Debug Tools for Orchideo

Debugging in an MDSD and Aspect-oriented Development Environment

Tim Felgentreff
Lysann Kessler
Christina Palm
Stephanie Platz
Frank Schlegel
Philipp Tessenow

Software Architecture Group, Hasso-Plattner-Institut, Universität Potsdam, D-14482 Potsdam, Germany,
`{tim.felgentreff, lysann.kessler, christina.palm, stephanie.platz, frank.schlegel, philipp.tessenow}@student.hpi.uni-potsdam.de`

ex|xcellent
solutions

Table of Contents

Debug Tools for Orchideo 1

Developing an Application with the orchideo Framework 9
Christina Palm

Static Analysis of orchideo Advice Weaving 39
Stephanie Platz

Debug Support for orchideo 71
Lysann Kessler

Post-mortem Analysis of Debug Traces 103
Tim Felgentreff

Exception Visualization 129
Philipp Tessenow

Continuous Integration For Eclipse Plug-ins 161
Frank Schlegel

Summary 183

Erklärungen 199

Debug Tools for orchideo

Our World is complex, so is our Software Since computers have come into widespread use, the number and complexitiy of problems that are solved using software has grown progressively. To deal with the quickening demands for more and more sophisticated software integrated development environments grow ever more to supply developers with tools to better meet the demands of consumers. The Eclipse Platform is designed for building integrated development environments (IDEs) [1]. Since the release of Eclipse 3 in 2004 the lines of code have increased twelve-fold[1].

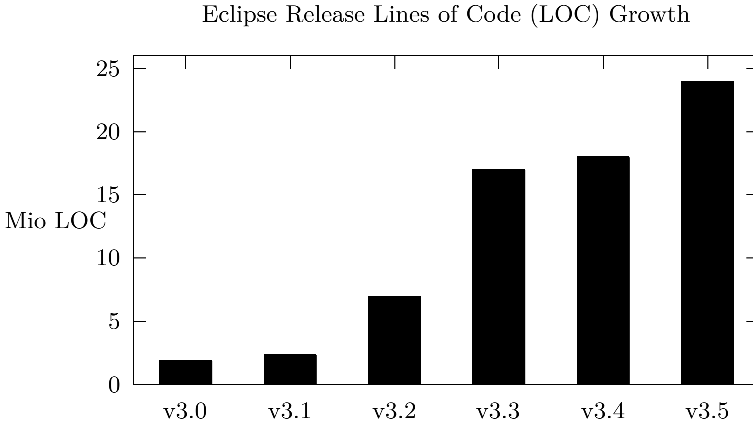

Fig. 1: Eclipse Releases since 3.0

To manage the growing size and complexity of software, the two software development approaches aspect-oriented programming (AOP) and model-driven software development (MDSD) became very popular in the last few years.

orchideo is a framework built on top of Eclipse that unites AOP and MDSD which we will elaborate on in the first part of this work [?]. We have built tools to extend the Eclipse SDK with orchideo specific abstractions. Tools to show errors and support understanding a program at the time of writing, such as source code analysis, are presented in part [?] of this study.

MDSD and AOP, however, do not relief a developer of the burden of debugging. When programs break, the programming abstraction is lost [2]. Debugging

[1] http://www.coderfriendly.com/wp-content/uploads/2009/07/eclipse_galileo.png

on a different level of abstraction costs time as the problem has to be understood and solved from two different persepectives.

Numerous other techniques for interactive debugging, logging of program state and analysis and visualization thereof help the programmer regain a certain abstraction level and solve problems more quickly. We have built tools to apply these debugging techniques to orchideo which we present in parts [?], [?] and [?].

Different debugging techniques apply to different phases of development. Initially, new software systems are small and tracing their various execution flows either statically or dynamically is easily possible. This is also the phase where static analysis techniques are most useful. However, as software grows, a domain specific structure or better yet domain specific debugging tools become more important. As software is deployed and enters the maintenance cycle, logging and post-mortem crash analysis is used [3].

Finally, during maintenance when patches to the software have to be applied to deployed systems, the system stability is of utter importance. Continuous integration systems help to ensure tests in large software systems are run upon each change in a source code repository. This ensures that new code does not break old functionality. We present such a system for orchideo and Eclipse in part [?] of this discourse.

Bachelor Thesis

Developing an Application with the orchideo Framework

Christina Palm

Supervisors:

Dr. Michael Haupt, Malte Appeltauer
Prof. Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute,
Potsdam, Germany

June 25, 2010

Developing an Application with the orchideo Framework

Christina Palm

Hasso Plattner Institute
Potsdam, Germany
christina.palm@student.hpi.uni-potsdam.de

Abstract. The two software development approaches aspect-oriented programming and model-driven software development became very popular in the last few years. orchideo is a framework that provides a combination of both concepts for developing business applications. This paper describes how aspect-oriented programming and model-driven software development are realized in orchideo and how we used the orchideo|suite to develop TeltowCar, an application for managing a car repair shop.

1 Introduction

The orchideo|suite [4] is a model-driven, aspect-oriented framework which is based on the Eclipse integrated development environment (IDE) [1], the Eclipse Modeling Framework (EMF) [5], and the Eclipse Graphical Modeling Framework (GMF) [6]. The orchideo|suite contains a set of different tools for developing business applications.

The orchideo|engine is the core and aspect-oriented part of the orchideo framework. It coordinates the other components and provides the functionality to develop orchideo aspects. Aspect models can be created with the help of a graphical editor. The base implementation of orchideo aspects is generated from these models by the orchideo|engine. The aspects cooperate with the orchideo|engine. The engine enables a combination and configuration of aspects by letting the user toggle aspects on and off. In that way aspects should become reusable and different projects should be able to use different (and individually adjusted) functionality. The orchideo|engine works at runtime and is responsible for weaving the aspects. Aspects can be used for generating source code from models.

orchideo|objects contains a set of predefined features in form of aspects. The aspects of orchideo|objects can be used for the management of objects. These aspects handle for example persistence (`PersistenceAspect`) and creation and destruction of objects (`ObjectAspect`) [7]. Furthermore orchideo|objects provides a graphical editor for application models. Some of the aspects in orchideo|objects are responsible for generating source code from the application model. Hence orchideo|objects provides functionalities for model-driven development of enterprise applications.

Besides the orchideo|engine and orchideo|objects the orchideo|suite contains orchideo|documents, orchideo|ui-designer and orchideo|views. orchideo|documents

is a documentation generator, which can generate text documents from any EMF based model. orchideo|views is a platform independent modular construction system for creating graphical user interfaces. orchideo|ui-designer is a tool for creating definitions for graphical user interfaces with the help of a specific GMF diagram type which can be used as input for orchideo|documents as well. In the context of this work, only orchideo|engine and orchideo|objects were used and are examined.

As mentioned before, orchideo provides model-driven and aspect-oriented functionalities. The approaches of model-driven Software Development (MDSD) [8], especially in context of the model-driven Architecture (MDA) [9] of the Object Management Group (OMG) [10], and aspect-oriented programming (AOP) [11] are described in Section 2 and Section 3. The realization of MDSD and AOP in the orchideo|engine and in orchideo|objects is examined in Section 4. Here the modeling language of orchideo, the role of aspects in orchideo and the orchideo generator are examined. The application we developed with the help of the orchideo|suite, TeltowCar, is described in Section 5. TeltowCar is examined in scope of orchideo, especially the application model, the implemented aspects, and how the application model and aspects are used in the application to influence its execution. The problems we found during the development of TeltowCar are stated in Section 6 and a conclusion is drawn in Section 7.

2 Model-Driven Software Development

2.1 Motivation

Referring to [12] a *model* has to meet three criteria:

- Mapping: a model has to be based on an original (that not necessarily needs to exist yet).
- Reduction: a model does not show all properties of the modeled original. It simplifies it in some way.
- Pragmatic: a model should be useful instead of the original for a specific purpose.

So a model is something that provides a higher level of abstraction than the original through leaving out unnecessary information and should make the original (or a specific part of the original) easier to understand.

During the development of a software system there often exist several models. There can be models, for example, created on base of requirements or models of the system architecture. There are even different situations models are created for. Some examples are communication within the development team and with domain experts, for documentation and for a better understanding of a problem. If there is a model of the system architecture, developers have to write source code based on this model. Doing this, a developer has to do a lot of similar and easy tasks like creating classes and method skeletons, and properties for classes and methods modeled for example in a class diagram very often. These tasks take time and are error-prone.

2.2 Model-Driven Software Development

The goal of model-driven software development is to automate these tasks by generating source code from models. Not only code skeletons are generated, but application logic as well.

Another goal is to handle software complexity due to modularization. This goal builds on the assumption that models are better to understand than source code because they provide a higher level of abstraction than third generation languages and support the communication inside the development team and with domain experts.

In MDSD software is generated partially or completely from models. Data structures and application logic are specified in models. A model-driven framework has often the following structure: It contains one or more domain specific languages (DSL) to create different kinds of models, a metamodel for each of the DSLs and one or more model transformations and/or a generator for generating source code from models. The structure of a MDSD framework is shown in Figure 1.

A *metamodel* can be seen as a model of a model and specifies the abstract syntax of a language (in case of MDSD this is a domain specific language). A metamodel defines what can be modeled, for example classes that can have methods and properties. An example language to define metamodels is the Meta Object Facility(MOF) [9] of the OMG.

A *domain specific language* is limited to a certain problem domain in contrast to a general purpose language. A DSL is based on a metamodel that defines its abstract syntax. The DSL defines the concrete syntax which can be graphical. Furthermore it determines how the elements defined in the metamodel can be modeled.

A model can be transformed into another model with the help of *transformation rules*. There are model transformations that have to be done manually. For example if a model on a high level of abstraction should be transformed into a lower level model, information has to be added. But if there is a data model, it can be transformed automatically into java code because only information about the platform (Java) has to be added. A goal of MDSD is to use automatic transformations. Transformations can be defined as rules that transform elements of one metamodel into elements of another metamodel.

Code generation means to generate source code from a model. This can be done with transformation rules by specifying how elements of the model's metamodel are transformed into elements of the source code's metamodel. Another way to generate source code is to use *templates*. A template specifies how a metamodel element should be transformed into source code. The generated code is completed by developers with manually written code. When the model is adjusted, there will be the need to generate code from it again. If the generated code was adjusted by developers, this would be problematic. Hence there is the need to somehow distinguish between the generated and the manually written code and they should be separated from each other. There are different patterns

for separating the generated from the manually written code, for example the generation gap pattern [13].

The application code which contains the generated source code and the manually written source code runs on a platform. Referring to [9] a "*platform* is a set of subsystems and technologies that provide a coherent set of functionality through interfaces and specified usage patterns, which any application supported by that platform can use without concern for the details of how the functionality provided by the platform is implemented".

In Figure 1 the relationships between the terms explained above are shown. A model is created using a DSL. One or more DSLs can be based on one metamodel. As a model is an instance of the metamodel, it could have different representations using different DSLs all based on the same metamodel. The source code generator uses templates to generate source code from a model. The generated code has to be extended with manually written code by developers. The generated and the manually written code represent the application code. The application code runs on a platform.

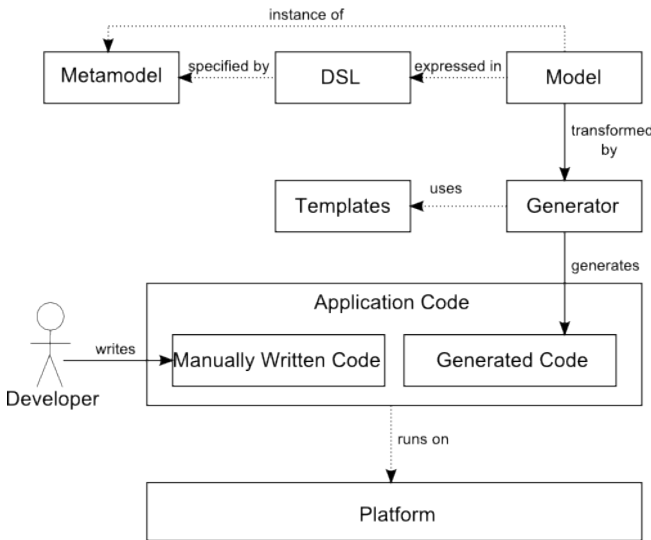

Fig. 1: The Structure of a MDSD Framework (on basis of [14])

2.3 Model-Driven-Architecture

The MDA is a set of standards and concepts created by the Object Management Group on the basis of MDSD. In context of the MDA there are three kinds of models:

- *Computation Independent Models(CIM)*: A CIM describes the system from a viewpoint that focuses on the requirements of the system and its environment. It is mainly created in cooperation with domain experts who have no knowledge about how the system would be realized, but have knowledge about the specific problem domain. This model shows no technical or structural details of the system and can be used to communicate with domain experts.
- *Platform Independent Models(PIM)*: A PIM describes the system from a viewpoint that hides information which are necessary for a specific platform. This model contains information that are similar for different platforms. So a PIM does not contain any implementation details.
- *Platform Specific Models(PSM)*: A PSM combines the PIM with details about a specific platform. For example source code is a PSM.

Using these three kinds of models in development is useful because they hide detailed information when it is not needed or when it is not available. Each of them provides a view on the system on a different level of abstraction. The transformations from CIM to PIM, from PIM to PSM, and from a PSM to source code, is illustrated in Figure 2. In the development process these three kinds of models are supposed to be used as follows.

The first model to be specified is the CIM. It can be created in cooperation with a domain expert. A software architect transforms the CIM into a PIM. This transformation has to be done manually because computational information is added. The transformation from a PIM into a PSM can be done automatically. In this step only information about a specific platform is added. Source code is a PSM, so source code can be generated either from a PIM or from a PSM. There can be more than one model of each kind. For example there can be more than one CIM, each describing a different part of the system and providing a different view on the system. The MDA uses the Unified Modeling Language (UML) [15] to create models.

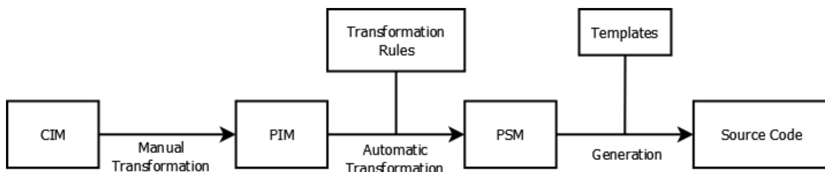

Fig. 2: An Overview of How CIM, PIM, PSM are Related

3 Aspect-oriented Programming

3.1 Motivation

Referring to the concept of separation of concerns [16] each concern that is relevant to a system should be handled separately. In software systems it is often not possible to encapsulate all different functionalities in separate modules. The concept of Object-Oriented Programming (OOP) provides constructs and functionality to encapsulate so called *core concerns* . In an ideal realization each object contains the implementation of a single concern. Core concerns are system requirements that contain central functionality and can be encapsulated in a module, e.g. a class in OOP. Functionality like multi-threading and error handling often cannot be encapsulated in one module, but affect a lot of modules and so are scattered through the whole system (or parts of it). The same code has to be executed in several modules (using for example OOP), which leads to duplicated code. There is no possibility to encapsulate crosscutting concerns in modules only with OOP. AOP provides constructs and functionality to solve this problem. It should entail better structured and modularized (and so probably better understandable) software and should support the reusability of crosscutting concerns by modularizing them. While MDSD supports OOP concepts with an easier way to achieve modularization and structured software, AOP enhances OOP with more possibilities and new techniques for modularization.

3.2 Concepts of Aspect-Oriented Programming

AOP [11] is a programming approach that provides constructs and functionalities to encapsulate so called *crosscutting concerns*. AOP is enhancing OOP with these constructs and functionalities.

A crosscutting concern is a specific requirement that crosses multiple other modules of a system that encapsulate the core concerns, for example as objects. A lot of examples for crosscutting concerns are stated by Ramnivas Laddad in [17], for example concurrency controls and transaction management. With AOP crosscutting concerns are encapsulated in *aspects* and are not scattered through the other modules. An aspect is the basic unit of AOP. Aspects are constructs that contain code that would be called in a large number of modules. Within an aspect it is also specified at which points this code has to be executed. Possible points in the execution of the system where an aspect can insert code are called *join points*. In AspectJ for example any identifiable execution point in a system is a join point.

A *pointcut* describes a set of join points [11]. It so defines where the advice code should be executed. A pointcut can decide whether to select a join point or not aided by the information in the context of a join point. An *advice* is the code that will be executed at a pointcut. There are three kind of advice:

- An *before advice* will be executed before the join point.
- An *after advice* will be executed after the join point.

– An *around advice* will be executed instead of the join point or parallel to the join point.

Pointcuts and advice are defined within an aspect. The pointcut selects the join points at which the code defined in the advice will be executed. The process of combining the aspects with the remaining code of the system is called *weaving*.

Weaving An aspect weaver is used to control the execution of the code specified in the aspects. A weaver is responsible for weaving the aspects with the rest of the systems code. Weaving could be done before compiling, after compiling, or at runtime.

In [18] weaving is classified in *invasive* and *noninvasive* weaving. Invasive weaving describes the weaving before compiling (source weaving) and after compiling (binary weaving). A source code weaver combines the aspect code with the rest of the source code before compiling, the woven source code is then compiled as a whole. A binary weaver weaves the aspect code and the base source code after compiling. While using invasive weaving it is not possible to consider dynamically loaded classes with the aspects. Noninvasive weaving means that the base program is not changed in order to enable aspect functionality. This kind of weaving is executed at the runtime of a system. The base program is not changed to enable aspect functionality. In this class of AOP frameworks the usage of a custom runtime environment is opposed to runtime interception as another possibility to enable the aspect functionality. When the weaving takes place at runtime the aspects can influence the systems behavior depending on runtime parameters and dynamically loaded classes.

4 Realization of MDSD and AOP in orchideo|engine and orchideo|objects

The orchideo|suite combines the two approaches MDSD and AOP. In the orchideo|engine aspects are created as models. Their base implementation is generated from these models. In orchideo|objects the data model of an application is created with the help of a DSL. orchideo|objects contains aspects that provide templates which are responsible for generating source code from these models. When developing business application with orchideo|objects and orchideo|engine different aspects implement different aspects of the software, for example: persistence, constraints and object management.

4.1 Aspects in orchideo

In orchideo there are two ways aspects can be used [7]:

– to control runtime behavior of an application
– to generate textual artifacts like source code

An aspect is defined as an aspect model. Two different diagrams provide views on this model. When developing an application with orchideo, aspects and their advice have to be configured within a session configuration. Aspects and advice can be toggled on and off in this configuration. An orchideo|engine session is created with a session configuration and so knows which aspects and advice it has to weave.

Aspects at orchideo|engine runtime Different from AspectJ [17], where a join point is any identifiable execution point in a system, the join points in orchideo are so called *actions*. An action is a programming construct which is defined within an aspect. Actions can be done and undone completely. So actions are similar to Java methods in some way, but can be, in contrast to Java methods, undone. In fact, for each orchideo action two Java methods exist in the aspects class: a `doAction()` method and an `undoAction()` method. An action can have `in`, `out`, and `return` parameters. `in` and `return` parameters are similar to method parameters and return types of Java methods. `out` parameter are used, for example for undoing an action. So if an action changes parameters, the old values can be stored in `out` parameters and when undoing the action these values can be reset by accessing the `out` parameters. An action can be called within applications by clients.

Besides actions so called *services* can be defined within an aspect. A service provides functionality to clients and can, like an action, be called by clients. In contrast to an action it cannot be undone, must not define any `out` parameters and is not a join point. The main functionality of a service is to provide some aspect specific functionality. Services are mainly used by clients for accessing information.

Advice differ from AspectJ advice, where the term advice names the code to be executed at a specific join point that was selected by a pointcut. In orchideo an advice is a construct that defines its pointcut action and the action or actions to be woven at a pointcut. The pointcut action has not necessarily to be a pointcut for the advice. The pointcut can be defined in more detail in the advice's `apply()` method (which is located in the class for the advice generated from the model) using the context information of the pointcut action. So the real pointcut is either the sum of all actions of the type of the pointcut actions or a subset of them.

An orchideo *aspect* is a construct, that defines actions, services and advice. It can also define templates, which is examined later. As only actions are join points, only actions can be woven in orchideo.

There are before, after and around advice:

- Actions woven by a *before advice* are executed before the join point action will be executed.
- Actions woven by an *after advice* are woven after the join point action was executed.
- In the context of an *around advice* exact one action is executed instead of the join point action. If there are more than one actions advised by one advice,

only the first one in the queue of actions will be executed. If there are more than one around advice, beginning with the first advice every advice has to call `proceed` when following advice should also be executed.

An advice can define *predecessors* and *successors*. With them it is possible to define the overall advice precedence. There is a special advice, the `any` advice, in orchideo. It can be defined as predecessor or successor of an advice in orchideo. Defining it as predecessor means, that the advice is executed after any other advice which has the same pointcut action. Defining the any advice as successor of an advice means that the advice is executed before any other advice that has the same pointcut action. There is no defined order in which advice are called when they have the same pointcut when not defining predecessors and successors. So the precedence of the advice can vary every time the system is executed.

Whenever an action is executed (or is about to be executed) the orchideo|engine checks whether this action is declared as pointcut action of any advice. For advice, that declare this action as pointcut, the engine weaves the action or actions declared by the advice.

As there is an `any` advice, there is also an `any` action in orchideo. If an advice declares this action as its pointcut, the advice's `apply()` method is executed every time any action is executed or about to be executed. Using this action as woven action enables the advice to weave any type of action.

An example of an aspect is shown in Figure 3 with an aspect diagram. We can see the aspect diagram of the `InitializationAspect` (5.4) of the application TeltowCar (5). In the example an after advice, the `invokeInitialize` advice is shown. the advice's pointcut action is the `CreateObject` action defined by the `ObjectAspect` [7] of orchideo|objects. The woven action of the `invokeInitialize` advice is defined within the `InitializationAspect`. The woven action is the `Initialize` action. As shown in Figure 3 the actions have parameters. The `CreateObject` action has an `in` parameter `classifier` of the type `Classifier` and cardinality `[1]` and a **`return`** parameter `createdObject` with cardinality `[0..1]` of the type `Any` (which is `java.lang.Object`). The `InvokeInitialize` advice defines the `RegisterCreatedObject` after advice as predecessor, which defines the same pointcut action. Every time the `CreateObject` action will be executed and both, the `ObjectAspect` and the `InitializationAspect` are activated within the active session configuration, the orchideo|engine weaves all advice that define the `CreateObject` action as pointcut action. It is guaranteed that the `RegisterCreatedObject` advice is executed before the `invokeInitialize` advice.

Aspect Implementation The orchideo generator generates the following artifacts important to the developer:

- an `AspectImpl` class for each aspect
- a method in this class for each service
- two methods for each action (`do` and `undo`)
- an `AdviceImpl` for each modeled advice

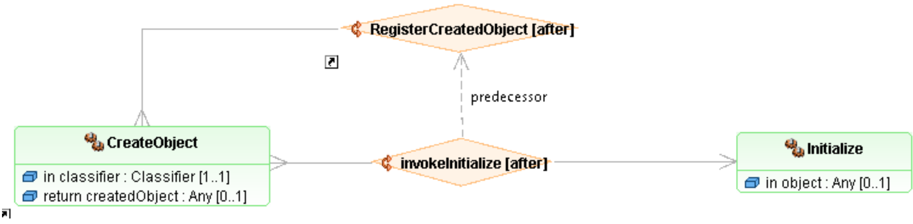

Fig. 3: The `InitializationAspect` aspect diagram

```
public InternalAction[] apply(InternalAction action) {
  assert action instanceof CreateObjectAction;
  // The newly created object
  Object object = action.getReturnValue();
  if (object == null) {
    return EMPTY_ACTION_ARRAY;
  } else {
    InternalAction initializeEmpty = new
        InitializeActionImpl(object);
    return new InternalAction[] { initializeEmpty };
  }
}
```

Fig. 4: `apply()` Method of an Advice

We describe the orchideo generator in 4.5. Because an action can be done and undone, for each action a `do` and an `undo` method have to be implemented in the `AspectImpl` class. Services are implemented as methods of the `AspectImpl` class. So for each modeled service a method declaration in the `AspectImpl` will be generated. For each modeled advice an `AdviceImpl` class is generated by the orchideo|engine. An `AdviceImpl` class has the method `apply()`. This method has to return an array of `InternalActions`. The `apply()` method of an advice is called when an action the advice declares as its pointcut action is about to be executed or actually executed. Within this method the developer can check whether the action meets the requirements of the advice. In Figure 4 the `apply()` method of an advice is shown. It is checked, whether the action, in whichs context the advices `apply()` method was called, is an action of the type `CreateObject` and if there is an object. The advice returns an `InitializeAction` with the object that was currently created by the `CreateObjectAction` and no action, if there was no object.

Aspects at Generation Time In orchideo|objects parts of the applications source code are generated with the help of aspects. Aspects can define templates

for code generation. There are two different kinds of templates in orchideo. On the one hand there are root templates. A root template takes a model element as input and generates artifacts out of it. For example the `ObjectAspect` specifies root templates for generating a Java package for each modeled package, special interfaces, and Java classes for each modeled class using the delegate pattern as described in Section 4.5. The other kind of templates are aspect templates [19]. With them aspect-orientation on template level is provided in orchideo. An aspect template inserts text fragments during the generation process. Here, the templates (root templates) serve as join points. For example the `PropertyAspect` uses aspect templates to generate a field for all properties of modeled classes and getters and setters for these properties in Java classes generated by the `ObjectAspect`.

Modeling Aspects An aspect in orchideo is created with the help of a modeling language. In orchideo there are two kinds of diagrams, each of them giving another view on the modeled aspect: aspect diagrams and container diagrams. The *container diagram* can give an overview of the actions, services and advice defined by the aspects in this package and how these aspects are related with each other and with other aspects. This kind of diagram gives no information about the advice kind, pointcut actions of the advice, predecessors and successors of the advice and the actions woven by the advice. An example of a container diagram is shown in Figure 5. The `InitializationAspect` defines one action, the `Initialize` action and one advice, the `invokeInitialize` advice. When there where more than one aspects defined in this package, each aspect would be shown as such a container and the relations between the aspects too (generalization and dependencies).

*Aspect diagrams*can show information about advice of an aspect. The advice kind, the woven action, the pointcut action and the predecessors and successors defined by advice can be examined in aspect diagrams. Both diagram types are based on the same model of the aspects and if one of them is changed by the developer the other diagram will be automatically adjusted. In that way it is possible to adjust the model in its tree view, in the aspect diagram and in the container diagram. From the aspect model the base implementation of an aspect is generated. The developer then has to implement the logic of the advice, actions and services of the aspect.

4.2 orchideo as Aspect-Oriented Framework

Following the categorization in Section 3.2 the orchideo AOP framework is a *noninvasive system* that uses a *custom runtime environment*. The orchideo AOP implementation performs the following AOP activities [18]:

– *dynamic aspect selection*: The aspects to be used in an orchideo session and their precedence are determined at runtime upon session creation, and depend on the provided session configuration.

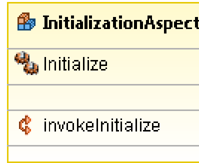

Fig. 5: Container Diagram of the `InitializationAspect`

- *aspect instantiation*: Scoping is achieved by the usage of different sessions in one application concurrently. As each session can be initialized with a different session configuration, different aspect configurations are possible for different sessions.
- *advice execution*: As this activity is required to qualify as an AOP framework, it is performed by the framework; although in orchideo it is split in advice and action execution, respectively.
- *bookkeeping*: The orchideo|engine keeps track of various additional information. The most obvious information is the execution history storing a subset of all execution context objects used so far. An execution context encapsulates the execution of actions. It is initialized with one action called by the client application and recursively weaves all advice into itself, with regard to the session configuration and advice precedence. This provides the possibility to undo previously performed actions.

4.3 Models in orchideo|engine and orchideo|objects

In orchideo there are the aspect models which are created with the functionality of the orchideo|engine and application models which are created with orchideo|objects. As mentioned in Section 4.1 there are two different types of diagrams based on one aspect model: the aspect diagram and the container diagram. Both of them provide a different view on the aspect.

Application models in orchideo|objects Application models can be created with the help of a DSL provided by orchideo|objects. Each model element has properties that are not necessarily shown in the diagram. The main elements that can be modeled with the help of the DSL provided by orchideo|objects are:

- *Classifiers*: Classes, interfaces and data types are the classifiers that can be modeled. Classes can be modeled as abstract. Additionally there is the possibility to use external interfaces and external data types in the model. Classifiers can have the following child nodes:

- *Attributes*[1]: Attributes can be represented in diagrams with a name, a type and a cardinality. An attribute of a classifier cannot be modeled as private or public. Classes, and data types can define attributes.
- *Operations*: Operations can be modeled as abstract, but not as static. Like attributes they cannot be modeled as public or private. All classifiers can define operations.
- *Constraints*: There are different types of constraints in orchideo|objects. These are examined in Section 4.4. Classes, and datatypes can define constraints.

