResponseStatus(HttpStatus.OK)
3  public Double avgExecTime(
4       @RequestHeader(Headers.ACCESS_TOKEN) final String accessToken,
5       @RequestHeader(Headers.APPLICATION_ID) final String applicationId,
6       @RequestParam(Params.CLASS_NAME) final String className,
7       @RequestParam(Params.PROCEDURE_NAME) final String procedureName,
8       @RequestParam(Params.ARGUMENTS) final String arguments) {
9     final DbApplication application = handleUnauthorized(applicationId,
          accessToken);
10    final Optional<Double> averageExecTime =
          metricRepository.aggregateExecutionTime(application, className,
          procedureName, Splitters.arrayOnComma(arguments));
11    return averageExecTime.isPresent() ? averageExecTime.get() : null;
12 }
```

Listing 3.6: An example of a REST controller method.

3.8 Eclipse Plug-in Component

The Eclipse plug-in is the client component of the FDD system. While the FDD approach suggests to support multiple IDE's (see Section 3.3), the prototype so far only supports the Eclipse IDE. However, the other components of the system (i.e., the Monitoring component and the Feedback Handler) are completely independent of the IDE. Therefore, the system is designed to be extendable to support further IDE's. The Eclipse IDE component has also been designed to maximize the reusability of the sub-components. However, some of the logic cannot be isolated from the underlying IDE technologies. This lies in the nature of an FDD system due to its basic characteristic of tight IDE integration (see Section 3.3.1).

The architecture of the Eclipse platform is implemented on top of Equinox[36], which is an implementation of the OSGi framework specification[37]. The Eclipse platform itself consists of a small core only, most of the functionality is added in form of plug-ins [BW14]. Plug-in's are the main

[35] http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html
[36] http://eclipse.org/equinox
[37] http://www.osgi.org/Main/HomePage

building blocks that bundle related functionality. A plug-in contains the Java code (JAR) and describes its dependencies to other plug-ins [BW14]. On top of the plug-ins deployed with the Eclipse IDE itself, developers can implement their own plug-in's in order to extend the generic functionality. The Eclipse plug-in of PerformanceHat is built on top of the core component and the Eclipse JDT[38]. In the following sections, the most important implementation aspects of the plug-in are illustrated.

3.8.1 FDD Resources Extension

The Eclipse resources plug-in is a substantial plug-in for the Eclipse IDE that provides an API to access projects, folders, and files inside the workspace the developer is working with [39]. The resources plug-in works on top of the Eclipse File System (EFS), which is an abstract file system API that provides an abstraction of the concrete details of particular underlying file systems [40].

The resources API provides comprehensive functionality to deal with Eclipse resources. It provides methods to create, copy, move, and delete resources, traverse hierarchical resource trees, add and remove markers, and acess and modify the resources content and metadata. Part of this generic functionality is used by the FDD plug-in. On top of it, a lot of FDD-specific functionality is implemented. To optimally integrate the FDD-specific functionality into the existing resources plug-in, a so called *FDD Resources Extension* has been designed and implemented. It extends the resource types from the Eclispe resource plug-in using the Decorator pattern [GHJV94]. Figure 3.12 contains a class diagram of the FDD Resources Extension. The topmost section of the diagram (the gray box named Eclipse Resource Plug-in) is not part of the extension, but shows the involved types of the Eclipse resources API: `IResource` is the basic interface for all kinds of resources in an Eclipse workspace, `IProject` and `IFile` are generalizations of `IResource` for the respective concrete resource types. Those three interfaces are the ones that are decorated with functionality specific to Feedback-Driven Development in the extension layer. The FDD Resources Extension itself (the blue part in Figure 3.12) consists of three layers.

The *Basic Decorators Layer* includes an abstract implementation of a Decorator for all of the three decorated interfaces having the `AbstractBaseResourceDecorator` as basic class. This class specifies an abstract method `resource()` that has to be implemented by each subclass in order to provide the decorated elements. The `AbstractBaseResourceDecorator` itself simply implements all the methods from the `IResource` interface and delegates all of the work to the decorated element. The other three abstract classes of this layer extend the basic class and provide the concrete decorated element implementing the `resource()` method. Additionally, the `AbstractProjectDecorator` and the `AbstractFileDecorator` implement the methods of the according interface, also delegating all of the work to the decorated element. In Figure 3.12, the overridden methods are omitted in the decorator as well as in the interfaces due to the lack of space.

On the level of the Basic Decorators Layer, no FDD-related functionality is provided. The purpose of this layer is solely to provide abstract implementations of decorators that do the delegation required in order to implement concrete decorators [GHJV94]. The concrete decorators are provided by the *FDD Generic Layer*, which is divided into an API sub-layer providing the required interfaces and an Implementation sub-layer providing the respective implementations. The inheritance hierarchy is the same as in the Eclipse resources plug-in.

[38] https://eclipse.org/jdt
[39] http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresInt.htm&cp=2_0_10
[40] https://wiki.eclipse.org/EFS

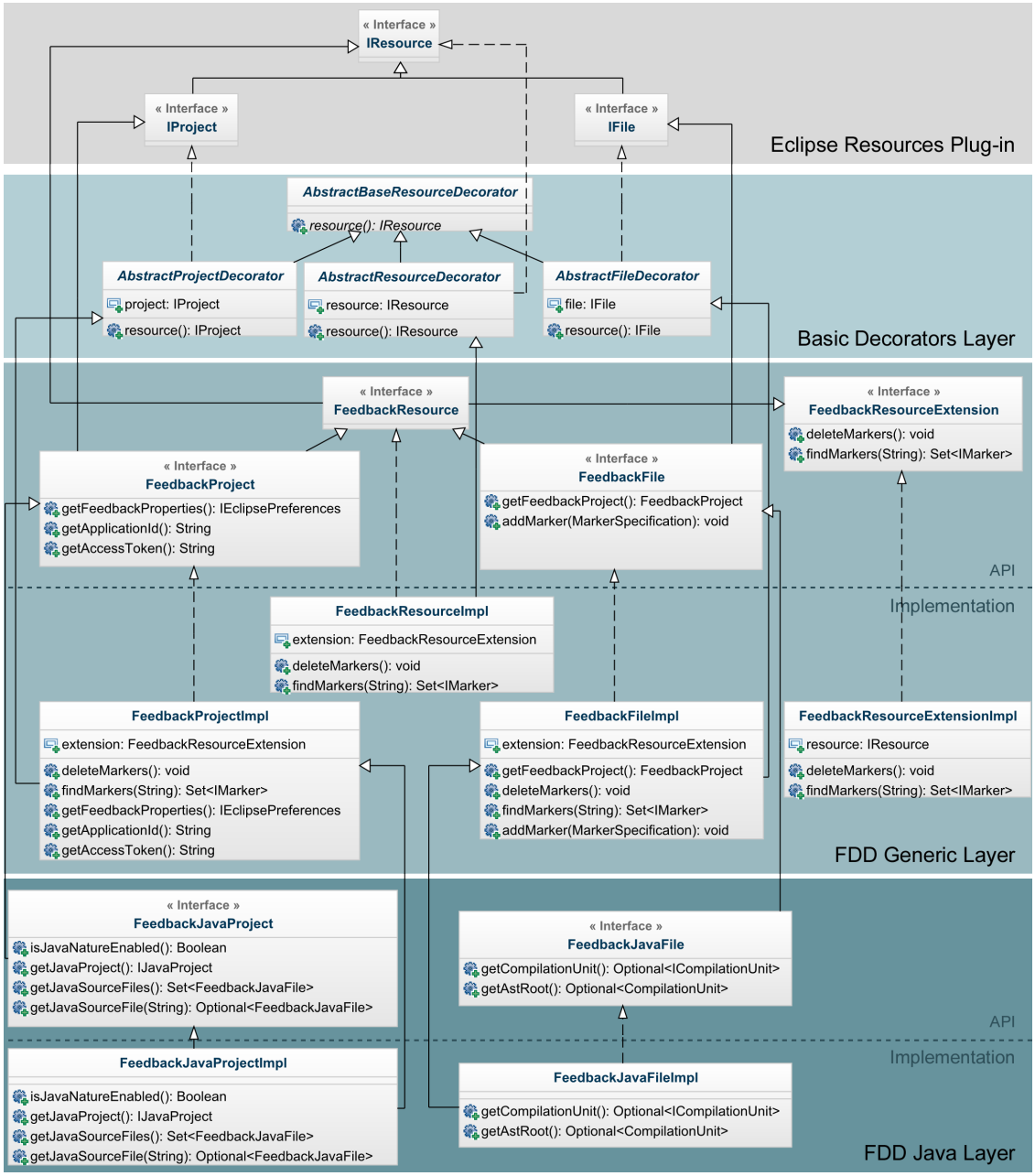


Figure 3.12: A class diagram of the Eclipse resources extension layer.

The fact that Java does not support multiple inheritance prevents the classes `FeedbackProjectImpl` and `FeedbackFileImpl` from extending their appropriate abstract decorator as well as the `FeedbackResourceImpl` class. To overcome this issue, the interface `FeedbackResourceExtension` and its associated implementation were introduced as a workaround. The `FeedbackResourceExtension` contains the methods that would have actually been designed in the `FeedbackResource` interface. This al-

lows the `FeedbackResourceImpl` and `FeedbackFileImpl` classes to use composition rather than inheritance to use the common functionality: Both these classes, as well as the `FeedbackResourceImpl`, delegate the resource-specific methods to an instance of `FeedbackResourceExtensionImpl`.

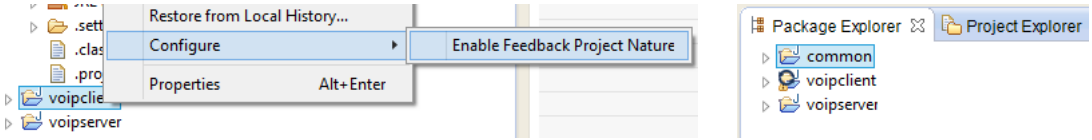
The functionality provided by the FDD Generic Layer is mostly related to the handling of markers (see Section 3.8.5). Additionally, the `FeedbackProject` provides some methods to access FDD-specific project properties (see Section 3.8.2). The functionality is not discussed in more detail here, but it is documented in the code.

The third layer of the FDD Resources Extension, the *FDD Java Layer*, works on top of the FDD Generic Layer and bundles all the Java-specific functionality related to FDD. In addition to the Eclipse resources plug-in, it has dependencies to the Eclipse JDT, which contains all the Java plug-ins from the Eclipse IDE. Similar to the FDD Generic Layer, the FDD Java Layer consists of an API and an implementation part. The interfaces in the API sub-layer simply extend the ones from the FDD Generic Layer and the classes in the implementation sub-layer provide the respective implementations. The separation from the Java-specific functionality into a separate layer decouples the FDD Generic Layer from the Eclipse JDT. This increases the reusability and makes the FDD Resources Extension expandable to support support other programming languages at a later point in development.

3.8.2 Project Nature

Eclipse's *Project Natures*[41] allow to tag a project as a specific type of project. Project Natures are defined by plug-ins and can be added to a project. This creates an association between the plug-in and the project [Art03]. The plug-in is now aware of the project and can conduct arbitrary actions on it [Art03]. Typically, a builder is registered for a project nature (see Section 3.8.3). A project can have multiple natures to interact with different plug-ins.

In the FDD Eclipse plug-in, the *Feedback Nature* is defined. Projects that want to use the functionality of PerformanceHat have to enable the Feedback Nature. This triggers the *Feedback Builder* on that project (see Section 3.8.3). To trigger (i.e., enable or disable, depending on the current state) the Feedback Nature, a context menu action is available in the Eclipse Project Explorer as well as in the Java Package Explorer (see Figure 3.13a). As soon as the nature is enabled, this is visible in the user interface through a decorator showing a blue feedback icon (see Figure 3.13b).



(a) Enabling the Feedback Nature with a context menu action. **(b)** Decorator of the Feedback Nature.

Figure 3.13: The Feedback Nature.

Project Properties

As soon as the Feedback Nature is enabled for a project, the project's properties are extended with a Feedback-Driven Development property page that allows to specify the properties related to the FDD plug-in (see Figure 3.14). Currently, two groups of properties are available: Authentication

[41] https://eclipse.org/articles/Article-Builders/builders.html

properties required to connect to the Feedback Handler (see Section 3.7.4) and thresholds that define the maximum allowable values for particular warning advices (see Section 3.3.2).

If the Feedback Nature is disabled, the FDD property page is not displayed anymore, but the properties remain stored in the project. This allows for temporary disabling of the Feedback Nature and re-enabling without the necessity for specifying the properties again. Temporarily disabling the nature is a common use case, because having the nature enabled causes longer build times due to the communication with the Feedback Handler.

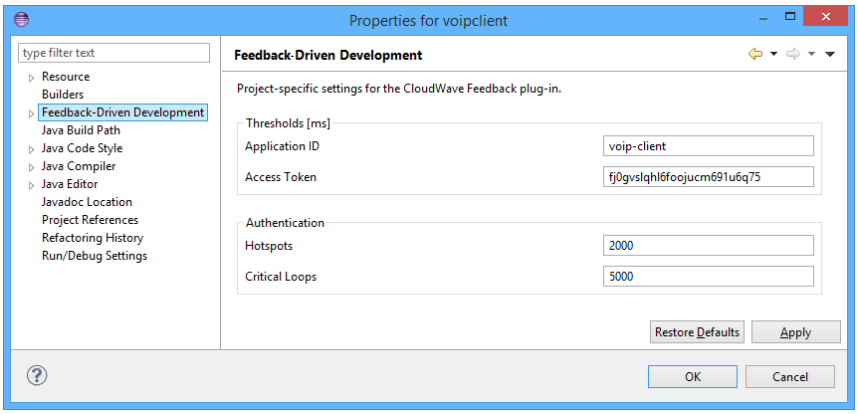


Figure 3.14: The FDD project property page.

3.8.3 Builder Mechanism

In Section 3.8.2 the Eclipse project natures were discussed. The concept of *Builder*'s is tightly coupled to project natures, each builder is associated with exactly one project nature. A builder is responsible for manipulating and creating resources in projects which have the builder's nature enabled [Vog09]. Plug-in's have the possibility to implement and register their own builder. Each registered builder is executed as soon as a build is triggered, which is either the case after one or more resources have been changed, if auto-build is activated, or through an explicit invocation of the user [Art03]. There are four different kinds of builds in Eclipse:

- **Incremental Build:** An incremental build happens in most of the cases when a build is processed in Eclipse. Incremental means, that the builder receives a resource delta, which describes what has been changed since the last build [Art03]. The incremental builder can analyze the delta and build the output that is affected by the changed resources only [Art03]. This prohibits the inefficiency of rebuilding the whole output after each change.
- **Auto-Build:** An auto-build is the same as an incremental build with the only difference that it is triggered automatically by the reason of changed resources in the workspace [Art03]. This happens, when the auto-build option is turned on for the workspace. If so, each time a resources is added, modified, or removed, all the registered builders are executed [Art03].
- **Full Build:** A full build uses the whole resource set as input instead of a resource delta and rebuilds everything from scratch [Art03]. Typically, this only occurs after executing a clean action.
- **Clean Build:** A clean build first deletes all files generated by the builder and then performs a full build. Just like the full build itself, it should theoretically never be used, if the incremental builder performs correctly [Art03].

The FDD plug-in associates its own builder called *Feedback Builder* with the plug-in's project nature. The Feedback Builder is responsible for parsing the resources of the project, gathering the corresponding runtime data, and assigning the warning advices (see Section 3.3.2) in form of markers to the source code. Markers will be discussed in more detail in the subsequent section. To divide the work of the builder into separate parts, the so called concept of *builder participants* is introduced. The purpose of the builder participant is to adopt one particular use case and create the respective advices. Listing 3.7 shows the simple interface specifying the API of a builder participant. The *build* method receives the respective project and the set of the changed files as input and is responsible for creating the appropriate advices. Figure 3.15 shows the type hierarchy of the builder participants. The base class `AbstractFeebackBuilderParticipant` implements the `build` method in a common way by iterating over the changed files and calling the abstract method `buildFile`, that has to be implemented by subclasses, for each file. Additionally it provides a helper method for the purpose of adding markers to a file. Concrete builder participants therefore only have to implement the `buildFile` method, which provides the content of the files as input (the `CompilationUnit` parameter).

