Extraction of User Behavior Profiles for Software Modernization

Master's Thesis

Gunnar Dittrich

May 11, 2016

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
M. Sc. Christian Wulf
Dr. Wolfgang Goerigk (b+m Informatik AG)

Eidesstattliche Erklärung

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

Kiel,

_____

Abstract

In the field of software modernization static and dynamic analysis of the system is a core part of the modernization process. The static analysis focuses on the extraction of structural and architectural information by searching the source code for dependencies. The dynamic analysis tracks method calls at runtime and enables statements about the real usage of the software in a productive environment. Often the monitoring is executed from the system's point of view and not from the user behavior perspective focusing on architectural information. The idea of user behavior profile extraction was picked up by research groups dealing with automatic generation of test cases. Based on user sessions, a user behavior model is created. The derived test cases are augmented with workload characteristics for simulating realistic load during the test execution.

This thesis we combine the concept of user behavior profiles and software modernization. We introduce a session extraction tool and a behavior model extraction tool based on the TeeTime framework. The Pipe-and-Filter architecture provides a good performance and reusability. The session extraction tool processes custom records created with the Kieker framework in order to create log files of user sessions. The session log files are analyzed with the behavior model extraction tool. It supports several visualizations, calculates think time statistics and is highly customizable for analyzing single processes.

We instrument the b+m bAV-Manager, an session-based, workflow-oriented administration software for customer and calculation data of insurers, developed by the b+m Informatik AG. The implementation of instrumentation components is realized with interceptors of the Spring framework. In an experiment we monitor the web application and record the user behavior with the session extraction tool. Based on the session logs we analyze the screenflow and the workflow with the behavior extraction tool. The results allow us, to make quantitative and qualitative statements about the user behavior in comparison with the application model. We derive several suggestions regarding the imminent modernization of the b+m bAV-Manager, that improve the matching of defined screen- and workflows and the extracted user behavior profiles.

Contents

List of Acronyms

API application programming interface

bAV betriebliche Altersversorgung

BPMN Business Process Model and Notation

CBMG customer behavior model graph

CRM customer relationship management

CSV comma-separated values

DSL domain specific language

DTO data transfer object

EMF Eclipse Modeling Framework

EMP Eclipse Modeling Project

FSM finite-state machine

GMF Graphical Modeling Framework

GQM goal, question, metric

HVI high value interface

IRL Instrumentation Record Language

JAR Java Archive

jBPM Java Business Process Model

JPA Java Persistence API

M2M model-to-model

M2T model-to-text

MDSD model-driven software development

MDSE model-driven software engineering

MDM model-driven modernization

MIT Massachusetts Institute of Technology

OCL Object Constraint Language

PDF Portable Document Format

RQ research question

SaaS software as a service

SHA-2 Secure Hash Algorithm 2

SUT system under test

T2M text-to-model

TCP Transmission Control Protocol

UI user interface

UML Unified Modeling Language

XMI XML Metadata Interchange

XML Extensible Markup Language

Chapter 1

Introduction

In this chapter we present the motivation for our work (Section 1.1) and the goals, we want to achieve (Section 1.2). Finally, we give an overview of the document's structure.

1.1 Motivation

A general problem of software system is software evolution over time. This process is accelerated by technical innovations and new standards. Aging software is more susceptible for errors, more difficult to adjust to new requirements, and therefore expensive in terms of maintenance. Usually the performance decreases and the software architecture degenerates. One countermeasure is continuous reengineering to prevent quality problems. But once the software reaches a certain size and complexity a software modernization is required. Classical approaches of translating an old system to a new system, maybe in another programming language, did not prove itself in the past. In preparation for a software modernization, an application system is typically analyzed regarding improvements. The static analysis focuses on the structure of an application. The dynamic analysis, however, explores the behavior of a system at runtime. This enables the monitoring of operation execution and timing measurements. With the information from timed activities in a software, the behavior of users can be tracked. The extraction of user behavior profiles via dynamic analysis has already been used for several projects. One example is the automatic generation of test cases and workload based on realistic user behavior. These approaches use the Kieker framework for writing monitoring records.

In this thesis, we want to use user behavior profiles for the software modernization process. Thereby we concentrate our work on the user behavior on the screenflow and the workflow level. In applications usually the user activities within a session can be seen as a movement from view to view, known as screenflow. Possible and sensible movements are specified by the application model, but the correlation with the actual usage is generally unknown. With the tools designed and implemented in this work, we want to make the comparison between application model and the real user behavior. Additionally, there are applications that define a model for the workflow level. Those workflow-oriented applications orchestrate the screenflows with system tasks to workflows defined by process definitions. Here, the same deviation between process definition and user behavior has to be assumed. For example, an user starts a process and at a certain point in the execution

he always leaves it to change a system setting. One could ask, if the change of the system setting should be integrated to the process or the process is not used correctly. In this work we want to monitor and analyze a session-based and workflow-oriented software, where a software modernization is imminent. The b+m bAV-Manager developed by the b+m Informatik AG can be considered as such a legacy software system and also contains a workflow engine. It is a Java web application for ensurers in the field of company pension schemes (betriebliche Altersversorgung (bAV)). It relies on an old version of the b+m Informatik AG's b+m gear development platform.

In this work we want to implement monitoring and analysis tools to visualize user behavior in session-based applications in general. In an experiment with the b+m bAV-Manager we want to evaluate them. The goal is the preparation of the modernization of the software by analyzing the user behavior in its screen- and workflows. A result might be the refinement of screenflows and process definitions for the next major release based on the latest platform version. Another important information would be a prioritization of screenflows and processes for a step-by-step modernization. Furthermore, we want to confirm usability problems discovered in a case study from another perspective.

1.2 Goals

We define four goals for our work.

1.2.1 G1: Implementation of Monitoring Components

The first goal of the master's thesis is the development of the required monitoring components for recording user behavior. For this purpose one or more record types have to be designed and implemented (see Chapter 3). They may carry information about sessions, workflows, screenflows, events, think times etc. We aim at reusing some components of similar approaches analyzing the user behavior in software systems like WESSBAS in Section 2.5.

The records will be emitted by the application at runtime. Therefore a configurable interceptor is needed to collect and aggregate the required pieces of information. Both, records and interceptors, are expected to be application-independent and configurable. A platform dependency of the interceptors has to be accepted, because they can not abstract from used technologies.

1.2.2 G2: Implementation of the User Behaviour Model Analysis

In order to implement the analysis node, we have to identify one or more user behavior model representations, that can ideally be plotted graphically. We give an overview of those representations in Section 2.3. Afterwards the required analysis components can be developed (see Chapter 4).

G2.1: Identification of Appropriate User Behavior Model Representations

In order to evaluate the user behavior a definition of an user behavior model is required. This goal is to identify possible model types and their graphical representation. The chosen model or models should improve the understanding of workflow usage and provide information for a subsequent software modernization. They should also be able to show workflow usage in a process like the creation of an opinion and their interactions.

G2.2: Implementation of Analysis Components

The records produced by the application represent the basis for user behavior modeling. They have to be processed quickly and persisted in a convenient way, especially regarding workload peaks of the application. The goal is the implementation with a high performance Pipe-and-Filter framework.

Predefined stages can be reused, but most likely stages for custom records and for graph models have to be created. The result is a data structure that enables graphical outputs and extraction of information for software modernization.

1.2.3 G3: Evaluation of the User Behaviour Model Analysis

In the evaluation in the Master's thesis we evaluate the results of the dynamic analysis. The results of instrumenting and monitoring the b+m bAV-Manager are discussed: important views in a screenflow, obsolete views, common workflows, weak spots in process definitions, switching between processes and think times. Particularly, statements about screenflow interaction and user behavior on the process level shall be made. It will be evaluated to which extent this information can be included in the modernization process of the b+m bAV-Manager, in terms of the model-driven modernization (MDM) approach.

1.2.4 G4: Proposition of Workflow and Screenflow Improvements

The final goal is the improvement of screenflow models and process definitions of the software based on the evaluation results. Possible correlations to usability issues of the software are considered.

1.3 Document Structure

The thesis is structured as follows:

- In Chapter 2 the relevant foundations and technologies for this work are presented.
- Chapter 3 illustrates the use of the Kieker framework for monitoring. We implement the required monitoring components like records and interceptors. Finally, we describe the instrumentation of the system under test (SUT), the b+m bAV-Manager.

- The fourth chapter focuses on the implementation of two tools for user behavior extraction with the TeeTime framework. First, we introduce a Java application for collecting records and for writing session log files. Second, the implementation of the behavior model extractor is described, which performs analyses on those session logs. The visualization of user behavior models concludes Chapter 4.
- In Chapter 5 the experiment regarding user behavior in the web application b+m bAV-Manager is explained. Afterwards, we discuss the results of the experiment.
- Related work in the context of user behavior analysis and software modernization is presented in Chapter 6.
- We conclude the thesis in Chapter 7 and look forward to future work.

Chapter 2

Foundations and Technologies

This chapter introduces concepts and tools, that are used in this work. First, we describe the monitoring framework Kieker and its Instrumentation Record Language (IRL) in Section 2.1. It is used for the monitoring of user activities at run time. Section 2.2 illustrates the TeeTime framework, which we use for our implementation of a session extractor and a behavior model extractor. It allows to design reusable Pipe-and-Filter-Architectures, while maintaining a high performance. In the subsequent section we define the concept of user behavior models and give examples for visualizations. After an introduction in the model-driven software development (MDSD) in Section 2.4, the WESSBAS approach is discussed in Section 2.5. It aims at monitoring user behavior in software systems in order to generate test plans with realistic workload information. Although, we do not consider test plan generation in this thesis, our behavior meta model reuses concepts of the WESSBAS approach.

Afterwards, the mentoring company b+m Informatik AG is presented. The company follows the MDSD approach with its development platform b+m gear. A software based on the b+m gear platform is the b+m bAV-Manager, which is used for the experiment later. Finally, we explain the DynaMod project, which focuses on the MDM of legacy software systems. In the context of this work, the b+m bAV-Manager is considered as an example legacy software system. In Section 2.8 we shortly describe the goal, question, metric (GQM) approach which is the basis for our evaluation.

2.1 The Kieker Monitoring and Analysis Framework

Kieker[1] is a framework for application performance monitoring, dynamic software analysis and visualization of analysis results [Van Hoorn et al. 2009; 2012]. It has been developed for Java-based systems but it also includes monitoring adapters for other platforms. Both online and offline analysis can be configured [Van Hoorn et al. 2009].

The Kieker Framework is separated in two main parts: monitoring and analysis (see Figure 2.1). First, monitoring probes collect measurements as monitoring records which are written to a monitoring log or stream by a monitoring writer. Afterwards, the monitoring records are imported by monitoring readers on the analysis side and passed to a configurable architecture of analysis plugins.

[1] http://kieker-monitoring.net/

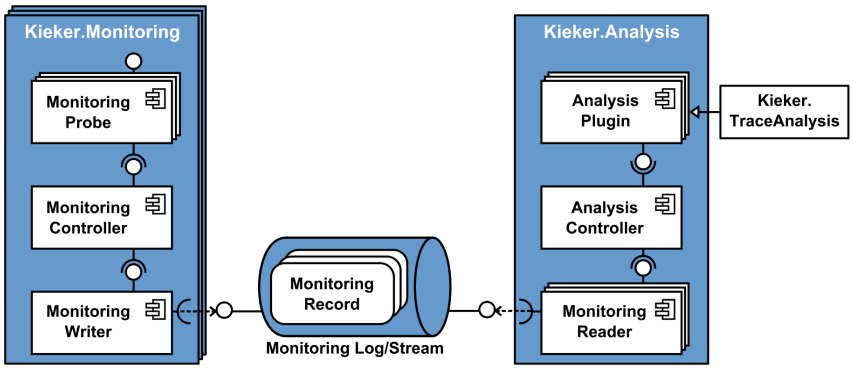

Figure 2.1. The Schematic Software Architecture of The Kieker Framework [Kieker 2015]

The data structure *Monitoring Record* represents a single measurement and contains a set of typed attributes. It also provides operations for serialization and deserialization of the attribute values. Typical applications are the measurement of resource utilization, operation executions and control flow activities. The Monitoring Records are transferred by *Monitoring Logs* or *Monitoring Streams*. They can be realized on a file system, databases, queues, channels or direct thread communication. The necessary serialization and deserialization of Monitoring Records to a respective medium-specific representation has to be handled by a matching pair of Monitoring Writer and Monitoring Reader. [Van Hoorn et al. 2009]

In contrast to static analysis, dynamic analysis with Kieker records runtime behavior of a system and is therefore able to monitor execution of operations, CPU utilization and memory usage. The final, graphical outputs are calling-dependency graphs, call trees and sequence diagrams. In practice, the framework also supports developers in locating problems and is applicable for software reverse engineering. [Van Hoorn et al. 2009]

Software systems are implemented in different programming languages. This requires monitoring components like records to be independent of programming languages. Kieker achieves the independence by defining common serialization standards and by implementing the corresponding runtime libraries. In order to define Monitoring Records for different programming languages only once, Jung [2013] introduces the Kieker IRL[2] based on Xtext[3]. It follows the MDSD approach by using models to describe the *Monitoring Record* and generating source code in the used programming language. This approach for model-driven instrumentation aims at ensuring compatibility of records across all languages.

The grammar of the IRL (see Listing 2.1) consists of four parts: header, service part, record declaration and literals/terminals.

```
1 grammar kieker.develop.rl.RecordLang with org.eclipse.xtext.common.Terminals
2 generate recordLang "http://www.cau.de/cs/se/instrumentation/rl/RecordLang"
```

[2] http://kieker-monitoring.net/blog/instrumentation-record-language-release-1-0/
[3] https://eclipse.org/Xtext/

```
3  import "http://www.eclipse.org/emf/2002/Ecore" as ecore
4  Model:
5    'package' name = QualifiedName
6    (imports += Import)*
7    (types += ComplexType)*;
8  Import:
9    'import' importedNamespace = QualifiedNameWithWildcard;
10 Type:
11   ComplexType | BaseType;
12 BaseType:
13   name=ID;
14 ComplexType:
15   RecordType | TemplateType;
16 TemplateType:
17   ('@author' author=STRING)?
18   ('@since' since=STRING)?
19   'template' name=ID (':' parents+=[TemplateType|QualifiedName] (',' parents+=[
         TemplateType|QualifiedName])*)?
20   ('{'
21     (properties+=Property | constants+=Constant)*
22   '}')?;
23 RecordType:
24   ('@author' author=STRING)?
25   ('@since' since=STRING)?
26   (abstract?='abstract')? 'entity' name=ID
27   ('extends' parent=[RecordType|QualifiedName])?
28   (':' parents+=[TemplateType|QualifiedName] (',' parents+=[TemplateType|
         QualifiedName])*)?
29   ('{'
30     (properties+=Property | constants+=Constant)*
31   '}')?;
32 Constant:
33   'const' type=Classifier name=ID '=' value=Literal;
34 Property:
35   (modifiers+=PropertyModifier)* (foreignKey=ForeignKey)?
36   (type=Classifier | 'alias' referTo=[Property|ID] 'as') name=ID
37   ('=' value=Literal)?;
38 ForeignKey:
39   'grouped' 'by' recordType=[RecordType|ID] '.' propertyRef=[Property|ID];
40 enum PropertyModifier:
41   TRANSIENT = 'transient' |
```

```
42    INCREMENT = 'auto-increment';
43  Classifier:
44    type=[BaseType|ID] (sizes+=ArraySize)*;
45  ArraySize: {ArraySize}
46    '[' (size=INT)? ']' ;
47  Literal:
48    StringLiteral | IntLiteral | FloatLiteral | BooleanLiteral | ConstantLiteral |
          ArrayLiteral | BuiltInValueLiteral;
49  ArrayLiteral:
50    '{' literals+=Literal (',' literals+=Literal)* '}';
51  StringLiteral:
52    value=STRING;
53  IntLiteral:
54    value=INT;
55  FloatLiteral:
56    value=FLOAT;
57  BooleanLiteral:
58    value=BOOLEAN;
59  ConstantLiteral:
60    value=[Constant|ID];
61  BuiltInValueLiteral: {BuiltInValueLiteral}
62    value='KIEKER_VERSION';
63  QualifiedName:
64    ID (=>'.' ID)*;
65  QualifiedNameWithWildcard:
66    QualifiedName ('.' '*')?;
67  terminal fragment NUMBER:
68      '0'..'9';
69  terminal INT returns ecore::EInt:
70      '-'? NUMBER+;
71  terminal FLOAT returns ecore::EFloatObject :
72      '-'? NUMBER+ ('.' NUMBER*) (("e"|"E") ("+"|"-")? NUMBER+)? 'f'? |
73      '-'? NUMBER+ 'f';
74  terminal BOOLEAN returns ecore::EBooleanObject :
75      'true' | 'false';
```

Listing 2.1. Xtext Grammar of the Kieker IRL

The header defines the parser, imports the terminals grammer and also imports the ecore package (lines 1-3). The service part contains the package name of the model and allows the import of record entities (lines 4-9). In the record declaration in lines 10-46 multiple records can be created. They include attributes with name and type. Finally, the language supports literals for all supported primitive types and references to default

constants (lines 47-66). The data structures of the IRL are flat and therefore only allow primitive types as data types. The terminals defined in lines 67-75 allow negative numbers for `INT`, introduce boolean values, and make sure the `FLOAT` rule does not shadow the `INT` rule [Jung 2013].

2.2 The Pipe-and-Filter-Framework TeeTime

The original Pipe-and-Filter architectural style [Shaw 1989] consists of two key components: pipes and filters. The considered task is divided in multiple, independent filters which represent subtasks. The communication and data transfer between filters is handled by pipes. The configuration is the third component in a Pipe-and-Filter architecture. It describes instances of pipes and filters and their connections. As a high level filter, configurations can be used to build a hierarchy. Ports are the fourth first-class entity and receive/send data of a particular type only. That means, filters read data from one or more of their input ports, process it, and write the results to one or more of their output ports [Allen and Garlan 1992].

Furthermore, ports allow dataflow variations. They allow them, because data may return to a previous filter (loop) or branches divide the flow by multiple output ports (if-then-else). In the original Pipe-and-Filter architectural styles (for example by Allen and Garlan [1992]; Shaw [1989]) each filter has an own internal state. This state can not be shared with other filters. However, there are applications where data sharing is needed. In that case the strict architectural style can be preserved and an update notification element has to be passed through the whole pipeline. Another option is the combination of the Pipe-and-Filter architectural style with a data-centered architecture providing a shared repository.

The TeeTime Framework[4] is an implementation of the Pipe-and-Filter architectural style focusing on performance and reusability, while addressing the challenges mentioned above [Wulf and Hasselbring 2015]. In contrast to earlier work the project shows that Pipe-and-Filter architectures can get along with little runtime overhead.

In the context of the framework filters, as well as data sources and data sinks, are called stages. TeeTime stages have an arbitrary number of input and output ports and can be composed of other stages. Ports are typed and connect stages with pipes either synchronized or unsynchronized. Figure 2.2 shows an excerpt of the TeeTime framework architecture with the main classes: configuration, stage, port, and pipe.

Below we summarize problems of the Pipe- and Filter architectural style that Wulf and Hasselbring [2015] address in their work on the TeeTime framework.

The communication between stages is handled by pipes. On a distributed system the pipe is the network and the data flow has to be converted, serialized and synchronized.

[4] `https://teetime-framework.github.io/`

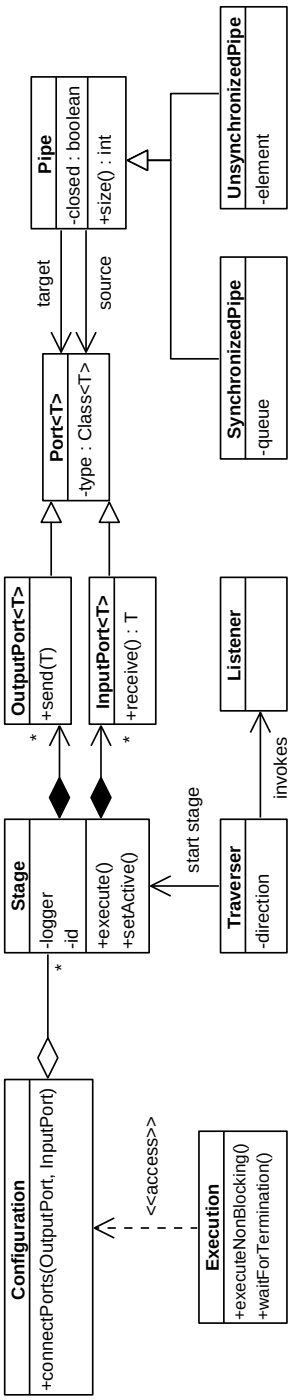

Figure 2.2. An excerpt of the TeeTime framework architecture [Wulf and Hasselbring 2015]

The TeeTime framework, however, exchanges data via references and synchronization is only required in stages running in different threads. Two stages in the same thread are connected by an unsynchronized pipe. This enables direct method calls and also ensures back-pressure. Back-pressure means, that the execution priority is higher for stages at the end of the pipeline. Two stages in different threads are connected by a synchronized queue.

TeeTime uses a two-phase approach for automatic pipe selection. When ports of two stages are connected in the configuration a placeholder pipe is created. Stages can be declared as active, if they actively pull input from the previous filter and push output to the following filter. Otherwise the stages are passive filters. After the declaration of active and passive stages TeeTime goes through the whole architecture beginning at one randomly chosen stage. All placeholders are replaced by synchronized pipes if the consuming stage is active or by unsynchronized pipes if it is passive. This process allows automatic stage connection at runtime. The threads for active stages and their passive successors are created automatically by the framework. If a stage does not define any input ports and creates multiple output elements, it is called a *producer*. A *self-terminating* producer emits only a finite number of elements, while a *non-terminating* producer runs an indefinite time. The latter is terminated when all self-terminating producers and consumers finished their execution. A *consumer* has only input ports and may produce output elements if it consumes input elements.