Relationships between classifiers can be modeled graphically with the help of the DSL of orchideo|objects, but are represented in the metamodel as properties of attributes or of classifiers. The relationships, that can be modeled with orchideo|objects are:

- *Associations*: Associations are represented in the metamodel as properties of classes and data types. There is the possibility to model bidirectional associations, these are represented by an opposite property of an attribute.
- *Aggregations*: Aggregations are represented by the `Aggregation` property of an attribute.
- *Realization*: Realization is represented as a property of classes and data types. These classifiers have a property `Realized Interfaces` which contains all interfaces the classifier realizes.
- *Generalization*: Modeling inheritance is provided by orchideo|objects to all classifiers. Generalization is also represented as a property of classifiers (`Super Types`). Modeling multiple inheritance is not possible.

Besides these main elements orchideo|objects provides the functionality to model packages, enumerations, and state-machines. The orchideo|objects DSL for creating application models is based on the class diagrams of the UML 2.0, but in a very simplified form of them.

4.4 Constraints and Derived Attributes

With the help of some Aspects of orchideo|objects it is possible to use constraints, derived attributes, persistence and Hibernate support. These aspects influence the modeling. Constraints and derived attributes can be modeled, but when the `ConstraintAspect` and the `DerivedAspect` are not activated in the session configuration there would be no impact during the execution of the system or during the code generation. For example when modeling a constraint `fullname` the developer can model this constraint and specify its type as *OCL* (explained later in this section). Then he can enter an OCL expression, but there would be no effect at runtime when the `ConstraintAspect` is not activated. When he specifies the type as *Java*, the `check` method in the `DelegateImpl` class will not be generated, when the `ConstraintAspect` is not activated.

[1] With attributes we actually mean properties of classifiers. We call them attributes to distinguish between those properties and properties of model elements.

Constraints Constraints for attributes of orchideo objects can be created in the application model. An orchideo object is an object that is an instance of a class modeled in the application model. It is possible to create constraints for attributes of different classifiers. These are classes and data types. It is not possible to use constraints in the context of external data types or enumerations. There are three kinds of constraints in orchideo: cardinality constraints, Java constraints and OCL constraints. Constraints could be specified depending on their type with Java code or the Object Constraint Language (OCL) [20].

Cardinality Constraints Cardinality constraints are generated automatically for every attribute of a class or a data type with the help of the `ConstraintAspect`. They are used by orchideo|objects to ensure that the upper and lower cardinality bounds are adhered at runtime. For example when modeling an attribute `name:String[1]` a cardinality constraint will be generated automatically to ensure that the concerning classifier has exact one name.

Java Constraints When defining the constraint's type as *Java*, explicit triggers must be defined by the developer. This is also possible in the properties view. A `check()` method declaration for the constraint is generated in the `DelegateImpl` class of the classifier the constraint is defined in. The developer has to implement the logic of the constraint in this method. For example, when a constraint of the type *Java* with the name `CurrentGreaterOrEqual` and with the explicit triggers `firstRegistrationDate` and `currentRegristrationDate` was modeled in the classifier `Vehicle`, the generator will generate a method `checkGreaterOrEqual()` as shown in Figure 6.

OCL Constraints If the type of the created constraint is *OCL*, it is possible to write the OCL expression in an editor in the properties view. Different to Java constraints the relevant properties for an OCL constraint are extracted from the OCL expression and so have not to be defined explicitly. The example from above is shown in Figure 7. The advantage of expressing constraints in OCL (beside being shorter in this case) is that the developer does not need to change the source code but can define the expression in the properties view of the application model.

Derived Attributes There is also the possibility to model derived attributes in orchideo— attributes that are computed on the basis of other attribute. Similar to constraints the computation of derived attributes can be done with the help of OCL expressions or can be implemented in Java code. As concerning constraints there is an editor for writing the OCL expression for the derived attribute. When specifying the type of the constraint as Java, a `compute()` method declaration for the derived attribute will be generated with the help of the `DerivedAspect` in the `DelegateImpl` class of the derived attributes classifier.

```
public boolean checkGreaterOrEqual(Vehicle contextObject) {
  Calendar first = contextObject.getFirstRegristrationDate();
  Calendar current =
      contextObject.getCurrentRegristrationDate();

  if (first != null && current != null)
    return first.equals(current) || first.before(current);
  else
    return true;
  }
```

Fig. 6: The `check()` method for a Java constraint

```
context Vehicle inv:
  firstRegistrationDate <= currentRegistrationDate
```

Fig. 7: An OCL Expression for an OCL Constraint

4.5 Source Code Generation in orchideo|engine and orchideo|objects

The generator is responsible for generating code and other textual artifacts out of models. The generator uses builders and natures [21] to run automatically and integrate in the Eclipse IDE. So the generator can run each time an orchideo project is built. The orchideo generator is generic, it just knows, with the help of orchideo aspects, what code it will generate and therefore needs a configuration (exact the same configuration as the orchideo|engine session configuration). The generator produces (dependent on aspects) different types of output. These are managed in so called outlets. An outlet is a symbolic name for a directory, for example `src` or `gen_src`. An aspect has to declare which outlet it is going to use for generating artifacts. Some of these outlets are cleared before the generation process, for example the `gen_src` outlet of an orchideo project. In that way some files are generated entirely new every time the generator runs. Other files are generated only once, for example the delegate implementation classes, which must be changed by the developer.

Separation of Generated and Non-Generated Source Code in orchideo
In orchideo the generated Java files are written in the outlets `src` and `gen_src`. The directory `gen_src` is cleared and so the files in this directory are generated new every time the generator runs. The files in the `src` directory are generated only once, when the class was modeled the first time. They have to be filled and maintained by the application developer. In the generated implementation

classes (that follow the scheme `ClassnameDelegate.java`) the Delegate Pattern [22] is used. Following the Delegate Pattern an object implements an outward interface but forwards method invocations to so called delegates. The Delegate Pattern in orchideo is basically used by the implementation classes generated by the `ObjectAspect`. For example if a class `Customer` is created in the model, the generator will generate an public interface `Customer` that can be accessed from within the application code, a Java implementation class `CustomerImpl`, an empty delegate interface `CustomerDelegate` and an empty delegate implementation class `CustomerDelegateImpl` which implements the delegate interface and contains the logic the developer has to add.

5 Using orchideo|engine and orchideo|objects to Develop the Application TeltowCar

For evaluating the orchideo framework we developed the application TeltowCar. TeltowCar is an application for managing customers and vehicles of a car repair shop. Customers and vehicles are stored in a database and the application provides functionality to add, remove, and alter the data of a customer or a vehicle, and to assign vehicles to customers.

5.1 Requirements

We will give a short overview of the requirements here:

- *Customer data*: A customer should have a first name, a last name and a title. Additionally a customer can have several addresses and phone numbers, an ID and a notes field which can be used for custom notes concerning this customer. One customer can be associated with several vehicles over roles like "owner" or "driver". Roles should be entered by the user, there should not be default roles. It should be possible to add, edit and delete customers.
- *Vehicle data*: As customers vehicles should be possible to add, change and remove. It should be possible to enter all fields that exist in the drivers license and again a custom memo field should exist. A vehicle can be associated with several customers over person roles. So for example a vehicle can be associated with one customer that is the owner of the vehicle and with two other customers who are drivers of the vehicle. It should be possible to create a vehicle when editing a customer so that the vehicle will automatically be associated with this customer.
- *Security*: Specified operations should only be possible to execute when entering a password, for example deleting and editing customers and vehicles. When the password was entered once, it should not be necessary to enter the password for a specified time — the time-out of the password.
- *Settings*: It should be possible to change settings like the time-out of the password and the password itself.
- *Searching*: There should be the possibility to search data sets via a search field. It should be possible to search for vehicles and for customers and the search should consider specified fields of customers and vehicles.

5.2 Infrastructure and Environment

The orchideo|suite builds on the Eclipse IDE. So when developing an orchideo application there has to be an orchideo project for the application model and a Eclipse Rich Client Platform (RCP) application project that uses the application model of the orchideo project. TeltowCar consists of an orchideo project named `TeltowCar`, three orchideo aspects (`SequentialIdAspect`, `InitializationAspect`, and `RefreshApplicationAspect`) and a Java project called `TeltowCarApplication`. In the project `TeltowCar` the application model is defined an the modeled classes are implemented. The project `TeltowCarApplication` contains the application logic and uses the application model, orchideo|objects aspects and the three aspects we implemented. For testing purposes we created two test projects, one for the application project and one for the orchideo application model project. The data sets are stored in a database. The database connection and management is done with the help of the `HibernateAspect`. Its usage is explained in Section 5.5.

5.3 The Application Model

In Figure 8 a diagram of the application model of TeltowCar is shown. This diagram shows not all entities and relationships of the application model, but provides a view on the main classes and their relationships. The main classes of TeltowCar are `Customer` and `Vehicle`. They are related with help of the class `PersonRole`. One customer can have no, one or more vehicles and a vehicle can have no, one or more customers associated with it. We used the class `PersonRole` for searching purposes. When searching for a specific tuple of customer and vehicle `PersonRole` provides tuples that match one customer to exactly one vehicle.

We created and changed the application model of TeltowCar mainly in one diagram and in the tree view of the model. In Figure 9 the tree view of the application model is shown. All classes, interfaces, and the external types used can be examined here. The `Preference` class is used for settings like passwords. The external interfaces `IIninialize` and `ICustomIdModel` are defined in the aspects we implemented (see Section 5.4). The other classes are used for storing information about customers or vehicles.

Searchable Map. orchideo|objects does not provide the possibility to model static methods. TeltowCar had the requirement to search through customers and vehicles. When searching a vehicle, it is necessary to search objects that are referenced by a vehicle, in fact the `PersonRole`. We do not want to search in all associations and all properties of the vehicle. If there would be a possibility to model static methods, we could have given the vehicle a static method that returns the fields to search in. To solve this problem, we created a separate class called `SearchableMap` and an interface called `IDescribable`. The class `SearchableMap` is located in the `src` folder of the application model project in an extra package. This class provides the functionality to define the properties it should be possible to search in. To indicate which classes are concerned, the

`IDescribableInterface` is used in the model. All classes, that should be searchable, have to implement this interface.

Constraints. Besides the cardinality constraints that are generated automatically, we used just one constraint in TeltowCar, that is the `CurrentRegistrationDateGreaterOrEqualFirstRegistrationDate` constraint of the type *Java*. It is the constraint that was discussed in 4.4. The vehicle has some attributes that have a cardinality of one or higher. Object are initialized in orchideo|objects with all properties having the value `null`. So after creating such an object, these cardinality constraints are not met. For solving that problem we wrote the `InitializationAspect` that is described in Section (5.4).

Derived Attribute. For the attribute `fullname` of `Customer` we used a derived attribute. The `fullname` consists of the `firstname` and the `lastname` of a `Customer`. We used a derived attribute of the type *Java*. In Figure 10 the `compute()` method of `fullname` is shown. The method computes the full name of a customer depending on whether the first or the last name exists or not.

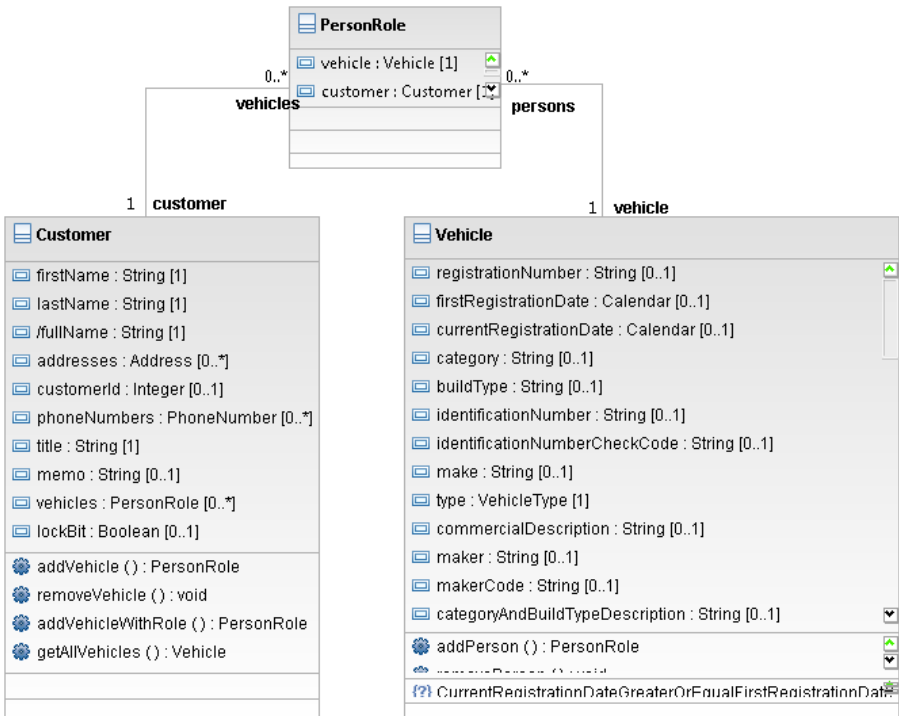

Fig. 8: The Application Model Diagram of TeltowCar

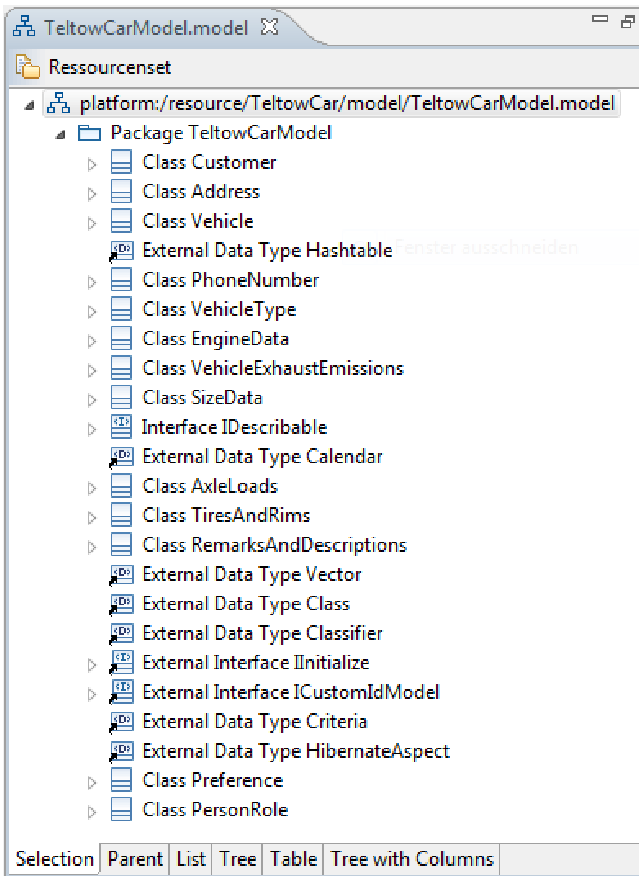

Fig. 9: The Application Model of TeltowCar

5.4 Implemented Aspects

We implemented three aspects in the context of TeltowCar. These are:

- `SequentialIdAspect`
- `InitializationAspect`
- `RefreshApplicationAspect`

The SequentialIdAspect. One requirement on a customer was, that each customer should have an ID and that these IDs should be set automatically. The IDs should be sequential. Assuming that the customer was not the only entity which needs an ID (for example bills need sequential invoice numbers) we wrote an orchideo aspect to calculate and set the IDs.

```
  public String computeFullNameValue(Customer contextObject)
     {
    if (contextObject.getFirstName() == null &&
        contextObject.getLastName() == null)
      return "";
    else if (contextObject.getFirstName() == null)
      return contextObject.getLastName();
    else if (contextObject.getLastName() == null)
      return contextObject.getFirstName();
    else
      return contextObject.getLastName() + ", " +
          contextObject.getFirstName();
  }
```

Fig. 10: `compute()` Method of the Derived Attribute `fullname`

```
private void delete(Object o) {
  closeEditorFor(o);
  Activator.getDefault().getObjectAspect().destroyObject(o);
  Activator.getDefault().getHibernateAspect().commit();
}
```

Fig. 11: The `delete()` Method for Deleting an Object from the Database

The aspects advice `InvokeSequentialId` works with an interface `ICustomIdModel`. A class implementing this interface will be considered by the advice. This is ensured in the `apply()` method of the advice that is shown in Figure (13).

`action.getSession().getAspect(HibernateAspect.class).getNewObjects` gives us all objects that should be committed to the database and are not already in the database. `getNewObjects` is a service provided by the `HibernateAspect`. A service does not change the state of the orchideo|engine, any objects or any aspect. Hence it is allowed to use services in the `apply()` methods of an advice.

From these objects we collect the objects that are of the type `ICustomIdModel`. When there are no objects or no objects of the type `ICustomIdModel` the method will return an `EMPTY_ACTION_ARRAY`, what means, no action is executed by this advice because the pointcut requirements are not met by the pointcut action of the advice and the context of this action. When there are objects of the type `ICustomIdModel` the advice creates a new `GenerateSequentialIdAction` with the object found. Then this action (defined in this aspect) is executed. In the case of

our application only the `Customer` class implements this interface. It is reasonable not to set the ID until the Customer was committed to the database. This is because when setting it earlier (before saving the customer) there would be more effort to assure that the ID is sequential. The `SequentialIdAspect` defines one advice, the `InvokeGenerateSequentialId` advice. The advice is shown in Figure 12. Objects are committed to the database with the help of the `HibernateAspect` of orchideo|objects. We defined the `commit` action of the `HibernateAspect` as the pointcut action of the advice. For handling the calculation of the IDs we created an own action, `GenerateSequentialId`. For each object without an ID this action searches in the database for all objects of the same type, that already have an ID. These objects are ordered to get the last ID given to such an object. The next ID is given to the new object.

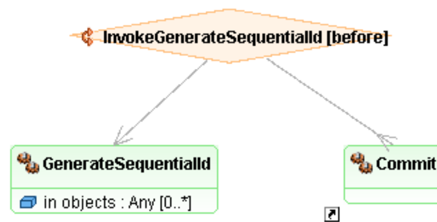

Fig. 12: The `SequentialIdAspect`

The InitializationAspect. As mentioned in 4.4, during the development of TeltowCar we encountered a problem with the initialization of objects, that are defined in the application model. When defining an attribute of a class with cardinality one or more, the generated cardinality constraint is validated after creating an instance of this class with `createObject()`. This is because the `ObjectAspect` of orchideo|objects initializes attributes with `null`. Hence every time such an object is created, it is necessary to initialize the concerned attributes manually.

This problem could be solved using an aspect that works every time after such an object is created. For that purpose we wrote the `InitializationAspect`. The aspect works with an interface `IInitialize`. All classes, that implement this interface, are considered by the `InitializationAspect`. The interface only tells them to implement an `initialize()` method. As shown in Figure 3 it defines the advice `InvokeInitialize` that is an after advice. The advices pointcut action is the `CreateObject` action of the object aspect which is responsible for creating objects. The aspect defines the action `Initialize`, which is woven by the advice. The advice defines the `RegisterCreatedObject` advice as predecessor. This is because an object has to be registered after its creation and before anyone can manipulate this object. The `apply()` method of the advice is shown in 4. The

```
public InternalAction[] apply(InternalAction action) {
  assert action instanceof CommitAction;
  Collection<?> objects = ((CommitAction)
      action).getSession()
          .getAspect(HibernateAspect.class).getNewObjects();
  if (objects == null) return EMPTY_ACTION_ARRAY;
  Vector<ICustomIdModel> new_objects = new
          Vector<ICustomIdModel>();
  for (Object o : objects) {
    if (o instanceof ICustomIdModel) {
      new_objects.add((ICustomIdModel) o);
    }
  }
  if (!new_objects.isEmpty()) {
    InternalAction generateIdAction = new
            GenerateSequentialIdActionImpl(new_objects);
    return new InternalAction[] { generateIdAction };
  } else {
    return EMPTY_ACTION_ARRAY;
  }
}
```

Fig. 13: `apply()` Method of the `InvokeGenerateSequentialId` Advice

advice just checks whether there is an object or not. The action `Initialize` calls the `initialize()` method of the newly created object which has to be of the type `IInitialize`.

The `InitializationAspect` in particular can be reused by other applications, because the problem of `null` initialized properties that have cardinalities of one or more is not only a problem of TeltowCar.

The RefreshApplicationAspect. When a customer or a vehicle is saved in the database, the search view should update and it should be visible in the editor, that the object was saved. The save button should no longer be enable, but the edit button should be enabled. An objecft was saved in the database after the `commit` action of the `HibernateAspect` of orchideo|objects was executed. The editor and the search view should only be updated, if the `commit` action was successfully executed. For updating the editor and the search view we implemented the `RefreshApplicationAspect`.

As shown in Figure 14, the `RefreshApplicationAspect` defines the after advice `invokeRefreshApplication` and two actions: the `RefreshEditors` action and the `RefreshViews` action. The advices pointcut action is the `commit` action and it weaves both the `RefreshEditors` and the `RefreshViewAction`. The `IRefreshable` interface is implemented by all editors and views that should be refreshable. It

tells them to implement the method `refreshView()`. The advice does not check anything in its `apply()` method, it just returns both actions, the `RefreshViews` and the `RefreshEditors` action. The `doRefreshEditors` is shown in Figure 15. The action mainly calls `refreshView()` on all editors that implement the `IRefreshable` interface and that are registered. The `RefreshViews` action does the same with views.

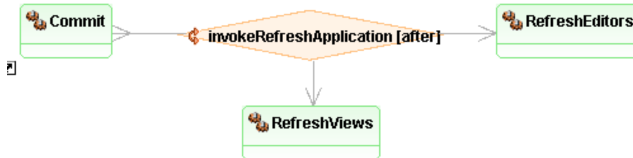

Fig. 14: The `RefreshApplicationAspect`

5.5 Our Application

The application project is an Eclipse RCP application. To develop the graphical user interface we used the Eclipse Standard Widget Toolkit (SWT) [23]. All of the aspects in orchideo|objects and their advice are enabled in the application. Furthermore the `RefreshApplicationAspect` described in Section 5.4 works at runtime of the application.

The Running Application. A screenshot of the TeltowCar application is shown in Figure 16. We provide an overview list of customers or vehicles ①. The items in the list can be expanded, so that the vehicles associated with a customer or the customers associated with a vehicle are shown. The overview can be switched from customers to vehicles and back with the button above the list ②. When double clicking on a list item, it will open in the editor area on the left ③. Open items are shown as tabs. The selected item can be examined in detail here. When the item is a customer (like in the example) the main information is shown in the top, above the tab. These are the title, the first name and the last name of the customer. In the area below, three different tabs are displayed ④. The first tab contains contact information, like addresses and phone numbers. The second tab provides an overview of the customer's vehicles. For each vehicle the role of the customer for this vehicle, the registration number, maker and an identifier are shown here ⑤. From within the vehicles tab of a customer, a new vehicle (that will automatically be associated with this customer) can be created. The third tab is intended to be used for custom notes about a customer. In the right top corner of this area the buttons for saving, editing and canceling (which are active or not active, depending on whether they are available or not)

```
public void doRefreshEditorsAction(RefreshEditorsActionImpl
    action) {
  if (PlatformUI.isWorkbenchRunning()) {
    IEditorReference[] references =
        PlatformUI.getWorkbench().getActiveWorkbenchWindow()
                                                        .getActivePage()
        .getEditorReferences();
    for (IEditorReference ref : references) {
      IEditorPart editor = ref.getEditor(false);
      if (editor != null && editor instanceof IRefreshable) {
        ((IRefreshable) editor).refreshView();
      }
    }
  }
}
```

Fig. 15: The `do()` Method for the `RefreshEditors` Action

are located ⑥. When opening a vehicle, an editor for the vehicle which provides all information of a drivers license is shown. The overview list on the right side provides the functionality to search for customers or vehicles ⑦. In this example, we are searching for "Ma" and a list of customer which have these letters in their names is given.

In the `Activator` of the application project an orchideo session has to be created with a session configuration. This has to be done with `Engine.INSTANCE.createSession("TeltowCarModelConfiguration")` in the case of TeltowCar. "TeltowCarModelConfiguration" is the name of the session configuration. Our session configuration has all aspects and advice enabled. Using the orchideo|engine session the access on orchideo aspects is managed. From within the application aspects (here the `ObjectAspect`) can be accessed with `session.getAspect(ObjectAspect.class)`. So every time a developer has to invoke an action of an orchideo aspect, he has to write (again in case of the `ObjectAsepct`) `Activator.getDefault().getSession.getAspect(ObjectAspect.class)`. So we decided to implement `getter` methods for the aspects we use more often in the `Activator`. These are the `ObjectAspect`, the `HibernateAspect` and the `ConstraintAspect`.

Persistence with the `HibernateAspect`. The `HibernateAspect` provides a bridge between orchideo objects and therefore it uses the functionality of Hibernate [24]. When a class is modeled as persistent, it will be considered by the `HibernateAspect`. Instances of such classes can be stored to and loaded from a relational database with the help of this aspect. To use the hibernate aspect, a relational database and a `hibernate.cfg.xml` have to be set up. This configu-

ration file contains information about the database connection, for example the location of the database (host and port), the driver class (for example for mysql) and the username and password. The `HibernateAspect` will generate a Hibernate XML (Extensible Markup Language) [25] mapping file for every class that is modeled as persistent in the application model. These mapping files have to be registered in the `hibernate.cfg.xml`

Instances of orchideo objects that are modeled as persistent then have to be created with the `createObject` action of the `ObjectAspect` and deleted with the `destroyObject` action. The developer has to execute the `commit` action of the `HibernateAspect` to commit the new, changed or deleted objects to the database. For example for our `DeleteAction` we use the method in Figure 11 to delete objects from the database. When executing the `destroyObject` action an advice of the `HibernateAspect` (the `delete` advice) is executed. It ensures internally, that the object is considered as deleted object the next time the `commit` action of the `HibernateAspect` is executed. Deleted, dirty and new objects are managed in lists by the `HibernateAspect` internally.

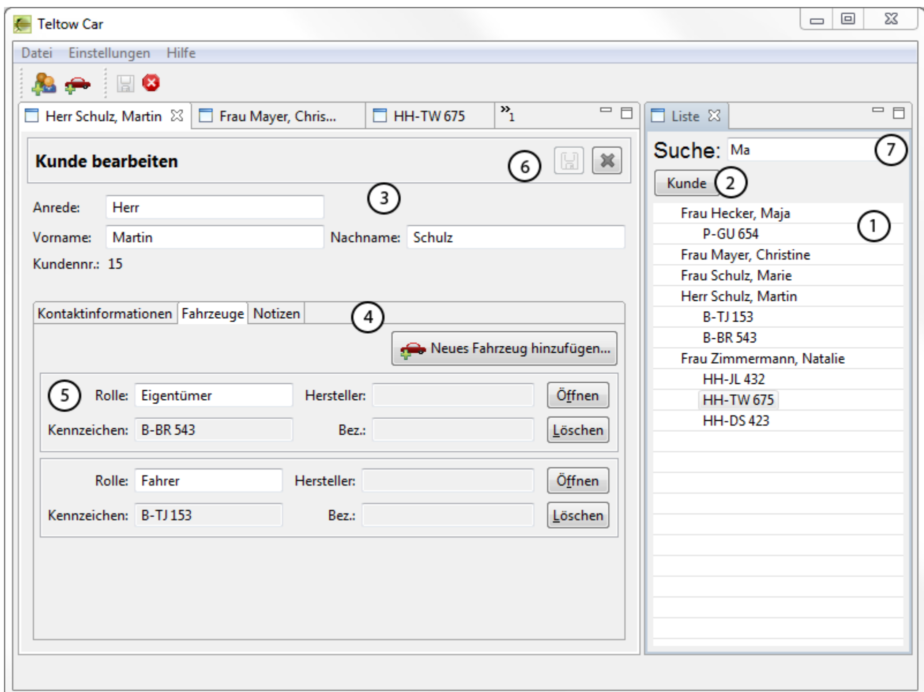

Fig. 16: A Screenshot of TeltowCar

5.6 Testing

For testing purpose in TeltowCar we used JUnit [26]. JUnit provides functionality to define test cases with assertions, that can be bundled in test suites and the possibility to use launch configurations for test running. We created one test project for the TeltowCar application model project, and one for the TeltowCar application. Both JUnit projects contain a launch configuration, so that they can be executed on a server [?]. For showing the status of the JUnit Tests we developed a traffic light plug-in [?]. For the tests we use a test database. Hence we needed a new Hibernate configuration. As we are testing orchideo projects (and for that purpose want to use aspects) we have to create an orchideo session in the set up of the tests. For testing purposes it would be suitable to use another session configuration than for the actual application.

After setting up such a testing environment, it is possible to write normal JUnit tests for orchideo projects. A test we wrote for the application model project is shown in Figure 17. A customer is committed to the database and the test asserts, that this customer is found in the database. We use the `ObjectAspect` to create a new customer (this is done in `createCustomer`), and the **`HibernateAspect`** to commit the customer to the database and to find the customer there after committing (this is done in **`findInDatabase`**).