```
1 public interface FeedbackBuilderParticipant {
2
3   public void build(final FeedbackJavaProject project, final
        Set<FeedbackJavaFile> files) throws CoreException;
4 }
```

Listing 3.7: The `FeedbackBuilderParticipant` interface defined by the FDD plug-in in order to split the work of the Feedback Builder.

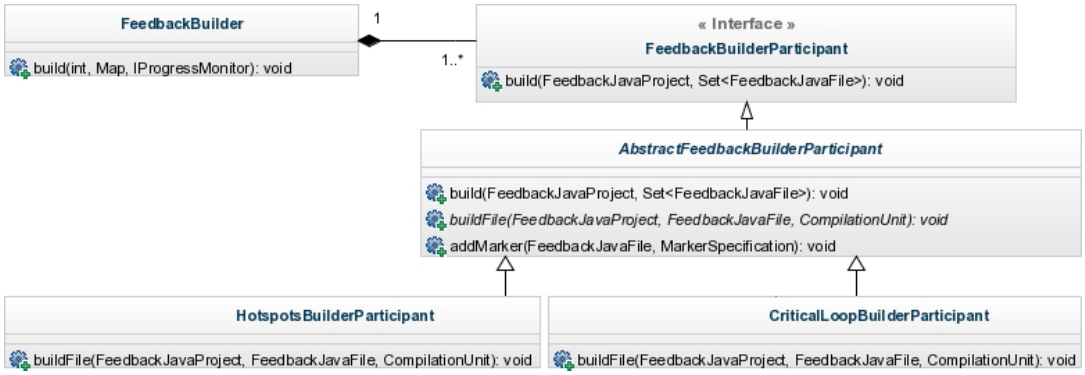


Figure 3.15: The Feedback Builder and builder participants.

The Feedback Builder itself receives the list of registered builder participants and simply delegates all the work to them. Listing 3.8 shows the source code of the method executing an incremental build from the `FeedbackBuilder` class. It computes the resource delta of the change as a first step and only proceeds if the delta could be successfully loaded. If so, the builder cleans all the sources affected by the delta, which means that all the FDD-related markers are deleted. Finally, it invokes all the builder participants and passes the resource delta to them in order to allow them to investigate the source code and distribute the markers.

Separating the build process into different builder participant for each use case (i.e., each type of warning advice, see Section 3.3.2) has several benefits. It allows to separate concerns without having to register a builder, and hence a project nature, for each advice. While registering

a builder is relatively expensive and requires to modify the plug-in's configuration file, creating a new builder participant simply requires creating a new Java class extending the abstract base builder participant. The design enables activating and deactivating particular builder participants at runtime without having to add and remove project natures. Although this feature is not implemented so far, deactivating a builder participant could be valuable to the developer to cut down the build time if he is not interested in some specific advices.

```
1  private void incrementalBuild(final FeedbackJavaProject project) throws
       CoreException {
2    final IResourceDelta resourceDelta = getDelta(getProject());
3    final Optional<? extends FeedbackJavaResourceDelta> feedbackDeltaOptional =
         feedbackJavaResourceFactory.create(resourceDelta);
4    if (feedbackDeltaOptional.isPresent()) {
5      final FeedbackJavaResourceDelta delta = feedbackDeltaOptional.get();
6      cleaner.cleanDelta(delta);
7      for (final FeedbackBuilderParticipant participant : participants) {
8        participant.build(project,
             feedbackDeltaOptional.get().getChangedJavaFiles());
9      }
10   }
11 }
```

Listing 3.8: The method `incrementalBuild` from the `FeedbackBuilder` class.

3.8.4 Static Source Code Analysis

In Section 3.3.2, the process from raw runtime data to valuable feedback was described. The last step in this process is the concatenation of the dynamic feedback to static source code artifacts. Static source code analysis is required to detect the right artifacts. The Eclipse JDT component provides an API to navigate the abstract syntax tree (AST) of Java sources. This API is used to determine the right location in the source code to attach dynamic feedback to. The AST is traversed using the Visitor pattern [GHJV94]. Listing 3.9 shows an extract of the `HotspotsBuilderParticipant`, showing how a Java source can be traversed by subclassing the `ASTVisitor`. The `ASTVisitor` is an abstract implementation that provides hook methods for all the AST nodes that can potentially be visited. A subclass, like the anonymous inner class in Listing 3.9, only has to override the methods of interest. In the case of hotspot methods, all method declarations and invocations are parsed. The actual matching of the runtime data to the right method is executed in AST node decorators. In the lines 5 and 11 in Listing 3.9, such decorators, called `MdethodDeclarationExtension` and `MethodInvocationExtension`, are created. These classes provide the methods to match runtime feedback with the respective AST nodes.

```
1  protected void buildFile(final FeedbackJavaProject project, final
       FeedbackJavaFile javaFile, final CompilationUnit astRoot) {
2    astRoot.accept(new ASTVisitor() {
3      @Override
4      public boolean visit(final MethodDeclaration methodDeclaration) {
5        visit(new MethodDeclarationExtension(methodDeclaration));
6        return true;
7      }
8
9      @Override
10     public boolean visit(final MethodInvocation methodInvocation) {
11       visit(new MethodInvocationExtension(methodInvocation));
```

```
12      return true;
13    }
14
15    private void visit(final AbstractMethodExtension<?> methodExt) {
16      for (final AggregatedProcedureMetricsDto hotspot : getHotspots()) {
17        if (methodExt.correlatesWith(hotspot.getProcedure())) {
18          final int startPosition = methodExt.getStartPosition();
19          final int line = astRoot.getLineNumber(startPosition);
20          addMarker(javaFile, createMarkerSpecification(new MarkerPosition(line,
                startPosition, methodExt.getEndPosition()), hotspot));
21        }
22      }
23    }
24  });
25 }
```

Listing 3.9: The AST visitor in the `HotspotsBuilderParticipant`.

The matching is as simple as comparing the qualified class name of the method, the name, and the qualified names of the parameter's classes. Listing 3.10 shows the respective code of the class `AbstractMethodExtension`, which is a base class for AST method node decorators. The `Procedure` of the runtime feedback is passed as input parameter and each of its attribute is compared to the respective attribute of the AST node. The methods returning the AST node attributes are implemented in the particular subclass.