TeeTime avoids runtime overhead by handling composite stages as wrappers. At runtime its child stages are integrated to the Pipe- and Filter-architecture like primitive stages. This addresses the problem of active composite stages in dedicated threads. They would require heavy synchronization to their child stages. Also the problem of non active composite stages is handled. They induce a high number of active child stages. Configurations may contain errors. For example, two different active producer stages send elements to a third passive consumer stage. Based on this information, it is impossible to decide, who executes the passive stage. TeeTime checks for inconsistent configurations with a graph coloring algorithm. It uniquely colors each active stage and their passive successor stages. If one passive stage is colored twice or is not colored once, a conflict is reported. Errors at runtime can have an internal cause (e.g., calculation error) or an external cause (e.g., network time out). They have to be identified and handled appropriately. Since a configuration may declare two instances of one stage type, TeeTime assigns unique identifiers and unique loggers to each stage. Furthermore the framework instantiates a customizable error handler for each stage. They decide in case of an error whether the execution is able to continue or it has to be aborted.

In their performance evaluation Wulf and Hasselbring [2015] show that the framework abstractions do not induce any perceptible runtime overhead. Additionally, the throughput of Pipe-and-Filter architectures on cache-coherent multi-core systems can be heavily increased with a lock-free pipe implementation restricted to a single producer and a single consumer thread. They disproved the assumed slow performance of Pipe-and-Filter architectures in general.

2.3 Modeling User Behavior

The modeling of user behavior is usually carried out in two steps. First, an Application Model [Van Hoorn et al. 2008] has to be created. It consists of the actions and the states (that can be reached via execution of actions) in an application from the user perspective. In Blum [2013] this type of model is called *interaction model* and Dallmeier et al. [2013] introduce the term *usage model*. The actual user behavior model is defined as an application model enriched by aspects of the user behavior.

Menascé et al. [1999] analyzed the workload characterization of e-commerce sites. Traditional methods based on hits per second, views per second, or visits per second are not appropriate any more. The users/customers of the system navigate through the application with consecutive and related requests, called *sessions*. A session is a single use of the application (from login to logout) by a single user. They introduce a state transition graph (customer behavior model graph (CBMG)) that describes the behavior of customer groups with similar navigation patterns. It has a node for each state and transitions between those states. However, the graph only tracks server side requests and does not include client side operations. Useful metrics like average session length, average visits per state etc. can be derived directly from the graph.

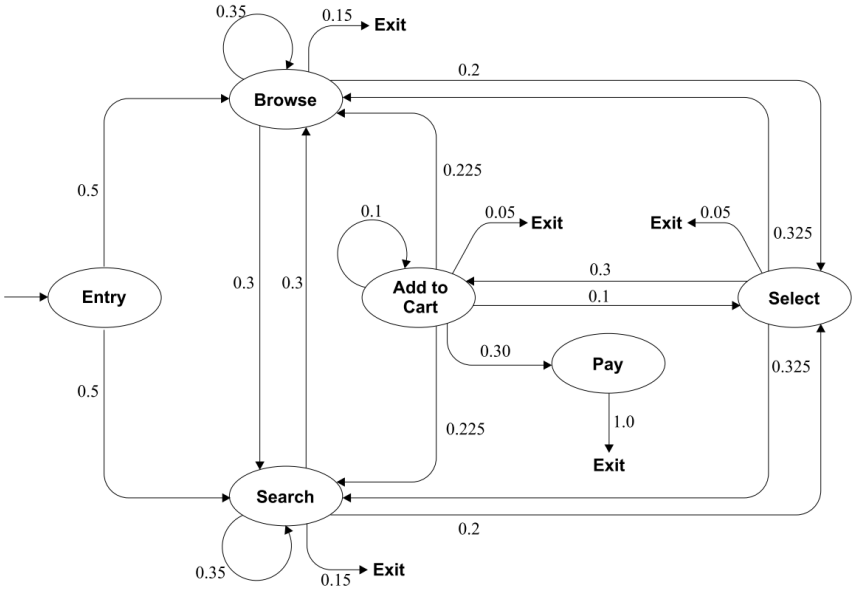

Figure 2.3. CBMG of an e-commerce site [Menascé et al. 1999]

Figure 2.3 gives an example for a CBMG. There are two artificial states for entry and exit. The user activities are represented as states and are connected with transition. Each transition denotes the probability of this transition being chosen from the source state.

Loops are allowed, so that customers can execute a search multiple times before continuing to the next activity. Menascé et al. [1999] call exits from states other than the "pay"-node as spontaneous exits.

A Markov Chain is a finite-state machine (FSM) consisting of states and transitions labeled with probabilities. The probability of a transition specifies the likelihood of being chosen as the next action of the user. The sum of all probabilities of outgoing transitions of a state has to add up to one. In Markov Chains of order one the future state only depends on the past stage.

Another element of user behavior models are *think times*. They are defined as the average time between completion of the user request and arrival of his next request on the server side [Menascé et al. 1999]. These times can be assigned to the states of a Markov Chain.

The b+m gear platform (see Section 2.6.1) already includes an application model for screenflows. A screenflow consists of multiple views (comparable to states) which define a number of events (comparable to transitions). Events either return to the same view or proceed to another view. Therefore those screenflow models represent a good base for user behavior analysis with Markov Chains.

2.4 Model Driven Software Development

In software engineering models play a central role as abstract representations of structures, functions, and behaviors. Stachowiak [1973] defines three characteristics of models: mapping, reduction/abstraction, and pragmatics. There is always an object (real, planned or fictive) corresponding to the model (mapping). A model reduces the original by uninteresting attributes but may extend it by additional attributes (reduction/abstraction). The model can replace the original in certain situations, because it covers the required attributes and possibly is easier to use (pragmatics).

A domain is a limited field of interest and knowledge, motivated by a technical or functional problem [Ludewig and Lichter 2010]. Meta-models are used to formalize a domain. Brambilla et al. [2012] define four layers of meta-modeling (see Figure 2.4).

Meta-meta-models M_3 provide the meta-models to define modeling languages in form of meta-models M_2 (language engineering). Based on these meta-models, models M_1 are created to describe real world objects M_0 of the target domain (domain engineering). Meta-meta-models like Ecore (see Section 2.4.2) are usually self-describing and conform to themselves.

A transformation in context of model-driven software development is a program which produces one or more target models based on one or more source models. Target models and source models usually conform to a meta-model (M_2). If one of the models is represented in source code, those transformation are called model-to-text (M2T) (generators) or text-to-model (T2M) respectively. If both models are expressed in explicit meta-models, they are called model-to-model (M2M) transformations instead. Transformations are defined by transformation languages. As shown in Figure 2.4 a transformation (M_1) is a model that

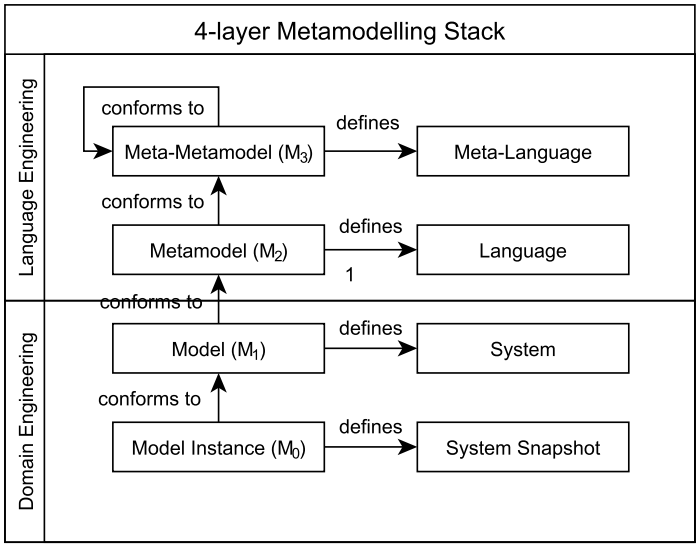

Figure 2.4. Meta-Modeling Stack [Brambilla et al. 2012]

conforms to a transformation language (M_2) [Stahl et al. 2007].

In contrast to the more general term model-driven software engineering (MDSE) introduced by Brambilla et al. [2012], we use the term model-driven software development (MDSD) of Stahl et al. [2007] which primarily looks at the generative software development. Figure 2.5 illustrates the basic idea of MDSD. First, the code of the reference application is analyzed and divided into three parts of code:

- The generic code, that is the same in each application.
- The schematic, repeating code, that has the same structure in each application but is dependent on the application's domain.
- The individual code, that can not be generalized and is different in each application

In the next step the three parts are separated into model and code components. The application model, which contains the application specific information, can be transformed into the schematic, repeating code. It can be defined in a DSL and/or graphically. A domain specific language (DSL) is a programming language or executable specification language that focuses on a particular problem domain [Van Deursen et al. 2000]. The DSL and the transformation logic is provided by the platform, that also includes the generic code. In the model-driven software development process only the application model has to be defined and the individual code has to be written, because the schematic, repeating code is now generated based on the application model through transformations.

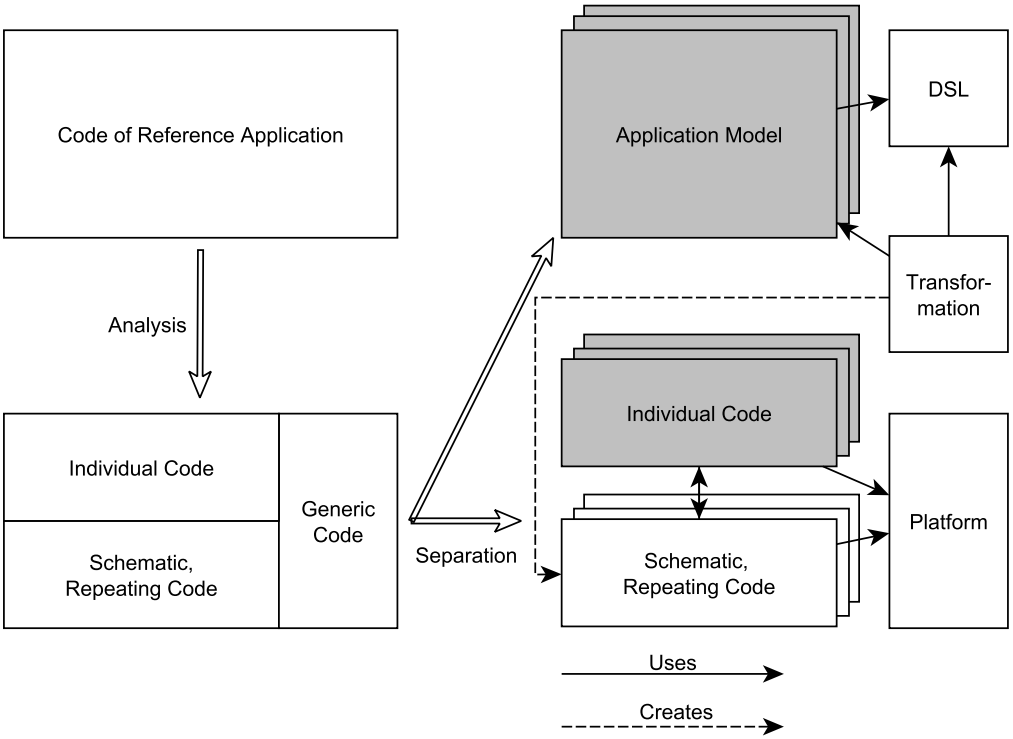

Figure 2.5. Basic Idea of Model Driven Software Development [Stahl et al. 2007]

2.4.1 The Eclipse Modeling Project

The Eclipse Modeling Project (EMP) provides a set of modeling frameworks, tooling, and standard implementations for the evolution and promotion of model-based development technologies [Eclipse Foundation 2016]. The project is divided in several sub-projects. In the following we give an overview of the most popular technologies of the EMP:

- Eclipse Modeling Framework (EMF): The core technology of the EMP is described in Section 2.4.2.
- Server and Storage: These technologies provide the infrastructure to store and to version model instances (M_0) efficiently. They allow concurrent access of multiple users to these models in form of a client-server architecture and with transaction support.
- User Interface: These frameworks support the implementation of UIs for EMF-Models (M_2).

- Graphical Modeling: The visualization of models is done by the Graphical Modeling Framework (GMF). It provides support to implement graphical views to embed in diagram editors.
- Modeling Tools: There are many ready-to-use tools targeting end-users for creation and editing of EMF-Models.
- Transformation: These technologies focus on the transformations of models into other models (M2M) or into textual representation (M2T) as described above. Those are commonly used in MDSD and MDM. One example is Xpand[5] as a statically-typed template language.
- Textual Modeling: Textual modeling frameworks are used for the development of standalone DSLs or DSLs based on existing EMF-Models. They feature editors with syntax highlighting, auto-completion etc. Xtext[6] is a popular framework in this category that allows the desciption of custom languages by using Xtext's grammar language.

2.4.2 The Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is the basis for all technologies of the EMP [Steinberg et al. 2008]. Its focus is (meta-)modeling and code generation. Additionally the EMF provides model validation, persistence, graphical editing and many more features.

The (meta-)modeling component is called Ecore and represents a self-describing meta-meta-model (M_3). Meta-models (M_2) which conform to Ecore are called EMF-Models. There are three equivalent other representations: Extensible Markup Language (XML), Unified Modeling Language (UML) and Java. Models can be serialized to XML Metadata Interchange (XMI), a standard for exchanging metadata information via XML. Many UML editors feature an export to XMI. Finally, the EMF has a code generator, that creates Java classes based on EMF-Models.

The core modeling classes of Ecore are `EPackage`, `EClass`, `EDataType`, `EAttribute` and `EReference` (see Figure 2.6).

An `EPackage` contains any number of classifiers, which can either be `EClasses` or `EDataTypes`. An `EClass` can have any number of super types and may be declared as abstract. It contains `EAttributes` and `EReferences` as structureal features. Structural features define a lower and an upper bound. An `EReference` references another `EClass` and can be linked with the opposing `EReferences` as opposite. An `EAttribute` has an `EDataType` as attribute type. The EMF provides graphical editors to edit EMF-Models based on Ecore [Steinberg et al. 2008].

In Chapter 4 we use the EMF to create a meta model for session entries and for user behavior.

[5] `https://wiki.eclipse.org/Xpand/`

[6] `https://eclipse.org/Xtext/`

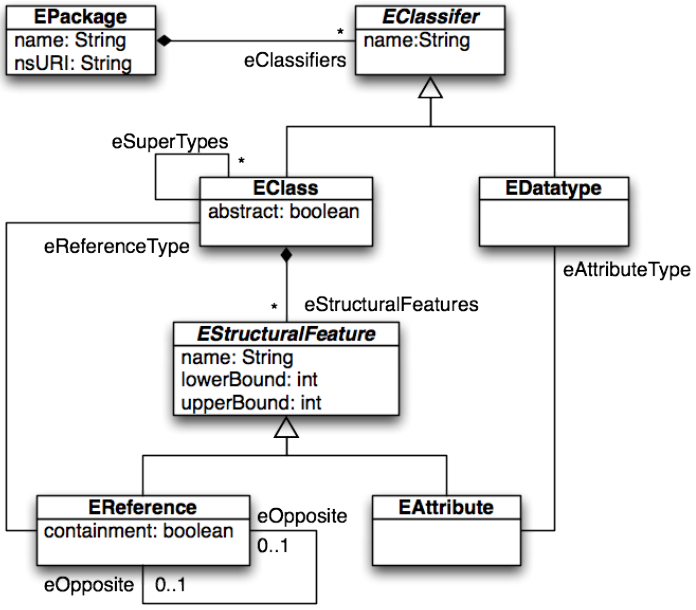

Figure 2.6. Ecore Modeling Classes (simplified) [Eclipse Foundation 2016]

2.5 The WESSBAS Approach

The WESSBAS approach exposes a software system to system to synthetic workload for load tests [Van Hoorn et al. 2014]. The workload specification is extracted automatically via dynamic analysis and generation of behavior models. The approach considers only session-based systems. Those systems are characterized by time-bounded user sessions and requests or events triggered by them (see Section 2.3). The focus of their work is the production of workload similar to the SUT usage profile. The WESSBAS approach comprises a DSL, an extractor for session logs and a generator for workload specification. Figure 2.7 gives an overview of the WESSBAS approach.

First, the SUT is monitored e.g. by Kieker. The behavior model extractor creates behavior models and workload intensity records based on the session logs. Those results are transformed to an instance of the WESSBAS-DSL by the model generator. Finally, a test plan can be generated based on the WESSBAS-DSL instance, which then can be executed by existing load testing tools, i.e. Apache JMeter[7].

Van Hoorn et al. [2014] introduce the DSL called WESSBAS-DSL for modeling of workload specifications. It is implemented as an EMF-model (see Section 2.4.2) enriched by constraints in the Object Constraint Language (OCL). Figure 2.8 shows the classes and

[7] https://jmeter.apache.org/

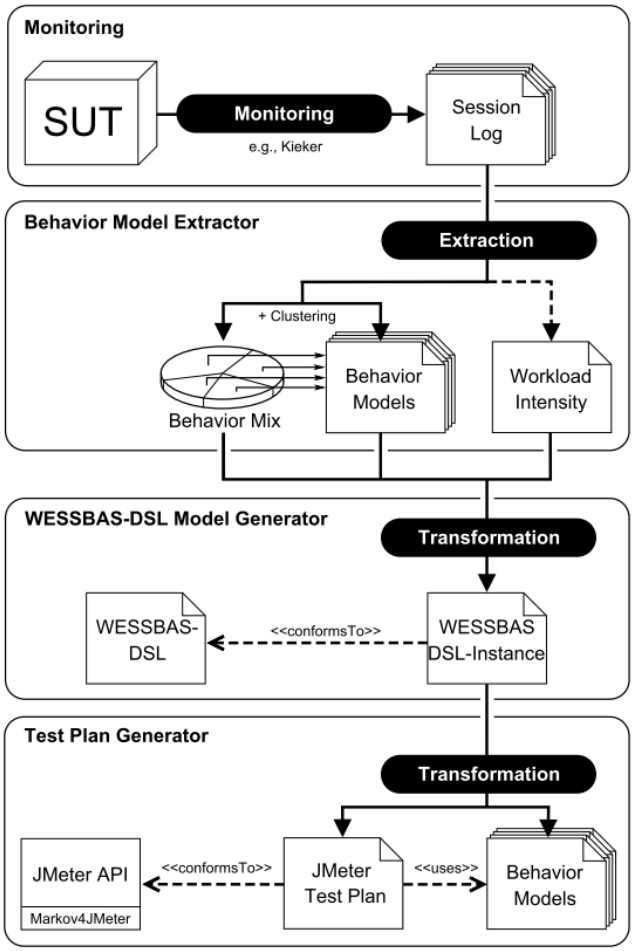

Figure 2.7. Overview of the WESSBAS approach [Van Hoorn et al. 2014]

relationships of the WESSBAS-DSL.

The behavior model includes the main components `ApplicationModel`, `BehaviorMix` and `WorkloadIntensity` [Schulz 2014; Van Hoorn et al. 2008]. The `ApplicationModel` consists of two layers: `SessionLayer` and `ProtocolLayer`. The `SessionLayer` contains the allowed sequences of service calls provided by the system. The `ProtocolLayer` defines the related protocol-specific workflow of the system for each service. The `BehaviorMix` specifies the relative frequency of the behavior of a user type to occur. The `BehaviorModel` corresponds to a Markov chain with services as states and transitions labeled with probabilities as explained in Section 2.3. The `WorkloadIntensity` defines the number of users emulated during a workload test, which is based on a formula.

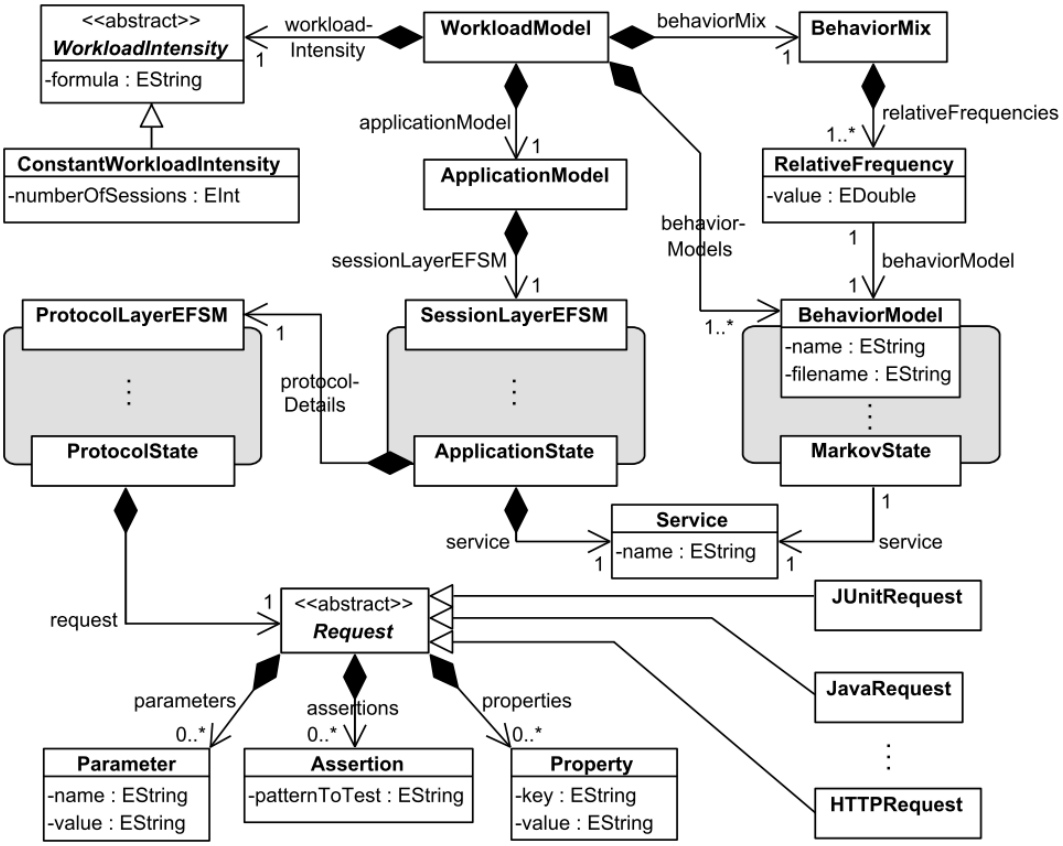

Figure 2.8. WESSBAS-DSL classes and relationships [Van Hoorn et al. 2014]

The extraction of WESSBAS-DSL instances from a running application system is based on session logs. Session logs are created by Van Hoorn et al. [2014] via adding session identifiers and timestamps to request logs. Request logs are commonly provided by web servers or can be obtained from monitoring tools like the Kieker framework (see Section 2.1). In the session log the requests are grouped by session identifier and enriched by timing information. The behaviour extraction process includes clustering techniques for grouping users with similar action patterns. This allows the optimization of the SUT based on the usage patterns. Furthermore, the impact of certain user groups on the system's performance can be evaluated.