6 Lessons Learned

The development of TeltowCar was a good way to become acquainted with the orchideo framework. We found that orchideo|objects provides good functionality for MDSD. Because the DSL of orchideo|objects is a simplified form of UML 2.0 class diagrams, it is good to understand. Modeling the application model feels natural after a short time. Working in the different views for the application model (the application diagram and the tree view of the model) and in the source code is easy. The aspect model and the join point model of orchideo are comparatively simple and thus easy to understand and clear. But due to its simplicity it is less capable. During the development of TeltowCar we gained enough experience about the orchideo framework, and especially about the development of applications with it, to discover some problems of orchideo.

The application developer does not know what code will be woven at join points during code writing. Depending on the session configuration he uses, different aspects can influence the control flow at the join points. To get to know which actions are possibly woven, he has to open the session configuration and select the action he uses in the code. If there is more than one session configuration, he possibly has to check all of them. So when writing code and invoking an action, important information is not visible for the developer at this point. In [?] this and other problems of AOP and especially of orchideo are examined and a solution for orchideo, the *join point marker plug-in*, is provided.

While debugging the developer has no direct access to the state of the engine and the aspect configuration. He may want to know about the session configuration, the enabled aspects and advice within this configuration, and the join

```
@Test
public void testCommitNewCustomer() {
  int numberOfPersonsNamedJohnDoe;
  // find out how often there is a John Doe in the database
  Criterion[] crit = { Restrictions.like("firstName",
      "John"), Restrictions.like("lastName", "Doe") };
  numberOfPersonsNamedJohnDoe =
      findInDatabase(Customer.CLASS, crit).size();
  try {
    @SuppressWarnings("unused")
    Customer c = createCustomer("Mr.", "John", "Doe");

      // commit customer to database
    hibernateAspect.commit();
  } catch (Exception e) {
    e.printStackTrace();
    fail("commit failed");
  }

  // assert that the customer is in the database
  assertSame(findInDatabase(Customer.CLASS, crit).size(),
      numberOfPersonsNamedJohnDoe + 1);
}
```

Fig. 17: The JUnit Test `testCommitNewCustomer()` for the Application Model Project

points for this session configuration. The session object also provides a history that contains a log of the previously invoked actions. In the variables view, the session object can be found, but the developer is confronted with a lot of information about the session. In [?] the *session view*, displaying the most important information about the session, and the logical structure plug-in we developed are examined.

When stepping through source code, the developer may step into framework code he is not interested in. A developer does not want to step into generated code or into the implementation of an action. We provide a solution for this problem with the *step filter plug-in* [?].

During debugging the state of orchideo objects is not visible for the developer. The developer has no information about the constraints of orchideo objects. When a constraint is violated the `HibernateAspect.commit` action will fail. The developer then has to check all orchideo objects within a session and whether they meet their constraints. We developed a *constraint view* [?] that provides information about whether the constraints of an orchideo object are violated during the debugging or not.

Assuming that there is a constraint violation like mentioned above in this section and `HibernateAspect.commit` is executed, this will lead to an `ExecutionInterruptedException`. The exception trace, produced by this exception, is very long and hard to understand. It is not easy to find out that constraint violation was the cause of this error. Tools developed to solve this problem for orchideo are described in [?] and [?]. To find the cause of such an exception while the application is alive, we provide the *runtime rescue plug-in* [?].

During the development of TeltowCar, we decided to set up a *continuous integration system* (CI), to be able to execute integration tests and regular builds of the whole system. The requirements we defined for a CI and the CI system we used during the development of the plug-ins are described in [?]. The orchideo|suite does not provide a model-level debugger. Hence errors cannot be mapped back to the model.

When the developer uses modeled objects during code writing there is no information about the constraints of this object. Every time he wants to alter such an object he has to check back in the model, if it has any constraints and if any of them is violated. This could lead to accidental constraint violation. If the developer does not do this manual check, he will get an error followed by a stack trace during the execution.

When developing own aspects, the developer has to implement the `apply()` method of advice. The advice's woven actions are defined in the model. The advice's `apply()` method has to return actions that are defined in the model, but it is possible to return actions not defined in the model. There will be no warning in the code or in the model. The developer just has the chance to find out at runtime, when assertions are enabled in his orchideo IDE. Only then orchideo will throw an exception.

7 Conclusion

The orchideo framework combines the two software development approaches AOP and MDSD. For a better understanding of orchideo we shortly presented these approaches. We described how they are realized in orchideo and how we used the orchideo|engine and orchideo|objects to develop the application TeltowCar. We used TeltowCar to get to know orchideo and its possibilities. We found that orchideo|objects provides good tool-support for the model-driven development of business applications and that the aspect model of the orchideo|engine is comparatively simple. During the development of TeltowCar we found some problems as well. These were mainly typical problems a developer is faced with when creating an application with orchideo. A subset of these problems and solutions we found for them are examined in [?,?,?,?].

Bachelor Thesis

Static Analysis of orchideo Advice Weaving

Stephanie Platz

Supervisors:

Dr. Michael Haupt, Malte Appeltauer
Prof. Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute,
Potsdam, Germany

June 25, 2010

Static Analysis of orchideo Advice Weaving

Stephanie Platz

Hasso Plattner Institute
Potsdam, Germany
stephanie.platz@student.hpi.uni-potsdam.de

Abstract. Aspect-oriented programming has been proposed as a way to improve modularity and increase the complexity of a program. Aspects are used to encapsulate crosscutting concerns which tangle and scatter the object-oriented code. However, the impact of aspects to the object-oriented base program are difficult to understand. The developers have to keep all aspects in mind to comprehend the whole impact. In today's software development frameworks a good support to overview the program code is expected. We have extended the orchideo|suite with an analysis plug-in which visualizes aspect impacts.

1 Introduction

The orchideo|suite [4] is an Eclipse[1]-based hybrid of an aspect framework and a model-driven development environment as described in paper [**?**]. The orchideo|object part offers tool-support to create application code with models. Furthermore the orchideo|engine enables the software developers to use aspect-oriented programming (AOP) [27] in their application code. Similar to object-oriented programming (OOP), AOP offers a higher abstraction level of the actual code, for example to get a better overview of the whole program. As a result the developers have to deal with other arising difficulties. During program development, coherences of classes and methods are hard to overlook due to late binding and polymorphism [28]. Those constructs are a result of OOP. Using AOP can result in important information being not visible when a software developer needs it.

Another problem is the general difficulty in understanding the impact of aspects. An overview of problems caused by the paradigm AOP is listed and explained in Section 2.1. To support the developer in facing those problems IDE tools use analysis techniques. Analysis can be either static or dynamic, whereas both are used in various fields. Compilers use static analysis to optimize the performance of a program. Static analysis is also used to construct test data. Information of the control or data flow of a program can be visualized as an aid for a developer to debug and to comprehend the structure of a program [29].

To understand how analysis can be helpful to overcome those problems we give a closer look on how AOP is realized in orchideo in Section 2.2. There we state some profound information to Section 4 of paper [**?**]. Furthermore we list the problems with AOP, both, those that orchideo has to deal with, and those

orchideo does not have to deal with. In Section 2.3 we explain what should be analyzed to overcome the problems and classify our work in the wider field of analysis. In Section 2.4 we state similarities and differences between AspectJ [30] and the realization of AOP in orchideo. Both realizations are based on Java [31]. That is why we use AspectJ and tools for AspectJ to evaluate our approach.

As stated above orchideo is a tool that combines both paradigms AOP and MDSD. In this paper we will focus on the realization of AOP only. The nature of AOP is to modularize cross cutting concerns. Thus aspects can influence a lot of parts of the base program. When implementing AOP the developers using it have to be kept in mind. One of the difficulties is to make the developers aware of where code is influenced by aspects and in which way it is influenced. Additionally it is important to make the control flow at such locations visible to the developers.

Also an orchideo user has to deal with those exact same problems (Section 2). He has to collect and combine aspect information from many different places to face these problems (e.g. configuration files, framework aspect definitions, implementation files of own aspects, debug code of interest, and so forth). Our approach—the JoinPointMarker plug-in for orchideo—uses static control flow analysis to extend orchideo in visualizing information about aspect impacts. This includes displaying woven code at corresponding source code locations. Our goal is that a developer gets aspect information feedback while he writes code without switching files.

The JoinPointMarker and the AJDT [32] plug-in help the software developers to have an overview of the aspects that are contained in their program. Both are introduced in Section 3. While we only state the developers' point of view on our JoinPointMarker plug-in in Section 3, we will give a detailed description of how the JoinPointMarker plug-in works in the following Section 4. Then we discuss the support of our plug-in for the developers. We state which of the orchideo-specific AOP problems we solved and which may need improved. After a short overview of related work we conclude our paper.

2 Static Aspect Impact Analysis in orchideo: An Overview

2.1 Problems with aop

The terms of AOP are described in Section 3 of paper [?]. There we list most of the advantages and use cases of AOP. But common points of criticism are that AOP is good for tracing and logging only and that AOP can be obviated by annotations, application frameworks, design patterns or well-designed interfaces. Ramnivas therefore states in [33] that AOP itself does not solve any problems, but improves solutions for already fixable problems by decreasing the complexity.

Nevertheless other problems remain that AOP has to face. Frequently asked questions from developers are: Where is code woven to and what code is woven? Which advice is woven first? Which changes influence the matching of a pointcut?

The last question addresses the fragile pointcut problem [34]. In most of the current AOP languages the matched join points were described lexically. But

during system evolution the code of the base program is changed permanently and therefore causes a change in matched and unmatched join points. Pointcut delta analysis [35] or (automated) refactoring [36,37] tools can help to overcome the fragile pointcut problem. In pointcut delta analysis two versions of a program (before and after editing the code) are compared. The changes between these two versions are analyzed and checked if the set of matched join points changes. The resulting delta are removed and added join points. A developer can use this delta to decide if the changes he has made are those he has intended.

Using refactoring tools means that changes in the program do not change the behavior of the program. Thus if a developer edits his code and the set of join points alter, the respective adjustments to pointcuts are either implemented or suggested to the developer. Hence with a refactoring tool or point cut delta information the developer is aware of changes which influence the matching of pointcuts.

The remaining questions arise due to violations of information hiding principles [38,39]. AOP constructs, such as advice and pointcut, can introduce new data and control dependencies and can additionally alter or eliminate existing dependencies. Thus information for program comprehension is not visible where it is strongly required [40].

On the one hand a developer wants to know if there *is more behavior* than he expects while writing object-oriented code without memorizing all pointcuts. On the other hand, even if a developer knows the location of all insertions, he cannot be sure of their order. If several advice match the same join point—which is then defined as a *shared join point* [41]—the woven order can often be nondeterministic. This can lead to undefined behavior if the woven code depends on data that is changed by certain matching advice.

All in all there is a lack of clarity concerning the location at which a developer is working. The impacts of aspects are not visible. We state in Section 3 two approaches facing this problem, whereas one of them is our own approach to make hidden aspect information visible.

2.2 aop Problems Found in orchideo

In the above section we characterize two main problems with AOP: The fragile pointcut problem and the hidden impacts of aspects. The *fragile pointcut* problem does not exist in orchideo because the pointcut language is very simple and is not based on lexical properties of the code. Therefore we do not need to use pointcut delta analysis.

A pointcut in orchideo is one single action which is defined by the framework or by a self-modeled aspect. The orchideo join point model and aspect examples are described in [**?**]. This implies that in orchideo code can only be woven around an explicit call of a modeled action, which is in particular a method call to an aspect. So if a developer is aware of that fact, he knows where code *can* be woven to. But even this is about a fifty-fifty chance because not every call to an aspect is a join point of current enabled advice.

In 2.3 we state the conditions that influence the actions that are being used as a join point. As a developer does not really know where code is woven and especially does not know what is woven, orchideo does not handle *impacts of aspects* in the base program satisfyingly. In an extra editor a developer can search for actions of aspects and inspect woven actions (see Figure 2). But the aspect impacts are not visible in combination with the base program. And in particular if the developer does not know what kind of method calls are actions, he cannot expect impact of aspects and so has no reason to look into the configuration dependencies. In the next section we propose what must be done to overcome these problems.

2.3 Reducing Problems Using Static Analysis

To solve orchideo specific AOP problems we have to analyze what happens during a call of an orchideo action, since it can be used as a join point and alter data and control dependencies. Analysis can be split into static and dynamic analysis. In the static analysis, information from different locations in the source code is combined and results in an approximation to the executed program. The static analysis produces a contingently larger set of possibilities than what will happen during the execution of the program. Contrary to this information is collected during program execution in dynamic analysis. Such information can be arbitrary exact but never complete because there is no assurance that the program runs through every possible execution state.

Our goal is to get and visualize information about the control flow at source code locations where advice matches *while a developer writes code*. This fact excludes the usage of dynamic analysis. A developer does not wish to execute the program to get information about what code will be woven. He rather wishes to have feedback as soon as he writes code because switching back and forth between the environments drastically decreases his efficiency for writing code.

Additionally the execution of the program takes time and is not exact enough for this use case. During every execution another instance of the possible woven order may appear. A developer cannot be sure that the results from the dynamic analysis are complete and that all possible woven code is offered to him. This problem remains independent of the number of times the program is being executed.

In our solution we therefore use static analysis to visualize information about woven code in orchideo. The collected information can be displayed when and where a developer writes code. We will discuss the accuracy of the resulting information in Section 5.

In orchideo code can only be woven to actions defined by aspects due to the join point model defined in orchideo. An advice can only weave other actions either *before*, *after* or *around*. But actions themselves can call another action directly, which is then treated as a *nested* action. Further, other aspects *inject* actions to a specific action. The difference is that the aspect containing this specific action is not aware of these injected actions. In orchideo the control flow at a method call being an action is influenced by the following conditions:

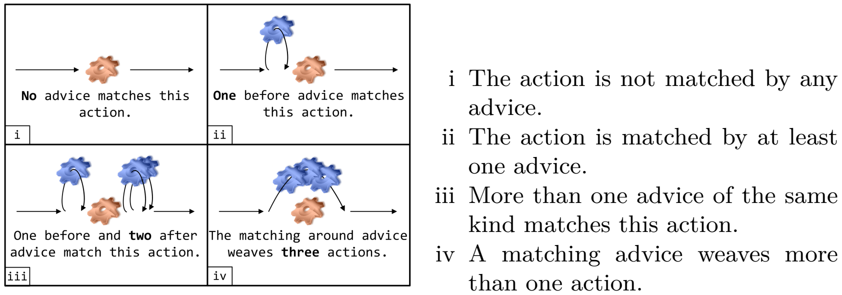

Fig. 1: Conditions influencing the Control Flow. The matching advice in conditions ii to iv weave at least one action. Otherwise the impact will be the same as in condition i.

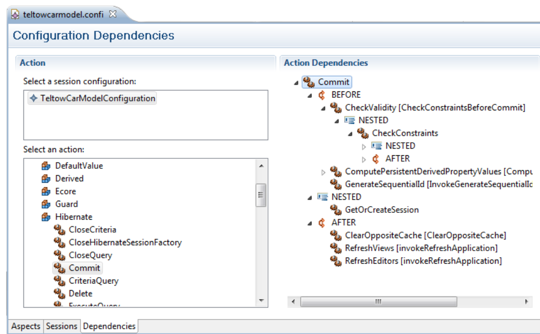

Fig. 2: Inspecting a Configuration File

Every condition causes other effects on the control flow and more than one of the four conditions can occur simultaneously. For each condition an example is given in the picture to the left of the listing in Figure 1. The first condition i implies that no single action will be woven to the action call. So only the method body of the action will be executed and the control flow continues as usual.

Condition ii means that other actions will be woven to this action. While writing source code that contains a call of an orchideo action, a developer does not know which of those two conditions (i or ii) affects that action call. He can manually look up through his configuration files and inspect what aspects and advices he has enabled. Once he has found the concerning action he can find out if there are other actions woven to it by selecting said action. He can also see which actions were woven before, after, around, nested or injected. Assuming a developer used the `commit` action of the `Hibernate` aspect in his code this can look like Figure 2.

In the displayed configuration file the `commit` action is inspected and additional actions woven before, after and nested can be unfolded. The problem here is that the information is stored in a separate file. That would result in the

undesired side effect of a developer changing back and forth between files. He wishes to have a complete overview of the current edited code at the location he changes it.

The control flow is influenced by condition ii if only one advice per kind (before, after or around) matches an action. The order of the advice is obvious: first the actions of the before advice, then the actions from the around advice and finally the actions of the after advice are woven.

Condition iii implies that the control flow is now influenced by advice precedence. An advice can be preceded and followed by an advice of the same kind. A developer can currently look into the configuration file editor and find out the woven order of advice with regard to the predecessors or successors. Nevertheless if advice specify for example the same predecessor and have no dependency to each other, the order of the woven advice is nondeterministic. Though there is no indication that the order remains the same as the displayed one during every execution, a developer might assume otherwise nonetheless.

An advice that is weaving more than one action also influences the control flow (iv). The behavior of the advice specifies the order and the amount of the woven actions. This means that not all woven actions defined by a single advice do always have to be woven. But the editor of the configuration file displays all actions the advice *might* weave. This is the same with nested and injected actions. In some executions of a specific action, not all nested or injected actions have to be called, depending on whether certain conditions are fulfilled to call those actions.

In Figure 2 we can see two other tabs. In the tab 'sessions', a developer can create and delete session configurations. With a click on the tab 'aspects', a developer can specify which aspects and advice of the orchideo|engine, or from the current program environment, should be enabled. This disabling and enabling will have an effect on the four conditions we stated in Figure 1 because this changes the actions woven to join points.

2.4 Comparison with AspectJs Ordering of Advice

AspectJ is another commonly used AOP realization for Java. The join point model of AspectJ is more powerful than the one used in orchideo. For example every method, constructor, exception or field access [30] can be used as a join point. Thus this totally different join point model of AspectJ causes other influences in the object-oriented Java source code than those appearing in orchideo. In orchideo code can only be woven to an explicit call of a modeled action which excludes a lot of code being a potential join point. Nevertheless a developer cannot be sure which method calls to aspects are actions (see Section 2.3). But in AspectJ he has no chance to guess where code will be woven to when looking at the bare source code. See [?] to have a closer look at the orchideo join point model. In Section 3 we describe an approach for that problem for both realizations of AOP.

Inspecting the advice ordering in AspectJ we can recognize some similarities with the orchideo weaving model. Figure 3 shows a comparison of advice precedence of both weaving models. AspectJ offers to specify the precedence of

aspects. An aspect has a higher precedence than another if it is explicitly declared by a developer or if an aspect is a specialization of the other one. Thus in the case that advice from different aspects have a shared join point,

- the aspect with higher precedence executes its before advice before the one with the lower precedence.
- the aspect with higher precedence executes its after advice after the one with the lower precedence.
- the aspect with higher precedence executes its around advice and encloses the one with the lower precedence. As a result the higher-precedence aspect decides if the lower ones are executed by calling `proceed()` or not.

In orchideo there is no abstraction defined that aspects functioning as a container of advice can specify advice precedence. Every single advice can specify its own predecessors or successors. If no precedence is specified—in both AspectJ and orchideo—the order of advice is nondeterministic.

However, AspectJ offers beside aspect precedence an ordering mechanism for advice defined by the same aspect. The lexical arrangement in an aspect determines the order of advice. So the first appearing before advice is running before the following before advice with the same matched join point. Like a lexically leading around advice likewise encloses subsequent before ① and after advices ② (see lower part in Figure 3).

This phenomenon cannot happen in orchideo. At a matched join point, all before advice are executed, followed by one to all around advice. Lastly, all after advice are executed, whereas those three stages take place with regard to the defined predecessors and successors (see upper part in Figure 3). Nevertheless in AspectJ and in orchideo, nondeterministic ordered advice remain.

3 Improvement of Awareness of Aspect Impacts

In Section 2.1 we state different problems with AOP. The fragile pointcut problem does not apply to orchideo, but orchideo does point out an impact problem. Therefore we focus on approaches for visualizing AOP impacts in the base program. Approaches facing this problem have to make hidden impacts visible, which leads us to two requirements:

- Disclose matched join points
- Clarify the control flow at matched join points

The first requirement means that locations where code is woven to should be offered to the developers while he writes object-oriented code. This helps the developers be aware of locations where aspects add additional behavior. Depending on how many advice match a shared join point the control flow at this join point becomes more obscure.

The second requirement claims that information concerning the order of woven code is prepared for the developers. In this section we will explain two different approaches for those two requirements. Since orchideo is based on Eclipse

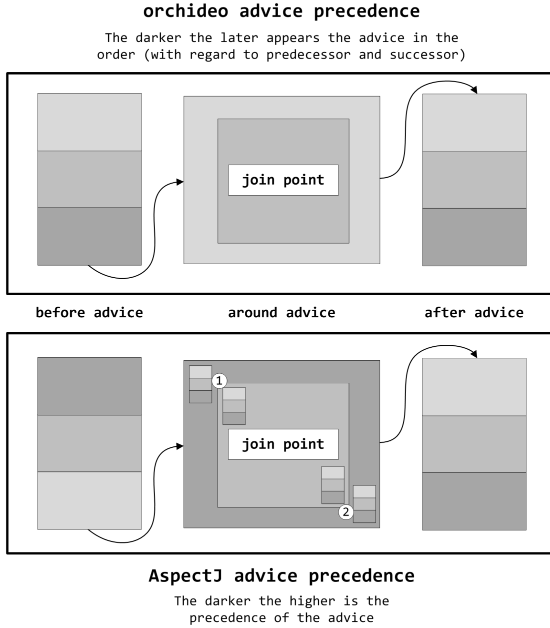

Fig. 3: Comparison: AspectJ and orchideo advice precedence.

we have done research on an Eclipse-based plug-in for AspectJ (AJDT). We choose AspectJ because it has a similar way to order advice as done by the orchideo|engine, which we mentioned in Section 2.4. We will first have a look at the AJDT plug-in and then at the JoinPointMarker plug-in, which is our approach for the hidden impact problem with AOP in orchideo.

Working with Eclipse. When starting Eclipse the developers are asked to choose a *workspace*. The workspace is the location where the Eclipse user wishes to save all his Eclipse settings and where he can create and save his source code. Eclipse users are accustomed to working in *perspectives* which are customized to a specific programming language or functionality (like debugging). Almost every perspective provides the developers with an editor to write code and a view in which the developers can explore the *projects* of his current workspace. Developers are also able to add or remove views to match their individual workflow.

3.1 The AJDT Plug-in

AJDT is a plug-in for Eclipse and the state of-the-art tool for understanding AspectJ systems [42]. Static analysis from AJDT provides valuable feedback for a developer to disclose matched join points. AJDT offers two strategies to disclose aspect information while writing code.

The first one is an extension of the *Outline view* and special *editor markers*, both well-known tools for Eclipse developers. The Outline view displays all

methods and field members contained in an opened Java source code file, while also displaying the import declarations and the package that contains the file (① in Figure 4). The symbol of each entry of a method or field member varies belonging on the type of accessibility. If a method is matched by an advice a little red triangle appears additionally in the according symbol. In very early versions of AJDT the advice matching such a method were assigned to that method in the outline view and could be unfolded by the Eclipse user. In newer AJDT version this information is displayed in an extra view—the *Cross Reference view*.

The Cross Reference view shows all influences of aspects on the currently opened Java file. Affected fields and methods are listed in a tree-based structure (② in Figure 4). For example a developer can inspect all matching advice for a listed method (③ in Figure 4) and by clicking on one of them he will be navigated to the aspect file containing that advice. This causes the Outline and Cross Reference view to change their content. The Cross Reference view now displays all methods and classes on which the aspect takes effect. This way AJDT guarantees a good overview of the effects of one single aspect in the whole program. The Outline view displays all specific members of an aspect similar as it is done for a Java file. These are for example methods and inner classes or interfaces and also pointcuts and advice. All in all the Outline view gives a structural view of the current file and the Cross Reference view shows all cross cutting influences in or of the current file.

Additionally AJDT sets editor markers at source code lines from advised methods. The markers are placed in the left ruler of the editor (④ in Figure 4). Clicking on the marker will highlight the whole line and the matching advice is displayed in a tool tip. But if more than one advice weaves code to a method the tool tip becomes useless because there is only the number of matching advice displayed. To figure out which advice matches the method a developer has to right click the marker and select the entry 'advised by' in the pop-up menu. Then all concerning advice are listed. By clicking on one of these entries a developer is navigated to the specific advice. The editor marker in an aspect file behaves the same way. They are set at code lines which affect the base program and a developer can also right click them to navigate to the affected code.

The second strategy from AJDT to disclose aspect information is the *Aspect Visualization perspective*. This perspective makes use of the *Visualizer* and the *Visualizer Menu*. AJDT offers an AspectJ Provider for the Visualizer which is enabled by default in the Aspect Visualization perspective. With an enabled AspectJ Provider, the Visualizer Menu displays all aspects affecting a selected project, package or source file. Every aspect is assigned an individual color and an own checkbox (⑥ in Figure 4). The Visualizer in combination with the AspectJ Provider displays source code files that are contained in the current selection of a project, package or simply a single file (⑤ in Figure 4). The source code files are represented by bars whereas their lengths depend on the number of lines of the file. Every affected file is striped in a specific color that depends on the aspects and is defined in the Visualizer Menu. The position of the colored stripe in the bar depends on the location in a file of the program elements that are affected

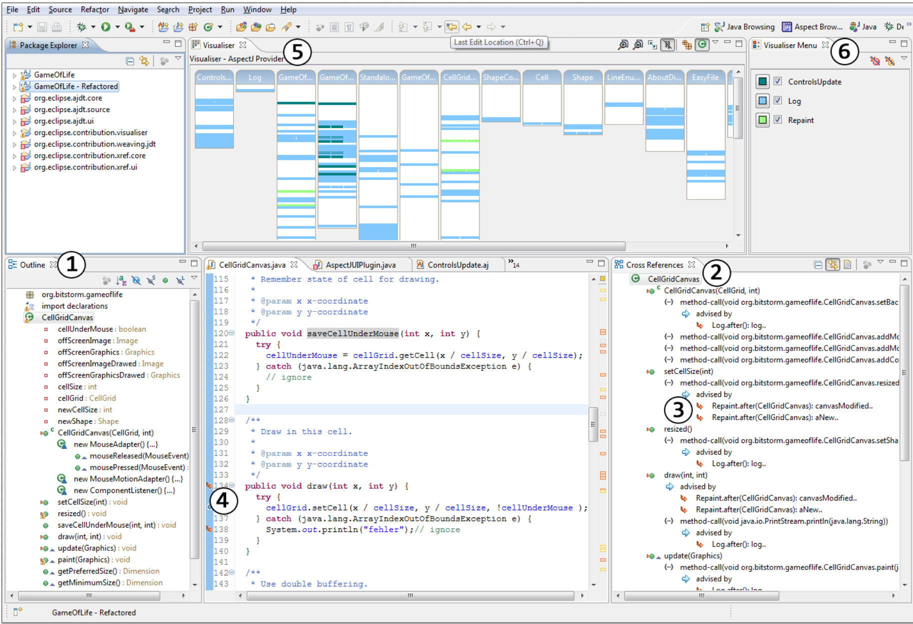

Fig. 4: The Aspect Visualization Perspective

by aspects. Aspects are only displayed in the Visualizer if the checkbox in the Visualizer Menu is enabled. This offers a developer to choose which aspects he wants to visualize.

With those two views a developer has an overview of where his aspects influence the base program, and by clicking on a stripe a developer is navigated to the corresponding source code line and can resume work there. In Figure 4 the Aspect Visualizer perspective is extended with the Cross Reference and Outline view. A developer using this customized Eclipse workspace has a good overview of aspect impacts in his base program.

But for our second requirement—clarification of the control flow at matched join points—a developer does not receive sufficient support from AJDT. AJDT provides the developers with the knowledge of which advice will weave at which join point. Via navigation a developer can look into every single advice to find out the woven behavior. He knows what will be woven but has no chance to get information about the order of advice from AJDT, simply debugging and observing the execution of the program remains the only mean to get examples of the possible order of woven advice. Unfortunately, debugging of aspect constructs is not well supported, too [43], [**?**].