```
1  public boolean correlatesWith(final Procedure procedure) {
2    return correlatesClassName(procedure) && correlatesMethodName(procedure) &&
         correlateArguments(procedure);
3  }
4
5  private boolean correlatesClassName(final Procedure procedure) {
6    return getQualifiedClassName().equals(procedure.getClassName());
7  }
8
9  private boolean correlatesMethodName(final Procedure procedure) {
10   return getMethodName().equals(procedure.getName());
11 }
12
13 private boolean correlateArguments(final Procedure procedure) {
14   return Arrays.equals(getArguments(), procedure.getArguments().toArray());
15 }
```

Listing 3.10: Matching runtime feedback with AST nodes.

3.8.5 Markers and Hovers

Section 3.8.3 described how the builder mechanism works. In the FDD plug-in, since no sources are generated, the only task of the builder is to contribute markers to the resources. Markers[42] are an Eclipse concept to communicate any kind of problem or other information. They are always attached to a resource. Their characteristics makes markers a perfect concept to visualize warning advices in the Eclipse UI. Using the existing markers concept to show advices allows the FDD plug-in to reuse a lot of functionality already integrated into the Eclipse IDE. The IDE provides a *Problem View* similar to the warnings overview in the suggested mockup in Figure 3.6. Markers

[42] http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_markers.htm

are further integrated into the Java Editor, and the respective locations in the source code can be highlighted and a small icon appears beside the respective line numbers. This concept is known from the Eclipse Java editor, displaying compiler warnings and errors directly in the source code. The builder of the FDD plug-in provides additional markers to Java source files - communicating warning advices to the developer. Figure 3.16 shows an example of two warning markers from the FDD plug-in visualized in the Java editor.

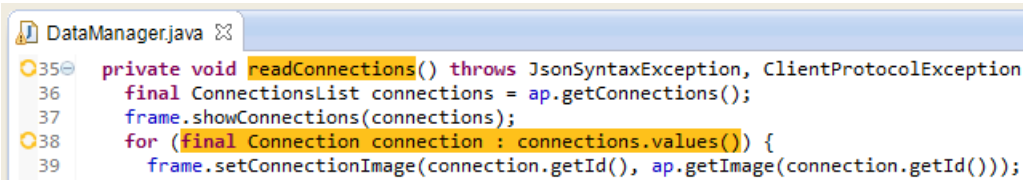


Figure 3.16: FDD markers showing warning advices in the Java editor.

In order to not have to register an own marker for each type of advice, the same marker is reused for all the different kinds of advices and a type-attribute is added to each marker specifying which type of advice the marker is identifying. This reduces the amount of required plug-in configuration and increases the reusability and maintainability of the marker module. Similar to the builder participants discussed in Section 3.8.3, this allows to easily add new marker types without having to change the plug-in configuration.

Displaying detailed Information in Hovers

While the markers are reasonable to indicate the location of a problem, they are not capable of communicating detailed information about the problem themselves. Indeed, a one-line description can be added to a marker that is displayed in the Problems View and as small tooltip when hovering a marker icon. However, a more elaborate UI element is required to illustrate all the runtime data available about an advice. For this purpose, the Java editor is extended with FDD *Hovers* displaying runtime data. The FDD hovers are similar to the existing Javadoc hovers known from the Java editor. Figure 3.17 shows an example of such a hover containing detailed information about a *critical loop*.

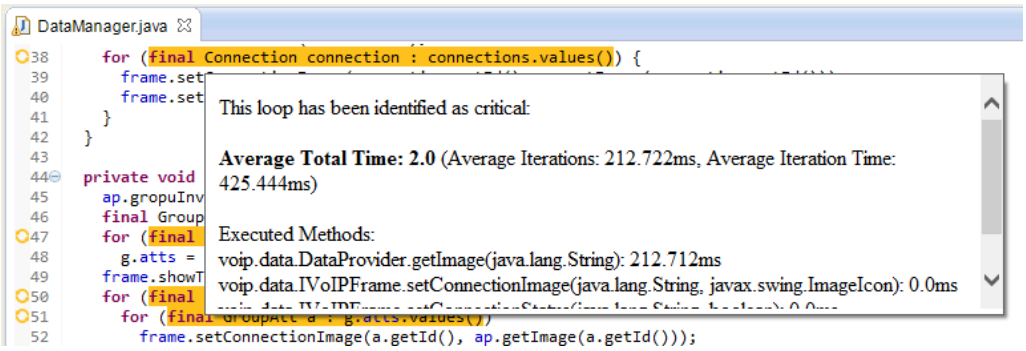


Figure 3.17: FDD hover displaying detailed information about a warning advice.

Technically, the main area of the hover is constructed of a browser control that is capable of rendering HTML, which makes the visualization very flexible. The content of a hover is provided by different content providers, which are registered in a content provider registry per type of advice. Adding a new marker type therefore entails the need for a new content provider, but does not require registering a new hover for the Java editor.

3.9 CloudWave Integration

CloudWave[43] is a research project, funded by the EU, with the goal of revolutionizing the cloud infrastructure in order to enable a higher level of agile development processes, dynamic adaption of cloud services to the environment, and optimization of resource utilization and service quality [BFKB+14]. Feedback-Driven Development is one of the three main pillars of the CloudWave project [BFKB+14]. Figure 3.18 visualizes the proposed framework of the CloudWave framework. Further details about the framework and the project as a whole can be found in [BFKB+14]. Our prototype is designed to be integratable into the CloudWave infrastructure to be used as part of it. The integration is not completed so far, but part of the required adaption code is already implemented.

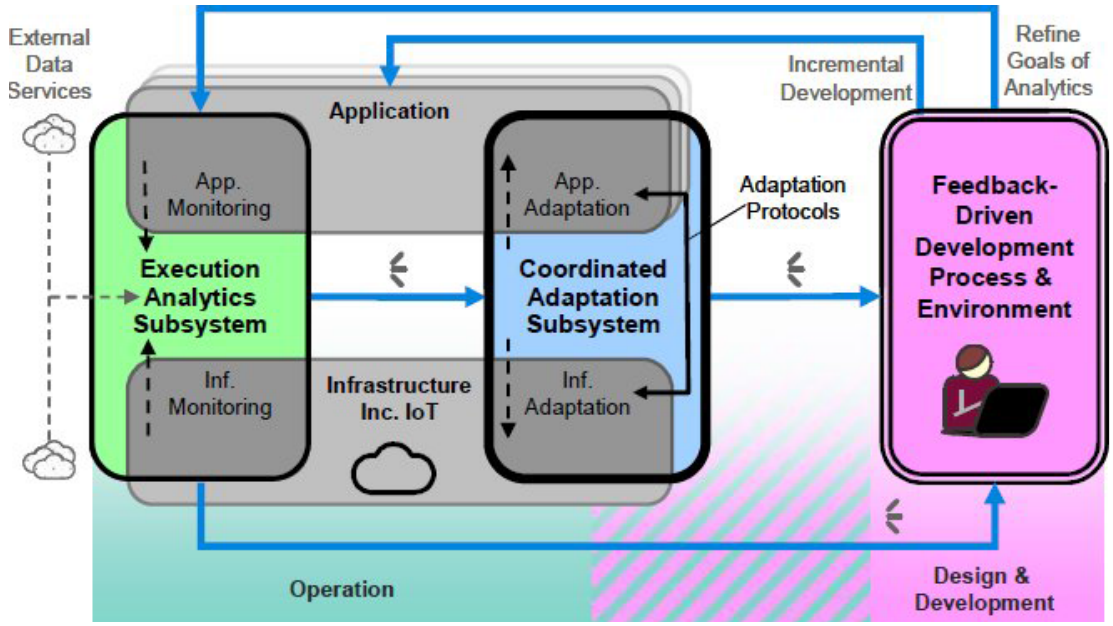


Figure 3.18: The proposed framework of the CloudWave project [BFKB+14].

Some of the components of the CloudWave environment are not implemented in Java. To communicate with those components, PerformanceHat uses a Java wrapper library from CloudWave, which uses the Java Native Interface[44] to access native code. One of the basic things that has to be done in order to enable the communication between PerformanceHat and the CloudWave environment is the translation of metrics. While a metric in PerformanceHat has the properties

[43] http://www.cloudwave-fp7.eu/
[44] http://docs.oracle.com/javase/7/docs/technotes/guides/jni

described in Section 3.5.1, a metric in CloudWave has a slightly different appearance, being composed of a name, some textual data, a unit, and a value. In order to transmit metrics between the two systems, a *metric converter* is implemented that converts FDD metrics into the CloudWave format (see Figure 3.19). The *CwMetric* type is a Java representation of the CloudWave metric format. The metric converter is capable of translating a `Metric` into a `CwMetric` and vice versa. Based on this converter, additional integration components can be developed at a later point in development.

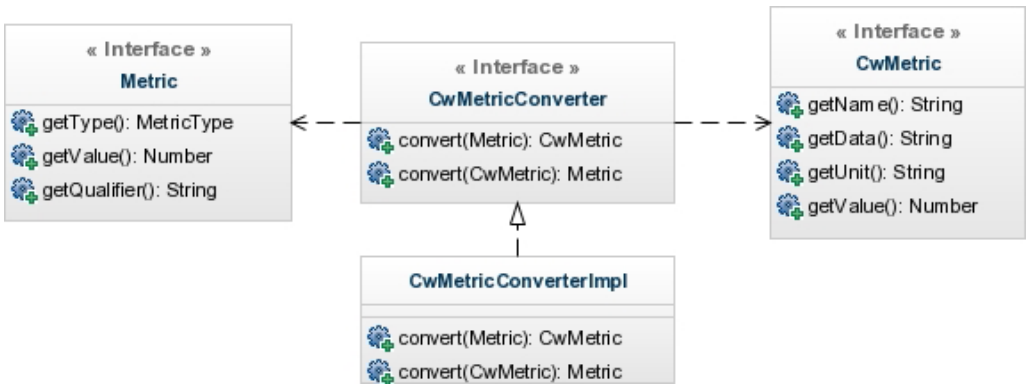


Figure 3.19: The metric converter.

Chapter 4

Evaluation

The evaluation of PerformanceHat is separated into two distinct parts. A qualitative evaluation compares the prototype with other performance management tools from research and industry. In the quantitative evaluation, the performance of PerformanceHat itself is reviewed.

4.1 Qualitative Evaluation

In a qualitative evaluation, PerformanceHat is compared to different kinds of tools in the area of performance engineering. Other research prototypes as well as industrial tools are considered. The foundation for the comparison is constructed out of six different dimensions targeting important features and characteristics. First, the dimensions are briefly explained and the selected tools are presented. Then, a tabular overview illustrates the comparison. Each dimension is discussed in detail. Finally, a conclusion is derived from the comparison.

4.1.1 Dimensions

The quantitative evaluation constitutes of the six dimensions listed below, which can either be fulfilled or not fulfilled by each of the compared tools.

- **Monitoring Capabilities:** This dimension covers the capability of collecting runtime metrics on application level (e.g., method execution time) and system level (e.g., CPU utilization, memory usage) as well as graphically visualizing these metrics to allow the user to observe the system state.
- **Problem Detection:** Problem detection describes the ability to automatically detect known performance problems, such as the performance anti patterns discussed in Section 2.2, based on the available runtime data.
- **Predictive Analysis:** Predictive analysis denominates the capability of predicting the future performance of an application based on of the performance data gathered in the past.
- **Production Data:** Some performance engineering tools conduct their analysis directly on the developer's local machine, others use data from productive environments. Since productive systems typically exhibit other conditions than development environments (different hardware, different underlying software stack), monitoring data at production is more valuable to detect performance problems. This dimension is fulfilled if the tool gathers runtime data on the production system rather than on the developer's machine.

- **Non-Intrusiveness:** In order to monitor a system, additional code besides the actual application code is required to execute the monitoring actions. If the monitoring logic is mixed with application- or business-logic, the maintainability of the system decreases. The code is less readable and additional dependencies are introduced to the components containing the business logic. Therefore, it is desirable to decouple the monitoring logic from the business logic. Systems which fulfill this property are called *non-intrusive* [vHRH+09].
- **IDE Integration:** Tools that are integrated into an Integrated Development Environment fulfill this dimension. IDE integration prohibits developers from context switches through having to change the application they are working in.

4.1.2 Selected Tools

Different kinds of tools, both from research as well as from industry, have been included into the comparison.

Research Prototypes

The following nine prototypes from research are discussed in the comparison:

- **Kieker** [vHWH12, vHRH+09] is an extensible framework that provides functionalities to monitor and analyze software systems. It contains various plug-ins to conduct different kinds of analyses on the monitored data.
- **Senseo** [RHV+09b, RHV+09a, RHB+12] is a tool that enhances the static source views of the Eclipse IDE with dynamic runtime data. It collects runtime type information as well as performance-related metrics and visualizes this data in the IDE.
- **VIVIDE** [TSH12] is a programming environment that combines static and dynamic data of an application. It uses tests to collect runtime information. The goal is to provide a task-oriented environment to the developer.
- **Profiling Blueprints** is a visualization concept proposed by Bergel *et al.* [BRB10] that helps to identify and remove performance bottlenecks. The implementation is integrated into the Pharo[1] development environment.
- In [NNH+14] Nguyen *et al.* describe a **performance regression detector** that is able to discover the cause of performance problems based on a repository that is filled with data about performance tests and information about the causes of past performance regressions.
- ***J** [DHV03] is a tool that provides functionalities to gather, compute, and visualize dynamic metrics from Java applications. It's aim is to support the developer in understanding the behavior of the application at runtime.
- In [Rei03] Reiss proposes a **dynamic Java visualizer** that focuses on minimal overhead in order to not slow the monitored application down. The main goal is to provide a tool developers can use almost all the times without being concerned about long waiting times.
- **DynaMetrics** [SS08] provides evaluation and analysis capabilities for dynamic data about object-oriented software systems. A main aspect of the approach is to compare the dynamic metrics with their static counterparts to determine the usefulness of individual metrics.
- **JInsight** [PJM+02] allows to visually explore the runtime behavior of a Java program. It is designed to be used for performance analysis and debugging activities.

[1] http://pharo.org

Java Profiler

Profilers are tools that allow for investigating the runtime behavior of an application during debugging. The functionality of the available, well-known profiler for the Java platform is very similar: the method call trace can be monitored and runtime metrics, such as CPU utilization and memory usage, can be observed. There are small differences briefly mentioned in the following and discussed in more detail in the comparison. Four different profilers have been considered for the comparison with PerformanceHat. **VisualVM**[2] is a tool that is delivered as part of the Java Development Kit (JDK). **JVM Monitor**[3] is a profiler that integrates into the Eclipse IDE. **JProfiler**[4] goes a step further and even provides multiple IDE support. Finally, **XRebel**[5] differs from the others in that it also provides detection mechanism for the N+1 performance anti pattern (see Section 2.2.4).

Industrial application performance management (APM) Tools

The collection of treated tools is complemented by a number of industrial application performance management (APM) tools. **New Relic**[6], **AppDynamics**[7], and **Datadog**[8] are web-based APM tools that allow for monitoring applications written in different programming languages as well as different relational database management systems. **DripStat**[9] is another web-based tool, but is restricted to the Java platform. **Glassbox**[10] differs from the others in that it is not web-based.

4.1.3 Comparison

Table 4.1 illustrates the complete comparison of the introduced tools. *Yes* means that a tool fulfills the respective dimension, *partial* means that some aspects of the dimension are considered, and *no* means that the dimension is not covered by the tool. All the statements about foreign tools made in the following paragraphs are based on the available documentation and to the best of the author's knowledge.

First of all, it can be stated that all the considered tools exhibit some *Monitoring Capabilities*. This is not surprising, since this dimension comprises basic features of performance monitoring tools. However, the degree to which the dimension is covered is different. Therefore, a finer granular comparison of the *Monitoring Capabilties* dimension is presented in the following subsection.

Problem Detection is differently supported by the compared tools. The only tool that exhibits a comprehensive detection mechanism is the performance regression detector of Nguyen *et al.*. Based on information from performance tests and past performance regression root causes, their system is able to identify root causes of performance issues. Kieker also provides some detection mechanisms, but focuses mainly on architecture discovery (i.e., extracting information about the structure and behavior and identifying architectural entities and their interaction). XRebel provides a detection mechanism dedicated to the N+1 problem (see Section 2.2.4). Glassbox is the only industrial APM tool that provides detection mechanisms. The other four at least fulfill the dimension partially through the possibility of specifying thresholds for particular transactions.

[2] http://visualvm.java.net
[3] http://www.jvmmonitor.org
[4] https://www.ej-technologies.com/products/jprofiler/overview.html
[5] http://zeroturnaround.com/software/xrebel/
[6] http://newrelic.com
[7] https://www.appdynamics.com
[8] https://www.datadoghq.com
[9] https://dripstat.com
[10] http://glassbox.sourceforge.net

As these thresholds are violated, a respective warning appears. PerformanceHat detects hotspot methods and critical loops, as described in this thesis. Further anti-patterns are not supported so far, therefore the dimension is fulfilled only partially.

With its predictive analysis, PerformanceHat provides a feature that is rarely covered so far in research as well as in industrial tools. The only tool that exhibits similar functionality is Kieker. Similar to PerformanceHat, Kieker conducts predictive analyses based on the gathered time series data.

	Monitoring Capabilities	Problem Detection	Predictive Analysis	Production Data	Non-Intrusiveness	IDE Integration
Research Prototypes						
Kieker [vHWH12, vHRH+09]	yes	yes	yes	yes	yes	no
Senseo [RHV+09b, RHV+09a, RHB+12]	yes	no	no	no	yes	yes
VIVIDE [TSH12]	yes	no	no	no	yes	yes
Profiling Blueprints [BRB10]	yes	no	no	no	yes	yes
Performance Regression Detector [NNH+14]	yes	yes	no	no	yes	no
*J [DHV03]	yes	no	no	no	yes	no
Visualizing Java in Action [Rei03]	yes	no	no	no	yes	no
DynaMetrics [SS08]	yes	no	no	no	yes	yes
JInsight [PJM+02]	yes	no	no	no	yes	no
Java Profilers						
VisualVM	yes	no	no	no	yes	no
JVM Monitor	yes	no	no	no	yes	yes
JProfiler	yes	no	no	no	yes	yes
XRebel	yes	yes	no	no	yes	no
Industrial APM Tools						
New Relic	yes	partial	no	yes	yes	no
AppDynamics	yes	partial	no	yes	partial	no
Datadog	yes	partial	no	yes	partial	no
DripStat	yes	partial	no	yes	yes	no
Glassbox	yes	yes	no	yes	yes	no
PerformanceHat	yes	partial	yes	yes	yes	yes

Table 4.1: Comparison of performance management tools along different dimensions. The assignment of *yes* (dimension is fulfilled), *partial* (dimension is partially fulfilled), and *no* (dimension is not fulfilled) is based on the available documentation and to the best of the author's knowledge

The benefit of using production data to conduct performance analyses has already been discussed. The treated profilers all run on the developers local machine and therefore none of them fulfills this dimension. The industrial APM tools in contrast are all embedded into productive environments. PerformanceHat and Kieker are, to the best of the author's knowledge, the only research prototypes that monitor productive environments. Senseo runs the monitored application in a different JVM, but on the same physical machine. Other systems, like VIVIDE, Profiling Blueprints, and the performance regression detector, use data from test execution. Tools from research like PerformanceHat exhibit a major benefit over these tools through being integrated into productive systems.

The non-intrusiveness is a critical characteristic of performance management tools, in order to not reduce the readability and maintainabilty of the monitored application's source code. AppDynamics and Datadog are the only tools that merely partially fulfill this criteria. The reason is that they break with the non-intrusiveness in case of loop observation (see the following subsection). All the other tools completely separate the monitoring logic from the application's actual business logic.

The last dimension of interest is the IDE integration. As already discussed in this thesis, its main purpose is to increase the developers awareness of the available runtime metrics. As PerformanceHat, Senseo integrates with the Eclipse IDE. VIVIDE and Profiling Blueprints are developed on top of the Pharo platform[11]. Two profilers also provide IDE integration. The JVM Monitor includes an Eclipse plug-in, while the JProfiler even supports multiple IDE's (Eclipse, IntelliJ[12], NetBeans[13], and Oracle JDeveloper[14]). None of the industrial APM tools provides IDE integration. However, NewRelic, AppDynamics, and Datadog at least provide an API that theoretically allows to integrate with an IDE plug-in.

Monitoring Capabilities Revisited

The *Monitoring Capabilities* dimension is at some extend covered by all tools. However, the dimension is spanned very broadly and there are significant differences in the degree of support between the tools. A tripartite classification, as given in the overall comparison, is not sufficient in order to make useful statements. Therefore, the following sub-dimensions are discussed:

- **Application-Level Monitoring**: Different tools provide different levels of support for monitoring application-level artifacts. Since **Methods**, **Loops**, and **Conditional Branches** are the source code artifacts with most interest for performance problems and anti patterns (see Section 2.2), those three constructs are treated in the comparison.

- **System-Level Monitoring**: Another important aspect of performance engineering is to observe how applications make use of system resources. For the comparison, **CPU utilization** and **memory usage** are treated, due to their relevance to the examined performance anti patterns (see Section 2.2).

- **Graphical Visualization**: The last sub-dimension investigates whether and to what degree the compared tools provide graphical visualizations for the data of the other two dimensions.

Table 4.2 shows the comparison of the monitoring capabilities sub-dimensions. Each dimension is discussed in the following.

[11] http://pharo.org
[12] https://www.jetbrains.com/idea
[13] https://netbeans.org
[14] http://www.oracle.com/technetwork/developer-tools/jdev/overview/index.html

	Application-Level Monitoring			System-Level Monitoring		Graphical Visualization
	Methods	Loops	Conditional Branches	CPU Utilization	Memory Usage	
Research Prototypes						
Kieker [vHWH12, vHRH+09]	yes	no	no	yes	yes	yes
Senseo [RHV+09b, RHV+09a, RHB+12]	yes	no	no	no	yes	yes
VIVIDE [TSH12]	yes	no	no	no	no	partial
Profiling Blueprints [BRB10]	yes	no	no	yes	no	yes
Performance Regression Detector [NNH+14]	yes	no	no	yes	yes	no
*J [DHV03]	yes	no	no	no	no	partial
Visualizing Java in Action [Rei03]	yes	no	no	no	no	yes
DynaMetrics [SS08]	yes	no	no	no	no	yes
JInsight [PJM+02]	yes	no	no	no	no	yes
Java Profilers						
VisualVM	yes	no	no	yes	yes	yes
JVM Monitor	yes	no	no	yes	yes	yes
JProfiler	yes	no	no	yes	yes	yes
XRebel	yes	no	no	no	no	yes
Industrial APM Tools						
New Relic	yes	no	no	yes	yes	yes
AppDynamics	yes	yes	no	yes	yes	yes
Datadog	yes	no	no	yes	yes	yes
DripStat	yes	yes	no	yes	yes	yes
Glassbox	yes	no	no	yes	yes	yes
PerformanceHat	yes	yes	no	yes	no	partial

Table 4.2: Comparison of performance management tools along their monitoring capabilities. The assignment of *yes* (supported), *partial* (partially supported), and *no* (not supported) is based on the available documentation and to the best of the author's knowledge

Looking at the application-level monitoring, it can be stated that all tools support the monitoring of metrics related to methods. This can be explained with the fact that methods are the main building blocks. In contrast, the other source code constructs are treated very rarely. Most of the tools only look at methods as a whole and do not monitor further constructs. Apart from PerformanceHat, only AppDynamics and DripStat support observing loops. Conditional branches are not treated by any of the tools, although information about code coverage may be crucial to performance optimizations. This has been comprehensively proven in the research domain of just-in-time (JIT) compilers. IBM applied some modifications to the reference implementation of the JVM

in order to optimize performance [SOT+00]. One of the optimizations looks at the code coverage, such as which conditional branch is taken how many times. Suganuma *et al.* [SYK+01a,SYK+01b] developed a JIT compiler framework that does similar optimizations. Haubl *et al.* [HWM13] developed a just-in-time compiler that conducts trace-based optimizations. Amongst other things, they store counts of the taken and not taken conditional branches. Soot [VRCG+99] is a Java bytecode optimization framework that contains a JIT compiler taking care of optimization based on conditional and unconditional branch elimination.

PerformanceHat, as well as the otherAPM tools considered, currently lacks the support of these optimization features present in JIT compilers. However, through loop support, PerformanceHat, together with AppDynamics and DripStat, provides the best support of application-level metrics.

System-level monitoring is a popular mechanism to detect performance problems. All industrial APM tools provide support for CPU utilization as well as memory usage. The functional differences are not significant. Profilers, with the exception of XRebel, also provide comprehensive support for system-level monitoring. Kieker and the performance regression detector of [NNH+14] support both of the considered metrics, Senseo only supports memory usage measurements, while PerformanceHat and Profiling Blueprints only measure CPU utilization. Hence, compared to other tools, PerformanceHat exhibits a weakness in this area.

Despite the performance regression detector, all the treated tools provide some level of graphical visualization of the gathered metrics from the two foregoing dimensions. VIVIDE, *J, and PerformanceHat have been classified as providing only partial support, since they do not cover the same amount of visualization possibilities as the other tools. The industrial APM tools as well as the Java Profilers provide the best visual support. They offer a large number of different graphs showing the application performance in the past. Profiling Blueprints is an approach that strongly focuses on visualizing dynamic metrics in general and the CPU usage in particular. PerformanceHat so far only provides a graph visualization of the execution time and the CPU utilization of methods.

4.1.4 Conclusion

From the discussed comparison some practical deductions can be derived. On application-level, most of the tools measure metrics only at the boundaries of methods. Through considering loops as separate units of interest, PerformanceHat provides a benefit over the other research prototypes. Some industrial tools offer similar functionality. Conditional branches are so far only examined in the research area of just-in-time compilation, performance analysis tools do not provide the respective functionality. Looking at system-level monitoring, PerformanceHat does not provide a functionality as extensive as other tools. The same is true for the graphical visualization of both application-level and system-level monitoring. PerformanceHat supports problem detection as well as predictive analysis, which results in a very broad functional range compared to the other tools.

While the requirement of non-intrusiveness is fulfilled by all tools at least partially, only a few research prototypes, including PerformanceHat, are capable of conducting analyses on production data. The industrial tools on the other hand lack support of other dimensions. Kieker fulfills the dimension of production data, but at the same time lacks the IDE support. Overall it can be stated that PerformanceHat combines the most important characteristics that are requried to support the DevOps methodology. Other tools provide a more elaborate functionality in particular areas, but do not cover the Feedback-Driven Development approach in its entirety.

4.2 Quantitative Evaluation

In order to make statements about the overhead PerformanceHat adds to its environment, a quantitative evaluation was conducted that reviews the performance of the prototype. Two different aspects were measured. First, the overhead which the monitoring component adds to the target application is investigated. Second, it is measured whether and how much the activation of the FDD plug-in slows down the build process in the IDE.

4.2.1 Monitoring Overhead Investigation

If a target application is monitored, there always arises an overhead in the execution time of the application due to the additional code that gathers the runtime data and sends it to the Feedback Handler. In order to review the amount of overhead that accrues, a study has been conducted comparing the response times of an application with and wihtout enabled monitoring.

Study Setup

The overhead measurement was conducted on a simple web application called *Tudu*[15]. *Tudu* is a simple task administration tool that allows to manage tasks and lists of tasks. It has the purpose of illustrating best practices when developing web applications with the Spring framework and is available as open-source project on Github[16]. To measure the overhead of the source code monitoring, different use cases have been conducted, each with and without monitoring. The following list briefly summarizes the 10 examined use cases:

- **Register:** Create a new user account for the *Tudu* application.
- **Login:** Login to the system with valid user credentials.
- **Add List:** Create a new, empty list of tasks with a specified name.
- **Edit List:** Edit the name of an existing list of tasks.
- **Delete List:** Delete an existing list of tasks.
- **Show Task:** Show all the tasks of a list.
- **Add Task:** Add a new task with a specified title, description, priority, and due date to a list.
- **Edit Task:** Edit the attributes of an existing task.
- **Delete Task:** Delete an existing task.
- **Refresh:** Refresh the task overview of a list.

The *Tudu* application was deployed to a Jetty[17] webserver inside a Java 7 runtime environment on a local development laptop with Windows 8.1. To measure the response times of the different requests, the built in debugger of Google Chrome called DevTools[18] was used. Since the server runs on a local machine, no network latency is integrated in the measured time values.

[15] http://www.julien-dubois.com/tudu-lists.html
[16] https://github.com/jdubois/Tudu-Lists
[17] http://eclipse.org/jetty
[18] https://developer.chrome.com/devtools

Results

In total, 10 different requests were conducted. Each request was executed 20 times without the FDD system and 20 times with the monitoring component integrated into the *Tudu* application. Table 4.3 gives an overview of the results for each one of the ten different requests. The listed values represent the average of the 20 repetitions of each request. Figure 4.1 provides a graphical illustration of the average values. A complete overview of all the measured values can be found in Appendix B.

The results illustrate that the overhead through the monitoring component is significant to the overall performance of the monitored application. Four out of the ten measured requests exhibit an increase of less than 30% through the overhead (*Show Tasks*, *Edit Task*, *Delete Task*, and *Refresh*), which might be acceptable in production environments. End users may not recognize such relatively small performance losses while using the application, as long as there are no long-running tasks (i.e., multiple seconds or even more). Two requests manifest an overhead of more than 30%, but at least less than 50% (*Login* and *Add List*), which should be classified as critical to the overall performance. The remaining four request types (*Show Tasks*, *Edit Task*, *Delete Task*, and *Refresh*) exhibit an overhead of more than 50%. While these overheads may not be problematic for small requests, long requests seem to be slowed down significantly through the source code monitoring.

use case	without monitoring	with monitoring	increase (%)
Register	97.3	148.5	52.62
Login	14.85	19.6	31.99
Add List	26.85	37.7	40.41
Edit List	40.8	66	61.76
Delete List	19.55	31.95	63.43
Show Tasks	18.65	23.8	27.61
Add Task	50.85	100.95	98.53
Edit Task	30.5	35.3	15.74
Delete Task	28.3	36.6	29.33
Refresh	5.3	6.7	26.42

Table 4.3: Average values of the monitoring overhead measurement in the Tudu application.

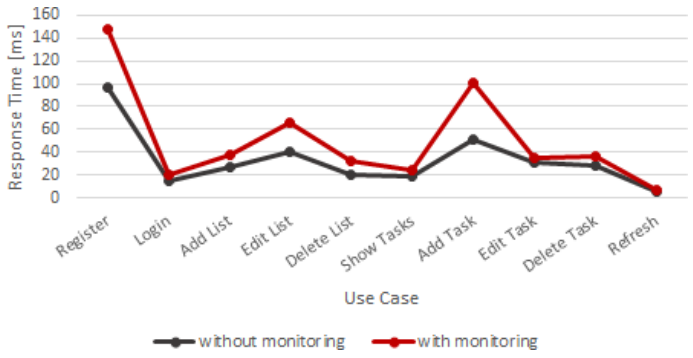


Figure 4.1: Graphical illustration of the average values of the monitoring overhead measurement in the *Tudu* application.

It is clearly visible, that the overhead percentage increases with ascending original response times: the three requests with the highest response times (*Register*, *Add Task*, and *Edit List*) all exhibit overheads of more than 52%. This indicates that the monitoring component potentially scales badly. However, a reliable statement about the scalability can not be made on the basis of the available measurements. *Delete Task*, which has a significantly smaller response time, exhibits an overhead of more than 60%, too. Furthermore, all the executed requests were relatively small (i.e., smaller than a second). Therefore, another experiment is required that is designed to determine the scalability of the monitoring component.

Scalability Measurement

In order to prove or confute the assumption of bad scalability, the monitoring component was subjected to another experiment. A very small web application has been created that simply serves GET requests to one single URL. As soon as the URL is requested, the application queries a number of items (personal data) from a NoSQL database, iterates over the result set, and concatenates the string representation of all the items. The resulting string is returned as response body in plain text. The number of treated items can be specified as a request parameter. Consequently, the volume of the request can be manually regulated.

Ten different requests were made, each with a different input parameter, starting with 50 and doubling the value each time. The environment of the deployed application was the same as for the precedent measurement and the request has been repeated 20 times each for all the different input parameters.

Table 4.4 provides the averages of the measured values. Looking at the percentage increase the overhead entails, it is evident that the assumption of the last experiment about scalability can be confirmed. The observation becomes even clearer when looking at Figure 4.2, which graphically visualizes the values. The monitoring overhead does not scale with increasing requests sizes. Small requests (i.e., <20ms) exhibit an overhead that is not significant. But as soon as the size of the request increases, the overhead gets more significant. Requests taking longer than 75 milliseconds, which is still a very short time, already exhibit an overhead of more than 100% in this experiment. This is even worse than in the previous measurements on the *Tudu* application.

The significant overhead of the source code monitoring is not surprising, since foregoing research has already come up with similar results [DDE08, AR01, NBR10]. However, the bad scalability is an additional problem that has not been expected in this dimension. The consequences derived from these findings are discussed in the subsequent section.

# of items	without monitoring	with monitoring	increase (%)
50	12.35	13.55	9.72
100	12.7	13.75	8.27
200	19.05	20.9	9.71
400	24.7	35.2	42.51
800	31.05	48.55	56.36
1600	38.1	55.3	56.36
3200	48	68.3	42.29
6400	77.85	170.65	119.20
12800	119.5	518.15	333.60
25600	316.35	1289.4	307.59

Table 4.4: Average values of the monitoring overhead scalability measurement.

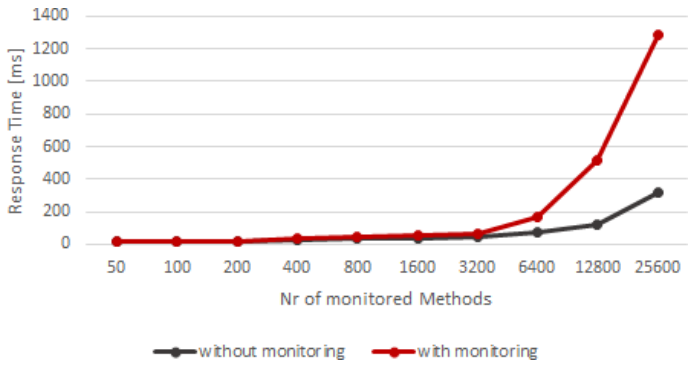


Figure 4.2: Graphical illustration of the average values of the monitoring overhead measurement on the sample application.

Discussion

From the results of the two examined exemplary applications, the following two basic statements can be derived:

- The source code monitoring entails a significant overhead to the overall performance of the monitored application.
- The entailed overhead does not scale with increasing request sizes.

Since the monitoring in an FDD system is conducted on productive systems, this observation is problematic, because typically the end-users of the system are affected by this significant performance loss. In the prototypical status the FDD system is currently in, performance is not a critical factor. The focus during development was on implementing the functionality in order to prove the approach. However, to make the FDD system ready to be used in real-world environments, the performance of the monitoring has to be addressed. Therefore, the following two improvements are suggested:

- **Improve parallelism in the source code of the Monitoring Component:** The current implementation of the Monitoring Component executes much of the work in the same thread in which the target application runs. It therefore blocks the execution of the target application after each particular method execution. This includes the construction of the domain model entities based on the monitored data. The only step that is executed in a separate thread is the transmission of the data to the Feedback Handler. To improve parallelism and thereby reduce the monitoring overhead, as much work as possible should be delegated to other threads. The only thing that has to be executed in the same thread is the actual measurement of the raw data.
- **Choose a sampling-based approach to monitor target applications:** Currently, the Monitoring Component instruments every single method execution of the target application. This enables the gathering of a large amount of data in short time periods. However, each individual request is affected by the additional overhead. To solve this problem, a sampling-based approach can be chosen to reduce the number of requests, and hence the number of users, that are affected by the monitoring overhead. Sampling reduces the number of affected users by only instrumenting a subset of all the events in the target application [ED05, AR01]. Various research has investigated different approaches to optimize the principles on which sampling is based [TFK11, DDE08, NBR10]. We suggest to introduce

one of the proven principles to enable sampling in the monitoring component of the FDD system.

4.2.2 IDE Plug-in Performance

As soon as the FDD plug-in nature is activated in the Eclipse IDE, the Feedback Builder is triggered with each build process (see Section 3.8.2). The execution of the Feedback Builder takes some time, since it requires the fetching of a huge amount of data from the Feedback Handler, as well as analyzing and combining this data with the static source code artifacts. The following evaluation investigates the effect of the Feedback Handler on the performance of the Eclipse build process.

Study Setup

To make statements about the performance of the Feedback Builder, it is compared with the Java and the Maven Builder, which are used to build a regular Java projects. Four different build types are investigated:

- **Incremental Build with 1 changed source file:** An incremental build after a change in only one single Java source file. This is the typical case if a source is modified and saved, which triggers the build process (if automatic build is activated).
- **Incremental Build with 3 changed source files:** An incremental build after changing three different source files. This is achieved through a method name refactoring. The name of the method is changed in the defining source file and two other files using the changed method are also modified.
- **Full Build with 50 sources:** A full build of a project with 50 Java source files. A full build typically only happens if a project is imported into the Eclipse workspace or after it has been cleaned.
- **Full Build with 80 sources:** A full build of a project with 80 Java source files.

If Eclipse is started in the debug mode, the time each builder takes during a build process can be read from the console output. For this purpose a file `.options` with the content shown in Listing 4.1 has to be created in the root folder of the Eclipse installation. If Eclipse is now launched in the debug mode, which can be achieved through passing the `-debug` parameter, the build time for each particular builder is written onto the console. The measurements were conducted with the Eclipse Version Luna (Eclipse 4.4) running inside a Java runtime environment 1.7 on Windows 8.1. The Feedback Handler operated on an external Ubuntu server.