The generation of WESSBAS-DSL instances consists of three phases. First, the application model is built based on the states and transitions of the SUT. Second, the extracted behavior is integrated, and finally, the information concerning workload intensity is added. Afterwards, the WESSBAS-DSL instance can be transformed into a JMeter test plan. The

core process is the mapping of WESSBAS concepts to Markov4JMeter[8] elements. In their evaluation Van Hoorn et al. [2014] show that the clustering of similar behavior model is very accurate by using normalized Euclidean distance. The accuracy of workload characteristics however has some weaknesses. They observe, that very long sessions are generated and the number of session generated deviates from the characteristics of the original workload in their approach.

2.6 The b+m Informatik AG

The b+m Informatik AG[9] offers software products and IT services for banks and insurers. The company was founded as becker & mohnberg Softwaregesellschaft mbH in 1994 by Kai-Christian Becker and Andreas Mohnberg. It emerged from the insolvency of the PBS Professional Business Software GmbH and took over 17 employees. After a consistent growth the company built an own building in Melsdorf and changed its name to b+m Informatik AG in the year 2000. The managing board was formed by the two founders and Frank Mielke. In the following years the number of employees grew to 200 with sites in Kiel, Hamburg, Hannover and Berlin until the financial crisis 2009 brought the company in economic difficulties. In line of the reorganization b+m was acquired by the Allgeier Group[10] in 2012 [Hahn 2015].

In December 2015 Allgeier sold the company to Main Capital[11] from the Netherlands. Main Capital is a private equity investor with an exclusive focus on the software sector in the Benelux and Germany. Currently, b+m has nearly 90 employees and recently returned to the list of the 50 biggest companies (regarding employee count) in Schleswig-Holstein.

Software solutions developed by the b+m Informatik AG base on the b+m gear platform (see Section 2.6.1). It realizes the MDSD approach described in Section 2.4. In this thesis we instrument one of those b+m gear Java applications: the b+m bAV-Manager (see Section 2.6.2). Other products for insurers[12] are the b+m AGS and the b+m AGS Portal. The main solutions for banks[13] are the b+m FGCenter as well as the b+m Credit Controller and the b+m Fonds Manager. The software b+m FGCenter is the market leader in Germany and supports banks in the entire application process for promotional loans [Hahn 2015].

2.6.1 The b+m gear Platform

The b+m Informatik AG has developed the b+m gear platform[14] for the creation of client-server web applications in Java [Reimer 2013]. The term "gear" is an acronym for "Generative

[8] https://sourceforge.net/projects/markov4jmeter/
[9] https://bmiag.de/
[10] https://www.allgeier.com/de/
[11] https://www.main.nl/en/
[12] http://bmiag.de/loesungen/versicherungen/
[13] http://bmiag.de/loesungen/banken/
[14] https://bmiag.de/loesungen/engineering-solutions/b-m-gear/

Architecture".

The platform follows the concept of MDSD seen in Section 2.4 and features a three-tier architecture. This ensures that the generative applications share and reuse common technical aspects and concepts. The idea of the platform is, that new projects can be set up with minimal initial effort, while being flexible for extensions and individual specifications.

As mentioned above the architecture consists of three layers: *Frontend* (presentation tier), *Service* (logic tier) and *Entity* (data tier) (see Figure 2.9). Each layer is described in an own DSL. This clearly separates technical and business logic and adds robustness against platform changes. Since version 3 the b+m gear platform provides an additional DSL to model the user interface (UI). The data layer defines entities and relationships whose data is mapped to the database with a Java Persistence API[15] (JPA) implementation of Hibernate[16]. The service methods in the logic layer contain the business logic and have access to the entities via report- and entity-factories. The communication with the presentation layer happens with a data transfer object (DTO). The frontend controller exchanges data with the users via views. Those views and their controllers are combined in screenflows.

There is an optional fourth layer *Workflow* (left side of Figure 2.9) that can be added to a project of the b+m gear platform in order to enable workflow capabilities. Workflows are fully or partially automated business processes. They are often mixed up with pure business processes, but those are unrelated to information technology. Process definitions in the b+m gear platform are written in an extended version of Java Business Process Model (jBPM)[17] and define the workflow in the system. Screenflows and services that have been declared as business services (at the top of Figure 2.9) in their respective DSL, can be called from a process as *user task* (*light task*) or as *system task* (*dark task*). Those calls can be synchronous, where the calling instance has to wait for an answer, or asynchronous, where control is immediately returned so that tasks can be performed simultaneously. An example for a light task is the print dialog, where the user selects a template and enters free text. The print on a print-server is a dark task, that is executed automatically by the system. The print can happen independently, because all required information has been entered by the user in the configuration. At the same time the user might already take the next task.

[15] https://www.oracle.com/technetwork/java/javaee/tech/persistence-jsp-140049.html
[16] https://hibernate.org/orm/
[17] https://www.jbpm.org/

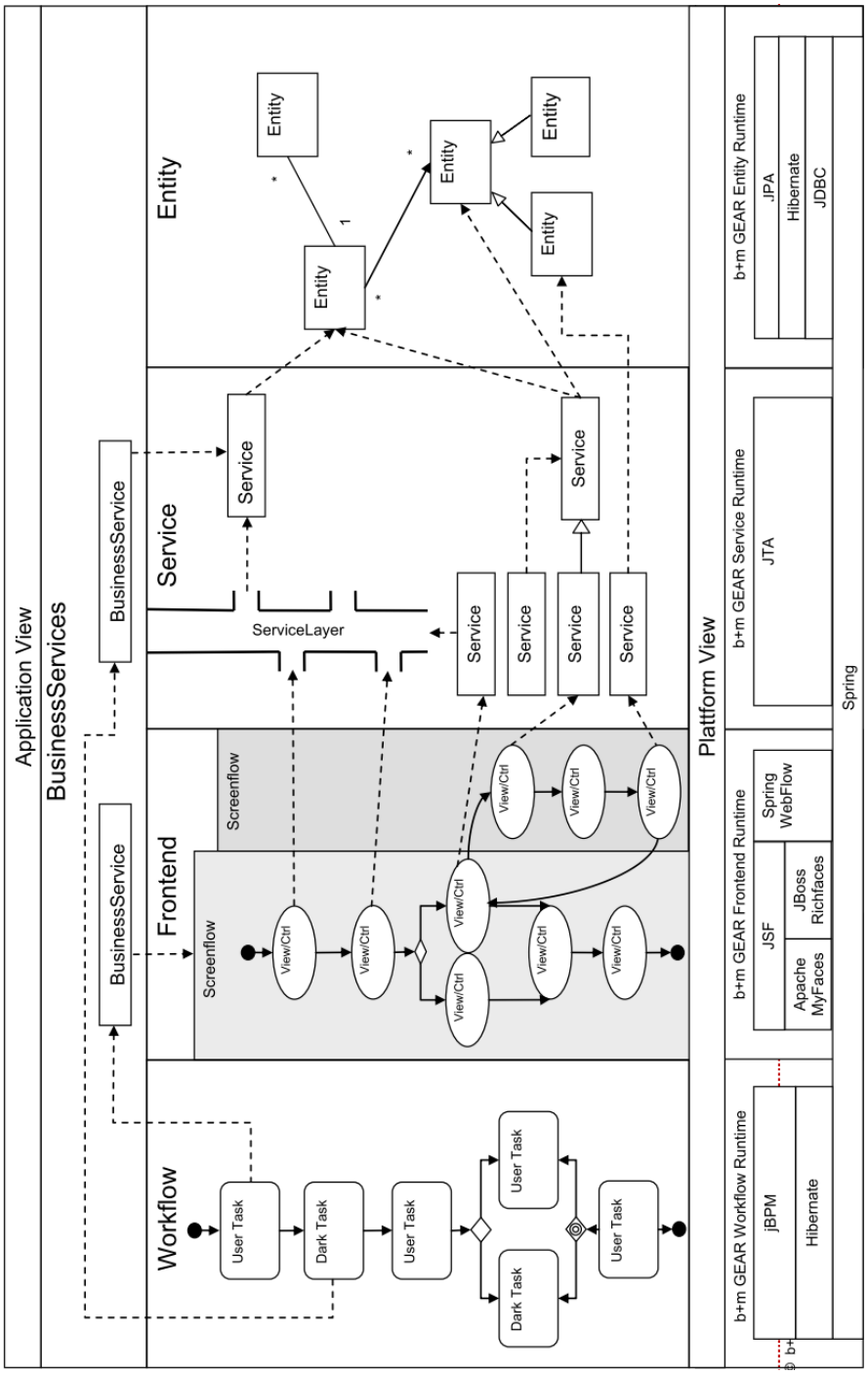

Figure 2.9. The Architecture of the b+m gear Platform [Reimer 2013]

```
1  package de.bmiag.bavmanager.screenflow.stammdaten;
2  flow AuftragAnlegen <|businessServiceName="AuftragAnlegen"|>{
3   input:
4    slot ErweiterteSuche;
5   output:
6    slot Firma;
7   start Start {
8    go to FirmaFindenWF;
9   }
10  sync call FirmaFindenWF to FirmaFindenWF {
11   input:
12    bind paramErweiterteSuche to ErweiterteSuche;
13   output:
14    bind FirmaPK to Firma;
15   else go to BearbeitungszeitraumErfassen;
16  }
17  view BearbeitungszeitraumErfassen {
18   on event content_back with action content_back validating None
19    go to FirmaFindenWF;
20   on event wfComplete with action wfComplete
21    go to Ende;
22  }
23  final Ende;
24 }
```

Listing 2.2. Screenflow DSL of the b+m gear Platform

Listing 2.2 shows the screenflow *AuftragAnlegen* (simplified) of the b+m bAV-Manager in the screenflow DSL. It can be used with the same name in process definitions as defined by the attribute *businessServiceName* (line 2). The idea of this task is, that one user starts a new process, selects an existing company in the database and sets a due date. Afterwards, the process can be taken over by any user in order to complete the work (before the due date is reached).

The screenflow *AuftragAnlegen* defines an input parameter for enabling and disabling the advanced search in line 4. The output is the primary key of a company data record (line 6). This identifier is determined in another screenflow named *FirmaFindenWF*. It is called synchronous at the beginning of *AuftragAnlegen* (line 10). Finally, the control flow reaches the view *BearbeitungszeitraumErfassen* which takes the user input for the due date. If the user leaves *BearbeitungszeitraumErfassen* via the *back* button, he returns to the company selection. Otherwise, the screenflow is completed.

Screenflows can be used as standalone group of views, but also in processes. Listing 2.3 illustrates the use of *AuftragAnlegen* in a process definition. Its an excerpt of the b+m

bAV-Manager process *Zwischenbilanz* (simplified version).

```
<?xml version="1.0" encoding="UTF-8"?>
<process name="Zwischenbilanz" xmlns="http://jbpm.org/4.0/jpdl">
 <swimlane name="Mitarbeiter"/>
 <swimlane name="Sachbearbeiter"/>
 <start>
  <transition to="ErweiterteSucheFestlegen"/>
 </start>
 <script name="ErweiterteSucheFestlegen" var="ErweiterteSuche" expr="${false}">
  <transition to="AuftragAnlegen"/>
 </script>
 <task name="AuftragAnlegen" swimlane="Mitarbeiter">
  <assignment-handler
   class="de.bmiag.gear.basis.workflow.task.TaskConfigurator">
   <field name="autoRelay"> <boolean value="true"/> </field>
  </assignment-handler>
  <on event="start">
   <event-listener
    class="de.bmiag.gear.basis.workflow.service.BusinessServiceCallSetter">
    <property name="serviceName"><string value="AuftragAnlegen"/></property>
    <property name="inputMapping">
     <string value="ErweiterteSuche->ErweiterteSuche"/>
    </property>
    <property name="outputMapping"><string value="FirmaPK->FirmaPK"/></property>
   </event-listener>
  </on>
  <on event="wakeUp">
   <event-listener class="de.bmiag.gear.basis.workflow.task.WakeUpTask"/>
  </on>
  <on event="escalate">
   <event-listener class="de.bmiag.gear.basis.workflow.task.EscalateTask"/>
  </on>
  <on event="end">
   <event-listener class="de.bmiag.gear.basis.workflow.task.TaskCleanup"/>
  </on>
  <transition to="DatenbasisErmitteln"/>
 </task>
 ...
</process>
```

Listing 2.3. Usage of jBPM in the b+m gear Platform

After the creation of two swimlanes (lines 3-4) the variable *ErweiterteSuche* is initialized with `false` (line 8). Then the task *AuftragAnlegen* is started, which is located in the swimlane *Mitarbeiter* (line 11). The property *serviceName* references the target screenflow (lines 19-21) and input/output parameters are bound to variables on the workflow level (lines 22-27). At last, event listeners are added for task suspension, reactivation and finalization (lines 30-38). Then the process *Zwischenbilanz* continues with the next task *DatenbasisErmitteln* (line 39). *DatenbasisErmitteln* uses the output parameter *FirmaPK* (line 26) for searching the current data base of the company. This functionality belongs to the historization concept of the b+m bAV-Manager.

The generation process of the b+m gear platform shown in Figure 2.10 is similar for all four layers. First, the developer creates the application design based on the business requirements. Via activating the generation process the infrastructure code is built. Together with the platform code it forms the basis for the manual code of the application.

The b+m gear platform does not use any protected regions and therefore strictly separates generated and manual code. The generated code is rebuilt every time, while the manual code is only created once. Dependencies are realized by Java inheritance. The manually implemented classes extend the abstract generated middle-classes which extend the abstract base-classes of the platform themselves.

The latest version of the b+m gear platform replaces Spring Webflow[18] and Richfaces[19] by the b+m Flow Controller and the Vaadin Framework[20]. The aspect of MDSD was extended by a new DSL for the UI of web applications. The basic structure of all views is now defined in a model and allows the generation of more parts of the Frontend layer. Listing 2.4 shows an example of a view defined in the b+m gear UI-DSL.

[18] `http://projects.spring.io/spring-webflow/`
[19] `http://richfaces.jboss.org/`
[20] `https://vaadin.com/`

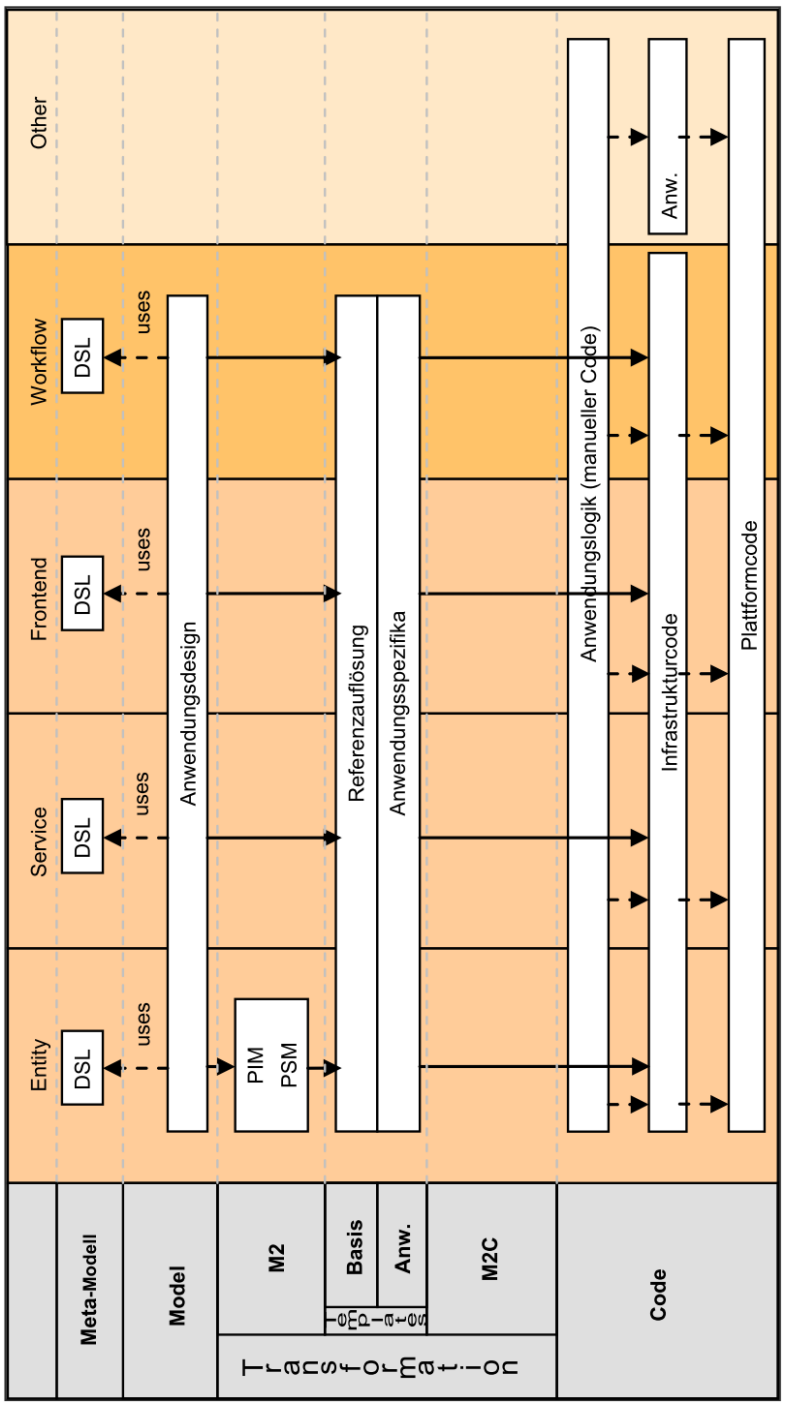

Figure 2.10. model-driven software development with the b+m gear Platform [Reimer 2013]

```
1  package de.bmiag.bavmanager.frontend.stammdaten;
2  viewModel FirmaInfo with data FirmaInfoHVI {
3    fields:
4      Kennung;
5      Bewertungsstichtag;
6      Datenbasisnummer;
7  }
8  view FirmaFindenWFView inherits BAVManagerWFView with layout
       FirmaFindenWFViewLayout : Vertical {
9    bindings:
10     FirmaInfo[] as firmaInfoList with id PK;
11   cells:
12     panel SuchkriteriumPanel : Panel with layout SuchkriteriumPanelLayout :
           Vertical {
13       container with layout ButtonLayout : LegacyButton {
14         button Suchen with trigger suchen decorated by Label;
15       }
16       container with layout KriteriumLayout : Horizontal {
17         custom Betreuer;
18         custom Firmenname;
19         custom Firmennummer;
20       }
21     }
22     panel SuchergebnisPanel : Panel with layout SuchergebnisPanelLayout : Vertical
           {
23       table FirmaTabelle : Table with model firmaInfoList {
24         trigger:
25           row default wfComplete;
26         columns:
27           Kennung;
28           Bewertungsstichtag;
29           Datenbasisnummer;
30       }
31     }
32 }
```

Listing 2.4. UI DSL of the b+m gear Platform

First, a view model is defined, that is based on a DTO named FirmaInfoHVI. High value interface (HVI) is the old name for DTOs. The view model uses the three attributes listed under fields in lines 4-6. Second, the view is created, which extends the base view of the application and uses a vertical layout. The view consists of two panels, a panel with

search criteria (lines 12-21) and a panel with the results in a table (line 22-31). The button "Suchen" in line 14 triggers the event suchen and the default event for double-clicking a table row is wfComplete. This example is taken from an early migration prototype of simple views of the b+m bAV-Manager to the new UI-DSL.

2.6.2 The b+m bAV-Manager Software

The b+m *bAV-Manager*[21] is one of the web application products developed by the b+m Informatik AG with the b+m gear platform [Harms 2014]. It serves as an administration software for customer and calculation data in the field of company pension schemes (bAV). In cooperation with a calculation engine which is developed by a partner company, it creates expert opinions for several valuation and accounting regulations. The software is used by international companies either on their own servers or as software as a service (SaaS) maintained by the b+m Informatik AG.

The core features of the application besides the administration of customer data are a workflow management, an access control system and historization of data bases. The b+m bAV-Manager has been extended multiple times due to additional requirements of its users. For instance, it offers web services for the exchange of customer data with a customer relationship management (CRM) system, writes metadata for document archiving purposes, distributes documents via e-mail server etc.

The current release is version 3.10 (autumn release 2015). The upcoming release 3.11 was postponed from spring 2016 to summer 2016. The b+m bAV-Manager still uses the b+m gear platform version 2.9.8 that is not developed further and relies on Spring Webflow and Richfaces. The management aims at modernizing the application in line with a customization request by a customer or as an investment project within the next years.

In Dittrich [2013] we analyze the modernization of the software's frontend layer. We concluded, that a model driven modernization from Richfaces to Vaadin is possible, but the additional features of the new platform (data binding, behavior definitions, etc.) can not be derived from the legacy code. Schmidt [2015] performs a case study regarding usability of the b+m bAV-Manager and lists suggestions for a software modernization while sketching them in an Axure[22] prototype.

2.7 The DynaMod Project

The *DynaMod project*[23] has been initiated by the Software Engineering Group of Kiel University and industrial partners from Kiel and the surrounding area. It focuses on MDM of legacy software systems. Their approach consists of three phases which align to the Horseshoe Model [Van Hoorn et al. 2011] (see Figure 2.11).