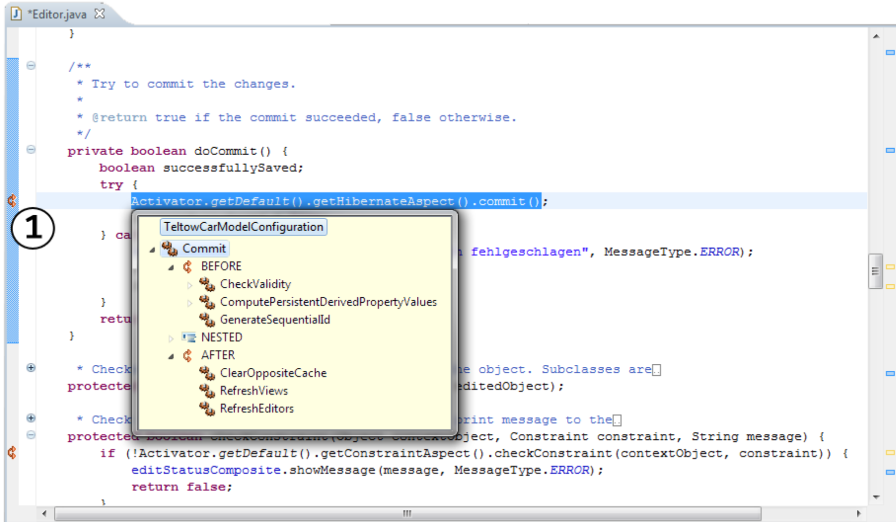

Fig. 5: JoinPointMarker plug-in in orchideo

3.2 The JoinPointMarker Plug-in

For orchideo we created a solution that supports a developer in getting an overview of the impact created by aspects. We use static control flow analysis to visualize what code is being woven at which location and in which order. In orchideo code can only be woven to a call of an aspect action (see Section 2.3). To help a developer in getting an overview of aspect impacts our first requirement is to disclose matched join points. In orchideo this means to disclose aspect actions which are currently used as join points. Our plug-in currently marks matched actions in the editor. This is done while a developer writes code and it automatically adjusts to the newly generated input. That allows a developer to receive instant feedback from our plug-in. We mark matched join points similar to AJDT. Markers are placed on the left ruler of the editor where the method call of a matched action is written (① in Figure 5) A marker signalizes that the action in this line is matched by at least one advice, which visualizes the difference between the conditions i and ii mentioned in Section 2.3. Therefore a developer knows that other actions will be woven to this action.

Having those markers, a developer is now aware of the location that code is woven to but he is still unaware of what is woven to it. To get this information he can click on the marker and a pop-up will appear displaying woven actions (see Figure 5). In orchideo the woven actions depend on the configuration file that a developer uses to create sessions in his program code. Because of this our plug-in will display a comparison of woven actions of different configurations on demand. There are two ways to choose which configurations should be

compared. A developer can select configurations in the preference page of our plug-in which is reachable by a right click on our marker. A simple click on a join point marker will compare enabled session configurations of the preference page only. In case that a developer has not defined such settings he can choose for the configurations to be compared in the pop-up. This capability is skipped when there is only one configuration with a matching pointcut. In Figure 2 only the `TeltowCarModelConfiguration` contains an advice matching the action called in this line.

Now a developer can inspect woven actions in the pop-up just like he does in the configuration editor. Actions are grouped by before, after, around, nested and injected action and for every woven action additional woven actions can be unfolded. An example is shown in Figure 5 where `commit()` is called and all woven actions can be inspected. Our plug-in pays attention to the order of the woven actions and does *not* display them lexically. The pop-up therefore visualizes the effects of conditions iii and iv mentioned in Section 2.3. In Section 5 we discuss the benefits of our visualization.

During the evolution of the program, orchideo specific configuration files are changed in addition to source code files. Aspects and advice can be added, changed or removed and therefore the set of matching advice at a shared join point can vary. This causes changes to the amount and order of advice and accordingly to the order of actions (see Section 2.3). That is why our plug-in also notices of changes in configuration files.

4 JoinPointMarker Plug-In in Detail

In the previous section we have shown that our orchideo plug-in has to complete several tasks when certain files have been changed. Because orchideo is based on Eclipse, we have been able to make use of its automatic build process. This process will be explained further before discussing the details of our plug-in. While developing our plug-in several questions arose such as: Which projects in the workspace should be observed by the plug in? What must be observed to ensure all current information is available? In which order should the advice be listed? How do we make a matched join point visible in source code? What happens if the source code is changed? All questions are answered in detail in the following sections.

4.1 Eclipse Automated Build Process

This section will focus on how Eclipse uses its background work to compile code and support a developer during programming. In order to handle all information of projects found in the workspace, Eclipse offers two mechanisms, *nature* and *incremental project builder*.

Eclipse uses its builders to produce output based on the raw materials entered. Such a build process can be either manually triggered or automated. The

Java compiler in Eclipse for example is a builder provided by the Java Development Tools (JDT) plug-in [44] for Eclipse. Build processes are incremental which means that an initial build is developed, followed by incremental builds. The advantage of using this process is that only changes to the previous version must be rebuild. We focus on that specific advantage and therefore implemented two individual builders.

In Eclipse one builder is associated to one single project only and the builder operates on the containing project resources. Associating a builder with a project is in general done by natures. They can install and uninstall builders to a project by changing the project's build specification. The build specification is held in the `IProjectDescription` which is assigned to a project and is represented as an array of `ICommand` objects. The builder is added or removed from this array by a nature. Once a builder is assigned to a project, it is informed of all resources that changed in the project.

The build process can be triggered explicitly or automatically, for either a specific project or the entire workspace. Incremental project builders are invoked implicitly by the Eclipse platform during an auto-build. If enabled, auto-builds run whenever the workspace is changed. The build process is then triggered by a project delta which contains all changed resources. The builder then inspects the changes and if any files of interest have changed it can work on those files. Further information can be found in [21] or in [45] at chapter 14.

As soon as the `OrchideoJoinpointMarkerProjectNature` has been assigned to a project, it manages the association of our two specific builders to the project. Once the builders have been assigned they can collect information needed to mark code lines where something is woven to. Figure 10 explains how our plug-in integrates in the Eclipse Automated Build process. The `AutoBuildJob` calls `build(kind, monitor)` on the Eclipse framework `BuildManager`, which runs in background. The build kind can be either full or incremental. Eclipse runs an automatic build job, when it is started (full build) and when resources of the workspace where changed (incremental build). The `BuildManager` then builds every concerned project with every assigned builder until no more builders request a rebuild. In the sequence diagram in Figure 10 we show as an example how one single builder—in this case our Advice Cache Builder—is triggered in that loop.

4.2 Project Selection

In this section we describe to which projects we assign our two builders, so that our plug-in hooks into the automatic build process.

Typically in Eclipse more than one project is found in the workspace for a single software program and each project contains a set of related functionality. The orchideo project nature is assigned to projects in which the application data is modeled or aspects are defined. This nature takes care of the generator creating code from the modeled diagrams [**?**]. There are however, other projects in the Eclipse workspace which establish on the orchideo projects and especially make use of aspect actions.

An example is given in the *TeltowCarApplication* which is introduced in [?]. In this application exists some custom aspect projects, the `TeltowCar` project and the `TeltowCarApplication` project. Whereas the `TeltowCar` project defines the domain objects of the application and therefore has the orchideo nature assigned, the `TeltowCarApplication` depends on this project and uses those modeled domain objects. In this way the `TeltowCarApplication` has no orchideo nature assigned but can use aspect actions.

This is the reason why we have chosen to take care of different kinds of projects in order to highlight *all* affected code. We first add our nature to all created projects that have the orchideo project nature. Then we have to analyze the project dependencies and add our nature to all projects that depend (recursively) on an orchideo project. If the dependency to an orchideo project is lost or if the orchideo project nature is removed from a project our plug-in removes our nature in the same way.

Our plug-in additionally gives a developer the possibility to choose whether our nature should be assigned to a project or not using the project menu. In the case that a developer does not want our plug-in to change the project natures automatically, he can specify this in our preference page.

4.3 Join Point Highlight Builder

Developers using Eclipse are often accustomed to using tools the IDE offers, such as markers described in Section 3. The JDT plug-in makes use of problem markers and sets them up in locations where compilations errors have been detected. These markers are placed in the left ruler of the Eclipse editor. In order to determine why a piece of code is incorrect, the programmer can click on the marker and gets a pop-up providing error information.

Just as compilation errors are highlighted, a developer could be assisted by using the IDE to collect and display information about woven code. In order to meet this demand, we implemented two builders in our plug-in. The *Advice Cache Builder* collects any aspect information needed; this process is described in section 4.4. The *Join Point Highlight Builder* combines this information with the current source code and highlights it.

When the source code in a project our builder is assigned to changes, the builder collects all Java files contained in the delta. The collection includes any added Java files but excludes those that have been removed, as there is nothing to highlight. The builder parses each changed Java file and creates an abstract syntax tree (AST) to inspect the written source code. With the aid of a visitor our builder checks every node of the AST to see if it is a method invocation that represents a matched action. This check is done using two criteria:

- The method invocation has to be called on an orchideo aspect.
- The method invocation has to represent an orchideo action which is matched by at least one advice in the current advice cache.

The first criteria ensures that the method invocation is a method call to an orchideo aspect, meaning that the method call is potentially an orchideo action.

With the second criteria we can be sure that the method invocation is an orchideo action which is used as a join point and is currently matched by at least one defined pointcut in the program environment.

An example is given in figure 6. At the top of Figure 6 a screenshot of a code snippet is presented. It calls the `commit` action of the `Hibernate Aspect`. Below the snippet there is the respective abbreviated AST. Our visitor is only interested in `MethodInvocation` nodes. The visitor first checks if the declaring class implements `de.exxcellent.orchideo.engine.Aspect`. Looking at the interface inheritance structure of the AST in Figure 6, we can assess that the aspect interface is implemented and now we can be sure that `commit()` is a method implemented by an aspect. The next step is to check if this is an action that is used as a pointcut by a currently enabled advice. Therefore the visitor will equalize current aspect information from the Advice Cache Builder with information of the `MethodInvocation`. The action, aspect and package name identify one-to-one and onto a join point. These identifiers are marked in Figure 6. If a join point following those identifiers is currently used as a pointcut the visitor memorizes this method invocation. Once the visitor finishes visiting the AST of a file, it hands all collected method invocations to the Join Point Highlight Builder.

The builder then deletes all old markers and sets new ones into each source code file that has been changed. The deletion of old markers ensures that the markers visible are displayed at the right location while the code evolves. When a developer clicks on a marker the action matched by the marker is highlighted in the source code and a pop-up appears, as seen in Figure 5. In this figure only one configuration contains an advice with a pointcut that is the `commit` action. Otherwise the other applicable configurations will appear next to the `TeltowCarModelConfiguration` and this provides a developer to easily compare both configurations.

Nested and injected actions are found by the Join Point Highlight Builder. This information is fast available from the model of the action. The woven before, after and around actions are requested from the Advice Cache Builder, whereas here we have to consider advice precedence with regard to the `Any` advice [**?**], action inheritance and pointcuts being the `Any` action [**?**].

4.4 Advice Cache Builder

The Join Point Highlight Builder marks whether certain method invocations are matched join points. An orchideo application developer uses configuration files to specify which aspects and advice are enabled for a certain session. Depending on the session configuration different actions can be woven at a certain join point. Therefore our plug-in must analyze all configuration files in order to mark used join points and list woven actions.

The join point markers have to be refreshed each time a Java file is saved and therefore the Join Point Highlight Builder is triggered. Saving files is done frequently during program evolution. As analyzing the configuration files takes time, the information is cached. The cache needs to be updated only when con-

Fig. 6: AST (lower part) of an orchideo Action Call (upper part)

figuration files change. This way we provide fast feedback when a Java file is changed, but still maintain up to date weaving information.

The Ordering of Advice. In order for our cache to benefit the Join Point Highlight Builder, it must contain the information where code is woven and what actions are woven in which order. All configuration files of the developed program must be analyzed in order to determine this information.

The listing in Figure 7 shows our approach to acquiring all advice information needed from those configuration files. The first step is to find all enabled advice by finding all enabled aspects in a session configuration. In line 7 of the listing the `findAllAdvice(sessionConfiguraion)` method gathers this information.

This information would be enough to know what actions are woven to which join point, but further steps are required to get the right order of the actions. Therefore the method `getAdvicePerKind(adviceList)` is used to group the advice into before, after and around. As mentioned in Section 2.3 advice can define their predecessor and successor, respectively. We use the orchideo|engine to arrange the individual lists. The engine is responsible to weave the advice in the executed program and sorts the before, after and around advice with regard to their predecessors and successors. The engine sorts the advice list using the `SortedAdviceList<AdviceType>` class. As we use the same class to determine the action ordering our results are similar to the ones in the executed program (see lines 12–15 in listing 7).

```
1
2  private void refreshAdvice() {
3    for (SessionConfiguration sc : sessionConfigurations){
4      List<Advice>[] advicePerKind;
5      Advice[][] advicePerKindArray = new Advice[3][];
6
7      advicePerKind = getAdvicePerKind(findAllAdvice(sc));
8
9      for( int i = 0; i < advicePerKind.length; i++) {
10       List<Advice> adviceList = advicePerKind[i];
11       // sort the advice list as done by the engine
12       SortedAdviceList<Advice> sortedAdviceList =
13         new SortedAdviceList<Advice>(
14           adviceList, SortedAdviceList.ADVICEMODEL);
15       adviceList = sortedAdviceList;
16       advicePerKindArray[i] = adviceList.toArray(
17         new Advice[adviceList.size()]);
18     }
19     AdviceCache.getAdvicePerConfiguration().remove(sc);
20     AdviceCache.getAdvicePerConfiguration().put(sc,
           advicePerKindArray);
21   }
22 }
```

Fig. 7: Collect and Sort all Current Advice

Unfortunately, this statement is not applicable to all advice. For example advice without a predecessor or successor were woven nondeterministic, because the orchideo|engine has no indication as to what priority is given to them. So during the sorting of the advice lists, those without precedence remain unaffected. But as soon as another advice is defined to be the predecessor or successor of the `Any` advice [**?**] the advice without predecessor or successor definitions will be moved into the respective position. Thus we cannot be sure which advice are ordered deterministically and which are not.

The final step is to remove the outdated advice information (see line 19 in listing 7) and to save the three arranged advice lists per configuration into our cache (see line 20 in listing 7).

The Ordering of Actions. The next step in our algorithm is to get the woven actions from each advice with regard to the order of advice. Once we have the sorted advice we can collect the woven actions in sequence from every advice list. At this point we have three advice lists per session configuration. Our objective is now to save all matched join points and the belonging woven actions which are defined by their according advice. The algorithm to sort the woven actions

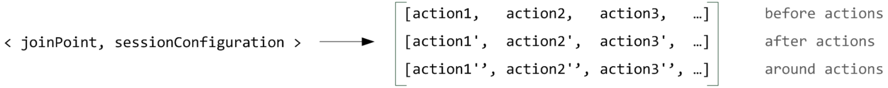

Fig. 8: Data Structure of Woven Actions

is shown in Figure 12. Thereby the order of actions depends on the order of the previously sorted advice list.

This information is assigned to the respective session configuration, so that the origin of the woven actions is clear. First of all we save a map that looks like the one in Figure 8 in our cache. We have a tuple of a join point (representing an orchideo action) and a session configuration which together map a two dimensional array of actions. The first dimension splits the woven actions into before, after and around advised actions. The second dimension includes the actions themselves (see Figure 8).

We iterate all session configurations and the three contained advice lists to establish this structure. The inner iteration over an advice list can be seen in the Appendix A in combination with the flow charts 12 and 13. We individually select each advice and create a new entry in a map for every pointcut we find. This map relates join points, which are orchideo actions, to their woven actions ①. In order to keep the kind defined by the advice, the woven actions are split in before, after and around actions.

After a new entry has been created we append the woven actions to the belonging before, after or around list ②. If an entry for a join point already exists, the woven actions of the current advice are appended to one of the three existing lists, depending on the kind of the advice ③.

After iterating over every single advice from the three advice lists, we can save our analysis into our cache. Therefore every entry that we created with the respective session configuration is put into the map described in Figure 8. If such an entry already exists it is deleted before adding the new one because it most likely contains outdated information.

There are two exceptions in the described algorithm:

1. During the iteration over every single advice it is necessary to pay attention to pointcuts that are the orchideo `Any` action [?]. All alternatives caused by the `Any` action are highlighted in dark gray in Figure 12. In case the `Any` action is the current pointcut, it is necessary to append the actions the current advice weaves, to all existing action lists of every join point entry ④. This is done with regard to the right type of list (either before, after or around). Besides we also have to put an entry with the `Any` action in our map and save all woven actions, just like we have done for every other pointcut.

Here is a short example of why we need this information: Whenever an advice x appears after an advice y which defines the `Any` action as a pointcut, all actions

woven from advice y have to append to the new created join point map entry from advice x before the woven action from advice x are appended. That is why advice y matches also the pointcut from advice x but appears in precede order. As a result we have to check if an entry of the `Any` action exists in the map. If that is the case we copy all woven actions to the newly created entry of the current iterating advice. Then we can append the woven actions of the current advice ⑤.

A list for the `Any` action is created before iterating over all advice lists and therefore we do not have to append the current woven actions to the woven actions of the `Any` action. (This was done when the woven actions were append to all existing lists.) Lastly, after all actions of the current configuration file are sorted, the list for the `Any` action is thrown away.

2. During the iteration of every single advice in the three sorted advice lists, we have to check if there are inherited join points. All alternatives caused by a base actions are highlighted in light gray in Figure 12.

A pointcut can have base actions or can be the base action of another pointcut. Thus we have to perform a check at two positions in our algorithm: If an advice has a pointcut with a base action, we have to check if such base action is a pointcut of a previous advice. This means we have to check if base actions appear as a key in our current map of join point to woven actions. Once we find such an entry, we have to put all woven actions of the base action as woven actions to the pointcut of the current advice ⑥. Further we do not have to check if there are previous woven actions to the `Any` action, because those should be contained in the woven actions of the base action. If we find no base action entry we have nothing to do, but check if there are actions woven to the any action as described above (⑤). Then we can add the woven actions of the current advice. Lastly, we have to check if the current pointcut is a base action of previous pointcuts. So we check all previous matched join points whether they inherit from the current advice. If so, we have to add the woven actions of the current advice ⑦.

The Workflow. We implemented the Advice Cache Builder to collect all the aspect information mentioned in this section and the Join Point Highlight Builder to display aspect impacts in the base program. Both builders are assigned to exactly the same projects. This is done and observed by our nature. In case any resources of the assigned project are changed, an automatic build is triggered as described in Section 4.1.

Once the `build()` method of our Advice Cache Builder is called from the `BuilderManager`, our Advice Cache Builder collects and inspects all added, changed and removed configuration files. This build job is shown in the sequence diagram in Figure 10. We then parse these configuration files and determine the session configurations they define. This is performed by the `findChangedResources(kind)` method. If we find removed session configurations we have to delete all the information saved in our cache assigned to them (`deleteRemovedSessionConfigurations()`).

Then the builder analyzes every single added or changed session configuration: First the builder searches for enabled aspects and then collects all advice

enabled in each aspect description which is done by method `refreshAdvice()` and is described in 'The Ordering of Advice'. The produced advice list is then sorted as described above in 'The Ordering of Actions'.

After the builder has sorted every single action it saves this information in our cache. Now the Join Point Highlight Builder can display up to date information concerning the location and the order actions are woven to. Whereas the Join Point Highlight Builder is only triggered by source code changes, the Advice Cache Builder has to make a request to several Join Point Highlight Builders to rebuild their assigned project. This is due to the fact that join points may have been changed by aspect definitions when the Advice Cache Builder has run. This concerns the project that the Advice Cache Builder is assigned to and all projects referencing it because they can all make use of the actions defined in the changed configuration files. In the sequence diagram of Figure 10 this is modeled as a loop. Each step of the loop is shown in Figure 11. For every concerning project a new project build job is created, which triggers an explicit build on that project with our Join Point Highlight Builder. This build is a full build so that in all resources of the project the markers are refreshed.

4.5 Integration of our Components with orchideo

When starting orchideo all builders of all workspace projects run. This first build always is a full build. In Figure 9 we can see how our builder integrates with the build process. The Java compiler compiles binaries from all Java sources of the assigned project and generates the application. Whereas the Java compiler ignores orchideo specific files. The orchideo|engine is responsible to weave advice where action calls occur at runtime and therefore uses aspect information contained in configuration and aspect source code files. The combined output of both is an application with deployed aspects (as displayed in Figure 9).

During the development process changing Java files will trigger a project delta which is then analyzed by the Java Compiler and according binaries are created incrementally. With our plug-in a modification of files triggers our builders, too. Changed orchideo specific files trigger the Advice Cache Builder to refresh the cache and to generate aspect metadata. Changed Java source code files additionally trigger our Join Point Highlight Builder to set markers in the editor with regard to the current aspect metadata.

5 Evaluation

We have stated several general problems with AOP for developers at the beginning of our paper. We have identified that the hidden impact problem applies to orchideo and thus decreases the comprehension of the program for the developers. Further we have specified two requirements to clarify the meaning of impacts. The first requirement is that join points matched by advice should be made visible in the source code. The other requirement is to clarify the control flow at those join points to the developers. Now we will discuss if our plug-in does help to overcome the orchideo-specific AOP problems.

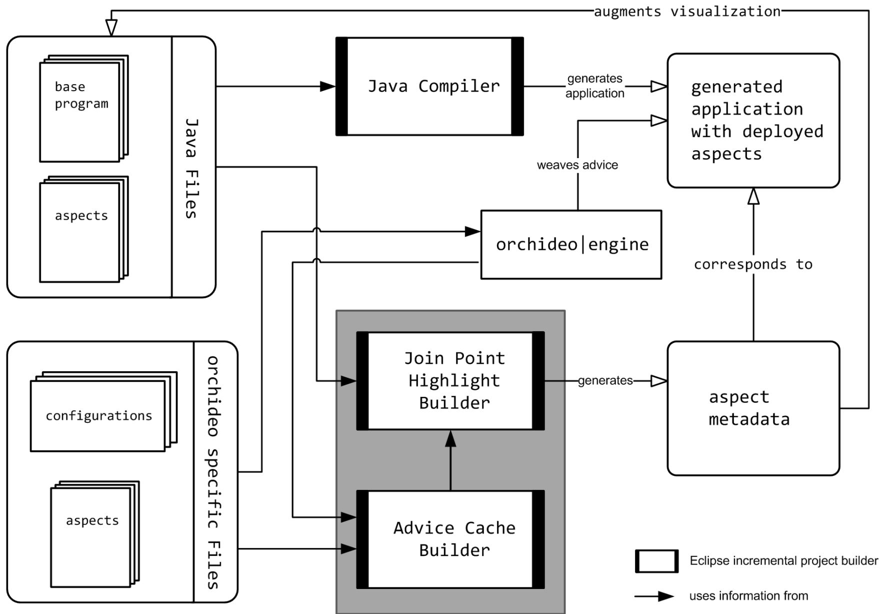

Fig. 9: Integration of our Components (highlighted in grey) with orchideo

Disclose Matched Join Points. In orchideo a programmer can make use of two different paradigms. He can use models to develop application code and draw on functionality with the aid of aspects. That functionality is for example creation, deletion or persistence of object instances of the modeled classes. Furthermore, a developer can define his own aspects, just like we did as described in [**?**]. While developing the object-oriented part of the application, markers on the left ruler of the editor visualize join points. Two such markers are set in Figure 5. Having a look at both code lines we can see that there are method calls to aspects which are the matched actions. But this method call does neither imply that every method call to an aspect is an action nor that this action is used as a join point by a currently enabled advice. It could also be a service (see Section 4.1 in [**?**]) or a simple method call defined in that aspect. Developers are not able to determine whether they are pointcut actions without the presence of our plug-in. In each line that our plug-in sets a marker, a join point matched by a currently enabled advice is now disclosed. With those markers a developer is now aware of all locations in the code *where* other code is woven to. But this does not help to comprehend what actually happens at these locations.

Clarify the Control Flow At Matched Join Points. Nothing will be woven to an action of an aspect if it is not used as a pointcut in any enabled advice of a configuration. The control flow remains the same. But if that is not the case, then

our plug-in will have to consider several conditions as mentioned in Section 2.3. It collects aspect information as described in Section 4, finds all woven actions for a specific join point and prepares the information for the developers. By clicking on a belonging marker a developer gets a pop-up with all woven actions separated by before, after, around, nested and injected. Further a developer can compare different configurations of woven actions in the pop-up. Thus the orchideo user is now aware of *what* is woven to a pointcut action.

Furthermore our plug-in pays attention to the order of actions. Our plug-in displays the weavings in the same order as the orchideo|engine weaves actions. It does that with full regard to the advices predecessors and successors, the `Any` advice, the `Any` action and action inheritance. Nevertheless we have to face that exceptions to this scenario can occur: There is always the possibility of the occurrence of nondeterministic woven advice (see Section 2.3). Because of the nondeterministic nature it is random if our displayed order is the same as the one that is woven by the orchideo|engine (which also weaves nondeterministically). Nevertheless we currently do not communicate this to the developers and they might therefore make wrong assumptions.

This deficit leaves room for future improvements:

1. An analysis of the predecessors and successors of advice with a shared join point (with regard to inheritance) has to be done. A point of interest are advice defining either successor or predecessor to any advice.

An interesting aspect for this might be to have an advice define either the successor or predecessor for the other advice in question. So advice with undefined successors or predecessors are indirectly affected and therefore might be woven deterministically. On the contrary, if multiple advice with a shared join point have the same advice as their successor or predecessor and do not define a dependency between themselves, their order is nondeterministic.

This is caused due to the weaving algorithm used by orchideo|engine. Because of those phenomena it is hard to determine the advice being woven nondeterministically. Nevertheless actions woven by nondeterministically ordered advice can for example then be highlighted.

2. In addition to this there is another discrepancy remaining. The *order* of woven actions of a single advice is not displayed. Our plug-in gets the woven actions from the modeled advice and displays them in the same order as they were modeled. But how they are actually woven is difficult to determine. The order of woven actions depends on the behavior of the advice weaving them. Every advice implements the method `apply(joinPointAction)` and the return value determines the amount and order of actions. This means that not all actions, which are defined as woven actions from an advice, have to be woven. Besides implemented conditions this can affect less, equal, or some duplicate actions.

An example is the `InvokeGenerateSequentialId` advice which weaves the `GenerateSequentialId` action before the `commit` action (see Section 5.4 in [**?**]). In Figure 5 it is the third woven before action. This action is only woven if objects of a certain type were committed. So the `apply(joinPointAction)` method

from the `InvokeGenerateSequentialIdAdviceImpl` returns an empty array or an array which contains one `GenerateSequentialId` action.

Analyzing the return values of such advice implementations can be included with an additional builder. But imagine an advice weaving three actions (`a, b, c`) on three different conditions, either action `a`, action `b` or action `c` will be the only one woven. This will be a challenge to display intuitively. The next issue is the impossibility of analyzing framework aspects and advice simply because orchideo does not provide the required resources. So this information has to be included by orchideo and not analyzed at development time. And questions as to how actions that specialize the actions from the framework are to be handled remain.

For visualizing the order of actions with special suggestions to nondeterministically woven actions a lot can be done. But perhaps it is important to consider the possible benefit beforehand. This task might be too specific for the developers. It might be more efficient to recheck their own code or to query the documentation of framework aspects compared to having it analyzed by a tool. Nevertheless there could already be a hint for the developers that there are nondeterministic woven actions.

Conclusion. All in all *what* and *where* code is woven is offered from our plug-in. This increases the comprehension and evolution of the program. But the *order* of actions *might* not always be the executed one. This information is currently not provided to the developers. It represents the larger set of possibilities produced by static analysis (see Section 2.3). Concerning the nondeterministic order of actions—which is caused by the orchideo|engine—we do not offer more information than a developer can get from the configuration editor. But we spare them from having to change their development environment. In particular they do not need to compare different configurations files manually but rather use our pop-up *where* they currently work.

Classification of Debugging. Another method to analyze woven code is debugging. The orchideo tool suite strives to support parallelized, multi-level distributed architectures. For the purpose of our work we will follow a classification of debugging techniques, modeled after [46, McDowell, 1989]:

– Traditional *dynamic debugging* using breakpoints and step filtering is presented in [**?**].
– *Static analysis* based on call, flow and configuration analysis is discussed in this paper.
– *Displaying and structuring information* about complex data is presented in [**?**] as well.
– Techniques for *recording, analyzing* and *displaying execution history* are the subjects of [**?**] and [**?**].

This classification gives an orientation for the various debugging techniques. The terms of debugging are explained in [**?**].

orchideo uses MDSD and AOP to decrease the complexity of the source code while developing software. But both constructs can introduce new source of complexity. For instance there is a lot of generated source code. Despite all efforts to develop software of high quality, defects can occur. To reproduce a defect using a *dynamic debugger* requires stepping through a mass of source code, which can be a tedious process. orchideo is no exception to this rule.

An orchideo user has almost no possibility to find out where a specific action is woven to. The order of actions is not interesting to a developer that only uses framework aspects because he can assume that the framework operates faultlessly and therefore weaving of framework aspects must not be observed. For aspect developers on the other hand, the order of own actions poses a topic of high interest. In the 'TeltowCar' application which we have developed with orchideo [?], it was hard to analyze when our implemented actions are woven. But with growing experience about orchideo, we recognized that setting either the right predecessor or successor seems to be the solution.