```
1 org.eclipse.core.resources/debug=true
2 org.eclipse.core.resources/build/invoking=true
```

Listing 4.1: Eclipse debug configuration in the `.options` file for measuring the builder times.

Results

Each of the four described build types was executed 20 times. Table 4.5 shows the average times required for the Java and the Maven Builders together as well as for the Feedback Builder. Figure 4.3 graphically visualizes the values. A complete overview of all the measured build times can

be found in Appendix B. The time the Feedback Builder requires is significantly higher than the time the other two builders require. This is not a surprising fact, since the Feedback Handler has to request a lot of data from the Feedback Handler. Therefore, a considerable amount of network latency is included in the build time.

Build Type	Java and Maven Builder	Feedback Builder
Incremental Build (1 changed source)	0.123s	0.844s
Incremental Build (3 changed sources)	0.779s	5.452s
Full Build (50 sources)	3.534s	48.405s
Full Build (80 sources)	5.736s	73.532s

Table 4.5: Average values of the Eclipse builder execution times.

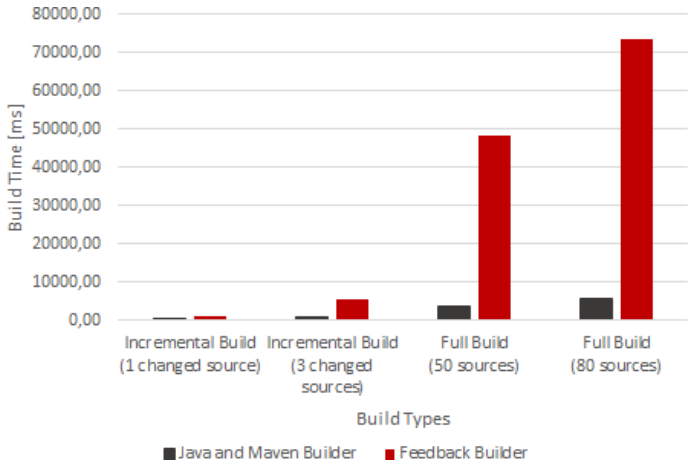


Figure 4.3: Graphical illustration of the values of Table 4.5.

Discussion

The comparison with the Java and Maven builder illustrates that fetching, and analyzing runtime data is an expensive task that takes significant additional effort to the build process. However, we argue that the relatively long build time is tolerable due to the following reasons:

- **Scope of executed work:** Feedback-Driven Development is an expensive activity by its nature. Network latency is included in the build time, due to the communication with the Feedback Handler, which makes performance optimizations on the application-level difficult. As discussed in the approach, as much of the analysis and filtering part as possible is done on the server (i.e., the Feedback Handler). However, some of the analysis part has to be done on the client, because it is tightly connected to the static analyis part. Moving this part to the server would require moving IDE concepts to the server, too. It needs to be investigated, whether the Eclipse Flux project is capable of supporting such a scenario (see Section 6.2.2 for further details).
- **Few full builds:** The Feedback Builder reaches its substantially high build times only in the case of a full build. Full builds, however, are typically executed very rarely. If a project is

imported into the Eclipse workspace, a full build is triggered. Apart from that, a full build is only executed after a project has been completely cleaned up. The number of full builds is therefore very small. If automatic build is activated, incremental builds are triggered each time a source file is saved. This means, that in most cases, incremental builds happen after only one file has been modified. An exception is a refactoring over multiple source files. Even though, building one or a few files also results in significantly higher build times, the absolute times shift from less than a second to a few seconds, which is still acceptable.

- **Non-blocking:** The Eclipse build process is executed in a separate task and neither freezes the user interface of the Workbench nor blocks the user from continuing with the development. Hence, the developer can continue working, while the runtime data is analyzed. The developer still obtains the runtime Feedback very fast and is able to detect new performance problems as soon as they are introduced.

Apart from these reasons, speeding up the build time of the Feedback Builder can improve the developers experience when working with PerformanceHat. A possible way of reducing the build time is to deactivate part of the Feedback Builder and only perform the analysis the developer is actually interested in. From a technical point of view, this feature would be seamlessly integratable into a future release of PerformanceHat, since the Feedback Builder is already partitioned into different builder participants. Further details about introducing a feature to activate and deactivate builder participants are described in Section 6.2.2.

Chapter 5

Related Work

Performance Analysis in software systems has been, and still is, an active field of research. This chapter lists and summarizes previous work from research that is related to Feedback-Driven Development.

5.1 Detection of Performance Problems

Various research has been conducted investigating ways to detect performance regressions automatically and furthermore detect the source of the problem. In [NNH+14], Nguyen *et al.* propose to assemble a regression cause repository containing data about performance tests as well as information about the causes of past regressions. Mining this repository shall asisst performance teams in identifying the causes of performance regressions. They investigate the effect of source code changes to the performance and were able to prove that even a single additional control statement in the source code can lead to significant performance issues. They classified performance regression causes into different categories and determined the most frequent categories through interviewing performance engineers of industrial software systems. On the basis of the outcomes, they developed the regression cause repository and the mining approach for the automatic detection of the regression causes.

A very similar principle has been developed by Foo *et al.* [FJA+10]. They extracted performance metrics during the execution of performance tests and stored this information in the repository. New test results were then compared to the stored data to identify performance regressions. By conducting case studies, they could prove that their tool is capable of detecting performance issues that are often overlooked in practice. They furthermore proved their approach to be scalable to large industrial software systems.

Jiang *et al.* also used performance tests to identify performance anomalies [JHHF08]. Altman *et al.* [AAFM10] present the design and methodology of the tool *WAIT*, a tool to identify the root cause of idle time in modern enterprise-class server applications.

In [vHWH12], Van Hoorn *et al.* present Kieker, a framework that is capable of monitoring and analyzing an application's runtime behavior. As in FDD, applications are monitored in production. The analysis is done by different plug-ins. Kieker itself includes a number of plug-ins, providing different analyses and visualizations of the monitored data. The extensible architecture allows to add further plug-ins tailored to specific needs. The main difference of Kieker compared to FDD is its lack of IDE integration.

5.2 Visualization of Runtime Information

Often, information about how an application behaves at runtime is indeed available, but developers do not use it. Different research has been conducted on trying to visualize runtime information in a way that enables developer to benefit of it during development. In [BRB10] Bergel *et al.* propose two profiling blueprints visualizing runtime information in form of polymetric views [LD03]. The goal is to identify performance bottlenecks and provide hints on how to remove them by reducing the execution time of frequently used methods on the one hand and eliminating invocations of slow methods on the other hand. The *structural distribution blueprint* visualizes the total execution time, the number of executions, and the number of different receivers (i.e., the number of different objects the method has been invoked on) for each method called in a program execution. These metrics are displayed alongside the static structure of the code (i.e., the class hierarchy). While this allows to detect classes and methods that are central to the execution, it does not provide any information about the call hierarchy of the invoked methods. This information is visualized by the *behavioral distribution blueprint* which displays the metrics alongside the method call invocations. The first two metrics, namely number of calls and total execution of methods, are also considered in this view and are complemented with information about what the method returns, whereat it is distinguished between void (which implies the method likely performs a side effect), constant values per receiver, and all remaining cases. This information allows to determine whether caching can be used and whether it could bring any performance improvements. The approach has been proven by conducting a case study that illustrated how the proposed blueprints help developers to detect the bottlenecks of a program execution. After introducing caching mechanisms according to the information gained from the blueprints, they achieved execution speedups from 43-45%.

In [RGN07] Röthlisberger *et al.* develop a tool that supports developers performing maintenance activities on complex object-oriented software systems. They consider a software system as a set of features and want to enhance traditional IDEs with a feature-centric perspective. Their tool consists of three different views. The *Compact Feature Overview* shows a list of all methods used in a feature, which allows to visually compare different features. Each method is colored according to its *Feature Affinity* [Gr] measuring the relevancy of a method: *singleFeature* methods participate in exactly one feature, *lowGroupFeature* methods in less than 50%, *highGroupFeature* in more than 50%, and *infrastructuralFeature* methods in all of the features. The *Feature Tree* view visualizes the method call tree of a selected feature. To reduce the size of the tree and increase the expressiveness common subexpressions and recurring method calls, as a result of loops are removed. Finally, the *Feature Artifact Browser* displays the source code artifacts (packages, classes, methods) actually used in the feature. This allows the developer to focus on the artifacts that are of interest for the feature he is working on.

To validate their tool Röthlisberger *et al.* conducted a user study with twelve subjects, each of them fixing two different bugs in a software system, one of them with a traditional IDE and the other one with the feature-centric environment. They were able to prove that their approach has a positive effect on the program comprehension of developers and reduces the amount of time required to localize and resolve bugs in a system.

In [RDT08] Röthlisberger *et al.* discuss the so called *unanticipated partial behavioral reflection* (UPBR), an approach that allows for applying changes to a running application without having to shut it down (dynamic adaption). The main purpose of such a system is to enable on-the-fly debugging and monitoring of applications. While there are a some existing proposals for such reflective computations in Smalltalk as well as in Scala, they all lack in one of the important characteristics of UPBR. Some of them need to be prepared at load time (because bytecode has to be transformed) which prohibits the system from being unanticipated, others do not allow the selection of operation occurrences (spatial and/or temporal) that are of interest. Röthlisberger *et al.*

developed a system for Smalltalk called *GEPPETTO* that addresses all the required characteristics and enhances the existing reflective model of the Smalltalk language.

5.3 Integrating Runtime Information into the IDE

Apart from FDD, previous research already pursued the idea of integrating and visualizing operational data in the IDE. In [RHV+09a] Röthlisberger *et al.* describe an approach to integrate dynamic information about applications into traditional IDEs to help developers understand the behavior and structure of the system at runtime. They want to overcome the weaknesses of debuggers and profilers, which in fact allow to inspect the dynamic behavior of a system, but they both only provide volatile information (i.e., the information gets lost after the session has been terminated) and lack the ability to aggregate the metrics of multiple runs of the target application. To seamlessly integrate the dynamic metrics into the IDE, they developed an Eclipse plug-in called Senseo. Senseo uses an aspect-oriented approach to gather the runtime data and construct a Calling Context Tree [ABL97], a generic data structure that allows to store different metrics for each calling context during execution. The collected data, comprising receiver-, argument-, and return types of message invocations, the number of invocations and created objects, and the amount of allocated memory, is displayed in the IDE in different places. The Java editor is extended with popups when hovering a method header or body. The rulers (vertical bars on the left and the right side of the editor) are complemented with dynamic metrics about the respective methods using a heat coloring scheme (blue, yellow, and red). Finally the artefacts in the package explorer (packages and classes) are decorated with an icon indicating the aggregated degree of contribution to a selected dynamic metric (e.g., allocated memory). The overall goal of these visual pieces of information is to quickly find hot spots in the source code which then allows to eliminate performance issues. A functional overview of Senseo is provided in detail in [RHV+09b].

In [RHB+12] Röthlisberger *et al.* first present an extended version of Senseo. A collaboration view located next to the source code editor displays a list of all dynamic collaborators (callers and callees) of the currently selected source code artifact. The Calling Context Ring Chart (CCRC) visualizes the underlying Calling Context Tree (CCT) and provides mechanisms to navigate and explore subtrees. The authors validated the benefits of this extended version of Senseo by conducting an extensive user evaluation with 30 professional Java developers. Typical software maintenance tasks had to be solved by an experimental group using Senseo and by a control group using the standard Eclipse IDE. The result of the evaluation showed that the time spent on solving the maintenance tasks is statistically significantly lower (17.5%) when having Senseo available. Even the hypothesis that Senseo increases the correctness of the tasks could be accepted (increase in correctness: 33.5%).

A performance validation furthermore indicated that Senseo is fast enough to be used with large projects even when updating the dynamic information in the IDE very frequently (i.e., once per second).

In [TSH12] Taeumel *et al.* present a new approach to directly integrate runtime information into programming environments together with a respective prototypical implementation called *VIVIDE*. The approach addresses the problem that developers cannot directly view runtime data as questions arise, which results in numerous context switches in thinking during development and maintenance of a system. They further investigated that often programmers prefer to mentally simulate the program execution in order to understand the dynamic behavior of the system, which increases their cognitive load and is therefore very error-prone.

Taeumel *et al.* suggest to provide information to the user in a more task-oriented way. They introduce a so called horizontal unbounded tape which is a scrollable view allowing to place any number of editors horizontally beside each other. On the left side of the scrollable view there

is a fixed project outline that allows to navigate through the whole project. The various editors in the scrollable view allow to combine static with dynamic data of the currently visible source code artifacts. For example, beside a traditional source code editor, an object explorer can be displayed showing runtime information and a call tree showing all invoked methods. Overlays are used to indicate relations between elements in different editors.Besides focusing on how to graphically present the data to the user, they further implemented a way to gather the runtime data by executing the existing test cases.

These approaches all have one common difference compared to Feedback-Driven Development. They all gather the runtime data from the developers local machine rather than from applications deployed to productive environments. This entails that the analysis and visualizations conducted by these tools do not provide inferences about the application's behavior and performance in production. Since performance problems are often connected to productive systems and do not appear during development, this is an essential difference.

Chapter 6

Final Remarks

This chapter summarizes the contribution of this thesis and provides a conclusion of the discussed material. Moreover, some possible directions for future work which have not been addressed so far are briefly summarized.

6.1 Conclusion

This thesis underlines the importance of integrating performance engineering aspects into techniques of empirical software engineering in order to support the DevOps approach in cloud computing. As a foundation for the contribution of the thesis, relevant background information on performance engineering and statistical methods is summarized. Based on that knowledge, an approach to implement Feedback-Driven Development is introduced in order to answer the research questions:

> RQ1: *"How can performance data from cloud infrastructures be integrated into software development environments?"*

Chapter 3 starts with illustrating the basic problem of runtime data not being used in software development activities. Feedback-Driven Development is described as part of a Continuous Delivery Pipeline and proposed as a solution to increase the developer's awareness of operational data. IDE integration is suggested as the main required characteristic of an FDD system. It is illustrated how raw runtime data can be gradually transformed into valuable feedback through filtering and aggregation techniques and how this feedback can be integrated into the static source code views of traditional IDE's. A conceptual architecture visualizes how these concepts can be embedded into an FDD system.

> RQ2: *"How can this data be used to predict performance problems in advance?"*

On the basis of the statistical methods and concepts introduced in Section 2.3, Section 3.3.3 proposes the application of statistical models to predict the evolution of an application's performance based on the available operational data. The moving average is suggested as a method to calculate the performance from available time series data and it is investigated how change point analysis can be combined with static data from version control systems in order to detect performance critical change sets.

Part of the thesis contribution is a prototypical implementation of the proposed FDD concept. The prototype gathers the required runtime data through instrumenting the running application on productive environments. The frontend is integrated into the Eclipse IDE to provide the developer with the determined runtime feedback. The architecture as well as the most important

implementation details of the system are explained in Section 3.4. The exploratory study in Chapter 4 shows that PerformanceHat combines important characteristics of the DevOps methodology that have not been covered in their entirety in previous research. A quantitative evaluation discloses the monitoring overhead of the FDD system as well as the execution times of the builder mechanism in the Eclipse IDE and lists possible improvements.

6.2 Future Work

The following section summarizes the most important improvements that can be implemented as future work.

6.2.1 Conceptual Improvements

The conceptual improvements address open points that improve or extend the Feedback-Driven Development approach and increase the benefits for a developer using an FDD system.

Statistical Methods

So far, the prediction of an application's performance is exclusively based on the principle of the moving average. The analysis module should be extended with other statistical methods, such as the Autoregressive (AR) model, the Autoregressive Moving Average (ARMA) model, or the Autoregressive Integrated Moving Average (ARIMA) model. Detailed information about these models can be found in [BJR13] and [SS10]. After having implemented them, the different statistical models can be evaluated against each other in order to find the best model to predict future application performance.

The Change Point Analysis method has been introduced in Section 2.3.3 to identify significant changes in the performance of an application. However, despite being part of the approach, this method has not been integrated into the prototype so far. Besides implementing a service to calculate change points inside the available procedure execution time series, this would include integrating the FDD system into version control systems, in order to combine the gained knowledge from the Change Point Analysis with source code changes, as proposed in 2.3.3. The FDD system should therefore be extended with a component integrating with established version control systems such as Git[1] or Mercurial[2].

Business Metrics

Our approach so far only focuses on technical feedback and metrics. From a business perspective it may be beneficial to extend the FDD approach to also monitor and analyze business-related metrics. A first step could be to analyze the importance of source code artifacts according to its relevance for particular business transactions. It could even be possible to calculate how much monetary value a source code artifact generates in production. Another possible improvement would be to gather metadata about the users of the application to determine which features are used in which geographical locations and at which times during the day. Such information may help to optimize the source code to individual conditions. However, further effort is required to investigate whether such analyses have the potential to provide additional benefits to the development process and whether business metrics are directly mappable to source code artifacts at all.

[1] http://git-scm.com
[2] http://mercurial.selenic.com

User Evaluation

In order to definitively prove whether the FDD approach is able to provide the targeted benefits to the user and to further investigate if developers do make proper use of the system and increase their productivity when working on performance related issues, an extensive user evaluation would be required. A possible study setup is to compare an intervention group, using PerformanceHat, and a control group using a plain IDE, both solving the same development tasks.

6.2.2 Technical Improvements

The technical improvements address possible extensions of the prototypical implementation to support nice-to-have requirements. To expand the research prototype into a releasable product, further work would be required that is not discussed here.

Extension of the Functional Range

Currently, our prototype is able to detect hotspot methods and critical loops. Those two use cases already make it possible to find major performance bottlenecks in the source code. However, the system is designed to support more use cases, as described in the conceptual part of the thesis (see Section 3.3.2). The system as a whole and especially the domain model are already implemented to support further use cases. Implementing new advices therefore takes much less time, since the whole technology stack is already implemented and a lot of code can be reused. The existing model still covers enough aspects of the runtime system to be able to detect most of the performance anti patterns discussed in Section 2.2.

Individual Control over Builder Participants

As discussed in the evaluation (see Section 4.2.2), the builder of the FDD IDE plug-in increases the build time significantly, which is due to the number of required remote calls. The builder of the FDD plug-in is already split into different builder participants, each of which does the work for one particular use case. The builder participants are registered in a registry and the builder itself delegates all the work to the registered builder participant. To narrow down the build time, the prototype could be extended such that a user can individually turn on or off particular builder participants. If, for example, a developer is not interested in loop execution time, she could completely deactivate the respective builder participant, and thereby decrease the required time to build the application.

Fine-Grained Thresholds

In the current version of PerformanceHat, threshold values that classify source code entities (procedures, loops) as performance critical are defined on the level of applications in the project properties (see Figure 3.14). However, in most cases, it would be more reasonable to define those thresholds for particular classes, or even methods, to define the threshold according to the particular code. A smart way of implementing this feature would be to allow the developer to specify a Java annotation (see Listing 6.1), which keeps the non-intrusiveness of the system alive.