[21] http://bmiag.de/loesungen/versicherungen/b-m-bav-manager/

[22] https://www.axure.com/

[23] http://kosse-sh.de/dynamod/

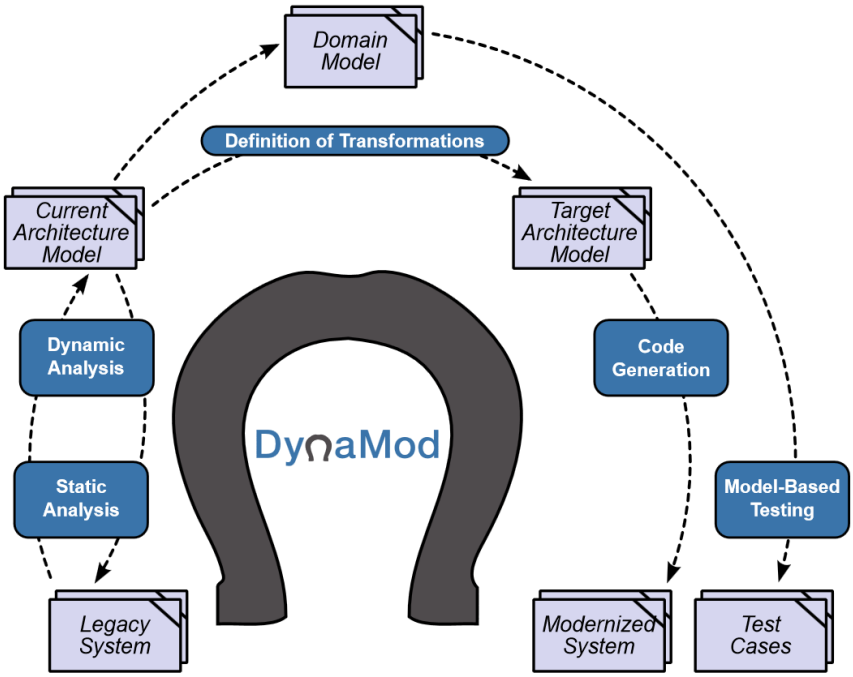

Figure 2.11. The Modernization Approach of The DynaMod Project [Van Hoorn et al. 2011]

First, they extract a model of the outdated architecture by using static and dynamic analysis, so-called hybrid analysis [Van Hoorn et al. 2013]. While the static part provides structural information about entities, relations and their arrangement, the dynamic part allows the analysis of runtime behavior. We described the latter in Section 2.1.

In the second phase the models of the outdated architecture are transformed to models of the desired system. Missing information can be supplemented by system experts.

Finally, generating techniques are used to develop the modernized system and to create test cases derived from the runtime behavior [Van Hoorn et al. 2011]. The MDSD paradigm (see Section 2.4) ensures the synchronization of architecture model and implementation. Additionally it provides good maintainability and reliability throughout the evolution of the software.

2.8 The Goal Question Metric Approach

Basili and Weiss [1984] developed the GQM approach to derive relevant metrics from problems. The measurement model consists of three levels:

1. **Conceptual level**: Define the relevant goals for the project.

2. **Operational level**: Derive questions that have to be answered to check if the goals were reached.

3. **Quantitative level**: Name metrics which help to answer each question.

This hierarchy is also illustrated by Figure 2.12.

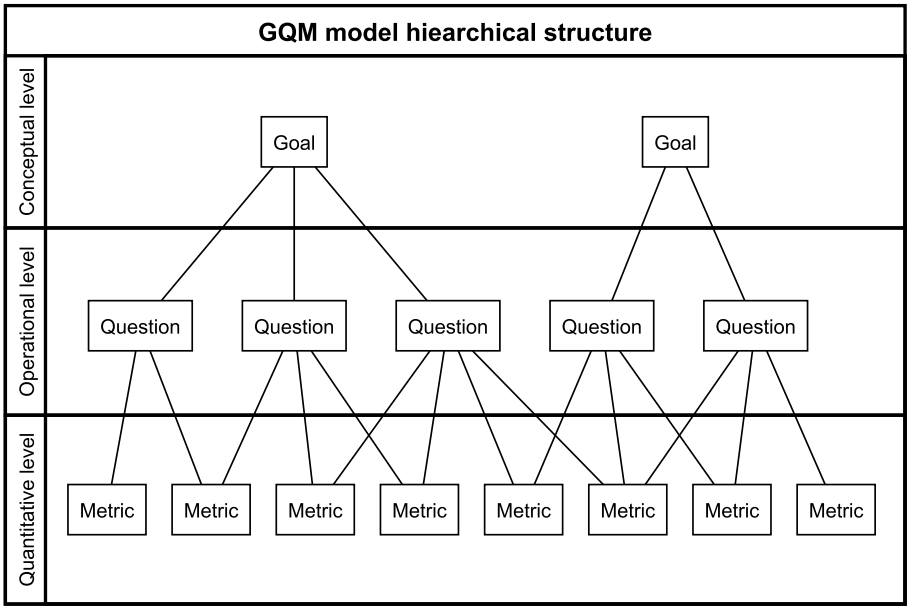

Figure 2.12. GQM model hierarchical structure

The process of setting goals is very important for a successful application of the GQM method. The questions on the operational level are developed based on the formulated goals. Finally, the questions are associated with appropriate metrics. Note, that the same metric can be used multiple times to answer different questions [Wohlin et al. 2000].

Chapter 3

Monitoring of Screen- and Workflow Activities

This section describes, how we implemented the monitoring of user activities in the b+m gear platform. Since the b+m bAV-Manager is based on an old version of the platform, the current version of the Kieker framework had to be integrated manually, which is explained in Section 3.1. Section 3.2 contains the implementation of a custom record via the Kieker IRL. Afterwards, in Section 3.3 we describe the implementation of the required monitoring parts for the b+m gear platform. This includes interceptors, configuration files, context definitions and program extensions. For the specific monitoring of the b+m bAV-Manager in this work, we extend the general interceptor (see Section 3.4). This approach allows a more detailed analysis of complex views composed of many sub views. Finally, problems and challenges during the implementation are discussed in Section 3.7.

3.1 Kieker Integration and Update in the b+m gear Platform

As mentioned in Section 2.6.2 the software b+m bAV-Manager is based on the b+m gear platform version 2.9.8. Table 3.1 gives an overview which b+m products use which version of the b+m gear platform and which version of the Kieker framework.

b+m products	b+m gear platform version	Kieker framework version
b+m bAV-Manager	2.9.8	1.5
b+m AGS Tarifrechner	2.9.8	1.5
FGCenter (up to 2.23)	2.9.8	1.5
FGCenter (since 2.24)	4.0.0	1.11
b+m AGS Vertriebsportal	4.1.0	1.11
none	4.2.0 (latest)	1.12 (latest)

Table 3.1. Overview of versions

In order to use the current version of the Kieker framework, several modifications had to be made. First, the common library of the b+m gear platform was replaced by the current

version. It contains helper classes and classes for monitoring and logging of a b+m gear application. This step replaced all legacy monitoring components based on version 1.5 of the Kieker framework by their updated counterparts based on version 1.11. However, the core libraries of the platform also contain code, that relies on the old Kieker version 1.5. Since the platform version 2.9.8 will only be changed in case of severe problems, we implemented a workaround.

The platform initializes the monitoring components during start-up in a lifecycle interceptor (`BAVManagerLifeCycleInterceptor`). If monitoring is activated the method `MonitoringController.initialize()` is called. However it does not exist any more in Kieker version 1.12. Therefore we manually disable monitoring in the project specific subclass, skipping the old initialization process. Then we are able to initialize the monitoring via the new common library.

Finally, the Java Archive (JAR) of the Kieker version 1.12 is added to the project specific libraries of the b+m bAV-Manager. Henceforth, it replaces the old library of the b+m gear platform.

3.2 Implementation of Custom Records via Instrumentation Record Language

The Kieker framework provides a convenient way for defining custom records with the IRL as described in Section 2.1. Combined with the Eclipse tooling and generators we produce the Java code for the monitoring records. Listing 3.1 shows the custom record defined in the IRL for recording user behavior.

```
1  package de.bmiag.gear.util.monitoring.record
2
3  @author "Gunnar Dittrich"
4  entity ScreenEntryRecord {
5    string userName
6    long   loginTime
7    string screenName
8    string flowName
9    string processName
10   string processExecutionId
11   long   entryTime
12   string eventName
13 }
```

Listing 3.1. Monitoring Record defined in the IRL

We decided to define only one record that carries the behavior information. When a user enters or reenters a screen, all pieces of information are collected at once. This includes

the session id (user name and login time combined), the unique name of the screen, the unique flow name which the screen belongs to and the name of the process, if the execution takes places in a predefined business process. In detail the attributes have the following meanings:

- **userName**: The unique name of the user (`null`, if no session has been started yet)
- **loginTime**: The login time of the user (`null`, if no session has been started yet)
- **screenName**: The name of the screen entered or reentered by the user
- **flowName**: The name of the flow containing the screen
- **processName**: The name of the process executing the flow (`null`, if it is a simple flow execution)
- **processExecutionId**: The process execution id differentiating multiple executions of one user (null, if it is a simple flow execution)
- **entryTime**: The time when the user enters or reenters the screen
- **eventName**: The name of the raised event to reach the screen (`null`, if the screen was called from another flow)

The alternative to one single record was the creation of records for screen calls, flow calls and process calls individually and combining them by session id at a later time. The problem is that users are able to execute several task at once. The switches to other tasks do not restart flows or processes. They only reactivate the suspended screen. Therefore, we have to remember the screenflow and workflow context for each screen anyway. This means, we need the complete information in any case and we decided to include everything in one record as shown above.

3.3 Implementation Details for General b+m gear Applications

The generation of monitoring records requires the implementation of instrumentation components. As defined in Section 1.2 those components should be reusable for all b+m gear applications. We achieve this by developing a general method interceptor, that can be applied to any view controller of a b+m gear application (see Section 3.3.1). Furthermore, we implement an additional service for workflow information retrieval as platform service in Section 3.3.2. Those services are provided to every application built with the b+m gear platform.

Since most of the implementations made for the instrumentation of b+m gear applications contain confidential information, we can not provide those elements in this work.

For an outline of all excluded sources see Appendix B. Therefore, the custom monitoring record defined in Section 3.2 serves as interface for other applications. Any software can be instrumented with the Kieker framework and emit those records. The records will be accepted by the SessionExtractor (introduced in Section 4.1) and its output by the BehaviorModelExtractor (introduced in Section 4.2) for further analysis.

3.3.1 Interceptor for Recording Screen- and Workflow Activities

The class `GearScreenEntryMethodInvocationInterceptor` represents the interceptor for recording screenflow activities. It intercepts method invocations of all view controllers in the application. First, the interceptor checks, if monitoring is enabled and if the intercepted method should be monitored. We decided to monitor the `prepareForm()`- and the `refreshForm()`-methods. Those are called, when a view is activated or reactivated.

```
1  public class GearScreenEntryMethodInvocationInterceptor implements
       MethodInterceptor, IMonitoringProbe {
2    @Override
3    public Object invoke(MethodInvocation invocation)
4      throws Throwable {
5        // Check if monitoring is enabled
6        ...
7        // Check if this invocation should be monitored
8        if (!shouldMonitor(invocation)) {
9            return invocation.proceed();
10       }
11       // Check if monitoring components are initialized
12       ...
13       // Gather screenflow and workflow information
14       ...
15       // Send the record
16       monitoringController.newMonitoringRecord(new ScreenEntryRecord(userName,
             loginTime, screenName, flowName, processName, processExecutionId,
             entryTime, eventName));
17       return invocation.proceed();
18     }
19
20     protected boolean shouldMonitor(MethodInvocation aInvocation) {
21     return aInvocation.getMethod().getName().equals("prepareForm") || aInvocation.
           getMethod().getName().equals("refreshForm");
22   }
23 }
```

Listing 3.2. Structure of the Interceptor for Recording Screen- and Workflow Activities

The interceptor initializes the Kieker monitoring components if necessary. Then, it assembles the required information for the `ScreenEntryRecord` (see Listing 3.2 and Listing 3.3).

```
1  public class FirmaFindenWFViewCtrl extends FirmaFindenWFViewCtrlBase {
2    @Override
3    public void prepareForm() {
4      // Initialize the view FirmaFindenWF
5    }
6    @Override
7    public void refreshForm() {
8      // Refresh the view FirmaFindenWF
9    }
10   @Override
11   public void validateForm() {
12     // Validate the view FirmaFindenWF
13   }
14 }
```

Listing 3.3. Structure of a View Controller in the b+m gear Platform

The `GearScreenEntryMethodInvocationInterceptor` has two central methods. It overrides the `invoke()` method of the `MethodInterceptor` interface. Due to space constraints and confidentiality reasons we just give a shortened, commented version. After checking if the monitoring is enabled, the interceptor checks if the invocation should be monitored with the `shouldMonitor()` method. As mentioned, we limit the monitoring to the `prepareForm()` and the `refreshForm()` methods of view controllers in the b+m gear platform (see Listing 3.3). Finally, a new instance of `ScreenEntryRecord` is created and passed to the `newMonitoringRecord()` method of the Kieker monitoring controller.

The `GearScreenEntryMethodInvocationInterceptor` retrieves the `userName` and the `loginTime` from the session context. The `screenName` is taken from the class name of the view controller without the suffix "ViewCtrl". In this way we save space in visualizations later. Views, screenflows and processes can be differentiated by shapes and/or colors. We get the `flowName` from the view controller by calling `getFlowExecutionInfo().getFlowId()`. The `processName` and the process `processExecutionId` is stored in the flow scope. They are determined by the monitoring service we implemented for the old b+m gear platform. The service is described in Section 3.3.2.

The monitoring controller of the Kieker framework provides a time source. We set the `entryTime` to the value from `getTime()` when the method call of `prepareForm` and `refreshForm` is intercepted. The `eventName` of the last raised event is provided by the view controller and the method `getLastEvent()`. Finally, it creates and emits the new monitoring record.

3.3.2 Service Implementation for Retrieving Workflow Information

The interceptor described in Section 3.3.1 collects the session and screenflow information, but has no access to the workflow information. The observed view controllers do not directly know in which process they are executed. However, all views that belong to a screenflow, which itself is executed in a process, have a task id. The task id should normally be enough information to resolve the corresponding process. However, the old b+m gear platform does not provide an appropriate platform service.

Therefore, we extended the service layer of the gear platform to identify the process execution id for a given task id. The workflow engine is queried for the process instance of the task. If the process is a sub-process, we continue with the parent process until the main process is found. Since there are only two small sub-processes in the b+m bAV-Manager currently, we will not monitor these processes individually. In a later work, when a more complex application is analyzed, that defines more complex processes, the monitoring service should be adapted. Instead of searching the parent process, the sub-process should be considered directly. This adds a new hierarchy to the behavior model, because processes may contain other processes.

As mentioned previously, in this work we consider only one layer of processes. The information from the monitoring service is added to the monitoring record and allows the assignment of screenflows (and their views) to processes.

3.4 Implementation Details for the b+m bAV-Manager

The b+m bAV-Manager consists of approximately one hundred views. However, there is one central, monolithic screen, that is composed of tabs: *AuftragVerwalten*. It is designed to display the complete data base of a company managed by the software. The tabs show general data, access restrictions, accounting information etc. Many processes use this screen as central task, because calculations are prepared and started on it. Additionally, the customers can make several configurations only on this complex view.

As a lot of user activities happen within this screen, we extended the general interceptor for b+m gear applications to a specific interceptor for the b+m bAV-Manager. It overrides the resolution of screen names by adding the name of the active tab to the screen name. In this way, the behavior within monolithic screens can be monitored. *VersicherungenImportieren* and *LeistungsplanVerwalten* are two other big screens organized with tabs. Their screen names are extended in the same way. This allows us to track user movement within a view, which does not add another layer to the general clustering with processes and flows. We split up the screenflow *AuftragVerwalten*, as if it consists of multiple views. A click on a tab will be represented by a transition between two of those views in the behavior model introduced later in this thesis.

3.5 Instrumentation of the b+m bAV-Manager

The instrumentation of the b+m bAV-Manager is done via Spring and its `BeanNameAutoProxyCreator`. The configuration in the application context is shown in Listing 3.4.

```
1  <bean id="screenEntryInterceptor" class="de.bmiag.gear.util.monitoring.
       BAVScreenEntryMethodInvocationInterceptor" />
2  <bean class="org.springframework.aop.framework.autoproxy.
3      BeanNameAutoProxyCreator">
4    <property name="interceptorNames">
5      <list>
6        <value>screenEntryInterceptor</value>
7      </list>
8    </property>
9    <property name="proxyTargetClass">
10     <value>true</value>
11   </property>
12   <property name="beanNames">
13       <list>
14         <value>login_LoginViewCtrl</value>
15         ...
16       </list>
17   </property>
18 </bean>
```

Listing 3.4. Monitoring Application Context

First, we define an interceptor bean based on the class `BAVScreenEntryMethodInvocationInterceptor`. Second, we apply it to the view controllers with the `BeanNameAutoProxyCreator` of the Spring Framework. In line 1 we define the interceptor bean. The interceptor is added to the proxy creator attribute `interceptorNames` in line 5. Finally, we list the beans, that have to be intercepted from line 13 onward. In the monitoring properties file the `AsyncBinaryNFsWriter` which writes to the file system is replaced by the `TCPWriter` (see Listing 3.5).

```
1 kieker.monitoring.writer=kieker.monitoring.writer.tcp.TCPWriter
2 kieker.monitoring.writer.tcp.TCPWriter.hostname=localhost
3 kieker.monitoring.writer.tcp.TCPWriter.port1=10133
4 kieker.monitoring.writer.tcp.TCPWriter.port2=10134
5 kieker.monitoring.writer.tcp.TCPWriter.bufferSize=65535
6 kieker.monitoring.writer.tcp.TCPWriter.flush=true
```

Listing 3.5. Monitoring Properties File (Excerpt)

We do this, because the session extraction tool, we introduce in Section 4.1, listens for records on a Transmission Control Protocol (TCP) port. In this way we also relocate the session logs from the SUT to the analysis server.

In our analysis in Chapter 4 we execute the session extractor on the same machine as the application. Therefore, the records are written to the default ports on localhost. Consider, that even multiple instances of the b+m bAV-Manager could be monitored at once with only one analysis node. The monitoring of each instance would be configured like described above, naming the same server as host. Then, all records could be collected and analyzed at a single point. This is especially useful for extracting user behavior from multiple demo-, test or production systems.

3.6 Instrumentation of the PM-System

We want to show, that our tools can be applied to any Java application, not only to the b+m bAV-Manager and the old b+m gear platform. Technically speaking, they can be applied to any application that can write `ScreenEntryRecords` to a TCP port with the Kieker framework since the IRL makes the records programming language independent. This can be done with an interceptor or other instrumentation techniques.

We instrument another web application of the b+m Informatik AG. The PM-System is based on the latest b+m gear platform and a relatively small system. It is used by the b+m Informatik AG sorely for the internal management of projects. As it is based on the latest platform we can not use the `GearScreenEntryMethodInvocationInterceptor` directly.

The following adaptations of the interceptor have to be performed to use the user behavior analysis tools (see Chapter 4). We get the session information about the user id and his login time from the logging context provided by the platform. Instead of `prepareForm()` and `refreshForm()` we now monitor the `prepareView()` and `refreshView()` methods. They have retained their functionality of initializing and refreshing the view. The workflow information is provided by the new flow manager. A custom service is not needed any more.

Since the `SessionExtractor` is independent of the platform and exchanges data with the `SessionEntryRecord`, there were no additional adaptations required.

3.7 Lessons Learned and Challenges Occured

There is one big challenge we encountered while monitoring the b+m bAV-Manager, which applies to all b+m gear applications. The central view provided by the platform is `WorkflowClientTasks`, similar to a dashboard in other applications. It shows all running processes, those that are assigned to the user's role and those that are assigned to the user personally. On this view the user can proceed with the execution of suspended processes assigned to him or to his role.

Consider a very simple process consisting of two successive light tasks. The user starts the process, sees view one, goes on to view two and finishes the process. The b+m platform however switches to `WorkflowClientTasks` between the two light tasks without showing it. When the first task is completed, the process returns to the dashboard. Since the next task is also light and is assigned to the same user, the process is restarted instantly. The user only sees the switch to the second task.

This behavior complicates the extraction of user behavior models, because after each light task a transition to `WorkflowClientTasks` and back is inserted, although the user never saw that view. Since it can not be determined if a user actively visited the dashboard or the workflow engine called it as intermediate view, we had to exclude it from our analysis. In Section 4.2 we introduce our analysis tool, which offers a command line option to exclude flows from the analysis.

Another challenge was the retrieval of workflow information on the screenflow level. The view controller of each screenflow provides a method `getFlowInfo()` that provides data about the corresponding process execution. A screenflow represents a task in a process. With the id of the task, the execution of the process can be determined. The process execution id allows the differentiation between instances of one process definition. For example, a view provides the task id "WorkflowTask.2345". The task is part of a process execution with id "StammdatenBearbeiten.6542". In comparison with a process execution with id "StammdatenBearbeiten.4532", they represent the same process *StammdatenBearbeiten* but different executions.

A problem arises, if a screenflow calls another screenflow. The platform does not pass on the flow information. The `getFlowInfo()` returns `null` in this case. For example, the process *StammdatenBearbeiten* uses the screenflow *AuftragBearbeiten* as central task. The screenflow *AuftragBearbeiten* calls the screenflow *RechnungBearbeiten*. In the screenflow *AuftragBearbeiten* the b+m gear platform provides the flow information, in *RechnungBearbeiten* it does not.

In order to eliminate this problem we added a variable to the flow context that stores the process execution id of the flow. This variable can be accessed from each view of the flow. Normally, these variables would not be passed during a flow call, but we added a specific prefix to the variable. All variables with this prefix are automatically mapped to the flow context of the called flow by the platform.

There is an additional problem concerning process information. The b+m bAV-Manager uses sub-processes to group tasks that occur regularly in processes. We do not want to analyze the sub-processes in our work but the main processes. Therefore, our monitoring service searches the main processes of the sub-processes. Their process information is then written to the flow contexts. This means, that all information concerning sub-processes gets lost, in the current implementation. As mentioned in Section 3.3.2, this part could be improved for future analyses.

Chapter 4

Analysis of Screen- and Workflow Activities

We divided the user behavior analysis in two main parts: the session extraction and the behavior model extraction. Figure 4.1 shows the structure of the analysis components.