Nevertheless the possibilities to find out the order of woven actions are limited. The developers can set a breakpoint to an action call but without any orchideo resources they cannot step into. Furthermore without our plug-in they have to guess which actions are currently used as a join point. With an additional breakpoint, the developers can go step by step through their action code, but they still have no information as to which actions are currently woven and which ones are about to be woven. Some of our other debug tools described in [?] and [?,?] *can* support the developers in this case.

We have implemented a view which searches for session objects in the current debugged stack frames (see *Session View* in [?]). A developer is then able to, for example look for the last entry in the execution history of the currently used session. It is not rare that the history itself has a thousand entries, but the entries themselves are difficult to understand due to their length and representation. Every entry of the execution history could be dumped and the output is similar to an orchideo stack trace which could be parsed by our implemented *Trace View* (see [?]). But this process would mean a major inconvenience to a developer (for details see Section 4.5 in [?]).

So all in all an interactive debug tool on AOP abstraction levels is still missing. When debugging becomes exhausting, programmers tend to avoid the dynamic debugger and try to analyze the program from its source [29]. And this is where our plug-in through *static analysis*, can help the programmer to actually debug. It is much more convenient for a developer if the IDE itself could collect and show aspect information. Object-oriented tools have evolved and therefore raise the bar for aspect-oriented tool integration. With our plug-in the program comprehension is improved and we support the evolution of the application. A developer now neither needs to switch the environment nor does he require extra interactive debugging to get information about aspect impacts.

AJDT vs JoinPointMarker. In Section 3 we can find two main differences between the AJDT plug-in and our JoinPointMarker plug-in. Whereas both—

AspectJ and the realization of AOP in orchideo—offer an according mechanism to order the advice, only our plug-in visualizes the order of advice at a shared join point. If a developer knows that nondeterministic advice weaving is occurring in either AspectJ or orchideo, he has the ability to determine the order: He can either define the predecessor or successor for the advice in orchideo; or he can define the aspect precedence which is transferred to containing advice in AspectJ (see Section 2.4). Nevertheless the order of advice is not displayed in AJDT.

The second difference is that AJDT provides an overview of aspect impacts in the whole base program and can split those impacts per aspect. Our plug-in on the other hand does not offer a project wide visualization of impacts but rather visualizes the impacts for the currently edited code only. This has several reasons: In orchideo aspects cannot change the behavior of a Java class. For example neither fields nor interfaces can be added to Java classes nor can methods be advised. In orchideo code is only woven to actions, which are explicitly called in the object-oriented code. The majority of these actions are object creations or deletions, persisting certain objects in a database, and so forth. Further we can identify that the most framework aspects of orchideo are used to connect the modeled and generated code with the application code (e.g. `PropertyAspect`, `OperationAspect`, `ObjectAspect`). This leads to the question if it is meaningful to know where in the base program for example objects are created. Therefore a whole impact visualization *might* not be necessary. This should be a part of further research concerning how much impact of orchideo aspects influences the base programs of orchideo applications.

Besides comparing Figure 4 and 5 we can state that only with one pop-up it is easier to get an overview of the current impacts and to stay focused on the current implementation details.

6 Additional Related Work

We already presented a comparison of AJDT and our plug-in in Sections 3 and 5. Kersten presents AJDT in [43,47]. Although the papers were written in 2003 nothing has been improved until today concerning the aspect impact visualization. The order of actions is still not displayed, whereas our plug-in provides this information.

Next to AJDT there are other IDE tools to support the comprehension of aspects, for example Borland®'s JBuilder®, Sun®'s NetBeans, Emacs and JDEE. All of them use a similar way to visualize aspect impact as AJDT and all concern AspectJ visualization. Kersten states in [47] that for commercial use of AOP such IDEs have to make the advice execution order on a particular join point clear. This affirms our solution for orchideo.

In [48] Zhang presents static analysis of aspect impacts with special regard to state and computation Impacts. orchideo aspects can have impact on the state of objects which are defined as `in` parameters (see Section 4.1 in [?]) of actions. As the orchideo actions are implemented with Java constructs rather than with aspect constructs state impact analysis can be difficult in orchideo

and is not performed by our plug-in. The computation impact analysis Zhang presents splits advice in invariant and variant advice. Variant advice do not modify the base code which we can compare with our condition i stated in Section 2.3. On the other hand invariant advice do have impacts on the base code which actually has the same meaning as conditions ii to iv. Zhang does not pay attention to the order of advice with a shared join point (condition iii), but we do.

7 Conclusion

In this paper we saw which general AOP problems apply to orchideo. Whereas no fragile pointcut problem can be detected, aspect impacts were not visible to a developer. We therefore statically analyze the advice weaving of the orchideo|engine with special regard to the order of advice sharing a join point. In Section 3 we describe how our plug-in realizes aspect impact visualization to the developers. While they write code, they can get informed of woven code by clicking on special code line markers. Further we described implementation and algorithm details of our plug-in.

We additionally showed how our plug-in integrates with the Eclipse automated build process. This integration is used to get source code deltas during the code evolves and thus display up to date aspect impacts. We evaluated our plug-in in comparison to AJDT, which is an aspect visualization tool also based on Eclipse. In summary AJDT provides a project wide overview of aspect impacts whereas our plug-in visualizes aspect impacts only at locations where code is woven to. Furthermore our plug-in pays attention to advice ordering, whereas AJDT is not capable of this.

With our plug-in the developers can stay focused on writing code and do not need to switch back and forth between files to comprehend aspect impacts. This drastically increases their efficiency for writing code.

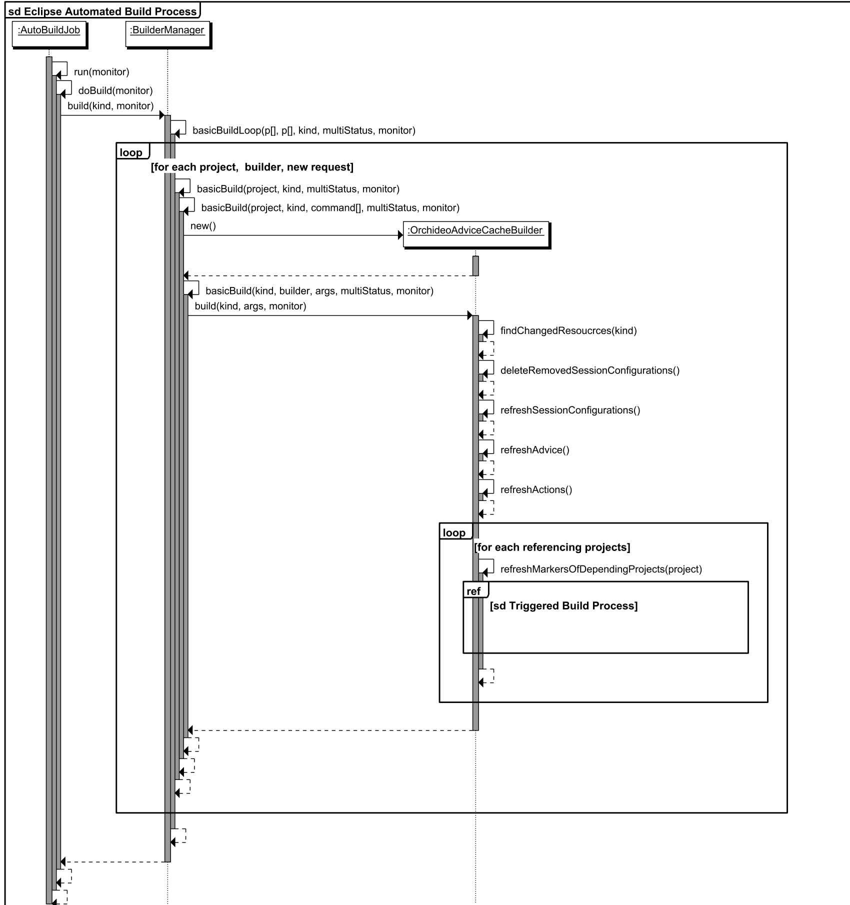

Fig. 10: Integration with the Eclipse Automated Build Process

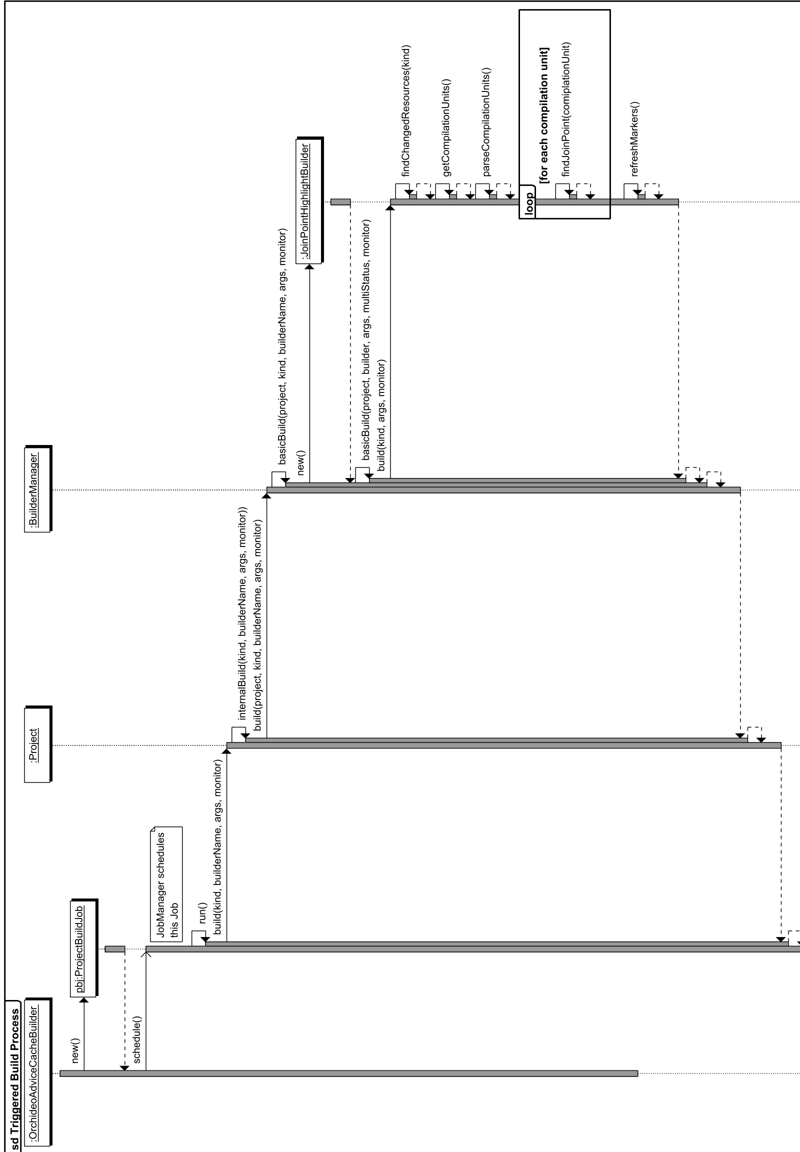

Fig. 11: Triggered Build Process

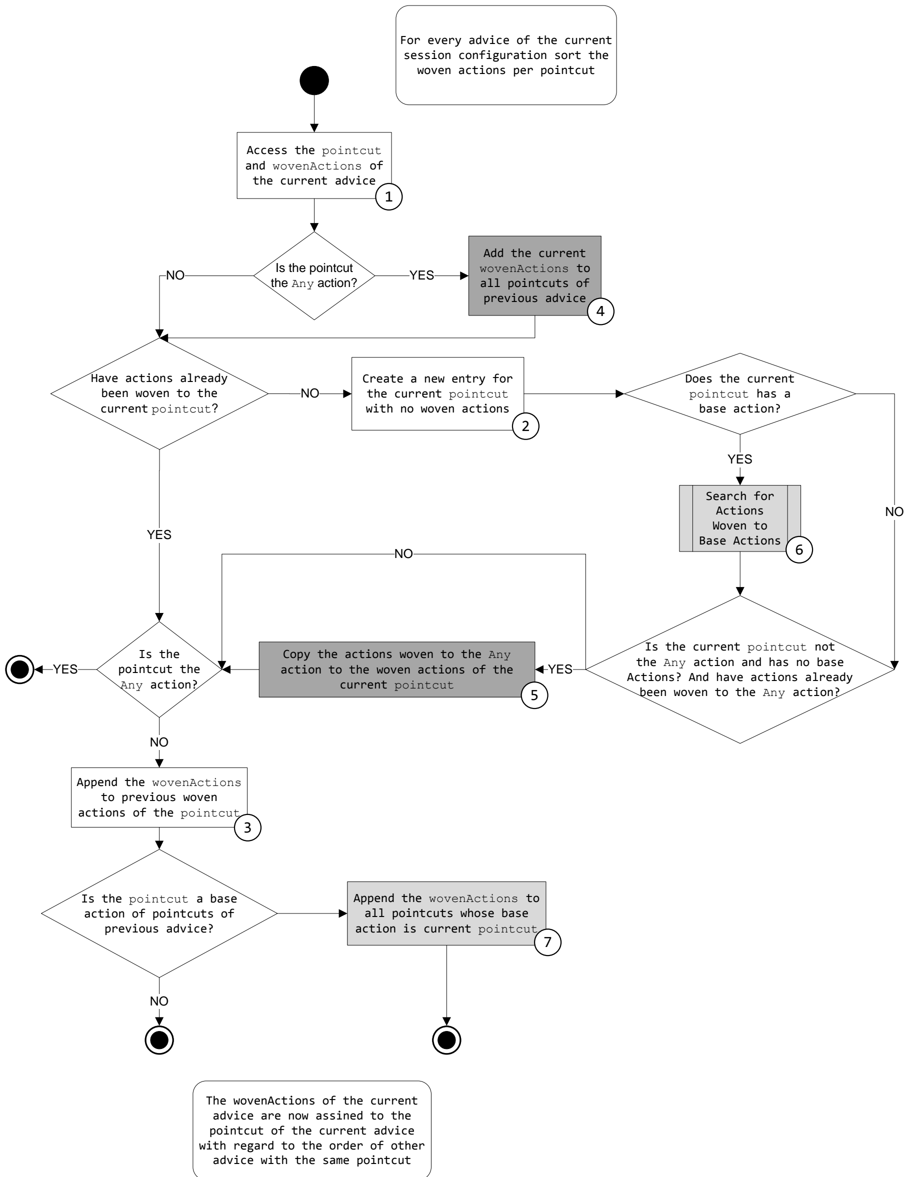

Fig. 12: Assigning Woven Actions to Pointcuts

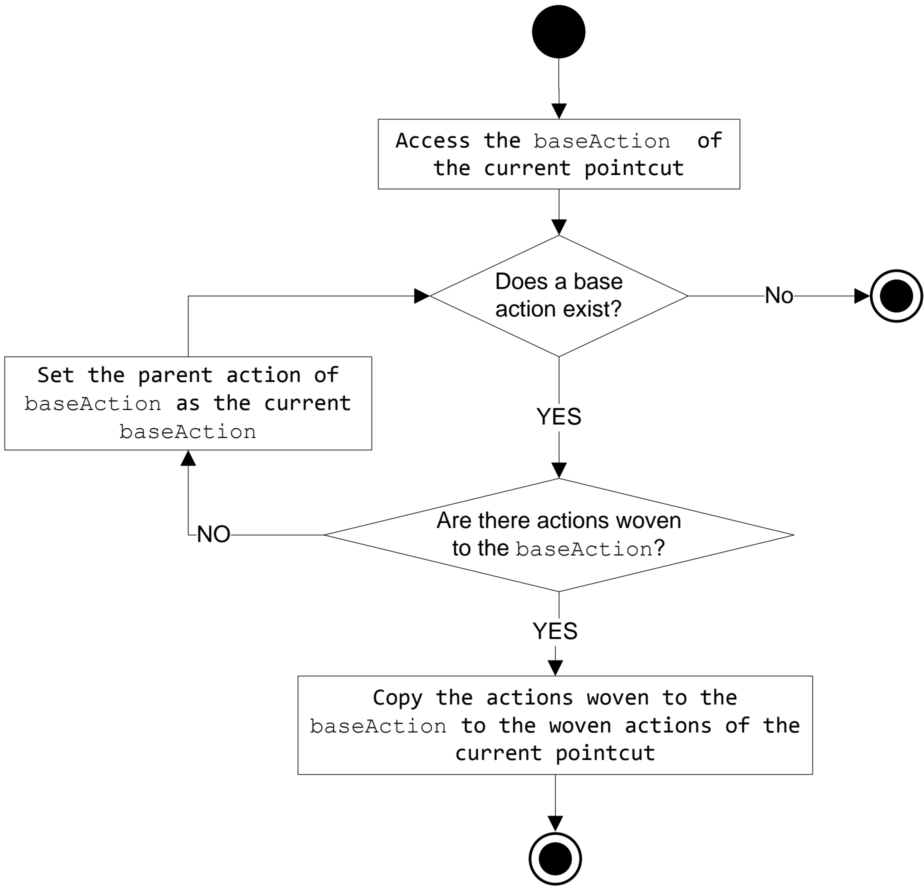

Fig. 13: Search for Actions Woven to Base Actions

Bachelor Thesis

Debug Support for orchideo

Lysann Kessler

Supervisors:

Dr. Michael Haupt, Malte Appeltauer
Prof. Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute,
Potsdam, Germany

June 25, 2010

Debug Support for orchideo

Lysann Kessler

Hasso Plattner Institute
Potsdam, Germany
lysann.kessler@student.hpi.uni-potsdam.de

Abstract. Model-driven software development and aspect-oriented programming support programmers in developing software that needs to meet complex requirements. With these approaches programmers can focus on the features they have to implement and can abstract from low level details. Unfortunately, frameworks using these approaches often lack appropriate debug support. Developers still have to debug with regard to the low level details they wanted to abstract from. In this paper, we describe tools that provide debugging solutions for a commercially used model-driven and aspect-oriented framework called orchideo. The tools help the programmer to comprehend the processes in the complex framework, but also allow them to debug their applications at a higher level of abstraction.

1 Motivation

Model-driven software development (MDSD) and aspect-oriented programming (AOP) support programmers to focus on the features they have to implement and abstract from low level details [**?**]. But despite all efforts to create software of high quality, even with these approaches failures can occur during development and after release of the software. The defect that caused the failure is often not identifiable without further analysis. Several methods have evolved that help to find such defects. One mature and commonly applied technique is the so-called *interactive debugging*.

Unfortunately both, MDSD and AOP frameworks, often lack the possibility of appropriate debugging support. For MDSD no common solution has been encountered yet that allows interactive debugging at the appropriate level of abstraction. Developers still need to debug at lower levels of abstraction. This is unsatisfactory, because the goal of this approach is to allow the programmer to abstract from implementation details and concentrate on the application's domain problems. For individual frameworks specific solutions can be constructed, but they typically depend on the concrete framework and its implementation. Currently there is no general tooling available to implement a model level debugger [49].

Interactive debugging of AOP applications introduces new problems for the debugger software and new complexity for the developer to cope with. For instance the control flow may magically change and is not easy to comprehend without additional runtime information.

The orchideo integrated development environment (IDE) as described in [?] is a framework that combines both, MDSD and AOP. It therefore inherits both approaches' advantages, but also their problems concerning interactive debugging.

This paper deals with the problems of interactive debugging in orchideo. We will first give an introduction to debugging terms and general methods applied in aid of debugging (sec. 2). We will then focus on interactive debugging and describe how this debugging method helps the programmer to find defects in a software system. Section 3 will describe problems that occur when debugging applications using MDSD and AOP frameworks, and evaluate the problems found in orchideo. In Section 4 we will then describe the solutions we developed to provide better support for interactive debugging of orchideo applications and the orchideo|engine in detail. Section 5 discusses our solutions and whether they improve the debugging process. In Section 6 we will overview solutions for other frameworks and discuss the differences to the orchideo debug tools. Finally we will give a brief summary and draw a conclusion (sec. 7).

2 An Introduction to Interactive Debugging

This section will introduce basic terms of debugging, general debugging approaches and debugging methods. We will then focus on the debugging method covered in this paper, *interactive debugging*.

2.1 Terminology

The following terms and their definition are adapted to [2] and [50].

Debugging. Debugging is the process of finding, isolating and removing defects from software to make it behave as intended.

Defect. A defect is a piece of code or of an software artifact that can make the software misbehave and cause an infection at runtime.

Infection. An infection is an incorrect state in the program execution caused by a defect. This infection can propagate through the program execution and manifest itself in incorrect states of other program parts. This does not necessarily have to happen: The infection may be overwritten or corrected. The infection eventually can but does not need to cause a failure.

Failure. A failure describes an externally observable malfunction in the behavior of the software. Hence a failure is always preceded by an infection and a defect. On the other hand the absence of failures does not imply the absence of defects [51].

Infection Chain. The cause-effect chain from the defect to the failure is called an infection chain.

2.2 General Approach to Debugging

Deduced from the above terms, debugging means to identify the infection chain, find the defect and then remove it. As described in [2] and [52], the general

approach to deduce the defect from the occurrence of a failure is as follows: Using debugging methods further described in Section 2.3, the developer isolates the infection chain to eventually find the defect itself. To achieve this, he uses a divide and conquer approach and repeatedly sets up and tests a hypothesis that narrows down the possible failure causes until he reaches the defect itself. The defect can then be corrected.

To test an individual hypothesis it is crucial that the failure occurrence is reproducible. It is also very useful if the test case is as simple as possible. This speeds up the whole process as well as it reduces the possible failure causes to some extent.

The concrete techniques used to isolate the infection chain can highly differ depending on the defect, the erroneous program, its execution environment and even on the programming language used. In certain situations it may even be possible to automate the process [52]. If debugging needs to be done manually, the time it takes can also depend very heavily on the developer who wrote the code and on the presence of a good and comprehensive documentation. It also depends on the developer debugging the program and his experience [53]. When it comes to debugging, some programmers can be three times as effective as others [2].

2.3 Debugging Methods

Different Methods have evolved to isolate the infection chain in an erroneous program. We propose the following categorization:

Printf Debugging. When using printf debugging, the programmer adds commands to print information to a simple output medium like the command line. The programmer can then observe the output medium to track the state of internal variables and whether the executed program reaches certain points in the control flow. It can also be handy to understand the sequence of actions in multi-threaded applications. Printf debugging in the context of the orchideo framework is discussed in [**?**].

Test-Driven Development. In test-driven development, small pieces of code, *tests*, are written and run. They are designated to test a certain functionality of the product. Given an adequate test coverage, an erroneous feature implementation is detected when the tests run. On the one hand this will cause failures of the system and therefore expose the existence of defects in the features. Additionally it can help to localize the defect because multiple tests checking different parts of the software can indicate which part is broken. Debugging of orchideo applications using test-driven development and continuous integration is discussed in [**?**].

Interactive Debugging. With interactive debugging the developer controls and monitors the program execution. He can inspect the internal state and alter it to cause certain effects, like simulating a certain state of the environment or causing a particular control sequence. To achieve this, the programmer uses a tool—the so-called *debugger*—that helps him controlling the

program execution and inspecting and altering the state. Interactive debugging normally only starts once a failure has been observed and the defect is unknown. The developer will then try to backtrack the propagation of the invalid software state. Interactive debugging will be introduced in detail in Section 2.4.

Listener and Control Interfaces. A software may also have built-in debugging support: It may provide listener and control interfaces that allow external observation and failure tracking. Depending on the range of the interface it can help to find failures and to isolate the infection cause. This method is applicable if the program cannot be debugged or tested directly. Therefore it is not a common solution but rather a solution for certain environments.

All of the introduced methods provide some kind of state information of the running software that is essential for tracking the problem, but is usually not accessible when the program is executed normally. Typically not all methods are applicable in various situations. Timing problems in multi threaded applications for example are usually hardly traceable and reproducible using an interactive debugging technique [46].

2.4 Interactive Debugging

As mentioned in Section 2.3 interactive debugging describes a debugging method where the developer directly monitors and manipulates the execution of a living software system. Interactive debugging is usually triggered when a failure has occurred and the causing defect is searched for. The goal is to understand the program's behavior and deduce the faulty statements. The developer uses a debugger tool that helps to analyze the software and perform the following typical interactive debugging techniques [2]:

- Execute the program and make it stop on specified conditions. Conditions are often locations in source code, variable state conditions or occurrence of exceptions during the program execution.
- Observe the state of the stopped program. This typically involves viewing the values of certain variables.
- Change the state of the stopped program. This can be achieved by assigning a value to a variable or by executing code with side effects.

Examples for interactive debugger tools are the command line GNU Project Debugger [54], the Java Debugger (JDB) [55], or the graphical Java Development Tools (JDT) [44] for the Eclipse environment, implementations of the Java Debug Interface (JDI) [56].

A debugger can read information built into the executable that it uses to communicate with the developer. For instance the debugger may provide the current method name and line number, or the name and declared type of a variable. For many programming languages and underlying platforms this information is not of interest at runtime. As the developer cannot and is often not able to debug byte or machine code, the debugger uses this more high level

information to communicate with the user. The problem of different levels of abstraction during debugging sessions is discussed in section 3.

Comparison In contrast to other debugging methods the execution of the program is controlled by the developer. He has the full insight and can acquire all information computationally accessible. He can hold the program execution, examine and change its state to accomplish certain side effects. This makes interactive debugging very dynamic. Other debugging methods require the software to be stopped, artifacts need to be changed and the program must be restarted in order to test the impact of executing a certain operation or changing the internal state. Using external observation tools like an interactive debugger, the developer can start debugging very fast or dynamically observe another detail without changing source code or recompilation [2].

The act of controlling the execution of the program can also create a major disadvantage: The runtime behavior of the software is changed extensively. Changes made by the developer during the debugging session could be forgotten and the potential solution to the problem unreproducible. Additionally any timing issues in concurrent programs may be unreproducible once the program is halted by the debugger. Although those Heisenbugs [57] are not unknown to other debugging methods [58], the probability of untraceable timing issues is very high for interactive debugging.

Concerning reproducibility there is another issue for interactive debugging: If the developer actively manipulates the system during a debugging session, he might fix the problem through the techniques described above. If the debugger does not exactly log the actions taken, they might be forgotten and the solution reinvented. In contrast to this, manipulations of the programs when using other debugging methods always manifest in the source code or other artifacts and are therefore not lost that easily.

Interactive debugging is a process that has to be done manually. Running tests in test-driven development in contrast is very automated. The test cases however are usually written by hand. Under certain circumstances even this process can be automated [52].

Like in test-driven development one test focuses on one feature, the developer focuses on one control flow path at a time when debugging interactively. The other debugging methods may provide information about other paths too. This can lead to information flooding, but can also provide an unconsidered piece of information crucial to finding the problem.

Interactive debugging does typically not explicitly help to expose unknown infections. Although a debugger can help with the process of reverse engineering and can therefore support the gaining of understanding about a system, this is merely a concern of dynamic analysis as described in section 2.3 of [**?**].

Interactive debugging needs a living system to work, whereas the other debugging techniques may produce and persist information that can be examined post mortem. Even if the debugger is attached, the program may reach a state where no additional information can be retrieved; the program cannot be rescued

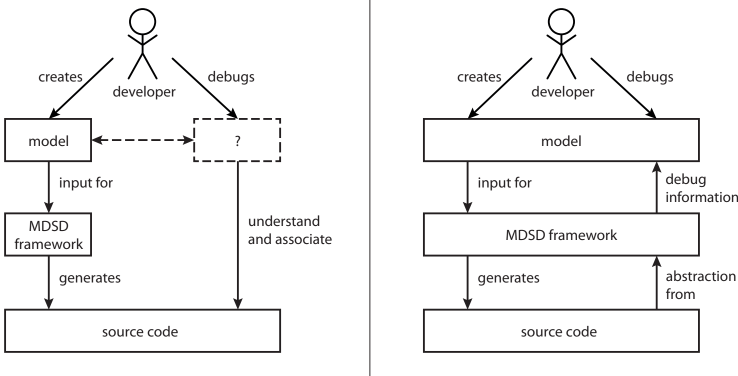

Fig. 1: Debugging of MDSD applications without and with a model level debugger, displayed in the left and right part, respectively

and has to be shut down. An example for this is the occurrence of an uncaught exception in a Java program.

3 Debug Support for MDSD and AOP Frameworks

In this section we will overview the general problems occurring in AOP and MDSD frameworks concerning interactive debugging, and what general approach is proposed to provide a remedy. For both programming paradigms we describe the problems that typically occur when debugging orchideo applications, orchideo aspects and the orchideo|engine.

3.1 Debugging Applications Developed With MDSD Frameworks

When debugging applications partially generated by a MDSD framework developers are often faced with a semantic gap between the software models they have created and the ones they have to debug. This gap exists because the debugger does not allow debugging the application at the model level. The left part of Figure 1 illustrates this problem. MDSD frameworks provide new levels of abstraction: Higher level models are created by the user, and the framework generates code from those models, usually through different steps of transformation. This helps the programmer to focus on the application specific problem instead of the low level implementation issues [?].