```
1 @Threshold(maxExecTime=400, maxCpuUsage=25)
2 public List<Item> getItems() {
3 }
```

Listing 6.1: A Java annotation defining thresholds for the method `getItems()`.

Integration of External APM Providers

The integration of external metric sources, such as NewRelic[3] or Kieker[4], is only in an experimental state so far and does not cover the whole functionality that would be required to seamlessly integrate those tools. In order to support development teams that already work with these tools, an integration of the metric source would increase the value of the tool.

Support of other Programming Languages:

Currently, the prototype only supports target applications written in Java. The whole FDD approach is very tightly coupled to object-oriented systems, an adaption to languages based on other paradigms could therefore be hard to achieve and would require major changes even on a conceptual level. Furthermore, a lot of the investigated performance anti-patterns (see Section 2.2) are also related to object-oriented systems anyway. However, an expansion of the tool to further support other object-oriented languages, such as C#, could be a possible improvement.

Multiple IDE Support

PerformanceHat so far only supports the Eclipse IDE. It would be desirable to also support other IDE's, such as IntelliJ[5] or Cloud9[6]. A lot of the existing system components could be reused: the Feedback Handler and the Monitoring component are completely independent of the IDE. However, a major part of the logic is implemented as part of the Eclipse plug-in. As discussed in Section 3.8, the tight integration into the IDE lies in the nature of an FDD system, due to the static code analysis. It is therefore difficult to separate part of its logic from concrete IDEs. A possible way to increase the amount of reusable code in a multiple IDE scenario could be to make use of the Eclipse Flux[7] project. Eclipse Flux has the goal of providing a new, flexible architecture and infrastructure that allows for integrating development tools across desktop, browsers, and servers. However, further investigations would be required to evaluate the usefulness of the platform for the FDD system.

[3] http://newrelic.com
[4] http://kieker-monitoring.net
[5] https://www.jetbrains.com/idea
[6] https://c9.io
[7] https://projects.eclipse.org/proposals/flight

Appendix A

Acronyms

AOP Aspect-Oriented Programming

API Application Programming Interface

APM application performance management

AR Autoregressive

ARIMA Autoregressive Integrated Moving Average

ARMA Autoregressive Moving Average

AST abstract syntax tree

CD Continuous Delivery

CI Continuous Integration

CPA Change Point Analysis

CRM customer relationship manamgenent

CUSUM cumulative sum

DSS decision support system

DTO data transfer object

FDD Feedback-Driven Development

IDE Integrated Development Environment

JDK Java Development Kit

JVM Java Virtual Machine

MA moving average

POJO plain old Java object

POM project object model

QoS Quality of Service

VCS version control system

Appendix B

Results of the Quantitative Evaluation

The following document contains the complete, raw data from the quantitative evaluation. The first two pages show the values of the monitoring overhead measurement and the third page the ones from the Eclipse build time measurement.

Monitoring Performance Measurement Results - Tudu Application

Use Case	Register		Login		Add List		Edit List		Delete List		Show Tasks		Add Task		Edit Task		Delete Task		Refresh	
	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w
	143	218	16	35	33	36	31	37	17	36	25	17	45	97	37	20	35	42	6	10
	108	169	19	22	34	31	43	86	21	38	26	38	51	98	38	28	29	41	6	6
	88	154	18	24	30	36	40	86	28	30	23	27	50	103	32	32	28	41	6	6
	87	143	19	21	25	39	41	27	21	32	17	29	50	106	42	41	27	42	5	7
	83	141	14	24	26	31	45	87	21	34	18	26	60	98	32	35	21	40	5	7
	94	140	18	22	23	59	41	34	20	30	18	28	60	94	21	34	27	32	5	7
	103	147	11	10	26	36	40	94	20	32	14	30	53	96	29	34	34	39	5	6
	101	153	14	10	27	35	40	114	19	31	21	17	53	112	21	49	26	38	5	6
	96	152	11	12	21	37	38	31	18	21	25	16	62	108	33	38	37	31	5	7
	93	147	17	21	26	41	29	103	19	35	19	16	51	105	31	42	35	37	5	7
	88	141	15	30	23	38	45	39	17	32	14	36	53	140	14		29	42	5	7
	86	142	14	22	26	36	43	95	20	33	15	26	41	98	34		24	33	5	6
	95	136	14	14	28	33	40	35	22	31	16	18	57	102	27		24	41	5	6
	94	132	13	12	29	35	43	84	21	28	16	24	47	105	33		25	39	6	7
	110	141	12	28	28	36	50	32	19	29	18	27	53	104	35		27	36	6	7
	114	136	16	22	27	35	47	87	19	33	20	21	51	41	24		37	38	5	6
	93	143	15	21	27	37	45	34	19	35	14	16	38	96	33		21	29	5	6
	89	144	14	11	24	40	41	91	17	35	22	16	51	107	32		32	29	6	6
	89	144	13	10	26	41	36	37	16	33	14	25	50	107	30		25	33	5	7
	92	147	14	21	28	42	38	87	17	31	18	23	41	102	32		23	29	5	7
Average	97,3	148,5	14,85	19,6	26,85	37,7	40,8	66	19,55	31,95	18,65	23,8	50,85	101	30,5	35,3	28,3	36,6	5,3	6,7

w/o: requests without monitoring
w: requests with monitoring

Monitoring Performance Measurement Results - Sample Web Application

Items	50		100		200		400		800		1600		3200		6400		12800		25600	
	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w	w/o	w
	8	7	10	6	9	16	29	32	30	48	34	46	48	63	93	170	121	343	322	1336
	9	6	9	7	24	19	24	32	31	47	34	35	56	74	96	175	112	481	318	1272
	7	6	8	19	21	20	18	40	28	49	40	65	52	67	125	157	96	432	310	1334
	8	7	14	18	22	18	26	22	26	41	42	71	58	65	55	335	113	493	295	1326
	7	11	7	8	20	21	26	38	31	52	38	59	44	66	61	182	113	700	361	1327
	9	13	8	8	22	23	26	38	36	53	34	68	54	70	56	153	132	512	304	1260
	7	16	17	18	24	21	26	33	40	49	38	60	50	62	59	230	144	349	341	1255
	13	19	7	18	21	21	20	35	38	71	38	36	28	62	82	154	134	442	320	1266
	7	18	14	16	22	20	21	37	38	46	38	61	44	72	81	158	142	441	302	1274
	17	19	17	15	17	25	23	46	32	48	30	49	54	67	78	226	130	678	301	1268
	17	17	20	7	18	19	31	33	34	43	44	68	42	58	60	144	108	750	303	1275
	18	21	17	8	23	22	21	23	28	47	44	62	38	61	78	151	113	397	290	1285
	19	14	14	13	20	22	24	29	24	28	40	50	56	83	101	160	115	394	295	1275
	17	14	6	8	17	21	23	23	33	60	38	49	52	62	80	143	113	751	312	1271
	17	16	15	18	8	27	27	45	32	54	32	64	56	59	89	153	120	604	316	1309
	7	17	7	21	20	23	23	42	31	49	38	52	56	85	93	149	118	518	317	1310
	20	7	14	18	14	18	32	36	28	53	44	53	46	75	60	133	118	386	345	1278
	17	18	14	12	20	21	25	42	24	49	32	49	34	81	66	157	115	499	331	1267
	6	12	20	19	19	21	28	41	29	42	50	55	46	73	69	140	125	563	320	1284
	17	13	16	18	20	20	21	37	28	42	34	54	46	61	75	143	108	630	324	1316
Average	12,35	13,55	12,7	13,75	19,1	20,9	24,7	35,2	31,05	48,55	38,1	55,3	48	68,3	77,85	170,65	119,5	518,15	316,35	1289,4

w/o: requests without monitoring
w: requests with monitoring

Eclipse Build Time Measurements

Build Type	Incremental Build (1 changed source)		Incremental Build (3 changed sources)		Full Build 50 sources)		Full Build (80 sources)	
Builder	JMB	FB	JMB	FB	JMB	FB	JMB	FB
	112	974	811,2	8932	4320	53607	8347	86884
	79	920	1639,2	10358	3764	48420	5634	76320
	83	836	559,2	9152	3810	45039	7010	75337
	78	1040	1146,4	5460	2983	49333	6792	72013
	204	798	576	3366	3010	51340	6720	69320
	173	786	671,2	5301	3056	51234	6340	72923
	99	786	658,4	4450	3457	45201	4301	73419
	97	740	803,2	4426	3842	43202	4582	73428
	94	794	819,2	4142	3987	56730	4296	69207
	75	830	790,4	4856	4100	44294	5092	75245
	103	792	756	6903	4117	46732	5039	73457
	125	932	667,2	3950	3749	46863	5140	71244
	179	937	620	4105	3529	49156	4998	75230
	183	1010	587,2	4035	2915	48857	6023	62341
	204	874	791,2	5254	3934	48332	5870	68273
	140	832	674,4	4832	2989	45700	5203	70035
	73	810	810,4	4992	3239	49134	5830	82273
	104	640	760	4920	3247	48114	5835	76320
	133	703	665,6	4154	3104	48412	5923	73846
Average	123,05	843,89	779,28	5452,00	3534,32	48405,26	5735,53	73532,37

JMB Java and Maven Builder
FB Feedback Builder

Appendix C

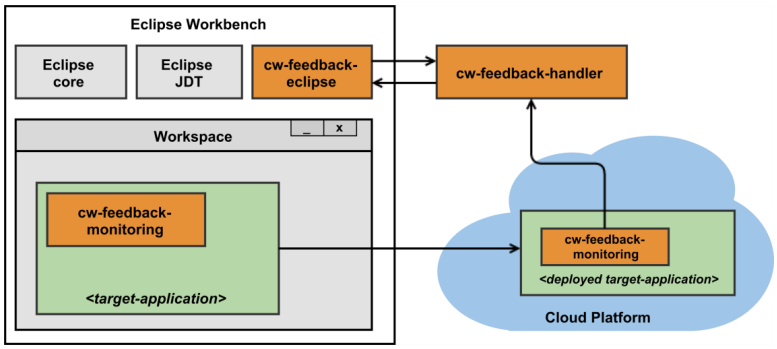
PerformanceHat User Guide

The `PerformanceHat User Guide` attached below contains information about how to use the FDD system to monitor a target application. Furthermore, technical details that are important for further development are sumnmarized.

PerformanceHat User Guide

System Overview

The picture below gives an overview of the FDD prototype. The prototype is integrated into the Eclipse IDE. In order to use it, the FDD Eclipse plugin (cw-feedback-eclipse) has to be installed into an Eclispe installation containing the Eclispe JDT (Java Development Tools). The Feedback Handler can be deployed to any Java webserver. The *target-application* is the application that wants to make use of the FDD system. In order to do so, the Monitoring component has to be integrated. The following guide describes how to set up a target application and the Eclipse IDE in order to use the FDD prototype. A second part is focused on technical details and how to build and deploy the particular components.



Usage

Set up a target application for Monitoring

Preconditions

The only pre-condition a Java application has to meet is having Maven as a build system. This allows the FDD Monitoring component to be injected using maven configurations.

Register the Application on the Feedback Handler

In order to use FDD, an application has to be registered on the Feedback Handler. This is done by calling the following URL:

`<FH-root-URL>/monitoring/register?name=<application-id>`

The `<application-id>` can be any identifier (i.e., a name without spaces) for the application. If an application with the same name is already registered, a respective error message is returned. Otherwise, the application is registered and an access-token is returned.

Create the required property File

Add a file `config.properties` into the folder `src/main/resources` with the following content:

```
monitoring.feedback_handler_url = < FH-root-URL >
monitoring.app_id = <application-id>
monitoring.access_token = <access-token>
```

Adapt the Project Object Model (`pom.xml`)

Several adaptions of the `pom.xml`-File are required. First, add the following properties:

```
<properties>
    <jdk.version>1.7</jdk.version>
    <aspectj.version>1.8.2</aspectj.version>
</properties>
```

Next, add the following dependencies to the `<dependency>`-section:

```
<dependency>
    <groupId>cloudwave</groupId>
    <artifactId>cw-feedback-monitoring</artifactId>
    <version>0.0.1-SNAPSHOT</version>
</dependency>
<dependency>
    <groupId>org.aspectj</groupId>
    <artifactId>aspectjrt</artifactId>
    <version>${aspectj.version}</version>
</dependency>
```

Add the following two plugin configurations to the `<build>`-section

```
<plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <executions>
    <execution>
        <phase>package</phase>
            <goals><goal>single</goal></goals>
        </execution>
    </executions>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <finalName>${project.build.finalName}-complete</finalName>
        <appendAssemblyId>false</appendAssemblyId>
    </configuration>
</plugin>
<plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>aspectj-maven-plugin</artifactId>
    <version>1.7</version>
    <configuration>
        <complianceLevel>${jdk.version}</complianceLevel>
        <aspectLibraries>
            <aspectLibrary>
                <groupId>cloudwave</groupId>
                <artifactId>cw-feedback-monitoring</artifactId>
            </aspectLibrary>
        </aspectLibraries>
        </configuration>
        <executions>
        <execution>
            <goals><goal>compile</goal></goals>
        </execution>
    </executions>
    <dependencies>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjtools</artifactId>
            <version>${aspectj.version}</version>
        </dependency>
    </dependencies>
</plugin>
```

Finally, in order to remove the error marker that eclipse shows now in the `pom.xml`, add the following configuration to the `<pluginManagement>`-part in the <build>-section:

```
<plugin>
	<groupId>org.eclipse.m2e</groupId>
	<artifactId>lifecycle-mapping</artifactId>
	<version>1.0.0</version>
	<configuration>
		<lifecycleMappingMetadata>
			<pluginExecutions>
				<pluginExecution>
					<pluginExecutionFilter>
						<groupId>org.codehaus.mojo</groupId>
						<artifactId>aspectj-maven-plugin</artifactId>
						<versionRange>[1.7,)</versionRange>
						<goals><goal>compile</goal></goals>
					</pluginExecutionFilter>
					<action>
						<ignore></ignore>
					</action>
				</pluginExecution>
			</pluginExecutions>
		</lifecycleMappingMetadata>
	</configuration>
</plugin>
```

Info: A complete example of a `pom.xml`-File can be found in the Appendix of this Guide.

Copy Maven Dependencies to the Classpath

This step is only required in development mode, if the Eclipse Instance used for the FDD system is started out of the development eclipse using the PDE target platform, because it isn't possible to install the m2e-plugin in the target platform. Therefore, the maven dependencies of the target application have to be added to its classpath with the following maven command:

```
mvn eclipse:eclipse
```

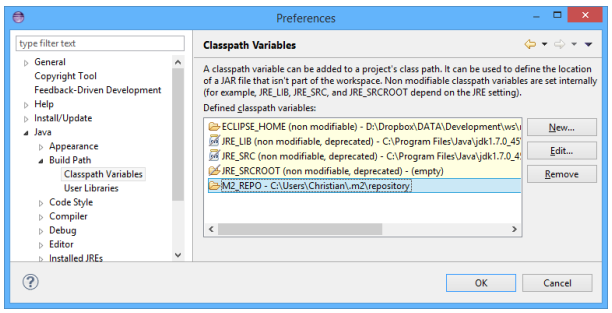
Set up the Eclipse IDE

Installation of the Plug-in

Install the FDD plug-in into the Eclipse workbench or start the target platform.

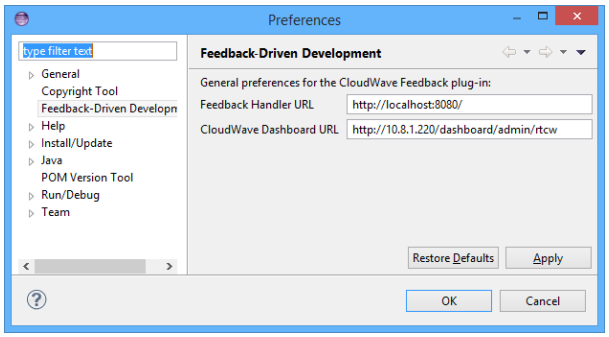
Set Maven Repository variable

Again, this step is only required if the Eclipse instance from the PDE target platform is used and the maven dependencies have been added to the classpath. To resolve the dependencies, Eclipse needs to know the path to the local maven repository. Add a variable 'M2_REPO' with the path of the local maven repository to the classpath variables (Preference → Java → Build Path → Classpath Variables).



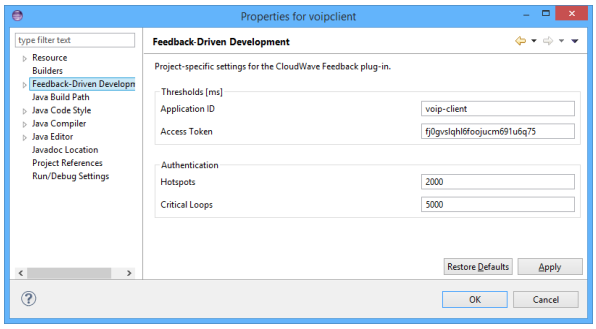
Edit the Workspace Preferences

Go to the Preferences and chose the Section ‘Feedback-Driven Development’. Specify the Root-URL of the Feedback-Handler and optionally the URL of the CloudWave Dashboard, if the FDD prototype is used together with CloudWave.



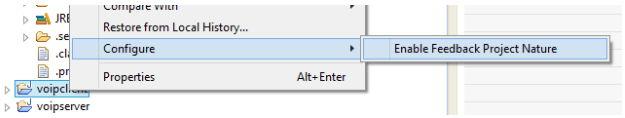
Edit the Project Properties

In the Package- or Project-Explorer of the IDE, right-click on the target application, go to the Project Properties and chose the Section ‘Feedback-Driven Development’. Add the Application-ID and the Access-Token of your target application and specify the desired threshold values for the respective warnings.

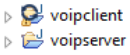


Enable the FDD Project Nature

Right-click on the target application and click on ‘Configure → Enable Feedback Project Nature’.



A blue decorator (FDD icon) should appear on the project to indicate that the nature is enabled:

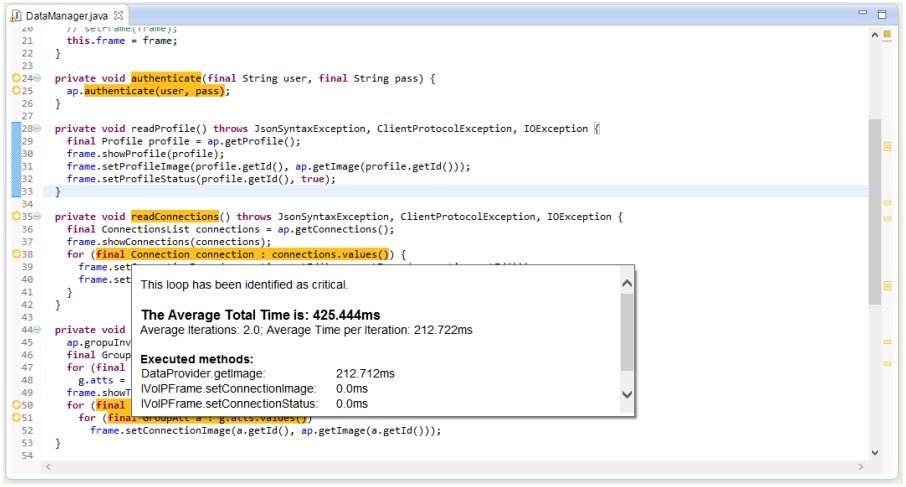


Working with the FDD plug-in

If everything works correctly, the runtime feedback should appear as soon as the application has been executed the first time with the Monitoring component integrated. Warnings should be distributed among the Java files. The following pictures indicate it should look like.

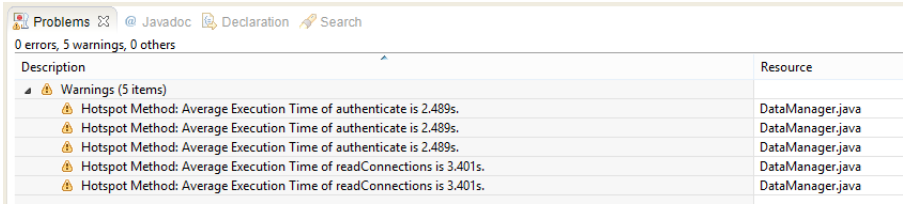
Editor

The warnings are shown in the Java Editor, as known from the Java compiler errors and warnings. The yellow circles on the left side indicate a warning of the FDD plug-in:

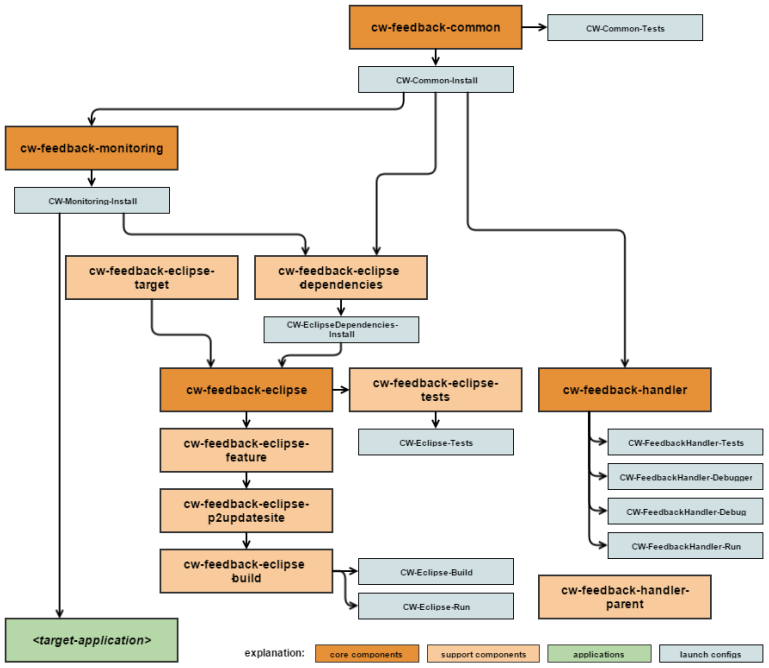


Problems View

As an other error and warning markers, the FDD warnings are also shown in the Problems View:



Overview of the System Components



Projects

This following Table briefly summarizes the different projects. Technical details are given further down.

cw-feedback-monitoring	Contains the code that has to be appended in an application that wants to make use of the Feedback Handler / Eclipse plug-in.
cw-feedback-common	Contains code that is shared among the two projects 'cw-feedback-handler', 'cw-feedback-monitoring' and 'cw-feedback-eclipse'.
cw-feedback-handler-parent	This project only contains a parent `pom.xml` for cw-feedback-common and cw-feedback-handler that allows to build both projects with one maven command during continuous integration
cw-feedback-handler	Comprises the code of the Feedback Handler.
cw-feedback-eclipse-target	Specifies the target platform against which the Eclipse plug-in ('cw-feedback-eclipse') is developed.
cw-feedback-eclipse-dependencies	Specifies the dependencies of the 'cw-feedback-eclipse'. This is a workaround to overcome the issue of using non-OSGi dependencies inside an Eclipse plug-in (more detailed information in Technical Documentation further down)
cw-feedback-eclipse	Comprises the code of the Eclipse plug-in.
cw-feedback-eclipse-tests	Contains the test cases for 'cw-feedback-eclipse'.

cw-feedback-eclipse-feature	Specifies a feature (containing the Eclipse plug-in) that can be installed in an Eclipse Workbench.
cw-feedback-eclipse-p2updatesite	Specifies the update-site providing the feature from 'cw-feedback-eclipse-feature'.
cw-feedback-eclipse-build	Provides the configuration to build the plug-in, feature, and update-site.
<target-application>	a placeholder for any application that is used in the described setup (i.e. that makes use of the Feedback Handler and the Eclipse plug-in)

Launch Configurations

The following launch configurations are provided:

CW-Common-Tests	Run all tests of the project 'cw-feedback-common'.
CW-Common-Install	Install the project 'cw-feedback-common' into the local maven repository.
CW-Monitoring-Install	Install the project 'cw-feedback-monitoring' into the local maven repository.
CW-EclipseDependencies-Install	Create a jar-File of all the dependencies in its pom.xml-File and copy the created jar into the lib-folder of the project 'cw-feedback-eclipse'.
CW-Eclipse-Tests	Run all (SWTBot)-tests of the project 'cw-feedback-eclipse'.
CW-Eclipse-Run	Run the Target Platform (Eclipse Workbench) with the plug-in 'cw-feedback-eclipse'.
CW-Eclipse-Build	Build the eclipse plug-in from 'cw-feedback-eclipse', the feature specified in 'cw-feedback-eclipse-feature', and the update site specified in 'cw-feedback-eclipse-p2updatesite'.
CW-FeedbackHandler-Tests	Run all tests of the project 'cw-feedback-handler'.
CW-FeedbackHandler-Run	Run the Feedback Handler on a local Jetty instance (on localhost:8080).
CW-FeedbackHandler-Debugger[1]	Run the Debugger for the Feedback Handler.
CW-FeedbackHandler-Debug	Run the Feedback Handler in Debug mode (requires 'CW-FeedbackHandler-Debugger' to be launched beforehand).

[1] To use this launch configuration a workspace variable MVN_EXEC has to be set containing the path to the maven execution file (on windows maven.bat). This can be done by

- Open Window → Preferences → Run/Debug → String Substitution
- Click New… and add a variable with the name=MVN_EXEC and value=<path-to-mvn-execution-file>

Technical Details

Deployment

The Eclipse Plugin can be deployed manually or using tycho:

Manual Deployment

Export the plugin `cw-feedback-eclipse-p2updatesite`:

- Right Click on project → Export → Deployable Feature → Next
- Select features
- Choose output directory
- Switch to 'Options' tab → Choose 'Categorize repository' → Browser to category file
- By default, the Eclipse update manager shows only features with a category
- Therefore, you should always include a category for your exported feature to make it easy for the user to install your feature
- Finish

Links

- Vogella - 6. Create update site for your plug-in

Tycho Deployment

- open console
- navigate to the directory `[gitrepo]/cw-feedback-eclipse-build`
- run `mvn verify` (or `mvn clean verify`)
- now the folder [gitrepo]/cw-feedback-eclipse-p2updatesite/target/repository contains the update site for the plugin

Links

- Vogella - Eclipse Tycho for building Eclipse Plugins and RCP applications - Tutorial
- Maven Tycho - Solution of the dependency problem

Eclipse Plug-in Projects

cw-feedback-eclipse

- Eclipse Project Type: Plug-in Project
- Contains the actual eclipse plugin with the IDE extensions
- Add external libraries
- Links
  - Vogella - Extending the Eclipse IDE - Plug-in development - Tutorial

cw-feedback-eclipse-dependencies

- Eclipse Project Type: General Project
- This project only specifies the (maven) dependencies of the actual eclipse plugin.
- This is a workaround to overcome the problem of referencing non-OSGi dependencies in the eclipse plug-in.
- The project is independent of all the other eclipse-related projects and not a child of 'cw-eclipse-build'
- The dependencies are specified as usual in the pom.xml file
- By executing maven install a jar-File is created that contains all the dependencies (cw-feedback-eclipse-dependencies-1.0.0-SNAPSHOT.jar) and the created jar is copied to the lib-folder of 'cw-feedback-eclipse'
  - This is done using the maven plugin 'org.apache.felix:maven-bundle-plugin'
- Remarks about the configuration in the pom.xml-file:
  - <Import-Package>;</Import-Package> is required, because otherwise some imports are added that cannot be resolved
  - the inline=true option allows to generate all the dependencies into one jar-File, otherwise all (transitive) dependencies have their own jar-Files

- Links
  - Maven Tycho - Solution of the dependency problem
  - Creating OSGi bundles of your Maven dependencies
  - Apache Felix - Bundle Plugin for Maven
  - bundle:bundleall
  - stackoverflow - OSGI transitive dependencies level
- Alternative Approach
  - There are some repositories containing OSGi-bundles of non-OSGi projects, e.g. SpringSource Enterprise Bundle Repository
  - Problem: For most of the projects old versions are provided

cw-feedback-eclipse-tests

- Eclipse Project Type: Fragment Project
- This project provides test cases for 'cw-feedback-eclipse'
- It is a fragment project with 'cw-feedback-eclipse' as its host
  - A fragment is always defined for another plug-in (host plug-in)
  - At runtime the fragment is merged with its host and both projects are just one
  - this makes fragments useful for tests, because classes of the host plug-in can be accessed, even if the host plug-in does not define them as external API
  - Remark: Fragments are always optional for their host plug-in
- There are three types of tests
  - Normal Junit Tests
  - Junit Plug-in Tests (Workbench-related tests)
  - SWTBot Tests (UI-related tests)
- Links
  - Vogella - Eclipse Fragment Projects - Tutorial
  - Testing RCP Applications in Tycho can cause Serious Harm to your Brain
  - SWTBot tests
    - Add SWTBot to target platform
    - SWTBot/UsersGuide
    - SWTBotTestUtil implementation

cw-feedback-eclipse-target

- Eclipse Project Type: General Project
- The target platform is the set of artifacts from which Tycho resolves the project's dependencies
  - You can either use a target definition based on
    - the Eclipse p2 update manager: This can be done by adding a repository to the pom in the build project, an example can be found here
    - Target Definition files (our case): In this case a separate project has to be created containing the target platform file
      - It expects exactly one file <artifactId>.target in the project's base directory
- Links
  - Vogella - Eclipse Target Platform - Tutorial
  - http://wiki.eclipse.org/Tycho/Target_Platform#Target_files
  - http://wiki.eclipse.org/Tycho/Packaging_Types#eclipse-target-definition

cw-feedback-eclipse-feature

- Eclipse Project Type: Feature Project
- A feature describes a list of plug-ins and other features which can be understood as a logical unit
- Links
  - Vogella - Eclipse Feature Projects - Tutorial

cw-feedback-eclipse-p2updatesite

- Eclipse Project Type: General Project
- Contains the category definition
  - a category contains a list of features
- By default, the Eclipse update manager shows only features with a category
  - Therefore, you should always include a category for your exported feature to make it easy for the user to install your feature.
- Links
  - Vogella - 6. Create update site for your plug-in

cw-feedback-eclipse-build

- Eclipse Project Type: General Project
- This is the parent project of all the other eclipse-related projects
  - it contains the parent pom.xml-file
- It specifies how the artifacts are built using tycho

Further Topics

Adding external libraries to Eclipse Plug-ins (OSGi bundles)

Adding an external library to the eclipse plugin can be done as follows

- open MANIFEST.MF → switch to Runtime tab → Classpath → Add…
  - Important: Each jar-File has to be added separately, it doesn't work by adding a whole folder
- Right-click on Project → Plug-in Tools → Update Classpath…

This should actually be avoided. Instead the library should be added as dependency to 'cw-feedback-eclipse-dependencies'. This (only) dependency is itself added as described above to the eclipse plug-in.

DataFormatException when launching the Target Platform

TODO

- Problem:
  - The following error occurs when launching the Runtime Workbench (i.e. target platform): org.eclipse.jface.resource.DataFormatException: For input string: "1x"
- Solution
  - The source of the problem is the package 'org.eclipse.ui.tests'
    - http://dev.eclipse.org/mhonarc/lists/jdt-ui-dev/msg01342.html
    - https://bugs.eclipse.org/bugs/show_bug.cgi?id=374292
  - To exclude the package
    - Configure Run Configuration
    - in Tab 'Plug-ins' → Launch with: select 'plug-ins selected below only'
    - exclude the test-fragment and 'org.junit'
- Links
  - Target Platform – how to deal with optional RAP dependencies

Appendix

Complete `pom.xml`-File Example

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">

    <modelVersion>4.0.0</modelVersion>

    <groupId>eu.cloudwave.samples</groupId>
    <artifactId>person-app</artifactId>
    <version>0.0.1-SNAPSHOT</version>

    <packaging>war</packaging>

    <properties>
        <jdk.version>1.7</jdk.version>
        <aspectj.version>1.8.2</aspectj.version>
    </properties>

    <dependencies>

        <dependency>
            <groupId>com.google.guava</groupId>
            <artifactId>guava</artifactId>
            <version>17.0</version>
        </dependency>

        <dependency>
            <groupId>org.springframework</groupId>
            <artifactId>spring-webmvc</artifactId>
            <version>4.0.6.RELEASE</version>
        </dependency>
        <dependency>
            <groupId>org.springframework.data</groupId>
            <artifactId>spring-data-mongodb</artifactId>
            <version>1.5.2.RELEASE</version>
        </dependency>

        <dependency>
            <groupId>cloudwave</groupId>
            <artifactId>cw-feedback-monitoring</artifactId>
            <version>0.0.1-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.aspectj</groupId>
            <artifactId>aspectjrt</artifactId>
            <version>${aspectj.version}</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <source>1.7</source>
                    <target>1.7</target>
                </configuration>
            </plugin>

            <plugin>
                <groupId>org.eclipse.jetty</groupId>
                <artifactId>jetty-maven-plugin</artifactId>
                <version>9.2.2.v20140723</version>
                <configuration>
                    <httpConnector><port>9000</port></httpConnector>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <executions>
                    <execution>
```

```
                        <phase>package</phase>
                        <goals><goal>single</goal></goals>
                    </execution>
                </executions>
                <configuration>
                    <descriptorRefs>
                        <descriptorRef>jar-with-dependencies</descriptorRef>
                    </descriptorRefs>
                    <finalName>${project.build.finalName}-complete</finalName>
                    <appendAssemblyId>false</appendAssemblyId>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>aspectj-maven-plugin</artifactId>
                <version>1.7</version>
                <configuration>
                    <complianceLevel>${jdk.version}</complianceLevel>
                    <aspectLibraries>
                        <aspectLibrary>
                            <groupId>cloudwave</groupId>
                            <artifactId>cw-feedback-monitoring</artifactId>
                        </aspectLibrary>
                    </aspectLibraries>
                </configuration>
                <executions>
                    <execution>
                        <goals><goal>compile</goal></goals>
                    </execution>
                </executions>
                <dependencies>
                    <dependency>
                        <groupId>org.aspectj</groupId>
                        <artifactId>aspectjrt</artifactId>
                        <version>${aspectj.version}</version>
                    </dependency>
                    <dependency>
                        <groupId>org.aspectj</groupId>
                        <artifactId>aspectjtools</artifactId>
                        <version>${aspectj.version}</version>
                    </dependency>
                </dependencies>
            </plugin>
        </plugins>
        <pluginManagement>
            <plugins>
                <plugin>
                    <groupId>org.eclipse.m2e</groupId>
                    <artifactId>lifecycle-mapping</artifactId>
                    <version>1.0.0</version>
                    <configuration>
                        <lifecycleMappingMetadata>
                            <pluginExecutions>
                                <pluginExecution>
                                    <pluginExecutionFilter>
                                        <groupId>org.codehaus.mojo</groupId>
                                        <artifactId>aspectj-maven-plugin</artifactId>
                                        <versionRange>[1.7,)</versionRange>
                                        <goals><goal>compile</goal></goals>
                                    </pluginExecutionFilter>
                                    <action><ignore></ignore></action>
                                </pluginExecution>
                            </pluginExecutions>
                        </lifecycleMappingMetadata>
                    </configuration>
                </plugin>
            </plugins>
        </pluginManagement>
    </build>
</project>
```

Bibliography

[AAFM10] Erik Altman, Matthew Arnold, Stephen Fink, and Nick Mitchell. Performance analysis of idle programs. In *ACM Sigplan Notices*, volume 45, pages 739–753. ACM, 2010.

[ABH03] U.A. Acar, G.E. Blelloch, and R. Harper. *Selective Memoization*. Research paper. School of Computer Science, Carnegie Mellon University, 2003.

[ABL97] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 Conference on Programming Language Design and Implementation*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM.