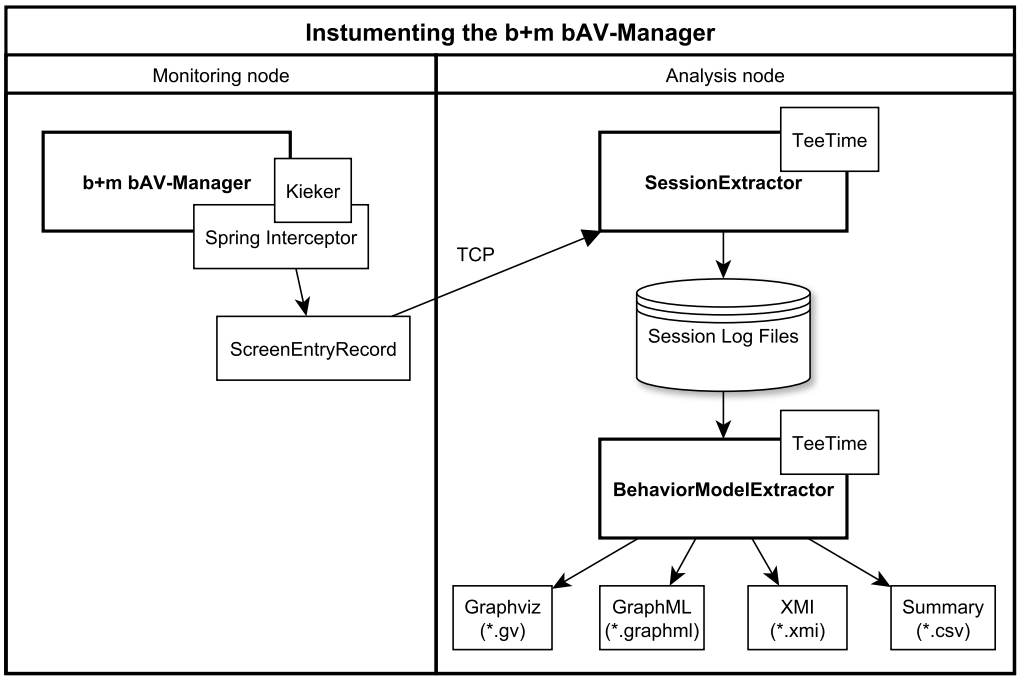

Figure 4.1. Structure of the Analysis Components

The monitoring node contains the b+m bAV-Manager which emits `ScreenEntryRecords` via the TCP. The records arrive at the analysis node and are processed by the `SessionExtractor`. The structure of the `SessionExtractor` is described in Section 4.1. It writes session log files to the disk of the analysis node. In this way, we take load off the application server and

move the processing to a dedicated analysis node. At any time the `BehaviorModelExtractor` can be started to build the behavior model and export its contents to Graphviz-, GraphML-, XMI-, and/or CSV-files (see Section 4.3). In Section 4.2 we describe the architecture and functionality of the `BehaviorModelExtractor`.

4.1 Structure of the Pipe-and-Filter Architecture for Session Extraction

The `SessionExtractor` is a Java application based on the TeeTime framework. Its Pipe-and-Filter-Architecture is shown in Figure 4.2.

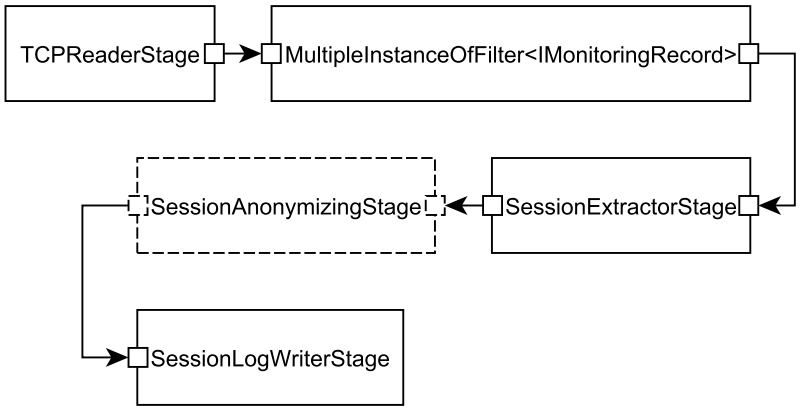

Figure 4.2. Pipe-and-Filter-Architecture of the `SessionExtractor`

The architecture consists of four mandatory stages and one optional stage. The first two are reused stages that are provided by the TeeTime project (see Section 4.1.2). The three other stages are custom stages implemented by us in this work (see Section 4.1.3). The `TCPReaderStage` receives all monitoring records sent to the configured port. With the `MultipleInstanceOfFilter<IMonitoringRecord>` we select only records with type `ScreenEntryRecord`. Those records are transferred to the meta-model for session entries (see Section 4.1.1) by the `SessionExtractorStage`. If the `-anonymize` option is set, the optional `SessionAnonymizingStage` is added to the architecture which hashes the sessionId. Finally, the session information is written to a log file using the `SessionLogWriterStage`.

The `SessionExtractor` provides the command line arguments listed in Table 4.1. Their functionality is explained in the stage description where they apply (see Section 4.1.2 and Section 4.1.3).

Name	Type	Required	Description

`-o`
`-output`
`-outputDirectory`
	`String`	yes	Output directory of the session log files.

`-anonymize`	`Boolean`	no	Encrypt session ids to prevent backtracking of sessions.
`-port1`	`Integer`	no	Port accepting `IMonitoringRecords`.
`-bufferSize`	`Integer`	no	Buffer size.
`-port2`	`Integer`	no	Port accepting `StringRecords`.

Table 4.1. Command line arguments of the `SessionExtractor`

4.1.1 The Meta-Model for Session Entries

For the internal use in the `SessionExtractor` we defined a simple Ecore-model (see Section 2.4.2) for session entries (see Figure 4.3).

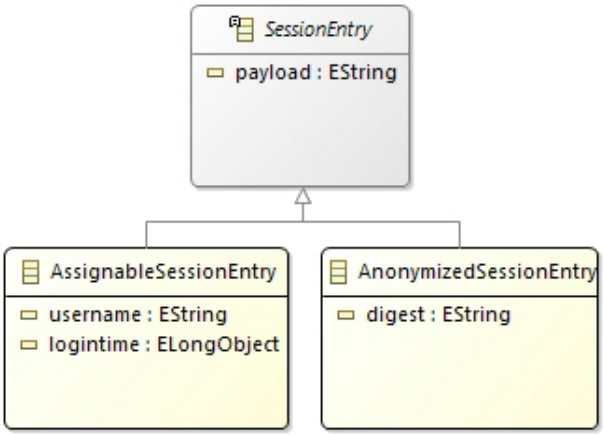

Figure 4.3. Class Diagram of the Meta-Model for `SessionEntries`

A session entry contains the payload (visited views, containing flows) extracted from the `ScreenEntryRecords` as comma-separated values (CSV). This is the content, that is written to the session log files later in the session extraction process. There are two types of `SessionEntries`: `AssignableSessionEntries` and `AnonymizedSessionEntries`. They provide different attributes to identify entries of the same sessions. The identifier of `AssignableSessionEntry` is the username and the login time. In the `AnonymizedSessionEntry` this information is hashed to a digest.

4.1.2 Reused Stages

The SessionExtractor reuses two stages of the TeeTime framework (see Table 4.2). The first column contains the stage name. The second column states the ratio of input to output ports. The TCPReaderStage for example has no input ports but an output port for received IMonitoringRecords. The third column represents the ratio of consumed to produced elements. The MultipleInstanceOfFilter<IMonitoringRecord> produces exactly one element for each consumed element. Other stages like the Directory2FilesFilter used in the BehaviorModelExtractor (see Section 4.2.2) produce multiple elements per consumed element.

Stage name	Input/Output signature	Consume/Produce signature
TCPReaderStage	0:1	0:1
MultipleInstanceOfFilter <IMonitoringRecord>	1:1	1:1

Table 4.2. Reused stages in the SessionExtractor

The TCPReaderStage reads monitoring records from a TCP port. On the first port it accepts IMonitoringRecords and on the second port it accepts StringRecords. Additionally, the capacity of the receiving buffer can be configured. The SessionExtractor uses the TCPReaderStage to receive all records produced by the monitored application.

The SessionExtractor only processes ScreenEntryRecords. Therefore, a MultipleInstanceOfFilter<IMonitoringRecord> selects them and discards other record types. The ScreenEntryRecords are passed on to the SessionExtractorStage.

4.1.3 Custom Stages

The custom stages we implemented for the SessionExtractor are shown in Table 4.3.

Stage name	Input/Output signature	Consume/Produce signature
SessionExtractorStage	1:1	1:1
SessionAnonymizingStage	1:1	1:1
SessionLogWriterStage	1:0	1:0

Table 4.3. Custom stages in the SessionExtractor

The SessionExtractorStage receives ScreenEntryRecords from the TCPReaderStage. It discards records, that can not be assigned to a session. One example are records from the login view. As the user is not logged in yet, no information about username and login time can be added to the record. Afterwards, the SessionExtractorStage builds a String by appending the values from the SessionEntryRecord seperated by semi-colons. An AssignableSessionEntry is instantiated with username and login time and sent to the output port.

In the `SessionAnonymizingStage` the `AssignableSessionEntry` is anonymized to an `AnonymizedSessionEntry`. The session id composed of username and logintime is encrypted with the Secure Hash Algorithm 2 (SHA-2). The digest and the payload form the new `AnonymizedSessionEntry` that is transferred to the next stage.

The `SessionLogWriterStage` accepts both types of `SessionEntries`. The session information is appended to the CSV file with the name "(username)-(logintime).dat" or "(digest).dat". In this way the `SessionExtractor` creates a log file for each session in the output directory. Those files can be used at any time for the user behavior analysis with the `BehaviorModelExtractor` which we introduce in Section 4.2.

4.2 Structure of the Pipe-and-Filter Architecture for Behavior Model Extraction

The `BehaviorModelExtractor` is also a Java application based on the TeeTime framework. Figure 4.4 shows it's Pipe-and-Filter-Architecture.

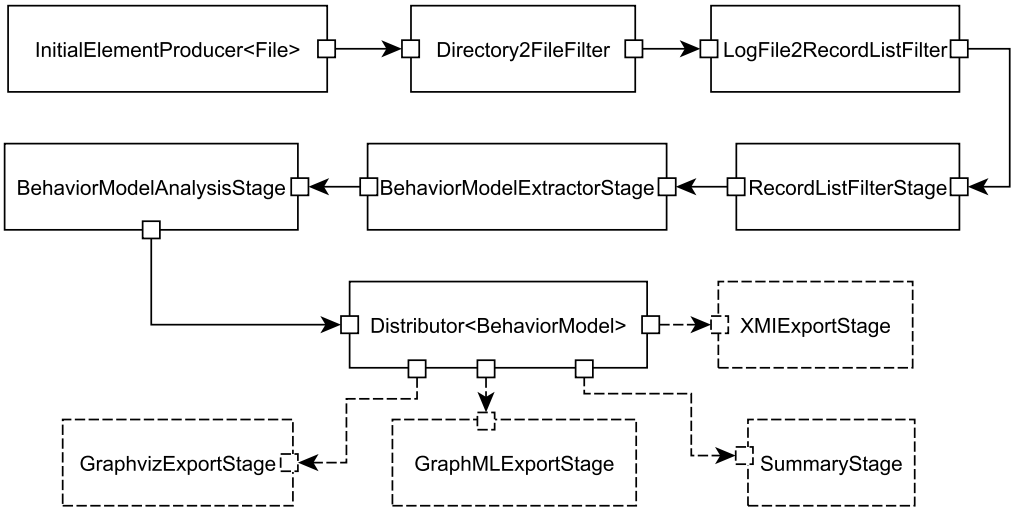

Figure 4.4. Pipe-and-Filter-Architecture of the `BehaviorModelExtractor`

The architecture consists of eleven stages in which the first two stages and the `Distributor` are reused stages of the TeeTime project. The first seven stages are traversed sequentially. The `InitialElementProducer<File>` determines the input directory of the session log files (option `-inputDirectory`). The `Directory2FilesFilter` lists all files with the file extension ".dat" which are converted to a list of SessionEntryRecords each in the `LogFile2RecordListFilter`. This list can be filtered via the option `-excludeFlow` in the

`RecordListFilterStage`. Afterwards, the main stage of the behavior extraction is executed. It builds the behavior model based on the meta-model we define in Section 4.2.1. The model is extended by probabilities and statistics in the `BehaviorModelAnalysisStage`. Finally, the reused `Distributor<BehaviorModel>` stage is called. It passes the model to all export stages simultaneously which can be activated by their respective command line argument.

The `BehaviorModelExtractor` provides the command line arguments listed in Table 4.4. Their functionality is explained in the stage description where they apply (see Section 4.2.2 and Section 4.2.3).

Name	Type	Required	Description

`-i`
`-input`
`-inputDirectory`
	`String`	yes	Input directory of the session log files.

`-o`
`-output`
`-outputDirectory`
	`String`	yes	Output directory of the visualization files.

`-h`
`-hist`
`-historic`
	`Boolean`	no	Treat all views as unique instances.

`-p`
`-proc`
`-process`
	`String`	no	Only extract behavior in given process.

`-e`
`-eflow`
`-excludeFlow`
	`List<String>`	no	Exclude this flow from monitoring.

`-s`
`-show`
`-showStatistic`
	`List<String>`	no	Show the statistic on state and/or transition label.

`-exportGraphviz`	`Boolean`	no	Export the behavior model to Graphviz file.
`-exportGraphML`	`Boolean`	no	Export the behavior model to GraphML file.
`-exportXMI`	`Boolean`	no	Export the behavior model to XMI file.
`-events` `-distinguishEvents`	`Boolean`	no	Distinguish transitions between two states by event name.
`-sum` `-summary`	`Boolean`	no	Summarize analysis in seperate CSV file.

Table 4.4. Command line arguments of the `BehaviorModelExtractor`

4.2.1 The Meta-Model for User Behavior

Van Hoorn et al. [2014] introduced the behavior model given in Figure 4.5 for the WESSBAS approach (see Section 2.5).

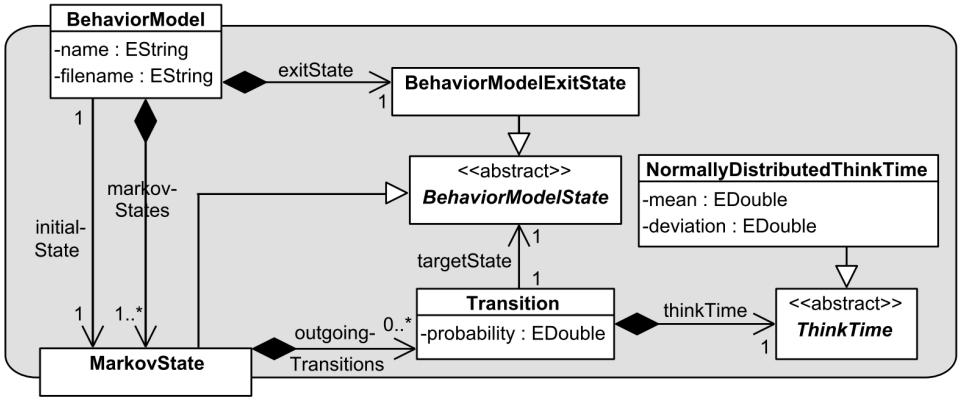

Figure 4.5. Behavior model of the WESSBAS DSL [Van Hoorn et al. 2014]

The `BehaviorModel` contains a set of `MarkovStates` and a dedicated exit state. In their model, exit states have to be modeled explicitly because they are not associated with services in the application model. Transitions are labeled with probabilities and contain a think time, which, in case of the WESSBAS approach, is given as a normal distribution. It is defined by the mean and the (standard) deviation.

Since the behavior model of the WESSBAS-DSL does not consider the influences of screenflow and workflow in the Application Model on the user behavior, we defined an similar Ecore-model (see Figure 4.6) to represent user behavior in our tool `BehaviorModelExtractor`.

Generally, it is based on the concepts of the WESSBAS meta-model. In comparison to their approach (see Section 2.5) we included two additional elements to group states in flows and processes. A state always belongs to a flow. A flow may be standalone or be part of a process. Consider that the same flow may appear in different processes. Just as in the WESSBAS meta-model we always include a synthetic end state in the behavior model. It is required to store statistics concerning think times of its preceding states. This information is referenced from the transitions. A state has any number of outgoing transitions with a certain probability. The transition references exactly one target state. The measured think times are saved in the transition, as well as the statistic calculated based on those think times. In addition to mean and deviation in the WESSBAS model we determine minimum and maximum think time for analysis reasons. Furthermore, we include the median as the most resistant statistic concerning outliers. The statistics calculated in the `BehaviorModelAnalysisStage` are discussed in Section 4.2.3.

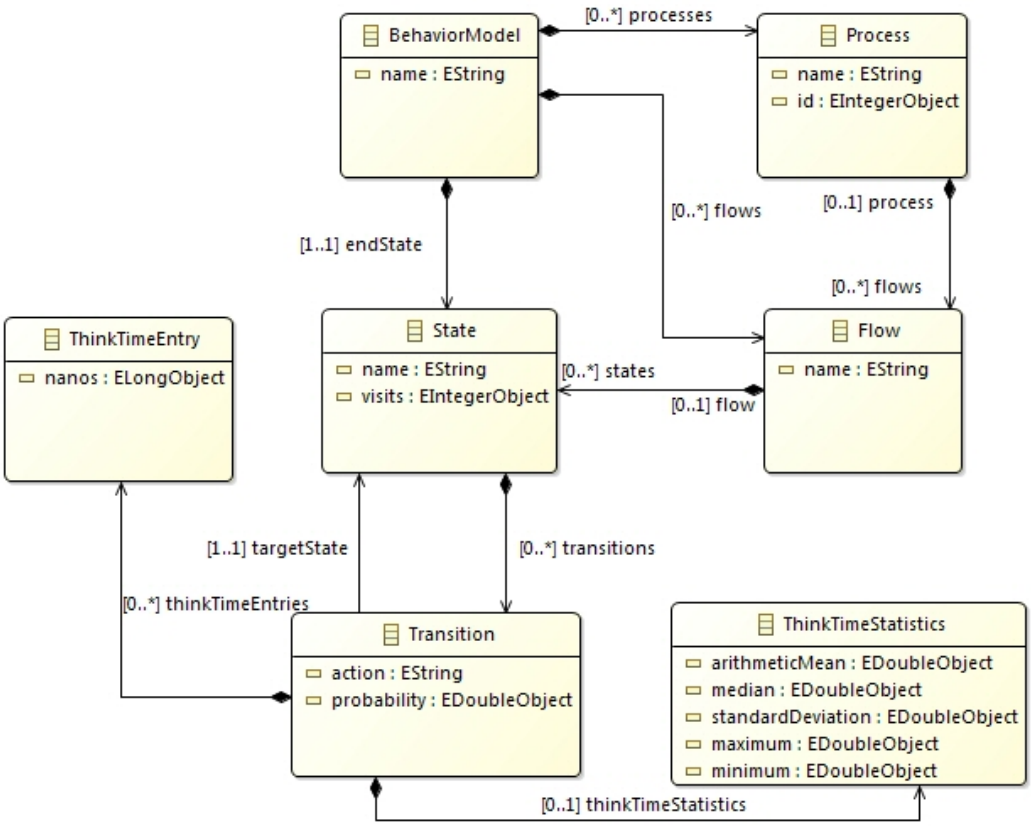

Figure 4.6. Class Diagram of the Ecore-based Meta-Model for the Behavior Model

4.2.2 Reused Stages

The `BehaviorModelExtractor` reuses three stages of the TeeTime framework. They are listed in Table 4.5 with the same structure as Table 4.2 in Section 4.1.2.

Stage name	Input/Output signature	Consume/Produce signature
`InitialElementProducer<File>`	0:1	0:1
`Directory2FilesFilter`	1:1	1:n
`Distributor<BehaviorModel>`	1:n	1:1

Table 4.5. Reused stages in the `BehaviorModelExtractor`

The `InitialElementProducer<File>` is a producer stage that is instantiated with a list of

elements. The `UserBehaviorExtractor` only passes one element to the stage. The element is the input directory of the session log files.

The list of files is determined by the `Directory2FilesFilter`. We implement the `File-Filter` interface to include only files with the ".dat" file extension. The result is a set of files that is sent to the `BehaviourModelExtractorStage`.

After the construction of the behavior model and the corresponding analysis, the `Distributor<BehaviorModel>` distributes the enriched behavior model among the export related stages `GraphvizExportStage`, `GraphMLExportStage`, `XMIExportStage`, and `SummaryStage`. These four stages can be declared as active, because they are independent and only read from the same behavior model. This enables a faster execution since the export tasks can be performed simultaneously.

4.2.3 Custom Stages

The custom stages of the `BehaviorModelExtractor` can be divided in three groups. The `LogFile2RecordListFilter` and the `RecordListFilterStage` prepare the behavior model extraction by reading the log files and filtering the input (based on the options `-historic`, `-process`, and `-excludeFlow`). The `BehaviourModelExtractorStage` and the `BehaviorModelAnalysisStage` build the model defined in Section 4.2.1, while the three final stages visualize the model or export it. Table 4.6 shows their signatures.

Stage name	Input/Output signature	Consume/Produce signature
`LogFile2RecordListFilter`	1:1	1:1
`RecordListFilterStage`	1:1	1:1
`BehaviourModelExtractorStage`	1:1	1:1
`BehaviorModelAnalysisStage`	1:1	1:1
`GraphvizExportStage`	1:0	1:0
`GraphMLExportStage`	1:0	1:0
`XMIExportStage`	1:0	1:0
`SummaryStage`	1:0	1:0

Table 4.6. Custom stages in the `BehaviorModelExtractor`

The `SessionLogFileReaderStage` is the counterpart of the `SessionLogFileWriterStage` (see Section 4.1.3). It creates a list of `SessionEntryRecords` based on the received CSV-files.

The list is sent to the `RecordListFilterStage`, which we only include in the architecture if at least one of the options `-historic` and `-process` is enabled. If the `-historic` is selected, the number of visits for all screens is remembered in a map. The number of the current visit is appended to the screen name. Therefore, in the later analysis recurring visits on the same view result in unique states. For example, consider following user behavior on the views A and B:

$$A \rightarrow B \rightarrow A \rightarrow B \rightarrow A$$

The result without the `-historic` option would be:

$$A \leftrightarrows B$$

With the `-historic` option we get:

$$A_1 \rightarrow B_1 \rightarrow A_2 \rightarrow B_2 \rightarrow A_3$$

This output is much closer to the user behavior, but statements about transition probabilities are more difficult, because the probability for $A \rightarrow B$ is now spread out to $A_1 \rightarrow B_1$ and $A_2 \rightarrow B_2$.

The second option is `-process` that expects a process name. If it is set, the `RecordListFilterStage` discards all `ScreenEntryRecords` that do not belong to the given process, but keeps `ScreenEntryRecords` from which the process has been entered or to which the process has been leaved. This allows us a concentrated analysis on one process, while maintaining the incoming and outgoing transitions.