Unfortunately, the frameworks do not adopt the debuggers to this situation [49]. Therefore the programmer has to debug the application at the low level of the generated code and has to manually associate issues occurring in the low level code to the high level models he created. This is no trivial process at all, especially because during code generation the compilers may perform several

optimizations. This may be one reason why model level debuggers do rarely exist [59]. The ability to trace the failure occurring in generated software artifacts to the original ones is called *model level observability* [60].

In traditional programming languages it is taken for granted to debug the program at the level of abstraction it was created in—which is for example the C source code the program was written in and not the machine code the compiler generated from it. For MDSD it is even more important to allow model level debugging because the semantic gap between original model and generated code is often even wider than for traditional languages. If the model level observability is too low the problems encountered during error tracing can nullify the advantages of model-driven software development [60].

According to [59] there are two possibilities to solve this issue: The MDSD framework could either provide the possibility to execute and debug the models themselves without a prior transformation into lower level representations. An example for this is described in [61]. This solution enables only model debugging, but not the debugging of the whole application at the model level.

The other solution is more analogue to traditional debugging: The framework generates code from the model and executes this code. When debugging it interprets the low level outputs and projects them onto the model level. This enables high level debugging of the whole application, as illustrated in the right part of Figure 1. As the second solution provides a more natural debugging experience we chose this one for our debug tools.

Debuggability of orchideo as a MDSD Framework. The main abstractions provided by orchideo|objects as a MDSD framework are classes having properties, methods and constraints (see 4.3 in [?]). During an interactive debug session, properties can be monitored in the Eclipse variables view, as the orchideo framework generates a Java attribute for each property defined in the model.

Methods are also propagated as Java methods. Due to the usage of the delegate pattern those methods are partially generated and the actual method implementation is defined in a different class. Debugging this method is therefore not as easy. As the feature is implemented using AOP the next section will describe how to improve debugging in this case.

Constraints are another important abstraction and are managed by the `ConstraintAspect` provided through orchideo|objects. Currently, the constraints of one certain object are not directly visible at runtime. The information whether those constraints are currently violated is not provided, either. To provide this information we propose a solution as described in 4.2.

3.2 Debugging Applications Developed With AOP Frameworks

As described in [18] the possibility to debug aspect-enabled programs is crucial for the usage of AOP. Unfortunately, AOP frameworks seldom provide a good debugging support, which makes it hard to understand and correct the program's control flow. Additionally the adoption of AOP to solve a software development

problem can introduce new sources of errors. A categorization and description of general debugging problems produced due to the usage of AOP can be found in [18].

Considering the orchideo AOP activities described in 4.2 of [?] and the comparatively simple join point model, orchideo applications can fall victim of the following AOP typical faults, classified and specified by [62], [63] and [18]:

1. *Incorrect aspect composition*: the execution order of different advice matching the same join point can be wrong, for example if the advice precedence has not been specified sufficiently. As in orchideo the execution order is nondeterministic if no precedence is specified [?], runtime execution order problems can be hard to identify and reproduce. Section 4.5 describes how to support debugging of this problem.
2. *Failure to establish expected postconditions or preserve state invariants*: defects in the action implementation or in the orchideo|engine can cause postconditions or state invariants to be violated, that would be met if aspects were disabled.
3. *Incorrect changes in the exceptional control flow*: exceptions occurring during action invocation and execution are caught by the orchideo execution context, and propagated to the domain application within an encapsulating `ExcecutionInterruptedException`. Additionally the actions executed so far in the current execution context are rolled back by calling the `Action.undo()` method existing for this purpose [?]. The `ExcecutionInterruptedException` should therefore always be handled on action invocation. The encapsulated exceptions depend on the session configuration and must be handled along with changes in the configuration. Unfortunately this is not enforced by the orchideo framework and can therefore easily be neglected, resulting in either unhandled or undetected exceptions. The runtime rescue and trace view plug-ins support the debugging process if confronted with the resulting problems (see sec. 4.6 and [?]).

Debuggability of orchideo as an AOP Framework. As stated in [18], an AOP debugging solution should enable either debug intimacy, or debug obliviousness, each when desired.

Debug intimacy, i.e. the ability to debug all activities introduced by the usage of the AOP framework, including the injected code, is fully supported by orchideo as long as the source code of orchideo|engine, orchideo|objects and the custom aspects is available to the developer. The only limitation lies in the lack of support when it comes to prediction of the runtime behavior and comprehension of the weaving process at runtime. Using static analysis the weaving can be predicted to a certain extend [?]. A possible solution to runtime comprehension is proposed in Section 4.5.

Debug obliviousness in constrast describes the possibility to hide all AOP activities during debugging. A solution for orchideo to support debug obliviousness has been implemented and is described in Section 4.1.

A debugging solution should also preserve the base program's debug information. As orchideo is a noninvasive framework this property is inherently given.

Dynamism and aspect introduction. These properties are not as easily to achieve within orchideo. They refer to the possibility to enable or disable aspects, and to the possibility to introduce new aspects, both dynamically at runtime. This supports the isolation of failure causes by dynamically enabling debug aspects, and by ruling out specific aspects during error search. Aspect weaving in orchideo is performed fully dynamically at runtime based on the join point and advice information configured in the session configuration [?]. This information is generated once upon session creation and never changed later at runtime. To support the above properties, this join point and advice information has to be manipulated at runtime. The programmer theoretically can do this, but as a matter of fact it is not a practical solution because the orchideo framework does not provide any convenience for this. As a workaround, during application development typically another session configuration exists that is used for debugging purposes.

Runtime modification is a property generally important to all debugging solutions. It provides the possibility to modify code at runtime that is automatically applied. This enables the programmer to quickly add trace statements, or try out a bug fix without the need to restart the application. For invasive AOP frameworks [?] this can be tricky to implement [18], whereas orchideo already features this property and is only restricted by the underlying Java hot code replacement capabilities [1].

Fault isolation describes the debugger's ability to isolate the fault location— whether it lies within the base program's code, the aspects code, or code of other AOP activities. This property is strongly connected with reproducibility and the ability to automatically rule out specific aspects. To determine whether aspect weaving is erroneous— which implies a defect in the orchideo|enigne code —static analysis of aspect precedence as described in [?] can be used. Other framework activity may not be as easy to separate from aspect or base code faults. Currently the debugger does not automatically point out runtime states that contradict the results of the static analysis. This would help to automatically detect failures in the orchideo|engine.

The separation of base and aspect code is not intended in orchideo. The base program usually cannot be executed without certain aspects enabled, as the main target of orchideo is to provide a MDSD implementation through the aspects defined in orchideo|objects. Executing the base program without those aspects does not make any sense. Some aspects can be disabled though to automatically determine whether or not the defect lies within such an aspect. An orchideo application developer must rely on the correctness of the provided orchideo|objects aspects.

[1] http://java.sun.com/j2se/1.4.2/docs/guide/jpda/enhancements.html: "HotSwap" Class File Replacement

Conclusion. orchideo, just as most other AOP frameworks does not provide ideal debug support. For some of the problems debug tools were developed within the scope of this project work and are described in the next section.

4 Debug Tools for orchideo

To provide better debugging support for orchideo application developers and orchideo|engine developers we needed to create interfaces to provide important information of the running application. This is the first step in elevating the level of abstraction and support debugging of orchideo applications with focus on AOP debuggability.

With "application developers" we refer to developers using the orchideo framework to develop domain specific applications. With "engine developers" we mean those who develop and improve the orchideo|engine and framework. In some cases developing an application may also include developing custom aspects. "Aspect developers" are programmers who need to do this. They are specialized application developers who need a deeper understanding of the orchideo|engine, but not as deep as engine developers.

The concrete solution and information visualization depends on the concrete problem to be solved. In this section each such problem will be described. Then we propose our solution and describe its implementation in detail. This is followed by an evaluation of the solution and a discussion of work that still needs to be done.

All solutions have at least one thing in common: They are all implemented as Eclipse plug-ins. This enables good integration into the existing development framework. The Eclipse plug-in architecture has been introduced in [45] and is taken as known in this section.

4.1 Step Filtering

The step filter plug-in targets the problem of debug obliviousness: orchideo is a complex framework that ships with a lot of code in the orchideo|engine and orchideo|objects that the application developer is not responsible for or interested in. The same situation applies for code generated by the orchideo framework, for example during model creation. During modeling and coding phases the developer can avoid this orchideo code, but not while debugging. The Eclipse debugger does not know which code is interesting to the developer and which is not. We provide a plug-in that makes the debugger ignore code from the orchideo framework with just a single click.

Problem Statement. An application developer debugging an orchideo application interactively may step into classes that he did not implement and is not interested in—he only wants to use them. A good example for this are all aspect actions, like the `CreateObjectAction` which is invoked by `ObjectAspect.createObject()` to create a new domain object.

The developer does not want to debug the method and is not interested in seeing its implementation. He has to take the correctness of the method as granted because he probably does not understand the complex processes in the orchideo|engine. Therefore he does not accidentally want to step into the action implementation (like when doing a "step return" from a deeper stack frame).

Solution. Eclipse provides a mechanism to ignore stack frames, method invocation states on a call stack [64] of certain classes or packages when debugging. It is called *Step Filtering*. If the step filters are configured correctly and the debugger supports them, the developer does not get in frames he is not interested in when stepping through the program execution.

The Java Development Tools provide step filtering support for Java applications. Such a step filter is a full package or class name, or a partial name defined as a regular expression. The filters can be configured in a preference dialog. Step filtering can be enabled or disabled for the current workspace, using a toggle button in the debug view. For Java applications the user can configure a list of step filters that will be stored in the workspace preferences. Using check boxes individual filters can be enabled or disabled. Disabled filters are ignored in the debugging sessions and are stored only for the purpose of quick re-enabling in the preference dialog. See the table below for some example filters.

Currently Proposed orchideo Step Filters
de.exxcellent.ocl*
de.exxcellent.orchideo*
org.eclipse*
lpg.lpgjavaruntime*
ActionImpl
AspectImplBase

Currently the orchideo application developers are encouraged to configure the step filters themselves. Of course configuring the list manually is error-prone and takes some time. But developers may not even know the possibility to configure the step filters.

We provide a plug-in that automatically configures the proposed step filters as in the list above and can enable or disable them using a toggle button in the Debug View. The current configuration is saved per workspace. Therefore a developer that debugs normal orchideo domain applications in one workspace is free to debug the orchideo|engine itself in another workspace without interferences. The orchideo step filters can be configured analogue to the Java step filters in its preference dialog. This way the plug-in can quickly be adapted to new package names. To update the proposed step filter list for all application developers the Eclipse software update can be used because the orchideo step filters are distributed as a regular Eclipse feature.

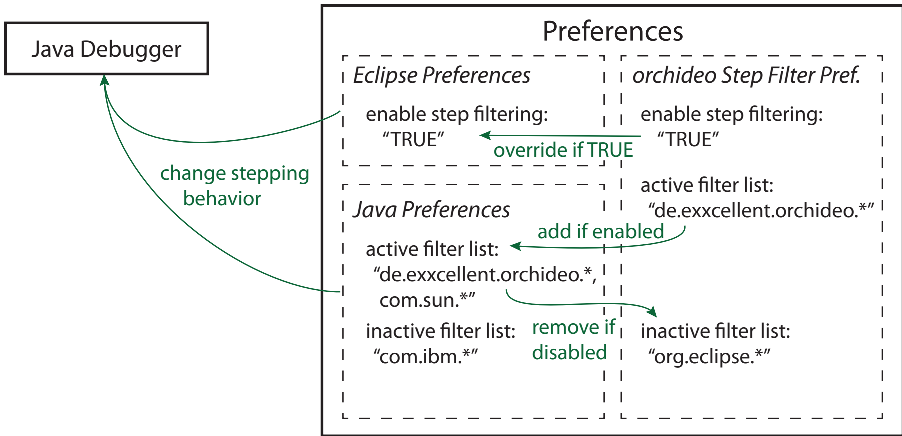

Fig. 2: Preference dependencies for the orchideo step filters

Implementation. The problems we faced when implementing the step filtering plug-in for orchideo were mainly API deficits. Although it is possible to programmatically enable or disable the Java step filtering there is no API to configure the concrete list used during the debug sessions. The list is intended to be configured by the user via the preference dialog. For us the only possibility to manipulate the filters was therefore to change them in the preference store using the Eclipse Preferences API [45].

The JDT plug-in maintains two workspace wide preferences related to step filtering: one comma-separated list of the currently active filters, and one comma-separated list of the currently inactive filters. The preference defining whether step filtering is enabled or not is a workspace wide eclipse preference. When changing the preferences the debugger is informed of the change as it is registered as a listener for the preferences. This way it can automatically apply the new settings.

When enabling the orchideo step filters, the orchideo step filters currently marked as active are added to the list of active Java step filters. Additionally Java step filtering is enabled using the provided API method, because we assume that the developer wants to enable global step filtering when enabling the orchideo step filter—else the enabling would not have an effect. When disabling orchideo step filtering, all orchideo step filters are disabled in the list of Java step filters. Figure 2 illustrates the dependencies of the step filtering preferences.

As we manipulate the Java step filter preferences, the orchideo filters also appear in the list of Java step filters and its preference dialog. Because of the architecture and functionality of the preference dialog inconsistencies can occur when manipulating Java and orchideo step filter lists at the same time. The trade-off we chose was to always override the activation settings for Java step filters with the orchideo lists, because the orchideo step filters are a special case of

step filtering in Java applications. To force this override the orchideo step filter plug-in listens for debug events and will check and override the settings each time the user starts a debug session or steps through the code.

To avoid asynchronism of the orchideo and Java step filter preferences, we synchronize the settings at Eclipse start-up, using the Eclipse early start-up mechanism [45]. Additionally the step filter toggle button is updated at start-up to reflect the current workspace settings.

Evaluation. The proposed plug-in can solve the debug obliviousness problem described in Section 3.2 without preventing debug intimacy to the following extent: If the standard step filters are used all orchideo|engine code is invisible to the programmer at runtime, because it is located inside the `de.exxcellent.orchideo` package. All aspects predefined by orchideo and used for the MDSD implementation are located in the same package and are therefore ignored by the debugger, too. Debug intimacy can be achieved when required at any time by disabling the step filter toggle button. Those three properties cover the most common cases of orchideo application debugging.

If the developer has defined custom aspects for the application, aspect and advice implementation as described in Section 4.1 of [?] are hidden with the standard plug-in configuration. We decided not to automatically add filters for custom aspects, because typically the aspects have to be debugged along with the application. To nevertheless hide a custom aspect's implementation, the programmer can add the corresponding aspect package to the filter list in the preference page for the orchideo step filters.

4.2 Constraint View

The constraint view plug-in shows whether the orchideo objects meet their constraints. This elevates the level of abstraction during debugging to the model level as discussed in Section 3.1 and provides the important information about the constraint state. For instance it helps the programmer to know if the next commit to the database would fail because of an invalid object state.

With *orchideo domain objects* we mean objects that are instances of domain models created in the orchideo development environment. Those objects are all adaptable to the orchideo type `XObject`, which is the object representation manageable by the `ObjectAspect`.

Problem Statement. When developing an orchideo application developers usually define constraints in their domain models [?]. One of the most common reasons for an action like `HibernateAspect.commit()` to fail is a constraint violation: The objects maintained by orchideo cannot be committed to the database because at least one object does not fulfill its requirements.

Assuming there is an application that uses a domain model including a class `Customer`. A customer has two attributes: a first name and a last name. The model constraints that both of those attributes must exist. The orchideo

```
Customer customer = (Customer)
    objectAspect.createObject(Customer.CLASS);

customer.setFirstName("Peter");
// customer.setLastName("Miller");

hibernateAspect.commit();
```

Fig. 3: The last statement will cause an `ExecutionInterruptedException` because the customer object is missing its last name.

framework automatically generates two cardinality constraints for this class: a `firstNameCardinalityConstraint` and a `lastNameCardinalityConstraint`. Both constraints have an upper and a lower bound of one. At runtime an instance of this class may be created using the `ObjectAspect`. If the session this aspect belongs to also has enabled the `ConstraintAspect` and the `HibernateAspect` the code snippet in Figure 3 will fail with an `ExecutionInterruptedException` because the customer object is missing its last name attribute.

In this simple example the cause of the problem is obvious. In a more complex situation with more objects and complex constraints the developer debugging the application now has to check all objects in the session whether their constraints are met. He could check the constraints manually by comparing the objects' attributes and the model requirements. The developer has to switch between the model or diagram file and the debugging session to get the information. This method is not well applicable to complex situations. Reading the orchideo failure trace or using the Trace View plug-in (see [**?**]) can help to identify the constraint that is violated and the object id of the invalid orchideo object. Matching all this information manually takes time and is error-prone.

Requirements. A solution to this problem is to provide a tool that automatically checks the orchideo constraints at runtime. It has to fulfill at least the following requirements:

Constraints have to be checked every time a change to the domain objects was applied. This means to check after every debug step and every time the user executes code on the target VM. More commonly also external changes should be considered as the system may be accessing a shared database.

The tool should give visual feedback whether constraints are violated. Changes in the violation state from one step to another should be marked, too. For an object it should list all constraints and their violation state. The user should have easy access to information that helps him to understand the cause of the violation. If possible the developer should not have to leave the debug perspective or switch to other files. Important information may include:

- the constraint type [**?**],
- all properties that influence the constraint's fulfillment (*triggers*),

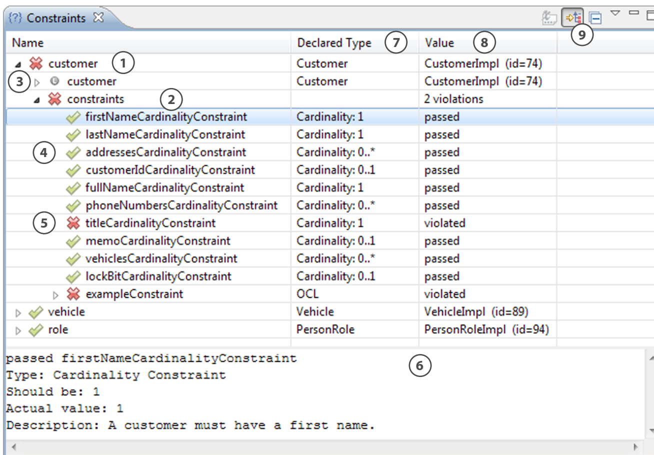

Fig. 4: Screenshot of the constraint view at runtime

– the current state of the triggers,
– the OCL expression for OCL constraints,
– the method checking the constraint and a *jump to code* action for Java constraints

The solution should integrate with the Eclipse debug perspective and debugging tools. It should have the "look and feel" of traditional debug elements in Eclipse.

The tool should not impair the performance significantly when debugging an application. If the user had to wait very long for the results of the constraint checking just to continue debugging, he would probably disable the feature and miss constraint violations.

Solution. We provide an Eclipse view that loads itself into the debug perspective and that shows for a set of objects whether they meet their constraints. The constraints are re-checked after every stepping, breakpoint hit, and remote code execution like when using the display view to execute code. The developer can stay in the debug perspective and does not have to switch back and forth between debug session and model file. Figure 4 shows a screenshot of the view.

The view has a tree similar to the tree in the Eclipse variables view, listing the orchideo objects and a symbol to show whether the constraints of this object are met ①. Each object can be expanded to list its constraints ②, and each constraint has a symbol to show whether the objects passed the check. In Figure 4 there are three orchideo objects. The first one, encapsulated in a variable called `customer` has some constraint violations. The constraint labeled ④ passed, whereas the constraint ⑤ is violated.

The Java variable itself is listed, too ③. Therefore all information about the orchideo object variable that is usually accessed using the variables view, is also

visible in the constraint view. This way the developer does not need to switch views and find the object in the variables view to get additional state information. Furthermore the view itself behaves very similar to the Eclipse variables view:

- It also has a detail pane ⑥, showing extended information about the selected variable, like the number of constraints and the number of constraints that are violated, or about the selected constraint itself. Information about a OCL constraint includes the OCL expression, whereas for cardinality constraint the number of actual and required values assigned to the property is shown.
- The user can enable columns that show additional information in each line along with the name of the variable or constraint. The columns available are the same as in the variables view. For instance the *declared type* ⑦ and the *actual type* show the type of the constraint. For cardinality constraints the former also shows the number of required values for the constrained property. The *value* column ⑧ shows whether a constraint is met. The *instance ID* column, reserved for object IDs in the Java virtual machine, is empty for constraints. For a Java variable, columns and detail pane show exactly the same information as in the variables view.
- The view actions are adopted to the variables view. For instance there is a button that toggles the logical structure state ⑨.

As thousands of domain objects may exist in an orchideo session object it is not applicable to check all constraints of all objects in the session each debug step. The current solution checks objects on the current stack frame only. This is a first quite simple to implement solution that has some practical advantages: The objects on the current stack frame most often are of highest interest. The developer already has isolated the object type and now wants to find out why such an object is taken into an invalid state. Additionally the performance is not impaired significantly.

Implementation. The `ConstraintView` class inherits from the `VariablesView` class to achieve good integration with the standard debug views. On debug state changes it gets automatically informed along with the other debug views and can react on it. The `setViewerInput()` method is called with input objects representing new debug states, for instance new threads being spawned or the activation of a new stack frame. A Java stack frame represents the current method invocation state on a thread's stack, including all local and static variables and the `this` variable.

Model Objects. The variables view lists `IJavaVariable` objects in its tree viewer, that have appropriate content and label providers. In the constraint view, we display `ConstrainedObjectVariable` objects, encapsulating the original variable and the list of constraints for that object.

Figure 5 shows an overview of the model entities used in the constraint view. The `ConstrainedObjectStackFrame` is the encapsulating input passed to the tree viewer. It is not displayed itself in the view. It holds `ConstrainedObjectVariable`

entities, which are the top most objects displayed in the view (① in Figure 4). It encapsulates one `ConstrainedObjectJavaVariable` ③ and one `ConstraintList` ②. A `ConstrainedObjectJavaVariable` simply holds its original `IJavaVariable` and encapsualtes its children in instances of `ConstrainedObjectJavaVariable`, too. Due to poor extensibility of the default content providers, this is required for the content provider mechanism to work (see below).

The `ConstraintList` can hold several `Constraints` ④, ⑤. The `Constraint` class is intended to be overwritten for the special constraints. Java constraints currently do not need to inherit, though, because they do not define any special behavior yet. Cardinality constraints override the *required value string* and *actual value string* behavior to additionally display the upper and lower bounds. OCL constraints have their OCL expression attached as a child entity.

Evaluating Constraints. As the Eclipse Java Debugger implements the client side of the Java Debug Interface, the variable values we can get from a stack frame have mirror objects [65] to the real objects in the debugged Java virtual machine. Those mirror objects implement `com.sun.jdi.Mirror`, a proxy used by the debugger to examine or manipulate entities in the virtual machine. `ObjectReference` is a specialized interface implemented by mirrors that point to objects. The constraint view uses this interface to evaluate whether an object on the stack frame adapts to the orchideo `XObject` interface (i.e. is an orchideo domain object), and whether it fulfills its constraints.

To evaluate the constraints for each orchideo object its session, object aspect and constraint aspect must be determined. Then the constraints can be enumerated and checked. As the remote method invocation is very inconvenient, we implemented a helper class that provides more abstraction from the `Mirror` API.

To improve the runtime performance constraints are only checked if necessary. The icon next to a orchideo object shows a red cross if at least one constraint is violated, or a green check mark if all constraints of that object are satisfied. Therefore probably not all constraints need to be checked to determine whether there is a constraint violation. If a constraint state or another property is evaluated, it is cached and does therefore not need to be checked again until the debug state changes (for instance because of a step event).

Content and Label Providers. The Eclipse debug views use a strong Model-View-Controller (MVC) pattern. Each model object type that should be displayed in the debug views must have a content and a label provider. Those providers are registered at the global adapter manager. The input proxies used by the debug views determine the matching content and label providers by searching for appropriate adapters.

The content provider, implementing `IElementContentProvider`, is responsible for defining the sub items of a given tree item. The label provider, implementing `IElementLabelProvider`, defines how to display a certain item. Subjects to be defined are text color, background color, label texts in the different columns and an icon to be displayed at the left of the line. We therefore defined content and label providers for the model object types mentioned above. Additionally we

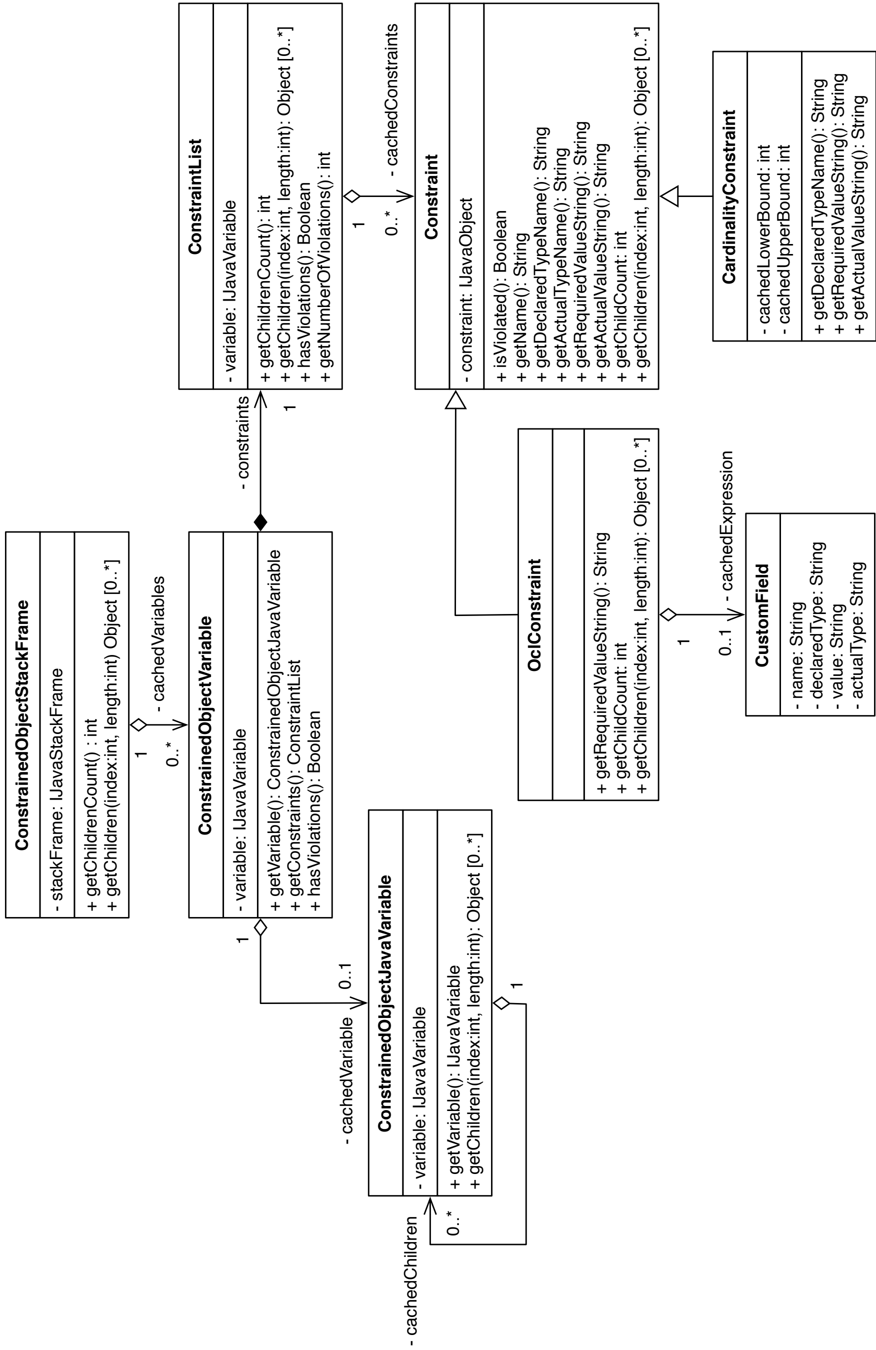

Fig. 5: Simplified class diagram of the constraint view model hierarchy

had to create an adapter factory used to register the providers at the adapter manager.

Detail Pane. Detail pane integration is achieved using an extension point for this purpose. The extension defines an `IDetailPaneFactory` implementation to be used as factory for the above model classes. The factory creates a detail pane which is capable of displaying extended information about the currently selected object in the constraint view. The `DefaultDetailPane` and its corresponding `DefaultDetailPaneFactory` provided by the Eclipse platform are capable of displaying information about `IVariables` and `IExpressions`. We inherited from those classes for our own detail pane and factory. By overriding the `display()` method we added the ability to display detailed information about our model objects, too.

Synchronization. As mentioned before all debug views are updated at the same time. To check the constraints of the orchideo objects at runtime we need to operate on the suspended thread object and on the variable objects. But other debug views also operate on the same objects. It is required that two such operations do not happen concurrently, else an exception is thrown.