[Acc] Telerik Data Access. How to: Detect and solve n+1 problems. http://docs.telerik.com/data-access/developers-guide/profiling-and-tuning/profiler-and-tuning-advisor/data-access-profiler-n-plus-one-problem, accessed October 2014.

[Ado06] Steve Adolph. What lessons can the agile community learn from a maverick fighter pilot? In *Agile Conference, 2006*, pages 6 pp.–99, July 2006.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, Feb 2009.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, April 2010.

[AI10] S.A. Ahson and M. Ilyas. *Cloud Computing and Software Services: Theory and Techniques*. An Auerbach book. Taylor & Francis, 2010.

[AIS+77] Christopher Alexander, Sara Ishikawa, Murray Silverstein, M Jacobson, I Fiksdahl-King, and S Angel. A pattern language. 1977. *Center for Environmental Structure Series*, 1977.

[AR01] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 168–179, New York, NY, USA, 2001. ACM.

[Art03] John Arthorne. Project builders and natures, January 2003.

[BBG10] R. Buyya, J. Broberg, and A.M. Goscinski. *Cloud Computing: Principles and Paradigms*. Wiley Series on Parallel and Distributed Computing. Wiley, 2010.

[BFKB+14] D. Bruneo, T. Fritz, S. Keidar-Barner, P. Leitner, F. Longo, C. Marquezan, A. Metzger, K. Pohl, A. Puliafito, D. Raz, A. Roth, E. Salant, I. Segall, M. Villari, Y. Wolfsthal, and C. Woods. Cloudwave: Where adaptive cloud management meets devops. In *Computers and Communication (ISCC), 2014 IEEE Symposium on*, volume Workshops, pages 1–6, June 2014.

[BGG14a] Martin Brandtner, Emanuel Giger, and Harald Gall. Supporting continuous integration by mashing-up software quality information. In *IEEE CSMR-WCRE 2014 Software Evolution Week (CSMR-WCRE)*, pages 109–118, Antwerp, Belgium, FEB 2014. IEEE.

[BGG14b] Martin Brandtner, Emanuel Giger, and Harald Gall. Supporting continuous integration by mashing-up software quality information. In *IEEE CSMR-WCRE 2014 Software Evolution Week (CSMR-WCRE)*, pages 109–118, Antwerp, Belgium, FEB 2014. IEEE.

[BHM07] Walter Binder, Jarle Hulaas, and Philippe Moret. Advanced java bytecode instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 135–144, New York, NY, USA, 2007. ACM.

[Bir14] Jens Birchler. Sqa-timeline: A timeline-based visualisation approach for software evolution data. Master's thesis, 2014.

[BJR13] G.E.P. Box, G.M. Jenkins, and G.C. Reinsel. *Time Series Analysis: Forecasting and Control*. Wiley Series in Probability and Statistics. Wiley, 2013.

[BKB00] Anna Bouch, Allan Kuchinsky, and Nina Bhatti. Quality is in the eye of the beholder: Meeting users' requirements for internet quality of service. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '00, pages 297–304, New York, NY, USA, 2000. ACM.

[BRB10] Alexandre Bergel, Romain Robbes, and Walter Binder. Visualizing dynamic metrics with profiling blueprints. In Jan Vitek, editor, *Objects, Models, Components, Patterns*, volume 6141 of *Lecture Notes in Computer Science*, pages 291–309. Springer Berlin Heidelberg, 2010.

[Bro98] W.J. Brown. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley computer publishing. Wiley, 1998.

[BW14] A. Brown and A.B.G. Wilson. *The Architecture of Open Source Applications*. Creative Commons, 2014.

[BYV+09] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. Cloud computing and emerging {IT} platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems*, 25(6):599 – 616, 2009.

[CBWDR10] Victor Chang, David Bacigalupo, Gary Wills, and David De Roure. A categorisation of cloud computing business models. In *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, pages 509–512, Washington, DC, USA, 2010. IEEE Computer Society.

[CC] Inc. Cunningham & Cunningham. Anti pattern. http://c2.com/cgi/wiki?AntiPattern, accessed December 2014.

[CG11] J. Chen and A.K. Gupta. *Parametric Statistical Change Point Analysis: With Applications to Genetics, Medicine, and Finance*. SpringerLink : Bücher. Springer, 2011.

[Che] Chef.io. What is continuous delivery? https://www.chef.io/solutions/continuous-delivery, accessed December 2014.

[Cit14] Jürgen Cito. Statistical methods in managing web performance. Master's thesis, 2014.

[CLFG14] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald C. Gall. The making of cloud applications an empirical study on software development for the cloud. *CoRR*, abs/1409.6502, 2014.

[CSLD14] Jürgen Cito, Dritan Suljoti, Philipp Leitner, and Schahram Dustdar. Identifying root causes of web performance degradation using changepoint analysis. In Sven Casteleyn, Gustavo Rossi, and Marco Winckler, editors, *Web Engineering*, volume 8541 of *Lecture Notes in Computer Science*, pages 181–199. Springer International Publishing, 2014.

[DDE08] M. B. Dwyer, M. Diep, and S. Elbaum. Reducing the cost of path property monitoring through sampling. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 228–237, Washington, DC, USA, 2008. IEEE Computer Society.

[DDN02] S. Demeyer, S. Ducasse, and O. Nierstrasz. *Object-Oriented Reengineering Patterns*. The Morgan Kaufmann Series in Software Engineering and Programming. Elsevier Science, 2002.

[DGS02] Robert F. Dugan, Jr., Ephraim P. Glinert, and Ali Shokoufandeh. The sisyphus database retrieval software performance antipattern. In *Proceedings of the 3rd International Workshop on Software and Performance*, WOSP '02, pages 10–16, New York, NY, USA, 2002. ACM.

[DHV03] Bruno Dufour, Laurie Hendren, and Clark Verbrugge. *j: A tool for dynamic analysis of java programs. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 306–307, New York, NY, USA, 2003. ACM.

[Doca] MongoDB Documentation. Aggregation pipeline.

[Docb] Phabricator Contributor Documentation. Performance: N+1 query problem. https://secure.phabricator.com/book/phabcontrib/article/n_plus_one, accessed October 2014.

[ED05] S. Elbaum and M. Diep. Profiling deployed software: assessing strategies and testing opportunities. *Software Engineering, IEEE Transactions on*, 31(4):312–327, April 2005.

[et12] F. et. *A First Course on Time Series Analysis: Examples with SAS*. epubli GmbH, 2012.

[FJA+10] King Foo, Zhen Ming Jiang, B. Adams, A.E. Hassan, Ying Zou, and P. Flora. Mining performance regression testing repositories for automated performance analysis. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 32–41, July 2010.

[Fow12] M. Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2012.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Pearson Education, 1994.

[GNJ] Menasce Daniel A. Smith Connie U. Williams Dr. Lloyd G. Gunther Neil, Maddox Michael and Raj Jain. Performance engineering. http://www.mitre.org/publications/systems-engineering-guide/acquisition-systems-engineering/acquisition-program-planning/performance-engineering-, accessed November 2014.

[Gr] Orla Gr, evy. PhD thesis.

[HF10] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[Hil09] David Hilley. Cloud computing: A taxonomy of platform and infrastructure-level offerings. *Georgia Institute of Technology, Tech. Rep. GIT-CERCS-09-13*, 2009.

[Hil14] Stefan Hiltebrand. Sqa-pattern: a recognition framework for violations of conventions in software engineering. Master's thesis, 2014.

[HWM13] Christian Häubl, Christian Wimmer, and Hanspeter Mössenböck. Deriving code coverage information from profiling data recorded for a trace-based just-in-time compiler. In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, PPPJ '13, pages 1–12, New York, NY, USA, 2013. ACM.

[Hü12] Michael Hüttermann. *DevOps for Developers*. Apress, 2012.

[JHHF08] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 307–316. IEEE, 2008.

[JSS+12] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. Understanding and detecting real-world performance bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, New York, NY, USA, 2012. ACM.

[KH01] Gregor Kiczales and Erik Hilsdale. Aspect-oriented programming. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 313–, New York, NY, USA, 2001. ACM.

[Kin03] A.B. King. *Speed Up Your Site: Web Site Optimization*. Voices Series. New Riders, 2003.

[Kla12] Mantas Klasavičius. Metrics-driven development, November 2012. http://www.infoq.com/articles/metrics-driven-development, accessed December 2014.

[LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, September 2003.

[Mad07] H. Madsen. *Time Series Analysis*. Chapman & Hall/CRC Texts in Statistical Science. Taylor & Francis, 2007.

[Man] Rajesh Mansharamani. Software performance engineering. http://www.softwareperformanceengineering.com/contact.html, accessed October 2014.

[Mar00a] Robert C Martin. Design principles and design patterns. *Object Mentor*, 1:34, 2000.

[Mar00b] Robert C Martin. The principles of ood. *Object Mentor*, 1:34, 2000.

[Mar04] Fowler Martin. Inversion of control containers and the dependency injection pattern, January 2004. http://martinfowler.com/articles/injection.html, accessed December 2014.

[Mar13a] Fowler Martin. Continuous delivery, May 2013. http://martinfowler.com/bliki/ContinuousDelivery.html, accessed December 2014.

[Mar13b] Fowler Martin. Deployment pipeline, May 2013. http://martinfowler.com/bliki/DeploymentPipeline.html, accessed December 2014.

[MG11] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[Mih14] Vlad Mihalcea. Mongodb time series: Introducing the aggregation framework, January 2014.

[Mur11] John Murphy. Performance engineering for cloud computing. In *Proceedings of the 8th European Conference on Computer Performance Engineering*, EPEW'11, pages 1–9, Berlin, Heidelberg, 2011. Springer-Verlag.

[NBR10] Harish Narayanappa, Mukul S. Bansal, and Hridesh Rajan. Property-aware program sampling. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 45–52, New York, NY, USA, 2010. ACM.

[NNH+14] Thanh H. D. Nguyen, Meiyappan Nagappan, Ahmed E. Hassan, Mohamed Nasser, and Parminder Flora. An industrial case study of automatically identifying performance regression-causes. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 232–241, New York, NY, USA, 2014. ACM.

[NSML13] A. Nistor, Linhai Song, D. Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 562–571, May 2013.

[Pal10] George Pallis. Cloud computing: The new frontier of internet computing. *IEEE Internet Computing*, 14(5):70–73, 2010.

[Par07] Trevor Parsons. *Automatic Detection of Performance Design and Deployment Antipatterns in Component Based Enterprise Systems*. PhD thesis, 2007.

[PJM+02] Wim De Pauw, Erik Jensen, Nick Mitchell, Gary Sevitsky, John M. Vlissides, and Jeaha Yang. Visualizing the execution of java programs. In *Revised Lectures on Software Visualization, International Seminar*, pages 151–162, London, UK, UK, 2002. Springer-Verlag.

[PM04a] Trevor Parsons and John Murphy. Data mining for performance antipatterns in component based systems using run-time and static analysis, 2004.

[PM04b] Trevor Parsons and John Murphy. A framework for automatically detecting and assessing performance antipatterns in component based systems using run-time analysis. In *The 9th International Workshop on Component Oriented Programming, part of ECOOP*, 2004.

[Pow02] D.J. Power. *Decision Support Systems: Concepts and Resources for Managers*. Quorum Books, 2002.

[RDT08] David Röthlisberger, Marcus Denker, and Éric Tanter. Unanticipated partial behavioral reflection: Adapting applications at runtime. *Comput. Lang. Syst. Struct.*, 34(2-3):46–65, July 2008.

[Rei03] Steven P. Reiss. Visualizing java in action. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 57–ff, New York, NY, USA, 2003. ACM.

[RGN07] David Röthlisberger, Orla Greevy, and Oscar Nierstrasz. Feature driven browsing. In *Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007*, ICDL '07, pages 79–100, New York, NY, USA, 2007. ACM.

[RHB+12] David Rothlisberger, Marcel Harry, Walter Binder, Philippe Moret, Danilo Ansaloni, Alex Villazon, and Oscar Nierstrasz. Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. *IEEE Trans. Softw. Eng.*, 38(3):579–591, May 2012.

[RHV+09a] D. Rothlisberger, M. Harry, A. Villazon, D. Ansaloni, W. Binder, O. Nierstrasz, and P. Moret. Augmenting static source views in ides with dynamic metrics. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 253–262, Sept 2009.

[RHV+09b] David Röthlisberger, Marcel Harry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Senseo: Enriching eclipse's static source views with dynamic metrics. In *ICSM'09*, pages 383–384, 2009.

[Rie96] A.J. Riel. *Object-oriented Design Heuristics*. Addison-Wesley Publishing Company, 1996.

[SC93] Robert C. Sharble and Samuel S. Cohen. The object-oriented brewery: A comparison of two object-oriented development methods. *SIGSOFT Softw. Eng. Notes*, 18(2):60–73, April 1993.

[SF12] P.J. Sadalage and M. Fowler. *NoSQL Distilled: A Brief Guide to the Emerging World of Polyglot Persistence*. Pearson Education, 2012.

[SMG11] Subhajyoti Bandyopadhyay Juheng Zhang Sean Marston, Zhi Li and Anand Ghalsasi. Cloud computing — the business perspective. *Decision Support Systems*, 51(1):176 – 189, 2011.

[Smi01] Connie U. Smith. Software performance antipatterns: Common performance problems and their solutions. In *In Int. CMG Conference*, pages 797–806, 2001.

[SOT+00] T. Suganuma, T. Ogasawara, M. Takeuchi, T. Yasue, M. Kawahito, K. Ishizaki, H. Komatsu, and T. Nakatani. Overview of the ibm java just-in-time compiler. *IBM Systems Journal*, 39(1):175–193, 2000.

[SS08] Paramvir Singh and Hardeep Singh. Dynametrics: A runtime metric-based analysis tool for object-oriented software systems. *SIGSOFT Softw. Eng. Notes*, 33(6):1–6, October 2008.

[SS10] R.H. Shumway and D.S. Stoffer. *Time Series Analysis and Its Applications: With R Examples*. Springer Texts in Statistics. Springer, 2010.

[ST13] F. Schulz and W. Theilmann. Towards systematic mobile cloud performance analysis. In *Wireless and Mobile Networking Conference (WMNC), 2013 6th Joint IFIP*, pages 1–4, April 2013.

[SW00] Connie U. Smith and Lloyd G. Williams. Software performance antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 127–136, New York, NY, USA, 2000. ACM.

[SW03] Connie U Smith and Lloyd G Williams. More new software performance antipatterns: Even more ways to shoot yourself in the foot. In *Computer Measurement Group Conference*, pages 717–725. Citeseer, 2003.

[SYK+01a] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. *SIGPLAN Not.*, 36(11):180–195, October 2001.

[SYK+01b] Toshio Suganuma, Toshiaki Yasue, Motohiro Kawahito, Hideaki Komatsu, and Toshio Nakatani. A dynamic optimization framework for a java just-in-time compiler. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 180–195, New York, NY, USA, 2001. ACM.

[Tay00] Wayne A. Taylor. Change-point analysis: A powerful new tool for detecting changes, 2000.

[TFK11] Johnson J. Thomas, Sebastian Fischmeister, and Deepak Kumar. Lowering overhead in sampling-based execution monitoring and tracing. In *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, LCTES '11, pages 101–110, New York, NY, USA, 2011. ACM.

[TSH12] Marcel Taeumel, Bastian Steinert, and Robert Hirschfeld. The vivide programming environment: Connecting run-time information with programmers' system knowledge. In *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! 2012, pages 117–126, New York, NY, USA, 2012. ACM.

[vHRH+09] André van Hoorn, Matthias Rohr, Wilhelm Hasselbring, Jan Waller, Jens Ehlers, Sören Frey, and Dennis Kieselhorst. Continuous monitoring of software services: Design and application of the kieker framework. Forschungsbericht, Kiel University, November 2009.

[vHWH12] André van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 247–248, New York, NY, USA, 2012. ACM.

[Vog09] Lars Vogel. Eclipse builder - tutorial, October 2009. http://www.softwareperformanceengineering.com/contact.html, accessed January 2015.

[VRCG+99] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, pages 13–. IBM Press, 1999.

[VRMCL08] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Rev.*, 39(1):50–55, December 2008.

[WFP07] M. Woodside, G. Franks, and D.C. Petriu. The future of software performance engineering. In *Future of Software Engineering, 2007. FOSE '07*, pages 171–187, May 2007.

[WTK+08] Lizhe Wang, Jie Tao, M. Kunze, A.C. Castellanos, D. Kramer, and W. Karl. Scientific cloud computing: Early definition and experience. In *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, pages 825–830, Sept 2008.