Since we encountered some problems during the instrumentation of the b+m bAV-Manager we added the option `-excludeFlow`. It can be used multiple times as command line argument to filter screenflows with the given name. Those screenflows and the contained view are not considered in further analyses as if they have never been visited. This option can distort the recorded user behavior, because resulting gaps are repaired by connecting previous and subsequent views directly.

The `BehaviourModelExtractorStage` reads the session log files that passed the `RecordListFilterStage` and builds the behavior model defined in Section 4.2.1. Since every list of `ScreenEntryRecords` represents a session, the model is expanded session by session. For each consecutive `ScreenEntryRecord` the `BehaviourModelExtractorStage` executes the following steps. First, the stage checks, if the current view has already been visited in the given flow and process. If not, a new state is instantiated and added to the flow's and/or process' context. Second, the stage creates a transition from the previous state to this state. If the `-distinguishEvents`-option is set, the `BehaviourModelExtractorStage` creates multiple transitions between the same two views, if the event name differs. With the activation of this option, probabilities for all possible actions on a view can be determined individually. However, the complexity of the user behavior increases with the higher number of transitions. The think time is calculated by subtracting the current timestamp from the previous timestamp. That means, it is the time from the entry of the previous state to the entry time of this state. This is not an absolute precise calculation, because the time for the server communication and system operations (dark tasks) is included. We consider this in our evaluation in Chapter 5.

Afterwards, the `BehaviourModelExtractorStage` remembers the current state as last visited state for the next iteration. After all `ScreenEntryRecords` of one session have been integrated into the `BehaviorModel`, a synthetic end state named "*" is added to the model. It is required, because all statistics and probabilities are connected to a transition. Therefore a final transition is added from the last state of the session to the end state.

The `BehaviorModelAnalysisStage` complements the created behavior model with probabilities and statistics regarding think times. Each transition with think times $t_1, t_2, ..., t_n$ and with s visits of the source state is traversed to calculate:

- the probability

$$p = \frac{n}{s}$$

- the maximum think time

$$t_{\max} = \max(t_1, t_2, ..., t_n)$$

- the minimum think time

$$t_{\min} = \min(t_1, t_2, ..., t_n)$$

- the arithmetic mean

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} t_i$$

- the variance

$$v = \frac{1}{n} \sum_{i=1}^{n} (t_i - \bar{x})^2$$

- the standard deviation

$$\sigma = \sqrt{v}$$

- and the median

$$\tilde{x} = \begin{cases} t_{\frac{n+1}{2}} & n \text{ uneven} \\ \frac{1}{2}(t_{\frac{n}{2}} + t_{\frac{n}{2}+1}) & n \text{ even} \end{cases}$$

In comparison to the WESSBAS approach described in Section 2.5 we explicitly included the maximum/minimum think time and the median. The maximum and minimum think time do not have a big influence on the analysis, but the median is very important. Van Hoorn et al. [2014] assume a normal distribution of think times composed of the arithmetic mean and the standard deviation. This assumption is problematic, if user keep a session while dealing with something else. The think time for one transition gets really big, although the user did not even look at the application. Another problem are measurement or recording errors. Those two causes for extreme think time outliers distort the arithmetic mean and the standard deviation. This is especially true, if we are dealing with small sample sizes (less frequently visited views) like in our evaluation. The problem can be avoided by eliminating/down-weighting of outliers or using resistant statistics. The median is a very

resistant statistic, because it reliably eliminates outlier values. Therefore we included the median in our think time analysis.

The screenflow `PersonSelektierenUndBetrachten` provides a good example for outliers and their influence on non resistant statistics like the mean and the standard deviation. The think times from the state `PersonSelektierenUndBetrachten` to the state `ZusageberBearbeiten` are listed below:

$$
\begin{aligned}
t_1 &= 7s \\
t_2 &= 3s \\
t_3 &= 2s \\
t_4 &= 17s \\
t_5 &= 5s \\
t_6 &= 776s \\
t_7 &= 24s \\
t_8 &= 3s
\end{aligned}
$$

This results in following statistics:

$$
\begin{aligned}
t_{\max} &= 776s \\
t_{\min} &= 2s \\
\bar{x} &\approx 105s \\
\sigma &\approx 272s \\
\tilde{x} &= 6s
\end{aligned}
$$

Clearly, the outlier t_6 severely influences the arithmetic mean $\bar{x}$ and the standard deviation σ. Especially, because the sample size is so small and the screenflow `PersonSelektierenUndBetrachten` is not frequently used. The median $\tilde{x}$ however gives a good approximation, how long the users stay on the state `PersonSelektierenUndBetrachten` before transitioning to state `ZusageberBearbeiten`. Those measurements were taken from our experiment (see Chapter 5) to emphasize careful evaluation of statistics in the context of user behavior.

In order to simplify the analysis of statistics we added a simple `SummaryStage`. It is activated by the `-summary` option and executed parallel to the visualization stages. It writes one CSV file for the state statistics and one for the transition statistics to the output directory. The state statistics file list the number of visits for all view, that have been visited at least once. The transition statistics file lists source and target states, probability, minimum,

maximum, arithmetic mean, standard deviation and median for all user transitions from view to view. For our analysis we simply imported the CSV files to a spreadsheet and added filters to the columns.

We describe the stages for visualization, namely `GraphvizExportStage` and `GraphMLExportStage`, in Section 4.3.

Finally, the `XMIExportStage` writes the behavior model to an XMI file. The file format based on XML is optimized for data exchange of objects based on meta-meta-models. In the modernization process the results of the behavior model analysis might be reused from this structure.

4.3 Visualization of the User Behavior Model

The final version of the `BehaviorModelExtractor` provides two options for visualization of the user behavior in a graph.

The first option is Graphviz[1], enabled by passing `-exportGraphviz` as command line argument. Graphviz is an open source graph visualization software, that focuses on representing structural information as diagrams of abstract graphs and networks. Figure 4.7 shows an example graph, which we trimmed down to make it reasonably readable. As illustrated, even with only a small number of nodes, Graphviz uses a lot of space. This happens, because its layouter places a lot of nodes side by side.

The graph was generated with the `BehaviorModelExtractor` based on the experiment data (see Chapter 5) with following options: `-exportGraphviz`, `-exportGraphML`, `-exportXMI`, `-summary`, `-p RueckfrageErstellen`, `-excludeFlow WorkflowClient`, `-s min`, `-s max`, `-s deviation`, `-s mean`, `-s median`, `-s visits`.

The views are represented by ellipses including their unique name and the number of visits n. They are grouped by rectangles. The rectangles are screenflows and the rectangles that include other rectangles represent processes (highest hierarchical level). The transitions are shows as arrows between the views. They denote the event name, the probability of a user choosing this transition from the source view and the statistics presented in Section 4.2.3.

Graphviz' strength is the visualization of smaller graphs and the automatic layout of states, transitions and their labels. It intelligently arranges shapes and arrows to avoid overlapping. However, the generated graphs get confusing, when a lot of states and transitions are added. Since one can not hide and show elements and generally the elements take a lot of space, we could not use Graphviz for our evaluation. In certain cases the tool could not even generate an output with the message "trouble in init_rank". Because of this, we started using GraphML during our work.

[1] `http://www.graphviz.org/`

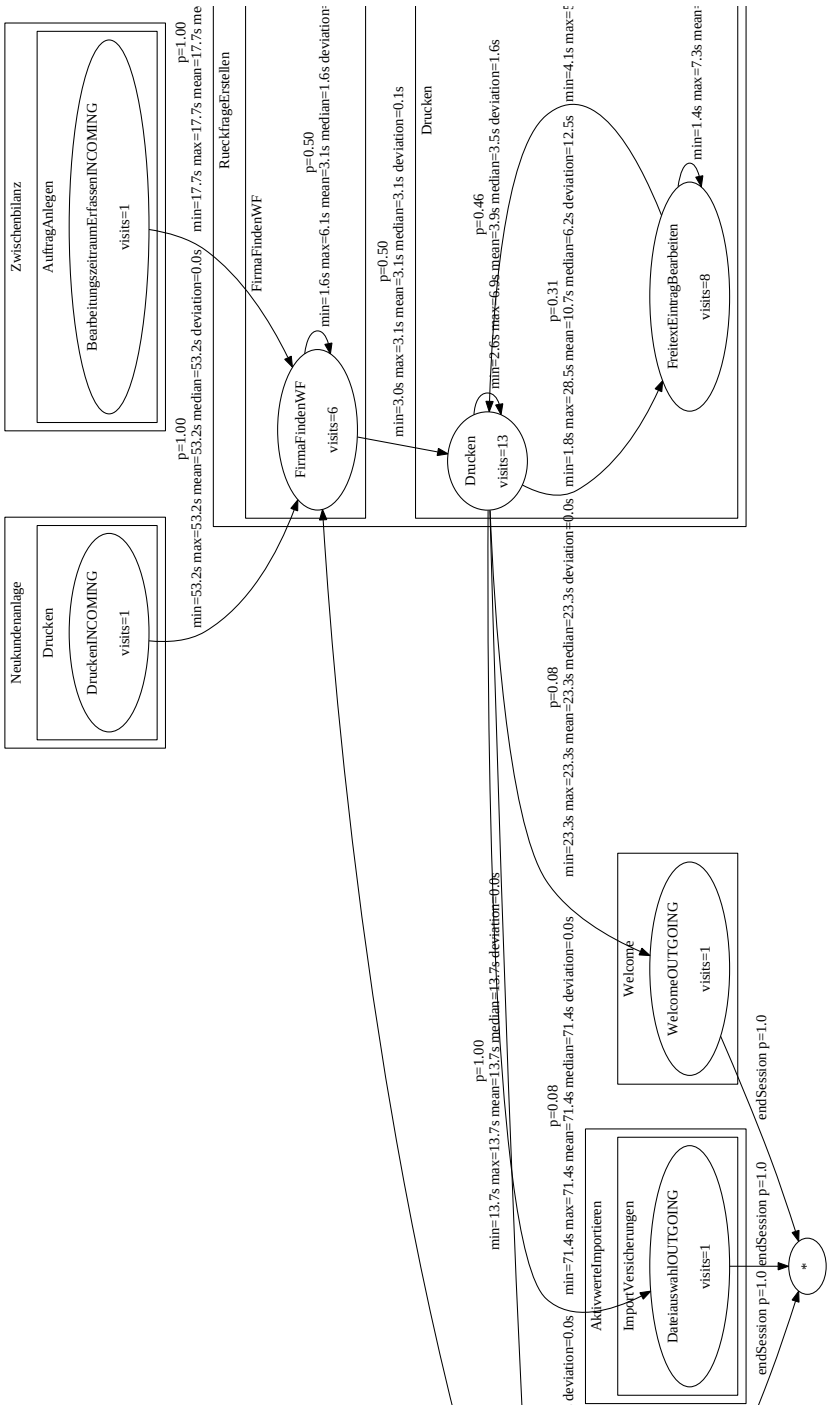

Figure 4.7. Example Visualization with Graphviz

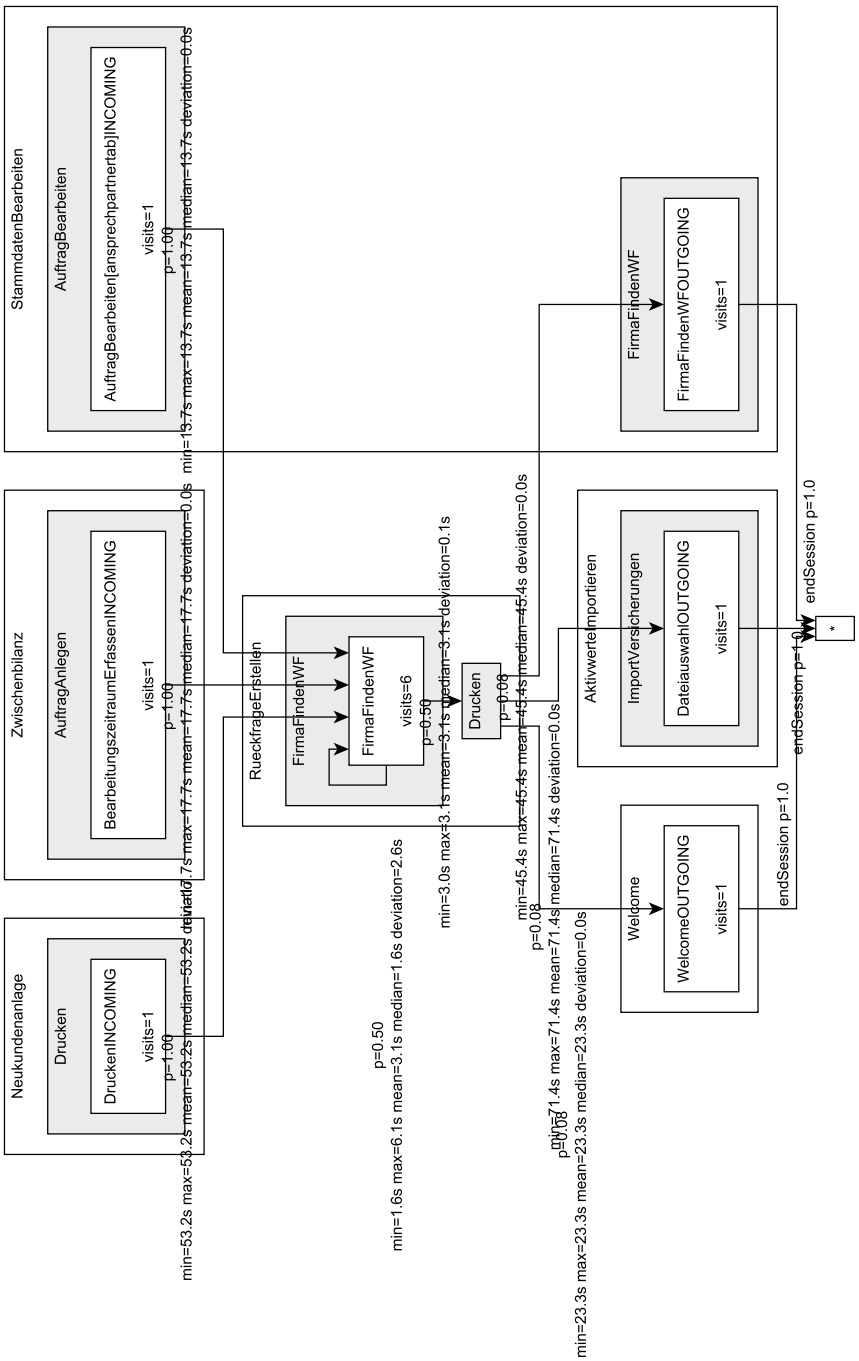

Figure 4.8. Example Visualization with GraphML

GraphML[2] (`-exportGraphML`) is the second visualization provided by the `BehaviorModelExtractor`. It offers a better clarity of hierarchical structures by folding and unfolding groups. Since the graphs reach a big size when looking at the behavior in the whole application, this feature is very helpful. The process and screenflow groups can be minimized and incoming and outgoing transitions simply touch the borders. An example visualization with GraphML is given in Figure 4.8. This is the same user behavior model as in Figure 4.7. We minimized the rectangle `Drucken` representing the screenflow for document generation to show the (un-)folding feature.

The structure and the labeling of the elements is similar to the Graphviz visualization. In order to customize the graphs generated by GraphML with additional labels and layout information, we used the extensions provided by the yEd Graph Editor[3]. Listing 4.1 illustrates a small part of a generated GraphML file for the use in yEd.

The whole visualization of the behavior model consists of the graph G. We globally define, that all edges are directed in line 3. In line 4 we define the synthetic end node $*$ with a yEd-label (NodeLabel). In line 12 the node `Welcome` is added. `Welcome` is a screenflow and therefore a group node (foldertype="group"). Additional information for yEd is provided in lines 14-17. The node contains a new graph called "Welcome:". The node `WelcomeWelcome` represents the view `Welcome` in the screenflow `Welcome`. Finally, we define an edge between the node `WelcomeWelcome` and the end state "*". The data for yEd states, that we do not want an arrow for the source state, but a standard arrow for the target state. The label is composed of the event name `endSession` and the probability.

The yED Graph Editor provides several windows, that support the navigation through complex graphs, that we generate in our evaluation in Chapter 5. The overview window shows the position in the graph while zooming in. The second window provides three tabs of information: neighborhood, predecessors and successors. Selecting a state, the neighborhood tab shows all connected states and the containing group. The predecessors tab shows all states that have an incoming transition to the selected state, while the successors tab shows all states reached by outgoing transitions. Selecting a transition the neighborhood tab only shows the source state and the target state in their respective groups. The latter is very helpful, when dealing with a lot of incoming and outgoing transitions, which often means overlapping labels. Group of nodes can be minimized and maximized with a "plus"/minus" button to further increase the overall clarity.

One disadvantage of GraphML in yEd compared to Graphviz is, that the graph is not layouted automatically. However, the yEd editor provides a lot of different layouts. The quality of the layout results differs quite a lot. The hierarchical layout is one of the few layouts, that provides a clear arrangement in our use case. Since a session in the b+m bAV-Manager always starts with the `Welcome` view, the hierarchical structure puts it at the top. Additional, the processes can be read top down, which is very useful. The resizing of nodes and the positioning of labels are each separate steps.