Unfortunately, there is no synchronization mechanisms in the Eclipse debug plug-ins designated to solve this problem. We had to implement an external synchronization mechanism that ensures that the view updates do not happen concurrently. The `SynchronizingJobManager` class from the `synchronization` package is responsible for this.

Synchronization is achieved using the `IJobManager` and `IJobChangeListener` interfaces provided by the Eclipse platform. The debug view updates are all performed asynchronously in so-called `Jobs`. The `SynchronizingJobManager` registers itself at the workspace for job change events. Each time it takes notice that a job of the other debug views is about to run it ensures that the constraint view is not updating at the moment. It may suspend the job and put it into a queue. This mechanism is implemented vice versa for a constraint view update job that is about to run. This way these update jobs should not run at the same time.

Future Work. At the time this paper was written the above synchronization is not working properly. The current implementation is also very dependent on a specific debug plug-in version, because the jobs to be synchronized are identified by string literals that are defined in the debug plug-in and are not accessible for external plug-ins like the constraint view plug-in. The synchronization mechanism is therefore subject to change.

Apart from that, the objects listed should not be taken from the current stack frame only. As stated before it is not possible to always list all objects of a session. At least recursive search should be implemented: Starting from the objects in the current stack frame their properties should be searched for orchideo objects and displayed appropriately until a given recursive depth is reached. This solution has to handle circular dependencies and use a good visualization technique.

Another possibility would be to provide the possibility to define expressions to be evaluated like in the Eclipse expression view. As the expression view class also inherits from `VariablesView` this might be not as hard to implement as it may suggest at the first thought. Though we have not yet explored this option further, it obviously provides the programmer much more freedom in choosing the objects to be checked.

In the requirements section the need for two types of additional information about the constraints was proposed: There currently is no action to jump to the code of a Java constraint implementation and the properties that act as triggers for the constraints are not directly accessible either. Those two features should be implemented to make the usage of the plug-in more convenient and reduce the need of switching to the model file in order to debug an application.

Evaluation. Currently the constraint triggers are not accessible in the view, because this is only profitable for complex constraints and has a minor importance. A *jump to code* action for Java constraints has not been implemented either, because it is more complicated to implement and there are only few Java constraints.

In the current version the constraint view enables the programmer to easily track the constraints of a certain object. The limitation that only objects on the current stack frame are displayed will be eliminated in near future using one of the above proposed solutions. This is important in orchideo as not only objects on the current stack frame can cause an operation to fail, but on a hibernate commit the whole object network has to be in a valid state. Apart from that the elevation of the level of abstraction concerning the constraints is provided by the plug-in and improves the debugging experience of orchideo applications. Our customers already use the plug-in for this purpose.

4.3 Session View

The session view targets to improve the debugging intimacy. When debugging the orchideo|engine it is necessary to have full insight into the engine activities. As long as the source code of the engine is available it is theoretically possible to debug the whole AOP framework. The current session state is also important, but this information is not as easily accessible while debugging. The session view plug-in enables the programmer to easily access information about the current orchideo sessions.

Problem Statement. If an engine developer suspects the existence of a defect in the aspect weaving implementation, he has to inspect the internals of the respective session object. Especially the list of advice, join points and the execution history are of interest.

Unfortunately session objects are not directly accessible. Until now, engine developers had to search the stack frames for orchideo domain objects which hold a reference to their session. This process is inconvenient and error-prone, because often multiple sessions exist in an orchideo application.

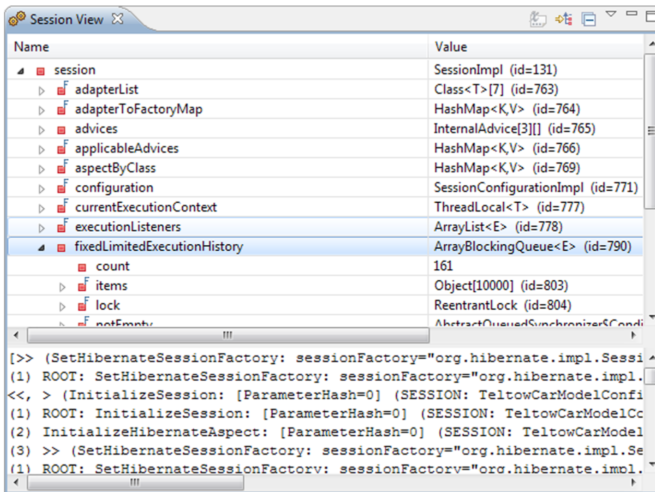

Fig. 6: Screenshot of the session view at runtime

Solution. The session view plug-in provides an Eclipse view that lists all session objects found in the current Java thread. The view automatically integrates itself into the debug perspective. Sessions are ordered by priority, as the customer demanded. The `this` object has highest priority if it is an orchideo session itself. Otherwise sessions near to the current stack frame have high priority, session objects farther away have a lower priority, determined by the number of indirections needed to reach the session.

This way the developer can easily tell which session is the "current" session. If he wants to see details about another session object he can study it nevertheless.

The session view furthermore acts like the Eclipse variables view, and therefore also adopts its advantages without reimplementing them, for example: View updates are fully asynchronous and do not block the Eclipse IDE. Attributes of a Java object can be inspected because the object is displayed in a tree structure. Changes to a variable or attribute are automatically highlighted to advise the programmer of this change.

Figure 6 shows a screenshot of the session view during a debug session. It contains one session object that has been expanded to view its attributes.

Implementation. The `SessionView` class also inherits from the `VariablesView` class. When `SessionView.setViewerInput()` is called, the view uses a helper class to recursively find all sessions in the thread of the provided stack frame. The search begins at the top most stack frame, checks the `this` variable and its properties, the orchideo domain objects and orchideo aspect objects and their properties for a session until a given recursion depth is reached. When all session objects are found they are passed as `IVariables` to a custom `IStackFrame` implementation that is only used to be displayed as model object in the session

view. This custom stack frame is then passed to `super.setViewerInput()` as the input to the tree viewer.

`IStackFrame` and `IVariable` objects can be displayed by the view because appropriate content and label providers are registered by the Java Development Tools plug-in.

Problems and Future Work. To achieve the displaying of `IStackFrame` and `IVariable` objects without providing own content and label providers it is necessary for the view to act as if it was a Variables View. This means it has to set its debug context ID, a string needed for initialization of the input provider service, to the same as the Variables View. As the same ID is also used to determine the ID of the preference responsible for persisting the column selection and ordering in the view, problems can occur when both views try to use this preference.

In the constraint view this string was changed to a new unique identifier to solve this problem. Unfortunately, the content providers check the debug context ID of the view. If it does not match either the variables view's, expression view's or launch view's context ID they will not work for the provided input object—although they are appropriate for it. Therefore classes that encapsulate the original `IVariable` objects have been implemented. The session view is an older view, and therefore those classes are currently not available to it. To solve the column presentation problem this will change in future and the session view will also get a unique debug context ID.

4.4 Logical Structures for orchideo|engine Objects

The logical structure plug-in improves the above session view plug-in by enabling the engine developer to filter information about the session object and other encapsulated entities.

Problem Statement. When an engine developer has isolated a session object he wants to analyze, he is confronted with many attributes of that objects. But typically only five attributes are of interest:

- The session configuration describes which aspects are enabled in the selected session.
- The session object holds a list of advice and
- a list of join points for the current session configuration.
- The current history that log which actions have been invoked lately. Dependent on what the user has enabled the history is either saved in the attribute for the *fixed limited execution history* or in the *undo marker based execution history* attribute. See [7] for more information about the history types.

The lists of advice and join points in turn are actually arrays of arrays. They are structured in a way that enables fast accessibility at runtime which is crucial for performance optimization of the orchideo|engine. Unfortunately this structure

often interferes with the engine developer's need to grasp the configured advice and join points. Figure 6 shows a screenshot of an expanded session object at runtime.

Solution. Obviously the information contained in the session objects should be preprocessed and filtered before shown in the variables or session view. Nevertheless it should still be possible to see the full information without filtering, if necessary.

The Java Development Tools plug-in provides a solution for this use case: For a variables view the user can enable so-called *logical structures*. Logical structures permit the user to define alternative display options for object instances of a certain type. The JDT plug-in for instance uses this mechanism itself to render collection objects in a clearer way. Instead of showing the internal attributes of a collection the user sees the simple array representation, because he is probably only interested in the number, identity and order of the objects in the collection. To view the internal structure of the objects it is still possible to disable the logical structures at any time. Figure 7 shows the toggle button (①) that enables or disables the logical structure rendering.

The customer requested a logical structure rendering for session objects that includes the above listed entities. There should be only one history attribute that reflects the current history. Another attribute should exist to show the type of the current history. Additionally the list of advice was to be separated by the advice type that is either *before*, *after* or *around*. The list of join points should lose its duplicate entries it had for performance reasons. The attributes of session configuration objects should be reordered to display the most important information, the name of the configuration, at the top. Our logical structure definitions has all those attributes, as shown in Figure 7. The screenshot shows a variables view with a session object expanded. As the logical structure rendering is a feature provided by the variables view and its content providers the orchideo logical structures of course also work in the variables view itself and in other views derived from it.

Implementation. Logical structures can either be configured in the preferences dialog, or defined using the `org.eclipse.jdt.debug.javaLogicalStructures` extension point. To make the logical structures easily distributable we decided to use the latter way. Listing 8 shows an excerpt of our logical structure definition for the orchideo session type. The `javaLogicalStructure` element defines the type that can be rendered differently. In the listing it is the orchideo `Session` interface. As it is only an interface we set the `subtypes` attribute to `true` because else the definition did not have any visual effect. The `variable` elements define which attributes the session objects get and in what order they are displayed in the tree viewer. The `value` attribute can be any valid Java code that can be executed in the context of the object itself. The result of the execution will be shown as the value of the defined variable.

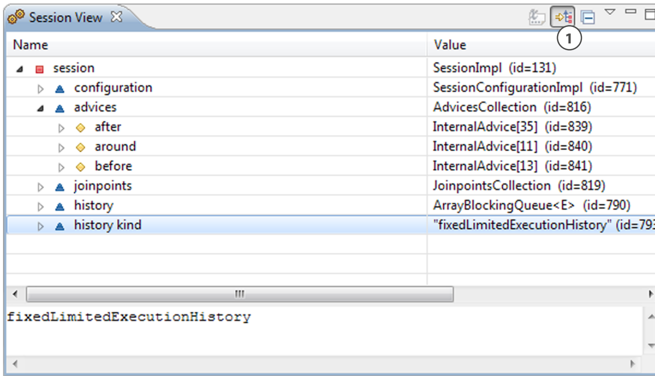

Fig. 7: Screenshot of the session view at runtime, with enabled logical structures

We decided to create a helper class, `SessionLogicalStructureProvider`, to calculate the value that is to be displayed. This way we have as few code as possible defined in the `plugin.xml` file itself that cannot be debugged. The `getAdvices()` method for instance creates an object of a new type that is only to be used in the logical structure mode. This enables us to alter the displaying of the array containing the array of advice by using a logical structure definition for this type, too. As the code that calculates the attribute values is executed in the session object's context, this helper class has to be defined in the same plug-in as the `Session` type. Therefore we had to extend the `de.exxcellent.orchideo.engine` package by our helper classes and the extension point definition.

Evaluation. Together the session view and the logical structure plug-ins provide a very simple possibility to reach internal session information for all session objects. The information is filtered and sorted so the developer can focus on important information only. Both improve the debugging experience by supporting debug intimacy.

4.5 Execution Context Visualization

In this subsection we propose a possibility to debug incorrect aspect compositions due to wrong advice definitions or engine errors. Therefore it is intended to be used by aspect developers or engine developers.

Problem Statement. An orchideo aspect developer has to define the advice pointcuts and precedence when defining a custom aspect. When using this aspect it might be possible that it is not invoked, or invoked at the wrong time. The error can either be caused by the orchideo|engine that wrongly weaves the advice, or by the advice definition and implementation. Debugging such issues is very hard because the developer quickly has the impression to be stuck in glue code that is hard to understand.

```
<javaLogicalStructure
    description="Reduces information of an orchideo session
       to its active history and its advices respectively
       its actions."
    subtypes="true"
    type="de.exxcellent.orchideo.engine.Session">
  <variable
    name="configuration"
    value="return de.exxcellent.orchideo.debug.engine.
      SessionLogicalStructureProvider.getConfiguration(this);">
  </variable>
  <variable
    name="advices"
    value="return de.exxcellent.orchideo.debug.engine.
      SessionLogicalStructureProvider.getAdvices(this);">
  </variable>
  <variable
    name="joinpoints"
    value="return de.exxcellent.orchideo.debug.engine.
      SessionLogicalStructureProvider.getJoinpoints(this);">
  </variable>
  <variable
    name="history"
    value="return de.exxcellent.orchideo.debug.engine.
      SessionLogicalStructureProvider.getHistory(this);">
  </variable>
  <variable
    name="history kind"
    value="return de.exxcellent.orchideo.debug.engine.
      SessionLogicalStructureProvider.getHistoryKind(this);">
  </variable>
</javaLogicalStructure>
```

Fig. 8: Example logical structure definition

Solution. The information the programmer is looking for is nevertheless possibly available: The session objects hold a history of execution contexts, which encapsulate the actions woven and even their order. If an action is currently being invoked, the current execution context is also available in the history. It is actually the last one in the list.

The developer can use the session view described above to look into a certain session and its execution history. Using this information the developer can determine whether or not certain aspects have been invoked, and in which order. If the developer would for instance set a breakpoint in the `do()` method of his custom aspect he can find out which actions have been invoked so far through the history. This information extends the information from the static analysis of advice precedence [**?**].

Unfortunately, the execution context is currently not visualized properly. They have the same format as the exception trace described in [?]. A better solution would therefore be an integration of the trace view described in [?] into the interactive debugging process. The trace view provides a more comprehensible visualization of the execution context, and could automatically update itself with the information from the current context. This would help the programmer to keep track of actions currently being invoked.

Currently the performance of the trace parser is not sufficient for repeatedly re-parsing execution contexts on each debug step [?], which would lead to undesirable latencies. On the other hand the developer can manually access the execution context trough the session view and paste it into the trace view to visualize the current execution context, when required. Due to the poor cost-benefit ratio we therefore did not explore the automation of this process any further, but instead concentrated on the value creation for our customer with regard to other unsolved problems.

4.6 Runtime Rescue

The runtime rescue plug-in is more passive than the other plug-ins. It targets the debuggability problem no. 3 described in Section 3.2: Because of the architecture of the orchideo|engine, exceptions occurring in orchideo applications are thrown deep inside the engine and not in the application code. The actual cause of the problem is forgotten and can be found only in the exception trace. This plug-in helps to find the original exceptions causing an `ExecutionInterruptedException` while the application is held alive and the objects can still be inspected to identify the problem causing the exceptions.

Problem Statement. In orchideo applications exceptions that are thrown during execution of an action are caught by the orchideo|engine and written to a stream. Then the actions in the current execution context invoked so far are rolled back to return to a valid state. The engine is then left with an `ExecutionInterruptedException` that holds the initial exception information. So when the user catches the exception the erroneous state inside the engine has vanished and the developer has to read and understand the orchideo exception trace to find the manner and origin of the initial exception. As described in [?] especially the comprehension of the trace was not easy to accomplish until recently.

Solution. For interactive debugging it would be more desirable to make the location of the original exception more comprehensible. As multiple problems can occur during one action invocation, these multiple problems must be displayed coequally [?].

Using the given capabilities of the Java Development Tools the user could define exception breakpoints to be informed of caught and uncaught exception occurring in the application. Defining the breakpoint for all types of exceptions

will probably hold the program too often. If the type of the exception is already known (i.e. if the problem has already occurred and should now be reproduced) this more special exception type can be defined as the break point condition. If the exception occurs the debugger will hold exactly at the location where it is thrown. This could help the engine developer to debug the orchideo framework if he suspects a defect in it. The orchideo application developer on the other hand is probably not interested in this location, but only what operation has caused this exception.

Our approach helps to find this location: We still make the orchideo|engine catch any exception that is thrown during the action execution. We can then use the capabilities of the orchideo framework to roll back the actions performed in the current execution context, so the application is brought back to a valid state. To fully roll back the application state, the thread's stack is un-winded to return to the application code instead of being stuck inside the engine code where the `ExecutionInterruptedException` is thrown. The debugger will hold at the beginning of the method of the application code that called the action which finally caused the crash and the line of this call is marked to indicate the problem. This process of returning to a valid state from an erroneous state gave the plug-in its name, *runtime rescue*.

As the problem actually manifests itself in one or multiple exceptions we can extract the stack trace and display the failure information to the user. After the runtime rescue has taken place the user is therefore at the point in the application that, if re-run, may cause the exception again. The developer can then use the debugging tools that have already been presented to analyze the current state and understand why the failure occurred. Especially the stack trace bears important information that is automatically processed using the trace view plug-in which is described in [**?**] and [**?**]. If the user wishes to test the application behavior on occurrence of exceptions, the runtime rescue feature can of course easily disabled and re-enabled using a toggle button in the debug view.

Implementation. When the user enables runtime rescue first of all an exception break point is created that will break the debugger on caught and uncaught exceptions of type `ExecutionInterruptedException`. The plug-in will then register a listener on debug launches to be informed of the start or stop of debugging sessions. The plug-in uses early start-up to determine whether runtime rescue is initially enabled and to register the break point and its listener, if appropriate. When a new program is launched in debug mode we start a thread that regularly checks whether a thread of a currently debugged Java program has stopped at the previously set exception breakpoint. When no more debugging sessions are running, the thread is stopped to not pointlessly impair the Eclipse runtime performance.

If the check is positive, the runtime rescue takes place: The orchideo trace is extracted from the thread that has the exception. The extraction is achieved using JDI mirrors just as in the constraint view plug-in. This way the current orchideo|engine execution context is dumped to a string and passed to the de-

bugger. This trace is then printed to the debugger console and parsed to be displayed in the orchideo trace view.

Additionally the thread's stack is un-winded to the top most method that has source code available in the current workspace. This location is most probably the last position before the exception which the developer can understand and have influence over. If no such location is found, the thread is resumed and therefore the exception is thrown.

Problems. The runtime rescue unwinds the stack until it reaches a method of the application code again. This results in the current instruction being the first instruction of that method. The Eclipse Debugger Framework does not provide any possibility to set the current instruction to the method that has been called and that resulted in an exception. The programmer has to step through the method to reach the erroneous method call. This can lead to side effects which may change the application state and prevent correct reproduction of the failure.

Another problem of stack unwinding is that `finally` blocks are currently not executed. Usually the Java virtual machine takes care of `finally` blocks being executed when an exception is thrown, until a matching `catch` statement is reached. When using stack unwinding this step is skipped. This is especially a problem because the orchideo|engine uses a `finally` block in the `ExecutionContextImpl` class throwing the `ExecutionInterruptedException`. Under certain circumstances skipping the execution of this `finally` block can leave the session in an incorrect state. This will lead to another exception as soon as the user tried to invoke another action in this session.

The general execution of `finally` statements is a problem that we currently cannot solve. Therefore the usage of the runtime rescue plug-in must happen with this detail in mind. Currently the only practical problem is the `finally` block in the `ExecutionContextImpl` class that is not executed. As a workaround this block can be executed manually when a runtime rescue takes place.

Evaluation. In the current version the runtime rescue is able to hold the program execution when an orchideo exception occurs. It automatically parses the stack into the trace view [?] to enable the developer to analyze the problem. The application is held alive by the debugger and the programmer can inspect its state. Unfortunately, the application may be in a state where any further action invocation may make the program crash again. As the application would crash anyway this is only a small disadvantage.

All in all, the objective of the plug-in can be seen as being achieved. Uncaught exceptions due to different aspect activation states are automatically caught and can be analyzed interactively by the programmer. Nevertheless it would be more convenient to reduce undesired side-effects to the application.

5 Overall Evaluation

Our goal was to build tools that assist the debugging process of orchideo applications, aspects and the orchideo|engine. In this section we summarize the benefits of our debugging solutions presented in Section 4.

Through our debug tools we can support the developer in debugging orchideo applications, both from the MDSD and the AOP point of view. The developer is provided additional information to understand the runtime processes in the orchideo|engine, and to inspect the current state of the domain objects. He can focus on the important software parts by hiding framework code and information that is not of interest. Unexpected errors can be caught by the debugger, allowing the programmer to inspect the cause of the error in the living system.

The plug-ins integrate with the orchideo IDE and make use of the mature implementations provided by the Eclipse debug platform. There is of course room for improvements, but the orchideo application and engine developers already benefit from the provided plug-ins. As we made a first big step towards the realization of characteristics good AOP debugger frameworks should provide, their development process is assisted and sped up.

Further research has to be done especially in the field of MDSD debugging support to eliminate the need of hand-constructing specialized solutions for individual MDSD frameworks.

6 Related Work

Cougaar MDA System Debugger. Although there is no general debugging solution for MDSD frameworks, debugging support for individual systems has been hand-constructed, just like the debug tools for orchideo presented in this paper. One such system is the Cougaar Model Driven Architecture System [66]. George et.al. present a model level debugger for this system [59]. The main difference of this debugger to our solution is that a whole new debugger has been implemented on top of the Eclipse Debug Platform [67] to provide the debugging support. This way the authors are very flexible in their implementation but also need to re-implement much convenience for such a debugging solution.

When implementing our tools we could rely on the JDI implementation provided by the Java Development Tools, instead of writing a new Java debugger. This way we were not as flexible as the above debugging solution, but could better focus on the MDSD and AOP problems. The implementation of a new debugger was therefore not an option to us.

AspectJ Development Tools. The AspectJ Development Tools (AJDT) provide tool support for editing, building and debugging AspectJ programs in Eclipse [43]. They provide basic debug support, but some important features are currently not present. Especially stepping into around advice is not supported and setting breakpoints in an around advice does not have any effect.

orchideo as a noninvasive system has a slight advantage here because debug information does not really need to be modified or extended, but the information needed by the debugger is automatically generated due to a normal Java build. Additionally the join point model in AspectJ is much more powerful, but also more complex, and the debugger therefore has to keep track of more information.

Model-Based Debugging. In contrast to model-level debugging, model-based debugging [68] does not emerge from model-driven software development. In model-based debugging models are generated from the existing source code of the software. When debugging the programmer can use additional information and visualization provided by the models.

Model-level debugging as applied by our tools bases on existing models. It supports the debugging process by elevating the level of abstraction to the level the programmer has created his models in. In model-based debugging the level of abstraction is elevated to a higher level than the one the programmer initially defined the program in. It therefore emerges from a different situation and provides merely a convenience, wheres model-level debugging in MDSD frameworks is a requirement for efficient debugging.

7 Conclusion

In our project work we developed several Eclipse plug-ins that can assist the developer during the debugging process in orchideo. Along with the static analysis of advice precedence [?] and the exception visualization [?], we filter and process information that is extremely important to the orchideo application developer to comprehend the control flow and find defects in the software. We saw that orchideo is not the only framework that suffered from missing debug support, especially concerning model-level debugging in MDSD applications and fault isolation in AOP programs.

We discussed that it is very important that developers can debug programs at the level of abstraction they created them. As software development continues to evolve into levels of higher abstraction, there will always be the need for appropriate debug support. On the other hand, programmers must always have the ability to inspect low-level and framework code when required, so they can comprehend the complex processes and understand why their implementation does not behave as expected.

We discussed the implementation details of our debug plug-ins and saw how to integrate with the powerful Eclipse IDE. We extended existing tooling and could therefore generally focus on high level tasks to realize the customer's needs. Our customers already use the tools and take advantage of them in their daily work.

Bachelor Thesis

Post-mortem Analysis of Debug Traces

Tim Felgentreff

Supervisors:

Dr. Michael Haupt, Malte Appeltauer
Prof. Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute,
Potsdam, Germany

June 25, 2010

Post-mortem Analysis of Debug Traces

Tim Felgentreff

Hasso Plattner Institute
Potsdam, Germany
tim.felgentreff@student.hpi.uni-potsdam.de

Abstract. Debugging techniques become ever more versatile and helpful for developers. Platforms like Eclipse, upon which the orchideo suite is built, have support for interactive debugging that allows programmers to inspect systems at runtime. Yet, printf statements are still used even today in cases where interactive debugging methods fail.
Such is the case with the orchideo|engine. Dealing with the output produced in engine's logs is, today, challenging for new users and sometimes impossible. We have built an analysis plug-in which solves this problem through automation.

1 Debug Traces in orchideo

Debug output has been a prime method of debugging ever since serial line terminals came into use and program output could be inspected while the program was running. It has never been subject to any specific method of formatting, no rules have been established as to what constitutes "debug output" [69]. This may partly be due to the fact that although developing and testing programs is taught in programming classes, debugging has traditionally been largely ignored in teaching [70] and most programmers experiences with it is acquired through the experience of writing programs [53].

The orchideo suite, too, uses debug output extensively. Within the engine which is responsible for running aspects, program failures are trapped, debug logs are collected and used by orchideo developers for further analysis.

This paper will discuss reasons for and ways of debugging programs using debug output and briefly compare this activity to other ways of debugging. It will then provide an analysis of orchideo|engine debug output and how we chose to automate this task.

1.1 Origin and Purpose of "printf Debugging" in orchideo

"Printf debugging" is a term coined by the function name for printing to the terminal in the C programming language [71]. Since C programs are compiled to machine code, live debugging of C binaries is severely limited even today [72]. As an alternative to runtime debugging programmers have often used *printf* statements to ask questions about program behavior and print state information to the terminal [69].

Debug code is irrelevant for the program execution and consequently not generally subject to scrutiny in terms of read- or usability. It usually only provides information to the programmer who created it and might not be easily understandable by any other person working on or with the software. Using printf debugging a programmer tries to monitor program state over time. The software in development writes a record of events to a log. Typically these events include such things as message calls, parameters and return values as well as key variable state changes [73]. Given a predefined input, correct variable states can be determined manually and violations can be easily spotted by the developer during test runs.

Over time, a programmer's confidence in the correctness of his solution grows and his debugging needs often shift to incorporate checking control flow and correct integration with the rest of the system [74]. Debug output used to this end often structures method calls and arguments in some way in order to facilitate reconstruction of the program flow through analysis of stack states. This kind of output can quite easily be mapped to desired output given a specific architecture and definitions of collaborations between key classes. Yet, growing systems contain ever more classes and debug statements have to provide increasingly more information to infer runtime state from it. The right decision about what to log becomes a key factor to successful debugging.

1.2 Alternative Debugging Methods

Continued development on programming abstraction levels such as model-driven software development (MDSD) and aspect-oriented programming (AOP) implemented by orchideo seem to aggravate aforementioned problem. However, software progress has added some support for more advanced debugging methods, too. Just like operating systems have provided hardware abstracted memory information for easier state monitoring, high level languages, sophisticated compilers and complex IDEs enable programmers to spot errors early on and interpreted languages in interactive systems allow stepping through code as it is executed [75].

Users of the orchideo framework can already make use of those debugging methods. Since orchideo is built on the Eclipse OSGi framework, the Eclipse IDE's excellent support for Java debugging (using conditional breakpoints and step filters) is available to the orchideo programmer as well.

We have also developed tools to support more specific use-cases to debug orchideo related problems [?]. These tools help solve a decent amount of mistakes which occur when writing applications with orchideo.

In addition, an orchideo application is usually made up of at least two OSGi bundles—one for the domain model and one for the application. This is the default structure provided when running the orchideo project wizard to create a new application. Unit tests, although no explicit aim of this separation, are easy to implement for the domain model and the application separately and one could argue that complete path and parameter coverage should ensure that all pieces of the system work as expected on their own.

1.3 Why keep printf Debugging

First, as software and its complexity grows, collaborations can become increasingly hard to predict and the system can no longer be easily verified [72].

These kinds of systems—examples include database servers with heartbeat, remote control agents in power plants and network intrusion detection systems—are often plain impossible to debug live. Reasons for this are:

– The system is split over multiple processes or even run on different hardware and cannot be stopped and examined synchronously.
– Software execution distributed over multiple processes or even hardware systems is, due to physical constraints, inherently non-repeatable and non-deterministic.
– Multi-threaded systems are prone to latencies and timing issues more than single-thread software. Missing a global "reference state" it is difficult to verify correct concurrent execution.

Such applications may most reasonably be debugged using facilities for collecting and structuring output [73].

Secondly, applications built on top of Eclipse tend to be a complex and often fragile conglomerate of OSGi bundles. Many hidden rules must be followed to prevent bugs in the system and even if the application as one part of this system could be proven correct all collaborations cannot possibly be tested. Developers outside the orchideo core team may be unaware of the API's proper usage patterns, both for orchideo and Eclipse, and introduce errors that are very hard to predict and debug. Such bugs are often only discovered some time after deployment [76].

Deployed orchideo systems fall in both categories, building on Eclipse and interacting with database in knowledge systems on different servers when deployed. Application developers using orchideo have limited access to the collaborating systems and errors can often not be reproduced locally for testing. Based on the above arguments, there is no substitute to tracing and logging.