[2] `http://graphml.graphdrawing.org/`

[3] `https://www.yworks.com/products/yed/`

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<graphml ... />
  <graph id="G" edgedefault="directed">
    <node id="*">
      <data key="d6">
        <y:ShapeNode>
          <y:NodeLabel>*</y:NodeLabel>
        </y:ShapeNode>
      </data>
    </node>
    <node id="Welcome" yfiles.foldertype="group">
      <data key="d6">
        <y:GroupNode>
          <y:NodeLabel autoSizePolicy="content" modelName="internal"
            modelPosition="t">Welcome</y:NodeLabel>
        </y:GroupNode>
      </data>
      <graph edgedefault="directed" id="Welcome:">
        <node id="WelcomeWelcome">
          <data key="d6">
            <y:ShapeNode>
              <y:NodeLabel>Welcome</y:NodeLabel>
            </y:ShapeNode>
          </data>
        </node>
      </graph>
    </node>
    <edge id="WelcomeWelcome--endSession->*" source="WelcomeWelcome"
      target="*">
      <data key="d10">
        <y:PolyLineEdge>
          <y:Arrows source="none" target="standard" />
          <y:EdgeLabel>endSession p=0.46</y:EdgeLabel>
        </y:PolyLineEdge>
      </data>
    </edge>
    ...
  </graph>
</graphml>
```

Listing 4.1. GraphML example extended by yEd data

For our analysis we followed these five steps:

1. Generate the GraphML file with the `BehaviorModelExtractor`
2. Open the GraphML file with yEd (all nodes lay on top of each other)
3. Select "Tools" and "Fit Node to Label" (corrects node size)
4. Select "Layout" and "Hierarchical" (best arrangement for nodes)
5. Select "Layout" and "Label Placement" (slightly improves the position of labels)

As illustrated in Figure 4.8 before, the positioning of labels very often is not optimal. Locations in the graph, where a lot of transition labels are displayed, the assignment to the correct lines is difficult. This problem, however, share both visualization approaches. An additional minor concern is, that the file size of GraphML is three times bigger than Graphviz. This is caused by the XML structure and the yEd extensions within the data elements.

Chapter 5

Evaluation

In this thesis we analyze the user behavior in the web application b+m bAV-Manager using an experiment in order to evaluate our analysis tools.

In Section 5.1 we describe the design of the experiment and the tasks. Section 5.2 deals with the execution of the experiment, that took place at the b+m Informatik AG. Afterwards, we explain how the data was collected in Section 5.3. The results of our evaluation are presented in Section 5.4 and discussed in Section 5.5. Section 5.6 summarizes the most important results of the evaluation We end the chapter with the threads to validity (Section 5.7) and how we reduced them.

5.1 Experimental Design

Following the GQM approach (see Section 2.8), we define the goal of our experiment as G: "Identifying abnormal use of screenflows and workflows by users in the monitored application". The identified spots should be considered specifically regarding the modernization of the software.

We define five research questions (RQ) based on our main goal G:

- RQ1: What are the least frequently used (or unused) views of the application?
- RQ2: What are the most frequently used views of the application?
- RQ3: On which views do the users take the most time and where do the think times vary a lot?
- RQ4: In which screenflows does the usage significantly differ from the application model?
- RQ5: In which workflows does the usage significantly differ from the process definition?

In order to make those questions measurable, we refine them as follows:

- RQ1.1: Which views have the least amount of visits?
- RQ2.1: Which views have the most amount of visits?
- RQ3.1: On which views do the think times show a high average/median?

- RQ3.2: On which views do the think times show a high variance/standard deviation?
- RQ4.1: Which events are rarely/never called?
- RQ4.2: Which views are visited in which order?
- RQ5.1: Which tasks are rarely/never executed?
- RQ5.2: Where do users leave/reenter processes?

To answer these questions, we determined the following metrics. The metrics M1 to M7 are supported by the analysis stage `BehaviorModelExtractor` (see Section 4.2.3). The M8 and M9 are not calculated directly but can be taken from the resulting visualizations and summary files.

- M1: Visits per view
- M2: Think time average
- M3: Think time median
- M4: Think time variance
- M5: Maximum think time
- M6: Minimum think time
- M7: Transition probability
- M8: Outgoing transitions mid-process
- M9: Visits on the escalation view

Since we execute the experiment with developers of the b+m bAV-Manager and not with its customers, we have to ensure, that the user behavior is realistic. Therefore, a good coverage of the use cases is required. Together with the project manager of the b+m bAV-Manager we examined that the defined processes in the application pretty much match those use cases. The processes in the software can be divided in two groups: general processes and customer specific processes. A typical general process, that is used by every customer, is "Stammdaten bearbeiten". It allows the selection and the editing of a company administrated with the b+m bAV-Manager. In contrast a typical customer specific process is "Gutachten erstellen". This is the yearly executed process for each administrated company to do calculations and generate documents. The process is customer specific, because they have different accounting procedures, archiving methods, connected third-party software etc.

In this experiment we use the general processes and the demonstration processes (as replacement for customer specific processes) as they represent a typical functionality. Table 5.1 shows the tasks, that were given to the participants.

ID	Task/Process	Customer-specific	Priority
T1	Stammdaten bearbeiten	no	high
T2	Gutachten	yes	high
T3	Gutachten (kurz)	yes	high
T4	Probeberechnung	yes	high
T5	Stammdaten betrachten	no	medium
T6	Einzelauskunft	yes	medium
T7	Neukundenanlage	yes	medium
T8	Rechnungsdaten importieren	no	low
T9	Aktivwerte importieren	no	low
T10	Rückfrage erstellen	yes	low
T11	Zwischenbilanz	yes	low

Table 5.1. Tasks to be fulfilled in the experiment ordered by priority

The process "Stammdaten bearbeiten", as mentioned above, is the main administration process for firm data in the b+m bAV-Manager. After the selection of a firm, everything related to that firm can be edited: employees, contacts, accounts, insurances, opinions, benefit plans etc. "Gutachten" is the calculation process, that is usually executed once per year per firm. Based on the firm data and the calculation results of the third party calculation engine an opinion in form of a document is created. In the configuration of the demonstration process the user chooses, if his work should be checked by a controller. If this option is selected, he must select another user with the controller role. The user can not progress with the process until the controller verifies his work. After the verification the document is printed and archived. Finally the user can create an invoice for the calculated firm and a revision of the data base is created.

The process "Gutachten (kurz)" is a shorter variation of the "Gutachten" process. After the calculation a verification is always required and no certain user is selected, but any user with the controller role may verify the document. The task of creating an invoice is omitted. "Stammdaten betrachten" is the read-only variation of "Stammdaten bearbeiten". It can be executed even if the data base was locked by another user. The process "Einzelauskunft" allows the users of the b+m bAV-Manager to search for persons across all firms. Their master data can be edited and documents can be generated for single persons. If a new firm should be managed with the software, the process "Neukundenanlage" is used. In the demonstration process it is simply "Stammdaten bearbeiten" followed by a print dialog for creating a document with information for the new firm. This document is addressed to a previously defined contact.

The processes "Rechnungsdaten importieren" and "Aktivwerte importieren" allow the import of CSV files with account data and insurance data respectively. Since these two processes are very short, we assigned only a low priority. The last two processes in the list have a low priority, too. "Rückfrage erstellen" consists of only two tasks: selection of a

firm and creating a document. The document can be sent to the contact person to ask for information etc. The process "Zwischenbilanz" is similar to "Gutachten" and therefore not as interesting for the analysis as the main calculation process.

5.2 Operation

The experiment took place at the b+m Informatik AG during April 2016. We deployed the latest version 3.11 of the b+m bAV-Manager extended by our monitoring components on a test server. The server runs Apache Tomcat 6.0[1] with Java 8[2] and Microsoft SQL Server 2012[3] on Microsoft Windows Server 2012[4]. The participants used their own computer to access the web application and received a short written guide (see Appendix A) in German. It gives a short introduction into the topic of the thesis and formulates tasks to be executed by the participants. In addition to the list of processes and their prioritization, we suggested supplementary tasks to be fulfilled by the participants. If they had completed the processes on their shortest path, the results would have been not very interesting. We proposed to change system settings, to add/modify products, schemes, firms, persons and to provoke errors in certain cases. Finally, in the guide we point out that the experiment focuses on realistic use cases and the participants should imitate the usage of real customers.

The participants were five software professionals with different focuses: software development, software architecture, and/or project management. Their experience with the b+m bAV-Manager also differs. Most of them work in the field of insurances, but developers from the banking division were included, too. Since their experience with the b+m bAV-Manager is very different, all participants were allowed to ask questions during the whole experiment. The less complex tasks (T6, T7, T8, T9, T10, T11) took approximately three minutes, while the more complex tasks (T1, T2, T3, T4, T5) took approximately five minutes.

5.3 Data Collection

All sessions started in the experiment period were monitored by the Kieker framework using the monitoring components implemented in Chapter 3. The application wrote to a local TCP port which was listened to by the `SessionExtractor`. The `SessionExtractor` wrote the session log files to a local directory. We applied the `-anonymize` option to prevent the assignment of sessions to users. At the end of the experiment we transferred all session log files to another computer for the analysis. The `BehaviorModelExtractor` was executed several times to generate graphs for the application usage in general and for each process in particular.

[1] `http://tomcat.apache.org/`

[2] `https://www.java.com/`

[3] `http://www.microsoft.com/de-de/server-cloud/products/sql-server/overview.aspx`

[4] `https://www.microsoft.com/de-de/server-cloud/products/windows-server-2012-r2/overview.aspx`

Additionally, the participants were asked to suggest improvements for the processes and the application in general. Since the focus of our experiment was the simulation of realistic use of the application, we did not design a questionnaire for the qualitative feedback. After the participation we directly asked for:

- Problems with screenflows and workflows
- Improvements for screenflows and workflows
- General problems with the b+m bAV-Manager
- General improvements for the b+m bAV-Manager

We present the results of the feedback at the end of Section 5.4.

5.4 Results

In this section we present the results of the evaluation. According to the research questions, we make some quantitative statements, before discussing the results in Section 5.5. We collected 53 session log files with 2381 recorded user activities.

RQ1 (What are the least frequently used (or unused) views of the application?): There are a lot of views that were visited less than ten times during the experiment (73 of 109). Since the behavior model only includes the views that have been visited at least once, we checked, if it contains all monitored views. We identified 23 of 109 total views, that have not been called by the users in the experiment at all (see Table 5.2):

The views 2, 3, 8, 14, 15, 16, 17, and 23 are obsolete views, that can not be reached by the user any more. They either have been removed completely or have been replaced by other views. The views 9, 11, 12, 18, and 19 are very specialized views, that are only required for certain calculations or configurations. The other unused views were mainly implemented for tests, for complex imports we could not cover in the experiment, and for use in customer-specific process definitions only.

RQ2 (What are the most frequently used views of the application?): The three most frequently used views are *WorkflowClientTasks* (963 visits), *AuftragBearbeiten* (313 visits), and *FirmaFindenWF* (143 visits). Since we split up the view *AuftragBearbeiten* into tabs, its number is the sum of these eight tab visit numbers.

RQ3 (On which views do the users take the most time and where do the think times vary a lot?): The longest think times we observed on the view *GutachtenVerifizieren*. This is not surprising because on this view controllers check the validity of calculations. The value however is very small compared to a realistic scenario. It seems as the participants of the experiment took the controlling aspect serious, but did not take a realistic time there. The controllers normally open the Portable Document Format (PDF) document created another user, which explains the long think time, and afterwards finish the task by accepting or declining the calculation. The shortest think times we observed on simple views, that

Number	View
1	AnlegenOderVerschieben
2	BruttoNettoRechner
3	ChooseBusinessService
4	CreateAccountListTest
5	DatenbasenAnzeigen
6	DatenbasisAuswaehlen
7	DatenbasisUeberschreiben
8	Eskalationshinweis
9	Gesellschaftszuordnungen
10	Imports
11	LeistungsplanEigenschaften
12	LeistungsplanPrognoseErweiterungVerwalten
13	LeistungsplanTemplateBearbeiten
14	OaseOnline
15	RechnungDrucken
16	SetUpBusinessService
17	SperreAktivieren
18	StacksEditieren
19	SteigerungTabelle
20	StichtagAktualisieren
21	TabellenUpDownload
22	UKasseErweiterungBearbeiten
23	Versorgungsanalyse

Table 5.2. Unused View of the b+m bAV-Manager in the Experiment

serve as starting point for other activities like the *Welcome* (0.5 seconds) view or the view *SystemkonfigurationStart* (0.3 seconds). The dashboard *WorkflowClientTasks* also has a low think time average, but this most likely results from the problem described in Section 3.7 The view *WorkflowClientTasks* is often called by the workflow engine with times smaller than 0.1 seconds. Considering the variance of think times, there are not a lot of conspicuous values. We found, that as expected the search views have less variance in their think times as the editing views. The sole exception is *PersonSelektierenUndBetrachten* which has to be the result of outlier values, because it is a simple, global search for persons administrated in the application.

RQ4 (In which screenflows does the usage significantly differ from the application model?): Even though the application has a clear structure, which view can be visited after which view (screenflow definitions) and which task has to be executed after which task (process definition). The number of possible transitions so high, that we could not check each for significant differences. We concentrated our analysis on recurring patterns and potential interesting parts of the behavior model based on experiences with the software.

We discuss this in Section 5.5.

RQ5 (In which workflows does the usage significantly differ from the process definition?): There are two simple ways of identifying deviations from processes, that are covered by our analysis tool. The first one is the number of visits on the view *WorkflowEskalation*, that is activated automatically, if an error occurs during a dark task. These errors are often raised by wrong configurations by the users. The only two processes where this occurred are *Gutachten* (7 visits) and *StammdatenBearbeiten* (4 visits). The second way is to count incoming but especially outgoing transitions mid-process. The most interruptions can be found in the user behavior model of *StammdatenBearbeiten*, *StammdatenBetrachten*, and *Gutachten*. Only a few occur on *Gutachten (kurz)* and *Zwischenbilanz*.

Finally, we want to summarize the results of our questions to the participants. A lot of the suggestions we got, align with the usability case study by Schmidt [2015]. The tool tips of the b+m bAV-Manager (which are provided by the platform) often overlay the buttons to which they belong. This often prevents, that the users can navigate through the views easily. Another mentioned problem is the structure of the dashboard which shows the active processes. For most users the separation of processes into "all", "group", and "personal" is not intuitive. They sometimes do not know where the process goes, if another person has to take the next task, and they do not understand, why they can not start processes from the "all" view. Additionally, the participants often felt lost in the process progress, because there is not indication, what task is active, what tasks have already been completed and which tasks still have to be done. When the user is in a process execution, he only sees the view corresponding to the current task. The headline of the views can be modified by property files, but the implementation is very inconsistent for the b+m bAV-Manager. Overall the workflow-orientation of the application is the biggest challenge for the user, even more for the customers than for the developers, that participated in our evaluation.

5.5 Discussion

In Section 5.4 we provided the analysis results according to our research questions. In this section we discuss the results focusing on the processes (as they represent the tasks in our experiment).

- ***Stammdaten bearbeiten*:** The main process for editing data bases shows some deviations from the process definition. The process is mainly left for changes in the system configuration (*Systemkonfiguration*) and the process *Datenbasenpflege*. Additionally, there are several visits on the error view *WorkflowEscalation*. This view is shown, when an error occurs during a dark task. This is a common problem in the application, when a user wants to edit a data base, that has already been opened by another user. It is the dark task *Sperre aktivieren*, that fails, because it can not lock an already locked data base. There two possibilities to solve the problem. The user can cancel the process execution, wait for the data base to unlock and then start a new process instance. The other option

is, to start a new process *Datenbasenpflege* and unlock the data base manually. This is not recommended, but the analysis shows, that both possibilities are chosen evenly often. Hereby, the outgoing transitions to *Datenbasenpflege* are explained, too. To avoid the creation of processes that are stuck in *WorkflowEscalation* because of locked data bases, the firm search could be extended. Either the search does not provide locked data bases for selection, or the view clearly indicates, that certain data bases are locked. Both alternatives would reduce the unnecessary start of *Stammdaten bearbeiten* and similar processes on locked data bases. The user could give up the dangerous unlocking in the process *Datenbasenpflege*. The transitions to *Systemkonfiguration* show, that often global settings have to be adjusted during the process execution. This could be improved by partially granting access to system configurations from the view *AuftragBearbeiten*. On this view, for example, products can be assigned to opinions. Opinions are firm specific, but products are globally defined in the configuration. If the application allowed the creation of products on the view *AuftragBearbeiten* directly, the user would not have to make a detour to the system configuration (leaving the process in this case).

- ***Gutachten*:** The central process for yearly calculations in the b+m bAV-Manager shows a similar user behavior model as the process *Stammdaten bearbeiten*. There are many incoming and outgoing transitions from the process itself. They mostly refer to other processes and not to the system configuration. This indicates, that the process is often executed parallel to other tasks of the user. We expected this behavior, because *Gutachten* is one of the few processes that contain multiple swimlanes. The validation of the opinion created by the responsible person is done by a controller. Therefore, the process has to be suspended and usually the responsible person will not wait for the controller to complete his work, but continue with other tasks. In terms of the modernization aspect, it might be helpful for the users to receive a notification, if a process is delegated to them. This could either be implemented as a pop-up window or as highlighted row in the dashboard. This might reduce the total time *Gutachten* take and also reduce the number of tasks executed simultaneously. Both effects influence the overall productivity of the users negatively.

- ***Gutachten (kurz)*:** The behavior model of the short version of the process *Gutachten* does not provide any incoming or outgoing transitions during the process executions. This might be the case, because the more complex task have been removed and the workflow is straightforward: Selection of a data base, calculation, and verification without a lot of options. The selection of a controller is also not included in the process definition. Therefore, the user can take the controlling task himself and simplify the overall execution. This process has proven to match the user behavior and can remain unchanged during the modernization.

- ***Probeberechnung*:** Similar to the process *Gutachten (kurz)*, this process does not show any interruptions. In comparison to other calculation processes, on the print view more

documents are generated.

$$\text{Drucken} \rightarrow \text{Drucken} \ (p = 0.63)$$

This supports the idea of the process *Probeberechnung* of testing calculations and print templates before the important calculation for the annual statements. Surprisingly, only a few users started a process *Datenbasenpflege* to clean up the data bases after their test calculations. Normally, the copied data base is deleted after its use in the calculation. In this case, either the copies remain in the application forever or the users delete it at a later time. For the modernization one might consider to make the visit of the data base administration mandatory. This would force the user to actively decide to keep the copied database.

- ***Stammdaten betrachten*:** This process compared to the read-only version of "Stammdaten bearbeiten" is used less frequently. We think, that it is mostly used if a data base is locked and the user wants to see the data. In other cases the users seem to just use *Stammdaten bearbeiten* and change no values. As one can see in the behavior model, a lot less different views where visited in the process *Stammdaten betrachten*. As it is one of the basic processes, a change in the modernization is not required.

- ***Einzelauskunft*:** The most interesting part of the behavior model is, that the *Einzelauskunft* is very often started after the data base administration in *Datenbasenpflegen*. We did not found a reasonable explanation for this user behavior. From the business perspective those two processes are completely independent. When the experiment is repeated at a customer of the b+m bAV-Manager, this incident has to be reviewed. Therefore, we can not give an advice for the modernization.

- ***Neukundenanlage*:** The behavior of users in the process *Neukundenanlage* is very similar to the process definition. Although the process was executed quite often, there are no outgoing or incoming transitions in between the main two tasks: the creation of a data base with the basic information about the new firm and the print dialog. In comparison to the general *Stammdaten bearbeiten* process, the tabs for the employees and contacts administration are more often called. For the modernization, these tabs could be reordered, especially because the creation of contacts is required. This however, might confuse user, which are used to the current order.

- ***Rechnungsdaten importieren* and *Aktivwerte importieren*:** Those two process require the user to prepare a CSV file for uploading accounting or insurance data to the application. Since the number of visits in these two processes is very low, the participants avoided this task predominantly. Therefore, we have to wait for the experiment on the customer servers, to make propositions for the software modernization.

- ***Rückfrage erstellen*:** This short process had a low priority and was only executed a few times. We can not make relevant statements about the process definition, but there were no interruptions, that indicate abnormal use of the process.

▻ ***Zwischenbilanz*:** *Zwischenbilanz* is a similar process as *Gutachten*. The difference is, that calculations are not done for the date of balance but for any date. The behavior model of this process show, that it is often executed parallel to other *Gutachten* processes. This is caused by the similarity of those two processes and we assume, that users of the b+m bAV-Manager execute similar tasks at the same time or consecutively. For the modernization, this is no problem.

In the following we want to discuss some non process-specific results. A typical pattern of screen- and workflows is the combination of a search view and a editing view (master-detail pattern). For example, on the first view A the user searches and selects a person. With the open button or a double click he enters the second view B, where he can edit the data related to that person. Nearly all flows in the b+m bAV-Manager organized like this, do not show any search results by default. The search would provide too many results. Therefore the users are obligated to filter the result beforehand. A perfect filter would allow the user to find their expected result in one step, resulting in following behavior model:

$$A \to A \ (p = 0.5)$$
$$A \to B \ (p = 0.5)$$

Looking at the results of the experiment, there are screenflows with this pattern which have worse probabilities like *FirmaFindenWF* ($A \to A \ (p = 0.55)$). These screenflows could be improved in the modernization by adding filters or adding columns in the result table.

In Section 3.4 we describe, how we split up the complex views organized with tabs into multiple views. Therefore statistics for each tab were recorded and we get the probabilities for transitions between the tabs. Table 5.3 shows the number of visits on each tab of the view *AuftragBearbeiten*. Surprisingly, the *GutachtenTab* has been visited more often than all other

Tab name	**Number of visits**
GutachtenTab	90
DatenTab	89
MitarbeiterTab	48
AnsprechpartnerTab	31
LeistungsplaeneTab	20
RechnungenTab	19
VersicherungenTab	12
BerechtigungenTab	4

Table 5.3. Number of Visits on Tabs of the *AuftragBearbeiten* View

tabs. Even the *DatenTab*, that is the start tab of the view and that has to be visited each time the view is called, has fewer views in total. For the modernization of the b+m bAV-Manager we could consider a rearrangement of the tabs based on the user behavior model results. The *GutachtenTab* is the candidate for the new start tab and the *BerechtigungenTab* could be moved to the last position.

As mentioned in Section 5.4 the participants suggested to improve the tool tips of the application. This will be modernized automatically by using the latest version of the b+m gear platform and Vaadin. However, the provided dashboard for the processes has not been improved yet. As suggested by Schmidt [2015] the categories "all", "group", and "personal" should be arranged in tabs to make this separation clear. Another idea is the visualization of the process progress in a linear progress bar at the top of views. The required information could be retrieved by an extended version of the monitoring service we have written in Section 3.3.2. For the modernized b+m bAV-Manager, the latest b+m gear version would also provide these information. The challenge is in this context to simplify the process definitions in Business Process Model and Notation (BPMN) to a simple progress bar. The system tasks could be omitted in such a visualization but decisions are difficult to illustrate appropriately.

5.6 Summary

For the evaluation of the `SessionExtractor` and the `BehaviorModelExtractor` we organized an experiment with software professionals at the b+m Informatik AG. The participants got a list of prioritized processes to execute them in the b+m bAV-Manager. The results show, that the user behavior focuses on few central views and several view have not been visited once during the experiment. On the screenflow and on the workflow level we determined several situations, where the application model or the process definition could be improved during the modernization. We suggest, for example, the rearrangement of tabs, the improvement of search filters, the elimination of unused views, the modification of the locking procedure for data bases, etc.

5.7 Threats to Validity

This section summarizes the threats to internal and external validity, that could have influenced our results. One threat to *internal validity* is, that the participants were not sufficiently experienced with the b+m bAV-Manager. This is true for a small part of the participants, that have not worked with the b+m bAV-Manager yet. However, they are familiar with the structure of b+m gear applications and have competences in the field of insurances. At the beginning of each experiment, we gave a short introduction to the b+m bAV-Manager, if the participant only had a small experience with the software.

Another problem is the motivation of the participants. Since they had to interrupt their daily work and take one hour time, they might have executed the task carelessly and in a hurry. All our participants took part voluntarily, assured that they have the time and said, that they would do the tasks seriously. We observed, that after around one hour the motivation decreased. We assume, that motivation is only a minor threat to our results, because the experiment time was limited to a short period (one hour) and the

participants for the most part want to advance the modernization of the b+m bAV-Manager. The difference between the usage of a software by their developers and the usage by their customers is another threat to internal validity. Software developers are used to test an application from the technical side. They quickly navigate through the application, fill only required fields with default values and choose shortcuts to save time. We tried to avoid this threat by emphasizing the goal of the experiment, simulating realistic user behavior in the application. But since the participants executed the tasks on their own, this might be the biggest threat to the result's validity. This particularly affects the think time measured by our monitoring components. If the pension schemes are not configured in detail but only the simple settings, the tasks take a lot less time.

The tasks, given to the participants, were derived from the process definitions of the demonstration data of the b+m bAV-Manager. This implies two additional threats to internal validity. First, the demonstration processes might not represent the actual productive processes at the customers. We reduced the threat by choosing the most common processes for our experiment and even prioritized important processes over unimportant ones. The demonstration processes are very similar to the customer processes, which mainly add a lot of dark tasks for e-mailing, archiving, printing and communication with third-party services. Usually the business logic does not differ from the default processes. The second threat is, that the processes do not represent the tasks fulfilled with the b+m bAV-Manager. Since the main processes are used similarly by all customers, we have to assume, that they represent the actual workflow. However, a task might me composed of consecutive processes and system configurations. We are not aware of such process chains in the b+m bAV-Manager, therefore we expect, that they do not influence the results of our experiment.

The prioritization of processes might also be a problem. We assigned the priorities "high", "medium", and "low" to the tasks in order to support a realistic usage overall. The threat to validity is, that the most frequently used processes are the most frequently used, because we have given this prioritization. We can not prove otherwise, but we are convinced, that no prioritization would have resulted in an even worse distribution. Without a focus on important processes, the participants would have completed the tasks in an uniform distribution as they would cover every process and would not focus on the important ones.

A threat to external validity is, that our experiment only shows the behavior of software professionals in the b+m bAV-Manager application. It might not represent the behavior of the real customers. The user behavior might even differ between the customers, because there are small and big insurers that use the software. We tried to reduce this threat by emphasizing a realistic use of the b+m bAV-Manager, but a follow-up experiment on the customer's servers is required. There are also parts of the application, that are only used by few customer, e.g. the view *Gesellschaftsverteilung*. Some views even can not be reached through the application, since they only appear in process definitions of a few customers.

Chapter 6

Related Work

In this chapter, we discuss related work on extraction of user behavior profiles and software modernization.

The methodology of Menascé et al. [1999] is often cited for the definition of the CBMG, the corresponding workload model and the introduced clustering algorithm. They recognized that hits/views/visits per seconds is not an appropriate characterization of e-commerce sites, but it can be represented by a behavior model graph. We applied their approach of collecting requests (`TCPReaderStage`), writing sessions to log files (`SessionExtractor`), and generating different CBMGs based on them (`BehaviorModelExtractor`).

The WESSBAS approach by Van Hoorn et al. [2014] (Section 2.5) uses the creation of behavior models for generating test cases with realistic workload behavior. It provides an behavior extractor similar to our `BehaviorModelExtractor`. The WESSBAS tooling focuses on clustering and extraction of workload intensity, while we specifically analyzed screenflows and their orchestration in workflows. The meta-model we designed for the behavior analysis reuses concepts from the WESSBAS behavior model. This includes the naming and the basic structure. Differences were described in Section 4.2.1. We discussed an extension of the WESSBAS tools with the authors but the goals of their and our behavior model analysis differed too much. The reason was mainly the workflow representation, but also the representation of the screenflow is difficult as illustrated by Schulz [2014]. They extended and used the WESSBAS approach. They integrated the generation of test plans in the b+m gear platform using the example of the reference application. They encountered some problems constructing an application model based on the screenflow definitions. Therefore, we did not include the extraction of an complete screenflow application model in our work.

Blum [2013] also generate test cases based on user behavior profiles. The difference to our and the WESSBAS approach is, that the behavior model is created based on a web crawler. It collects all possible transitions between views of a web application and adds them in even distribution to the model. The probability for a transition per default is one divided by the number of different transitions from that view. These probabilities can be adjusted by testers or experts, but these values have to be more unrealistic than monitoring the user interactions directly. The crawler, however, constructs the complete application model from the frontend perspective. This enables additional analysis and test possibilities, that our approach does not cover, because we restrict the behavior model on the actual

visited views.

A similar work is done by Dallmeier et al. [2013]. Their tool WebMate[1] also extracts a behavior model (called usage model) by crawling a web application. They abstract from probabilities and focus on the coverage of the complete website. They use the information gathered at a certain point in time to compare the behavior with earlier revisions of the website. Additionally, their tool searches for cross browser inconsistencies. This approach is another good example of analyzing application structures, but not for monitoring realistic user behavior.

Schmidt [2015] executed a case study for the b+m bAV-Manager regarding usability for the imminent modernization of the software. With eye-tracking and surveys they compiled a list of frontend improvements. We can support many of the suggestions with our results in Section 5.4 and Section 5.5. Their analysis, in comparison to our thesis, focused on the user behavior on the views. We took a more technical approach and compared the user behavior on the screenflow and the workflow level with the application model. Nevertheless, usability problems also have an effect on the user behavior. Abnormal use of screenflows and processes discovered in Section 5.4 may be caused by them.

The Dynamod project [Van Hoorn et al. 2011] introduce with MDM a concept for extracting models from legacy software systems. The extracted models are tranformed and serve as basis for MDSD. In our work we also extract models from a legacy software, but we do not focus on the application structure but on the user behavior. We also do not include the resulting user behavior models in the MDSD process directly. For the modernization we use the quantitative and qualitative results of the evaluation.

Schulz et al. [2014] introduce an approach for obtaining workload models from systems developed in the Dynamod project. Similar to the work of Van Hoorn et al. [2014], they extract user behavior models to generate realistic workload on the SUT. In comparison to this thesis, where we mainly looked for qualitative statements about processes and screenflows, they target performance and workload analysis.

[1] https://webmate.io/de/

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we designed and implemented components for monitoring and analyzing user behavior in session-based applications. From the results of the experiment instrumenting the b+m bAV-Manager, we derived suggestions for the imminent modernization of the software.

After we presented the foundations and technologies in Chapter 2, we first implemented monitoring components in Chapter 3. We updated the Kieker framework in the outdated b+m gear platform and defined an custom record for screen- and workflow activities. Then we implemented interceptors to instrument the b+m bAV-Manager and the PM-System (based on the current b+m gear platform). In Chapter 4 we developed two tools: The `SessionExtractor` for receiving the previously defined records and writing session log files, and the `BehaviorModelExtractor` for constructing a behavior model and generate visualizations with statistics concerning probabilities and think times. Both Java command line programs are based on the TeeTime framework. In Chapter 5 we designed an experiment for simulating realistic use of the b+m bAV-Manager on a test server.

The results show, that several propositions for the software modernization can be derived from the generated user behavior models. The suggestions contain screen- and workflow improvements, UI improvements, but also revealed weaknesses in the user behavior itself, in the process "Probeberechnung" for example. This work shows, that with the extraction of user behavior profiles, we can not only make quantitative statements about think times, probabilies and visits but also qualitative statements about processes and user behavior. The visualization of the extracted models in a graph help to understand the structure of the application on the frontend layer. In comparison to similar dynamic analysis, which usually focuses business logic and data layer, the user behavior analysis supports user's understanding of the workflow-orientation concept.

The Kieker framework has proven to be a stable and robust tool for dynamic analysis. The Kieker IRL allowed us to define a programming language independent data structure for recording user behavior. The IRL itself made some problems, e.g. overriding the `equals()` method of the `AbstractMonitoringRecord` which is declared as `final`. The TeeTime framework provides a very clear structure and is very customizable. We did not notice any performance issues during our analysis. The complete analysis of the experiment

data is executed in approximately one second on a laptop (Windows 7 Professional, Intel Core i7-4700MQ CPU @ 2.4GHz, 4 cores, 16 GB RAM). The advanced features of the TeeTime framework, we were had to take from the documentation, like exception handling with strategies and asynchronous execution of stages. It is more difficult to find suitable predefined stages for reuse. Since there is no fully documented repository of the built-in stages, we had to search in the TeeTime stages project or write our own custom stage directly.

7.2 Future Work

In our future work, we want to repeat the experiment on productive systems of b+m bAV-Manager customers. This would improve the validity of the results by using customized processes, behavior of real users not software professionals, behavior of daily work and a bigger amount of data. In the b+m bAV-Manager the users represent roles like administrator, controller etc. On the one hand we could group those users by group and monitor their behavior in each group. On the other hand we could monitor the users independent from their role and use clustering methods (see related work in Chapter 6) to find differences and similarities. There might be processes, that could be customized for certain roles, e.g a test calculation may look different for general users and controllers. Another option is the analysis of sessions of a single user. With the resulting visualizations, he might better understand his behavior in the application.

Concerning the modernization of the b+m bAV-Manager, we want to implement the changes suggested in this thesis and suggested in other work about the software. A high priority has the visualization of workflow information. Currently, the customers only have the static BPMN process definition and only few information at run time (at the dashboard). The new b+m gear platform provides the necessary services for accessing the workflow engine. The biggest challenge is, to reduce the complexity of the process definitions, hide dark tasks for example, and display the progress in a simple way. The first idea we want to implement, is a linear progress bar, that provides enough information to orientate in a workflow.

The in Chapter 4 implemented tools `SessionExtractor` and `BehaviorModelExtractor` also contain room for improvements in the future. For example, the `SessionExtractor` can encrypt sessions in order to protect the privacy of the users. We might move the encryption to the monitoring side, so that the `SessionExtractor` only receives the information, that a view has been visited by someone. The tool is then able to group the records to sessions, but the session can never be assigned to users again. Currently, the `SessionExtractor` only accepts record from a TCP port induced by the reused `TCPReaderStage`. In the future a reader for log files should be added as a stage to the Pipe-and-Filter architecture. This would require a new command line parameter to choose between the two input options. The `BehaviorModelExtractor` may be extended with additional filters, statistics and output formats. In order to determine the unused views, we had to compare the tracked views with

the application model manually. A better approach would be to integrate the application model into the analysis. This is the most complex addition, we anticipate in the future. Finally, the GraphML export could be improved by adding colors or optimizing the labels. This would underline the separation in view, screenflow and workflow and support the qualitative analysis in general. Transitions, that have a high probability could be represented with a broader line and states, that have a high number of visits could be represented with bigger boxes. Most likely we will not support Graphviz visualizations in the future, because it does not provide sufficient functionality for complex user behavior models.

Appendix A

Experiment Guide

Originally the experiment guide was distributed as Microsoft Word document in German.

Einleitung

Im Rahmen meiner Master Thesis an der CAU Kiel zum Thema „Erstellung von Benutzerverhaltensprofilen für die Software-Modernisierung“ möchte ich ein Experiment durchführen. Mit dem Experiment soll eine möglichst reale Nutzung des bAV-Managers nachgestellt werden, um mit meinen entwickelten Komponenten das Verhalten von Nutzern in der Anwendung zu analysieren. Die Ergebnisse sollen bei der anstehenden Modernisierung des bAV-Managers berücksichtigt werden. Diese Anleitung gibt euch einen kurzen Überblick, worum es in diesem Experiment geht und wie Monitoring und Analyse umgesetzt werden. Am Ende erhaltet ihr die eigentliche Aufgabe, um Sitzungsdaten zu erzeugen.

Details zum Experiment

Ich habe auf dem Testserver ps-bavmanager-windows die aktuellste Version des bAV-Managers (3.11) installiert. Sie ist um einige Monitoring-Komponenten erweitert, um mit dem Kieker Framework Benutzeraktivitäten aufzuzeichnen. Die Aktivitäten einer Sitzung werden als separate Log-Datei anonymisiert auf dem Server abgelegt. Am Ende des Experiments werde ich diese Dateien mit meinem Analysewerkzeug untersuchen. Das Ziel ist es, viel und wenig besuchte Masken zu identifizieren und Schwachstellen in den Screenflow- und Prozessdefinitionen aufzudecken. Außerdem suche ich nach Auffälligkeiten bei Denkzeiten/Verweildauer auf einzelnen Masken (hohe Durchschnittsverweildauer, große Schwankungen in den Verweildauern, ...).

Die Aufgabe

Nehmt euch ungefähr eine eine Stunde Zeit die unten in der Tabelle aufgelisteten Prozesse im bAV-Manager auf dem Testserver ps-bavmanager-windows durchzuführen (`http://`

ps-bavmanager-windows.intranet.kiel.bmiag.de:8080/bav-manager/). Um ein möglichst realistisches Verhalten zu simulieren, habe ich die Prozesse mit Prioritäten versehen. Prozesse mit einer hohen Priorität solltet ihr mindestens 5-mal durchführen, bei Prozessen mit einer geringeren Priorität reichen 2-3-mal. Außerdem habe ich Vorschläge gesammelt, welche Aktivitäten man innerhalb der Prozesse jeweils zusätzlich erledigen kann. Das sorgt für ein bisschen mehr Variabilität, als die Prozesse immer auf dem kürzesten Weg abzuschließen. Manche Aktivitäten, wie das Anlegen von Produkten und Hochladen von Druckvorlagen, finden außerhalb des Prozesses statt (→ Systemkonfiguration). Diejenigen, die sich fachlich mit dem bAV-Manager auskennen, sollten natürlich eine realistische Bearbeitung der Prozesse anstreben. Für die Anderen lohnt sich noch ein Blick in das bAV-Manager-Wiki (https://portal.bmiag.de/wiki/bavmanager/tiki-login.php?user=Anonymus&pass=**********).

Dank

Vielen Dank für eure Unterstützung!

Prozess	Priorität	Mögliche Zusatzaktivitäten
Stammdaten bearbeiten	Hoch	Firmendaten ändern, Leistungsplan ändern, Gutachten ändern, Gutachten-Varianten anlegen, Ansprechpartner ändern/zuordnen, Mitarbeiterdaten ändern, Versicherungen ändern, Rechnungen ändern, Produkte anlegen, Produktzuordnungen vornehmen, neue Druckvorlage anlegen, globale Ansprechpartner anlegen, Prognose berechnen, Hintergrundberechnung, Freitexte bearbeiten
Gutachten (kurz)	Hoch	Wie bei „Stammdaten bearbeiten“ und zusätzlich: Gutachten als fehlerhaft zurückgeben, Kontrolle delegieren
Gutachten	Hoch	Wie bei „Gutachten (kurz)“ und zusätzlich: Mit/Ohne Kontrolle durchführen, keine Rechnung erstellen, Rechnung als fehlerhaft zurückgeben
Probeberechnung	Hoch	Wie bei „Gutachten“ und zusätzlich: Datenbasen kopieren/löschen/umhängen
Stammdaten betrachten	Mittel	Wie bei „Stammdaten bearbeiten“ aber nur mit lesendem Zugriff
Einzelauskunft	Mittel	–
Neukundenanlage	Mittel	Wie „Stammdaten bearbeiten“ und zusätzlich: Dokument als fehlerhaft zurückgeben (auf Drucken-Maske)
Rechnungsdaten importieren	Niedrig	–
Aktivwerte importieren	Niedrig	–
Rückfrage erstellen	Niedrig	–
Zwischenbilanz	Niedrig	Wie bei „Gutachten“

Table A.1. Processes to be Executed During the Experiment

Appendix B

External Libraries and Excluded Components

Table B.1 and Table B.2 list the external libraries used by the SessionExtractor and the BehaviorModelExtractor.

Library file	License
hppc-0.7.1.jar	Apache License, Version 2.0
jcommander-1.48.jar	Apache License, Version 2.0
kieker-1.12-emf.jar	Apache License, Version 2.0
slf4j-api-1.7.12.jar	MIT License
slf4j-simple-1.7.12.jar	MIT License
teetime-2.1.jar	Apache License, Version 2.0
teetime-stages-2.1.jar	Apache License, Version 2.0

Table B.1. External libraries used in the SessionExtractor

Library file	License
commons-math3-3.6.1.jar	Apache License, Version 2.0
hppc-0.7.1.jar	Apache License, Version 2.0
jcommander-1.48.jar	Apache License, Version 2.0
jctools-core-1.2.jar	Apache License, Version 2.0
kieker-1.12-emf.jar	Apache License, Version 2.0
slf4j-api-1.7.12.jar	MIT License
slf4j-simple-1.7.12.jar	MIT License
teetime-2.1.jar	Apache License, Version 2.0
teetime-stages-2.1.jar	Apache License, Version 2.0

Table B.2. External libraries used in the BehaviorModelExtractor

Since the Massachusetts Institute of Technology (MIT) License[1] is compatible with the Apache License, Version 2.0[2] [Apache Software Foundation 2016], we distribute our work

[1] https://opensource.org/licenses/MIT/

[2] https://www.apache.org/licenses/LICENSE-2.0/

under the Apache License, Version 2.0 itself. We used following libraries in addition to frameworks already mentioned in this work: High Performance Primitive Collections for Java[3], JCommander[4], Simple Logging Facade for Java[5], The Apache Commons Mathematics Library[6] and Java Concurrency Tools[7].

There are parts of our implementation, that can not be published because they contain confidential information of the b+m Informatik AG. This includes the following components:

- **Interceptors:**
  - `BAVScreenEntryMethodInvocationInterceptor.java` (Specific interceptor for the b+m bAV-Manager)
  - `GearScreenEntryMethodInvocationInterceptor.java` (General interceptor for b+m gear applications)
- **Platform service for process information retrieval:**
  - `MonitoringModel.service` (Monitoring service defined in the b+m gear service layer DSL)
  - `MonitoringImpl.java` (Implementation of the monitoring service)
- **Workaround for including the latest Kieker version into the old b+m gear platform:**
  - `BAVManagerLifeCycleInterceptor.java` (Disable monitoring so that the activation of the platform monitoring is skipped, and afterwards initialize the monitoring manually)

Those components were described as detailed as possible in Chapter 3 and Chapter 4. Consider, that the interceptors have to be rewritten anyway, if a non gear application should be monitored. The platform service and the workaround are b+m gear specifics and become obsolete with the modernization to the current version 4 of the platform (explained in Section 3.4).

[3] `https://github.com/carrotsearch/hppc/`
[4] `https://github.com/cbeust/jcommander/`
[5] `http://www.slf4j.org/`
[6] `https://commons.apache.org/proper/commons-math/`
[7] `https://github.com/JCTools/JCTools/`

Bibliography

[Allen and Garlan 1992] R. Allen and D. Garlan. *Towards formalized software architectures*. Technical report CMU-CS-92-163. Carnegie Mellon University, School of Computer Science, July 1992. (Cited on page 9)

[Apache Software Foundation 2016] Apache Software Foundation. *ASF Legal Previously Asked Questions*. 2016. URL: http://apache.org/legal/resolved.html. (Cited on page 81)

[Basili and Weiss 1984] V. R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering* SE-10.6 (Nov. 1984). (Cited on page 29)

[Blum 2013] D. Blum. Konzeption einer Anwendung zur automatischen Extraktion von Nutzerverhaltensmodellen aus Webseiten und Erstellung von Lasttestskripten. Master's thesis. Fakultät für Informatik der technischen Universität München, 2013. (Cited on pages 12 and 71)

[Brambilla et al. 2012] M. Brambilla, J. Cabot, and M. Wimmer. *Model-Driven Software Engineering in Practice*. 1st edition. Morgan & Claypool Publishers, Sept. 2012. (Cited on pages 13, 14)

[Dallmeier et al. 2013] V. Dallmeier, M. Burger, T. Orth, and A. Zeller. "WebMate: Generating Test Cases for Web 2.0". In: *Software Quality. Increasing Value in Software and Systems Development*. Volume 133. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2013. (Cited on pages 12 and 72)

[Dittrich 2013] G. Dittrich. Analyse und Modernisierung einer Altanwendung mit Methoden der MDM (Model Driven Modernization). Bachelor's thesis. Wirtschaftsakademie Schleswig-Holstein, 2013. (Cited on page 28)

[Eclipse Foundation 2016] Eclipse Foundation. *The Eclipse Modeling Project*. 2016. URL: https://eclipse.org/modeling/. (Cited on pages 15 and 17)

[Hahn 2015] M. Hahn. *b+m Informatik AG. Employee handbook*. b+m Informatik AG. 2015. (Cited on page 20)

[Harms 2014] A. Harms. *b+m bAV-Manager. User manual*. b+m Informatik AG. 2014. (Cited on page 28)

[Jung 2013] R. Jung. *An instrumentation record language for kieker*. Technical report. Department of Computer Science, Kiel University, Germany, Aug. 2013. (Cited on pages 6 and 9)

[Kieker 2015] Kieker. *Kieker web site*. 2015. URL: http://kieker-monitoring.net/. (Cited on page 6)

[Ludewig and Lichter 2010] J. Ludewig and H. Lichter. *Software Engineering. Grundlagen, Menschen, Prozesse, Techniken*. 2nd edition. dpunkt.verlag, Mar. 2010. (Cited on page 13)

[Menascé et al. 1999] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes. A Methodology for Workload Characterization of E-commerce Sites. In: *Proceedings of the 1st ACM Conference on Electronic Commerce*. EC '99. New York, NY, USA: ACM, 1999. (Cited on pages 12, 13, and 71)

[Reimer 2013] S. Reimer. *b+m gear Java 2.9.7. Handbook*. b+m Informatik AG. 2013. (Cited on pages 20, 22, and 26)

[Schmidt 2015] A.-C. Schmidt. Entwicklung und Einsatz eines interaktiven Prototyps zur Verbesserung der Usability einer Software der b+m Informatik AG. Master's thesis. Fachhochschule Kiel - Fachbereich Medien, 2015. (Cited on pages 28, 65, 69, and 72)

[Schulz 2014] E. Schulz. Integrating performance tests in a generative software development platform. Diplomarbeit. Kiel University, Department of Computer Science, June 2014. (Cited on pages 18 and 71)

[Schulz et al. 2014] E. Schulz, W. Goerigk, W. Hasselbring, A. Van Hoorn, and H. Knoche. Model-Driven Load and Performance Test Engineering in DynaMod. In: *Proceedings of the Workshop on Model-based and Model-driven Software Modernization (MMSM '14)*. Volume 34. Softwaretechnik-Trends, Aug. 2014. (Cited on page 72)

[Shaw 1989] M. Shaw. Larger scale systems require higher-level abstractions. In: *Proceedings of the 5th International Workshop on Software Specification and Design*. IWSSD '89. Pittsburgh, Pennsylvania, USA, 1989. (Cited on page 9)

[Stachowiak 1973] H. Stachowiak. *Allgemeine Modelltheorie*. Springer-Verlag, 1973. (Cited on page 13)

[Stahl et al. 2007] T. Stahl, M. Völter, S. Efftinge, and A. Haase. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. 2nd edition. dpunkt, May 2007. (Cited on pages 14, 15)

[Steinberg et al. 2008] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro. *EMF: Eclipse Modeling Framework*. Pearson Education, 2008. (Cited on page 16)

[Van Deursen et al. 2000] A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: an annotated bibliography. *Sigplan Notices* 35.6 (2000). (Cited on page 14)

[Van Hoorn et al. 2013] A. Van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, and S. Köster. *DynaMod: Dynamische Analyse für modellgetriebene Software-Modernisierung*. Technical report TR-1305. Department of Computer Science, Kiel University, Germany, Aug. 2013. (Cited on page 29)

[Van Hoorn et al. 2011] A. Van Hoorn, S. Frey, W. Goerigk, W. Hasselbring, H. Knoche, S. Köster, H. Krause, M. Porembski, T. Stahl, M. Steinkamp, and N. Wittmüss. Dynamod project: Dynamic analysis for model-driven software modernization. In: *Joint Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM 2011) and the 5th International Workshop on Software Quality and Maintainability (SQM 2011)*. Volume 708. Mar. 2011. (Cited on pages 28, 29, and 72)

[Van Hoorn et al. 2008] A. Van Hoorn, M. Rohr, and W. Hasselbring. Generating probabilistic and intensity-varying workload for web-based software systems. In: *Performance Evaluation – Metrics, Models and Benchmarks: Proceedings of the SPEC International Performance Evaluation Workshop 2008 (SIPEW '08)*. Lecture Notes in Computer Science (LNCS). Heidelberg: Springer, June 2008. (Cited on pages 12 and 18)

[Van Hoorn et al. 2009] A. Van Hoorn, M. Rohr, W. Hasselbring, J. Waller, J. Ehlers, S. Frey, and D. Kieselhorst. *Continuous monitoring of software services: Design and application of the Kieker framework*. Technical report TR-0921. Department of Computer Science, Kiel University, Germany, Nov. 2009. (Cited on pages 5, 6)

[Van Hoorn et al. 2014] A. Van Hoorn, C. Vögele, E. Schulz, W. Hasselbring, and H. Krcmar. Automatic extraction of probabilistic workload specifications for load testing session-based application systems. In: *Proceedings of the 8th International Conference on Performance Evaluation Methodologies and Tools*. VALUETOOLS '14. ICST, Brussels, Belgium, 2014. (Cited on pages 17–20, 47, 51, 71, 72)

[Van Hoorn et al. 2012] A. Van Hoorn, J. Waller, and W. Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering (ICPE 2012)*. ACM, Apr. 2012. (Cited on page 5)

[Wohlin et al. 2000] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering*. Norwell, MA, USA: Kluwer Academic Publishers, 2000. (Cited on page 30)

[Wulf and Hasselbring 2015] C. Wulf and W. Hasselbring. The Pipe-and-Filter Architectural Style Revisited: From Basic Concepts toward Smart Framework Implementations. Submitted for publication. 2015. (Cited on pages 9–11)