1.4 Requirements on orchideo Trace Analysis

Developers at ex|xcellent and our own experiences with the TeltowCar [?] testing project provided us with a number of problems our analysis would have to solve. In this section we present those requirements and where they come from. They have been summarized in table 1.4.

Application Developer Support We found it to be common practice at ex|xcellent solutions to log engine errors in deployment and send log files to the developers for review. Errors found in the logs are, most of the time, constraint violations when trying to save dirty objects. In such cases the orchideo|engine uses actions indicating failure to throw an error. An example we have come across most frequently and which, according to developers of ex|xcellent , is the most common

Engine Developer	Aspect Developer	Application Developer
Failures in engine calls to aspects	Errors in aspect code	Errors in application code
Incorrect weaving in regards to session configuration	Improperly defined aspect dependencies	Object validation errors
	Wrong session configuration	Faulty constraints

cause for problems, is the `MarkConstraintViolation` action which is called when a constraint defined in the domain model is violated in application code. In the textual representation these errors could usually be found using a simple `grep` on the output. Constraint problems may arise easily when the developer has not considered all possible changes to domain objects that might have occurred before they are committed to the database. An application developer has to be able to easily conclude from the analyzed stack which constraints have failed on what kind of objects and why.

ex|xcellent solutions gave us a detailed description of the environment in which an application developer requires automatic analysis and from which sources logs might arrive that need to be analysed and have provided logs of different kind to us for dissection and testing purposes:

- Automatically sent files containing logs from remote sites
- Parts of logs sent inline as emails
- Logs intermingled with console output in local sessions

To support developers maintaining deployed applications, we are required to parse legacy traces. This means, that no or only minimal changes were to be made to the engine trace. Our solution had to be able to parse the existing format and not rely on changes to the trace. Any addition to the log format would have to be optional.

Engine Developer Support While application developers rely on the orchideo suite to build their application and are thus not interested in how it works exactly, engine developers are interested in correct execution of the orchideo|engine itself. Errors in the engine may result in incorrect weaving or exceptions in calls to aspects. This means that more execution context must be available in a pre-processed form for the engine developer to deduce errors from traces. Because automated analysis might not always provide all information required, the trace was to be kept in a human readable state for the developers to rely on. Markup languages were not considered human readable enough to be useful on the kind and amount of data the engine trace contains.

Apart from those requirements the engine developers will have to maintain the software at some point. Due to this non-functional requirements were added to avoid third party dependencies and to supply a solution written in Java language.

Aspect Developer Support Aspect developers think in terms of actions and join-points in the orchideo|engine. Aspects in orchideo are configured in a session configuration. This configuration determines what aspects are run. To debug aspects and dependencies between them aspect developers require session information in addition to the current logging to make sure aspect dependencies have been defined properly and the correct session has been used.

On these levels of development exceptions may occur which cause the application to fail and are of prime interest to all developers. Such exceptions and their stack traces need to be extracted and be easily recognizable.

1.5 The Problem with orchideo|engine Traces

The main problem faced when analysing program output is that simply reading logs is not sufficient to understand anything about a program, its execution flow or any errors that have occurred. Just like any kind of information it cannot be directly transferred to the programmer, in whichever way it is presented. Instead, the reader has to construct his own meaning from it [77].

The orchideo|engine is responsible for runing arbitrary actions whenever their associated join-points are called. Each action defines a `do` method which is called when the action is run. Most actions are never called directly by the application developer, but instead are run as advices in a nested tree of advices when the developer request a specific action to be executed (e.g. `HibernateAspect.commit` to submit objects to the database will cause a number of actions to run which check those objects for validity first). As the engine cannot know in which ways any action may fail, it keeps an execution history and catches all exceptions that might occur. In a catch clause, the history is rolled-back by calling all `undo` methods on actions that have successfully been run and an `ExecutionInterruptedException` is generated. The exception has a serialized form of the history and the original exception attached. This allows an application developer to simply catch this one type of exception and deal with it in application code (e.g. by writing to a log) without having to worry about inconsistent system state.

As with many debug systems that have grown into logging facilities the format of an orchideo|engine trace is not formally defined; it is, as we will show, not even decidable. The content of the output was extended over time to accomodate the logging needs of the developers. A typical orchideo trace may contain 10k–20k lines, almost each line representing an action, its parameters and fields, as well as possible nested values and session information (see figure 1). Developers using orchideo have little chance of understanding this output without having a look at the engine itself. Although it contains traditional Java stack traces those are often not sufficient to determine the cause of the engine exception. Different from stack traces, the engine's execution path is not simply linear and the problem is not necessarily to be found near the top or bottom of the trace. We discuss how to improve the usability and expressibility of the orchideo trace in the following sections.

```
  at org.eclipse.equinox.launcher.Main.run(Main.java:1311)
  at org.eclipse.equinox.launcher.Main.main(Main.java:1287)
[MainExceptionTrace:[Thread.java:java.lang.Thread#getStackT
> (Commit: wasCommitted="null" [ParameterHash=0] (SESSION:
(1) CheckValidity: [ParameterHash=0] (SESSION: TeltowCarMod
ParameterValue: [de.exxcellent.orchideo.objects.aspect.core
[no stacktrace available]
(2) >> (CheckConstraints: objects="null" constraints="null"
(1) ROOT: CheckConstraints: objects="null" constraints="nul
```

Fig. 1: An orchideo|engine trace

2 How to deal with Debug Output

Dealing with the problems in analyzing debug is no new idea: logging frameworks like log4j or the JUnit report created by the “junit” task for the Ant build system deal with this problem in different ways. We have looked at various solutions to the problem and how we could use them to devise our own solution to the problems with orchideo logs. In the following paragraphs we will present some of those solutions and discuss how we could apply them to our problem.

2.1 Core Dumps

An OS level debugging technique also used in some Java debuggers is core dumping. Core dumps are snapshots of process memory which are saved to a file in case of an error. A core dump contains all information present in the system at runtime when the error occured and (if source code and the original executable are available) may be analyzed using the same techniques used in step-wise debugging.

Using memory dumps as an alternative error log would enable application developers to use our previously created debug plugins in Eclipse to analyze and present the runtime information at the point in time when the original program crashed in a sand-boxed environment. To do this, we would need to extend the JVM to be able to load *any* memory image into it’s process space, re-create thread states and basically “halt” all re-created threads. Platform and JVM specific memory layout, however, would prevent direct loading of core dumps on another machine. Alternatively, instead of creating a memory dump we could try to serialize *all* reachable objects in *all* threads and deserialize them on the developer machine into a sandbox. A user could then inspect the complete application state as if he were sitting on the suspended system on which the error occurred.

2.2 Logging Frameworks

When logging program execution, not all output is of equal importance. There might be different “levels” of output, for debugging, for general information

about the state, for warnings if problems arise and for logging error messages. The log4j framework is just one example how to provide a flexible way to enable different levels of debugging output at runtime and use different "sinks" as output options for them [78]. This may be used to separate different kinds of messages into different logs to make allow easier filtering and make the error messages problems more visible.

The customer requirement constrained the changes to the log format to a minimum. Given that orchideo applications are deployed and running today and major changes to the log format would not solve the problem for those legacy applications. Employing a logging framework would be no solution for the existing system, but a rewrite of the part of the system we are building the solution for and consequently no solution at all. Using a logging framework would have to accomodate the need to analyse old as well as new logs:

1. Change the engine to use the a framework for logging. This includes adding information to actions—whether they are printing information, warnings or errors—and hooks for configuring the log level of the engine at runtime.
2. Build a list of patterns to convert old logs into the new format.
3. Develop a solution to automatically split logs based on the kind of information.
4. Automate analysis of the new log format.

2.3 Formatted Output

Text formats such as XML [25] or JSON [79] have been developed to be easily parseable and solutions exist to parse these formats. An example where XML is used to structure output of another program is the *junit* task included with the Ant [80] build system.

The Eclipse development environment (and by extension orchideo) offers excellent integration with the JUnit framework and offers a view with the progress and details of unit tests with visualization and inspection for errors and failures.

Most larger software projects ensure tests are run on each change in the software (e.g. upon each commit in a VCS) using some kind of continious integration system. This prevents the use of a view to focus the attention on errors. Short of extending JUnit, logging has to be used to on the CI.

The default output of JUnit is, while easily readable, not easily inspected automatically. The Ant junit task task logs test results as XML files for later analysis by other tools. The CI server Hudson in turn includes a plug-in which displays the results recorded in the XML files and keeps a log of past test runs (figure 2).

This situation is the result of custom logging in the case of the JUnit framework which now has to meet the requirements of new software development methodologies where the log cannot be viewed while the tests are run. The Ant developers have provided a more structured output, which in itself is not much of an improvement in regards to readability, but much easier to parse by other tools. This problem is similar to the problem faced by orchideo developers as

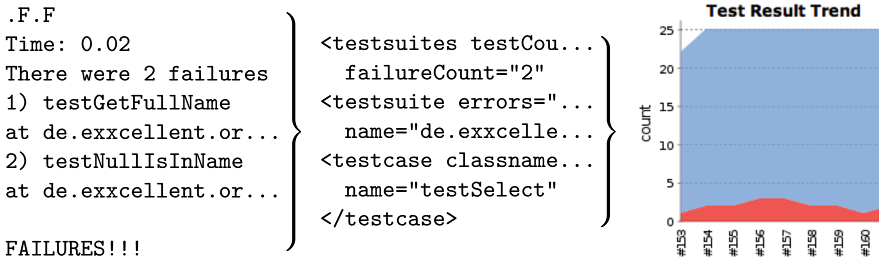

Fig. 2: From JUnit over XML to visual output

the engine log was initially not meant to be used as a debug log on deployed systems.

A solution built using this idea could be easily implemented by providing hooks in the orchideo|engine for listeners to pick up logging. Different listeners could then decide, based on the current log line, how (and if) this is formatted as XML.

2.4 Log Analysis Toolkits

Toolkits such as the logstash framework provide flexible indexing for log files [81]. The goal is to be able to index any kind of log. In the case of logstash the developers use the Grok [82] pattern matching tool which can, based on simple grammar definitions, create JSON representations of loglines.

The ideas behind logstash are close to what we want to accomplish: providing analysis of output with no impact on the existing engine logging solution. However, logstash/Grok produces JSON output, which, while easily parseable, is not integrated at all with the orchideo suite, even though the . It is just another intermediate representation. The tool itself cannot be integrated into the existing Eclipse based eco-system which would be a less-than-ideal solution given the present coherent user experience.

3 Solution Description

In co-ordination with ex|xcellent solutions we weighed the above options against one another. The customer set the requirement to keep the logs in a human readable form close to what it is now and explicitly asked not to switch to XML as the main log format, regardless of our analysis and visualization solution. Also, the idea of generating core dump is evidently impossible to implement with the Java language today and it was discarded directly.

To use a logging framework or a structured output format legacy system traces would have to be automatically translated. Regarding the current format of orchideo|engine traces analyzing the log to convert it to a new format just to parse it would mean translating one representation into another. This would

	Logging Framework	Output Formatter	Log Analyser
Easily Parseable	✓	✓	✗
Legacy Trace Support	✗	✗	✓
Requires No Engine Changes	✗	✗	✓
Applicable Solutions Exist	✓	✗	✗

essentially require the writing of two parsers and significant changes to the engine itself to support the new log format. Those solutions are thus inadequate for the given problem.

We believe that the idea behind logstash serves the needs of ex|xcellent solutions best. Since avoiding third part dependencies and integration with the existing system were requirements, we opted for writing our own parser as a plug-in for orchideo.

4 Syntactic Form of orchideo|engine Traces

Parsing is a well understood field in computer science and most parsers today are written using some kind of parsing framework. Popular Java-based parsing frameworks, like *ANTLR* [83] and *Rats!* [84], take parsing expression grammars (PEGs) and generate parsers from it.

As we previously said (section 1.1) no such grammatic definition for logs from orchideo exist which is why we had to create one.

Analyzing orchideo|engine traces and the code to generate them yielded information which we will present first to help the reader understand a number of design choices we made during our implementation. Errors we found in the traces we have analyzed include:

- Constraint violations on domain objects
- Java exceptions in application and engine code
- Misconfigured aspects (e.g. database connection configuration)

The following examination will use the log extract shown in figure 3 to demonstrate how such errors manifest in a trace. A complete set of grammar rules based on this analysis may be found in appendix B.

4.1 One Action per Line

Each action is logged on a single line with at least it's name and parameters. This will help to understand the following explanations. There are additional lines (mostly excluded from the example in figure 3), but the information immediately relevant to a specific action is always contained in one line.

Actions are numbered starting with (1) on each nesting level (more on levels below). Nested levels have a number in front of their opening brackets and all but the first opening bracket (the root joint-point for the logged engine execution history that led to the logged failure) are numbered.

```
> (Checks: objects="null" constraints="null" [ParameterHash=42])
 (1) ROOT: Checks: objects="null" constraints="null" [ParameterHash=42]
 (2) >> (MarkConstraint: contextObject="<caught exception>"        ↩
        constraint="de.exxcellent.orchideo.objects.dsl.model.impl. ↩
        ConstraintImpl@595bcd (description: null) (name: nameLength) ↩
        (oclExpression: self.name.size()>0, type:OCL)" wasMarked="true" ↩
        [ParameterHash=<caught exception>])
        ⋮
     <<
<
```

Fig. 3: Three line extract of a hand-formatted orchideo trace

4.2 Tree Structure

From what we learned about AOP and how actions work in part [?], we know that the orchideo|engine executes actions at join-points created by other actions. This creates levels of nested action executions where each action that acts as a join-point generates a sub-level of executions. These nested leves may be interpreted as trees. In figure 3 the formatting has been adjusted to the tree structure and how each sub-tree surrounded by opening and closing angle brackets (“>”).

Apart from the beginning and end of a line of execution caused by one join-point, more information is encoded in the opening and closing brackets. Closing brackets come in groups and are always on a new line. They match the last opening group of brackets with the same count of brackets. This cardinality encodes the nesting level of a subtree.

Actions are printed by the engine in their execution order—which means that *before-advice* is logged in front of the action which caused it within a tree. To be able to see the cause for a subtree easily, the orchideo developers added two redundant log entities:

- The the description of the action which acts as join-point is printed in regular brackets on the same line and immediately after the opening angular brackets for the level.
- The causing action has the string “ROOT:” as prefix when it is eventually printed at its proper point in the execution history.

This is an example for a piece text which has been included in logs simply to make interpretation for humans easier instead of encoding an actual surplus of information.

4.3 Action Information

The logged information for each action is built in the exception simply by concatenating various strings retrieved from objects accessible at the point of failure.

Action Name: We previously explained that the engine prints the string "ROOT:" in front of the root action (→ the *cause*) for a sub-sequence. After that follows the name of the action. In the example above we see the actions Checks and MarkConstraint have been run. The action name is not the name of the class implementing the action, but is determined by the string given in the aspect model [?]. This string is a valid Java identifier which is easy enough to parse [31].

Action Fields: Immediately after the action name, the parameters which have been passed to the implementor instance are listed as key=value pairs of field name and value. During our analysis we called these parameters "Action Fields" even though they are not necessarily fields in the action class.

The action fields are, next to the name, often the most interesting part of the action information as in them the object on which the action has worked is revealed. If the action has to do with constraint checking, a frequent source of errors, the attribute which a constraint is checked against is included in the fields as well. In figure 3 the MarkConstraint action is passed the constraint it marks. The string representation of this parameter contains the fully qualified class name and reveals in the `name` field that the checked attribute was `nameLength`. Extracting this information in order to make it more visible helps the developer to connect an error back to the domain model.

The actual values passed as parameters are printed using their `toString` methods and enclosed in quotation marks. In the above example, the Checks action takes two parameters named `objects` and `constraints`. Of course when writing new aspects, an engine developer has to avoid some character sequences (such as a single quotation mark) implicitely that could render the output ambigious. Otherwise, the log will become difficult to interpret both for the human reader as well as a computer program.

We found that many orchideo objects that are typically passed to actions—examples include the various constraint classes, the domain-specific objects and OCL expressions—have their `toString` methods overwritten in order to supply more information about their state. This can be seen in the above MarkConstraint action's parameter `constraint`. The ConstraintImpl prints its default string including full class identifier and address and appends some fields to it.

Fields in Parameters: Fields in parameters, if printed, are surrounded by regular brackets and the key–value pairs are separated using a colon (key: value). A pair of brackets may contain additional pairs which do not represent fields, but additional information the orchideo developers deemed useful. Such additional pairs are separated from one another using commas. The `type: OCL` pair in figure 3 is an example for such additional information.

Different from parameters' key-value pairs the values of subfields are not surrounded by hyphens. However, the fields may contain any type of object which can lead to problems with the aforementioned implicit restriction on character sequences in parameter values. See in the above example the field `oclExpression` in the `constraint` parameter value: we can see the OCL string which was entered by an application developer using the orchideo|objects module for MDSD.

We have found cases where such expressions were invalid, but did not prevent the application from starting. In such cases, however, an application developer unaware of this implicit restriction to his input may render the log output concerning this constraint unreadable.

Exceptions in Printing: In line 3 of the trace extract, the `contextObject` seems to have the value "`<caught exception>`". This is a fallback string the engine prints if an exception occurs while trying to retrieve the string representation of a parameter. The trace leading up to this exception is printed at the end of the execution sequence and passed up to all parent execution sequences. This means that exceptions occurring somewhere nested in the execution will propagate all the way up to the root level. Although the original source of the exception is thus obscured, it was allegedly done in order to increase visiblity of an exception.

The Parameter Hash: At the end of each action information a *parameter hash*, that is, a sum of all parameters' hashes, is printed. This hash has no informative value in itself. The reason for its existence can be found in the orchideo documentation which gives tips on how to use the textual log for debugging: if a problem occurs and an orchideo trace is generated, the parameter hash, which can be retrieved from each action implementor using `getParameterHash`, is supposed to be used as conditional breakpoint. Although we certainly hope that this use-case now, with the addition of our debug plugins, will become less important, we still decided to keep and parse this information as well to support users which might have found this useful before.

5 Parser Architecture

Our parser is structured into a lexer for reading and converting input, a parser to provide an API to other tools and modules for analyzing parts of interest in traces.

5.1 Lexer

To be able to abstract from different inputs (files, strings) and deal with different formats (platform specific line breaks, mixed whitespace) we have written a simple lexer to be used by the actual parser. The lexer operations are loosely modelled after the `Stream` protocol from Squeak/Smalltalk.

Our lexer is able to read from files, strings and streams, keeping track of line numbers along the way. It also hides away differences in line endings on reading and thus ensures that using the default `\n` for newlines in regular expressions matches the end of a line regardless of the originating operating system's line ending convention.

Our lexer mainly consists of different methods to *peek* ahead in a stream, either with a fixed character count, at an offset, or up to the end of the current line. This provides flexible ways to check for different non-trivial conditions when

parsing a log. Consider, for example, the action which is printed in brackets on line 1 of figure 3: Before the text is parsed, the lexer is used to peek ahead if the next character, excluding whitespace, is an opening bracket. If that is the case, the lexer method to peek ahead to the end of the line is used to look for the closing bracket. If unintentional linebreaks have been introduced (e.g. by sending the trace in an email), the lexer can peek ahead over the next lines to find the closing bracket. Once it is found, the complete action text may be parsed at once. If no closing brackets are found, the input stream position has not moved and another rule is tried to continue parsing. *Read* methods to move ahead in the stream mirror the *peek* methods and are used to consume the input.

Apart from that, we implemented rudimentary regex matching in the lexer, in order simplify checking for start conditions of our grammar rules. However, this functionality is limited to a few characters at the beginning of the current text positions for performance reasons.

5.2 Parser

We have determined that even in configurations where the engine logs more than 14k lines of action executions, the nesting depth hardly ever outgrew half a dozen. This led us to consider a simple recursive parsing algorithm early on as the stack level would not grow inconsiderably given such numbers. There are but a handful of rules and an easy solution simplifies building upon those for future extensions of the trace.

Our parser implementation consists of a `Parser` class exposing a simple API which may take the same types of input formats as the lexer, as we wanted to hide this implementation detail away from any visual frontend using the analyzer [**?**].

The parser is recursive in the way that it uses a number of classes that implement specific grammar rules. Each grammar rule adheres to an interface conveniently called `GrammarRule`. This interface contains methods which, given a lexer as input, decide whether a rule matches at the input's beginning or end and one method to consume from the input one textual instance of itself. As grammar rules themselves often consist of other rules, the input is passed down to the children of a rule for consumption. In this way, our parser acts much like a state machine, with different states depending on the parsed log lines and a finite, but not unequivocal number of possible transitions based on the current state. You may find a slightly simplified sequence diagram of the parsing process on a separate page (figure 6). In this example, a file handle is passed to the parser using the public API. The parser constructs a `Lexer` instance on that file which abstracts from the underlying representation of the text. That lexer is then passed to an instance of the `Level` class which represents a tree nesting level. As execution always starts with a join-point action, the top level enclosure is always a level (nesting level zero). A level then enters a loop to parse everything that is included within, until its end-condition matches. In the example, a trace item, which represents a line containing a number an an action text, announces a match on the lexer position and the root level passes control over the input stream (in form of the lexer) to the `Item` instance. The item

```
public class ActionName implements GrammarRule {
  @Override boolean canMatch(Lexer in) {
    return in.match("\\s*" + JavaIdentifierString + ":");
  }
  @Override boolean atEnd(Lexer in) {
    return in.peek() == ':' || in.isEmpty();
  }
  @Override public ActionName consume(Lexer in) {
    consumeWhitespace(input);
    while (!atEnd(in))
      name.append(in.read());

    if (!(in.peek() == ':'))
      throw new ParseException(in.peek(), ':',
          in.getLine());
    in.read(); // Consume the colon
    return this;
  }
}
```

Fig. 4: The implementation of the ActionName class for parsing

then consumes and parses its number from the lexer and passes control on to an `ActionText` instance and so on. This way, the control state is always passed to the next rule which can match at any given point and each rule extracts some information from the trace. After parsing is done, the root level is returned to the client—which now contains objects representing all parts of the trace.

5.3 Grammar Rules

Grammar rules in PEGs contain terminals and non-terminals (e.g. other grammar rules). Each terminal/non-terminal may have cardinality modifiers to make them required, optional or to allow multiple items of this terminal/non-terminal.

In our implementation rules as (see appendix B) are represented by instances inheriting from the `GrammarRule` class. Optimizations have been made in some cases were rules were so trivial that they could be expressed as a regular expression in a parent rule. Each class knows which terminals and non-terminals it requires and how often. Each contained rule is asked for a match on the current input using the `canMatch` method defined in the `GrammarRule` interface. Optional children are given a value of `null` if they do not match. If a terminal/non-terminal does not match although it is required, a `ParserException` is raised. As an example of the relevant methods in the class responsible for parsing action names from a trace contains is shown in figure 4. As shown in section 4.3 an action name must be a valid Java identifier and end with a colon. If the colon is not the next character after consuming the action name, an exception is raised.

The parser always consumes all leading text in its input until a `Level` matches—level being the name we gave the rule matching one sequence of actions. The first level would always start with a single, opening, angular bracket and the root action's text encompassed in regular brackets. This first root action is usually the action called by the application developer (often actions like `ObjectAspect.createObject` which is used to create domain objects or `HibernateAspect.commit` which is run to commit objects to the database).

Each nesting level contains a number of actions (or else the engine would have printed it directly without a level around it). If a `Level` matches at the beginning of the current input position and has consumed it's action text, it enters a loop trying to match children, which may either be more levels or simple actions, until it hits enough closing angular brackets to finish. If the end of the input is encountered before enough closing brackets could be read, an exception is raised.

5.4 Parsing Ambiguities

As specified in section 1.4, developers may receive logs by email. Mail clients often change the whitespace of messages and we found problems arising thereof are mainly:

1. Unintentional line breaks are introduced
2. Groups of spaces may be replaced with tabs
3. Different newline conventions might be mixed

In order to deal with these problems relating to whitespace, we implemented various helper methods, both in the lexer and in the `GrammarRule`. To inhibit the impact of erroneous matches due to problems with whitespace or missing line-endings rules, in some places, does not actually pass the global lexer, and thus global control of the parsing process, to their contained rules, but constructs a new lexer on some part of the input.

Consider an `ActionText` (appendix B) that has a few user supplied values in its nested fields. If such values were ambigious either because they contained a quotation mark or looked like another rule, we would be able to decide on a reasonable parsing strategy due to the fact that each action was logged on a separate line in the trace. However, with additional line breaks such as in mails, this decision cannot be based on line endings. Introducing new line ending characters was considered, but would lead to problems with legacy support which we are required to provide. Our solution to this problem matches both from the beginning and the end of an action text. An action text first consumes its text and the text which to be parsed by sub-rules directly, up until another action text matches or the end of a level is reached, ignoring all line-breaks in between. Action fields are parsed last and only with a lexer on the remaining unparsed string in the middle of the action text.

Using this technique, short of encountering a string that looks like a trace as part of a parameter value, we can prevent that matching on a broken trace will disrupt more than one action at a time and even then, only the parameter values would be parsed incorrectly, keeping the rest of the information intact.

5.5 Parsing Exceptions

As a result of our analyses we found that stack traces from exceptions which occur somewhere during action execution in the engine are simply appended directly to levels in which they occurred. Thus, they are actually part of a level, but occur outside of the level's brackets.

Stack traces per se have a distinct format which we can easily separate from other rules. It is common, however, to provide an explanatory string when throwing an exception which is printed at the beginning of the stack printout. These strings may be arbitrary text and at this point we can rely neither on line endings to check for the end of such a string, nor on pre-consumption through the containing level. This means that the parser must be able to separate exceptions correctly, but not confuse action texts with exception strings and possibly loose information. Thus, the `JavaTrace` consumption was implemented as a fallback, in case no other rule matched. While a `JavaTrace` consumes input, it continuously checks all other existing rules and finishes if any match is found. This special case ensures that under no circumstance an action's execution is lost because a stack trace precedes it.

6 Extensions for Requirements

Several of the aforementioned requirements cannot be satisfied with the information contained in the engine traces today. In the sections below we show how we have extended the trace to solve more of the problems related to the orchideo traces. The extended grammar may be found in appendix B.1.

6.1 Object State Logging

The most complex systems running on orchideo at the moment have more than fifty domain specific classes in their model alone. In these projects our analysis plug-in is a useful asset to help connect an error back to the model. Still, knowing that a violation has occurred on an attribute is quite different to knowing *why* this has happened. The information that a constraint was violated is already logged, however, the violating state of the object at the time the constraint failed is not. The current architecture of the orchideo|engine does not permit us to retrieve those values easily which is why the customer has made the requirement optional.

The orchideo|objects framework, which is built upon EMF. The "Eclipse Modeling Framework" is a modeling and code generation facility for building tools and other applications based on a structured data model [5]. When using orchideo objects in application code, we are not dealing with the domain models directly. In the engine, at the time the trace is generated, only (EMF-) objects representing the domain models are available, without holding the application objects' values.

We have solved this problem by inferring more information from the type and target of constraints passed to the `MarkConstraint` action: Constraint objects

print fields relating to their type and the object's field which they apply to. We parse those fields as part of the paramater value for analysis. This way users are now told whether the object in question validated an OCL constraint (and which) or a cardinality constraint. Additionally, the field which has been violated is emphasized.

6.2 Connecting back to Code

Initial user research (which we discuss in more detail in [?]) has shown the tool is being used actively by orchideo application developers. The previous workflow

Receiving a logged failure
➥ Analyzing the problem with help from engine developers
➥ Finding the bug in the application code
➥ Fixing the application

and in the process bother at least two developers with the problem is moving towards the more ideal

Receiving a logged failure automatically visualized
➥ Finding the bug in the application code
➥ Fixing the application

which not only occupies less developers, but also reduces the time needed until a problem is understood and a bugfix can be constructed.

However, the application developer still has to match a failure like a constraint violation manually to source code. This is often enough straightforward given the top-level root action and the failure. Yet, especially a developer only started working on an existing project, there is still manual work required when trying to match a failed action to the place in source code where it was called. The customer requested a method to automate recovering the way from errors back into application code.

To alleviate this problem, we have extended the engine trace and prepended stack information to it. It is important to know that, when an aspect's action is called from application code and fails, the trace will be printed in the context of that action. At that time, the stack will only be a few frames away from the application code's call.

We analyze the `ExecutionStackTrace` in the primary level and build instances of the `StackTraceElement` class from, it which is available in Eclipse as a representation of stack frames. Those can be used directly to find documents in the local workspace relating to the stack frames giving the developer conclusive hints to find bugs.

6.3 Engine Failures In Session Configuration

During development, orchideo developers often use different session configurations in order to adjust the engine's aspects according to their needs [?]. From the engine developers point-of-view, if any code, from the engine itself or the aspects it is calling, fails or if aspects have run their actions that should not have been active with the current setup, this failure can only be debugged knowing which session configuration was active at the point of failure. An aspect developer who configured a session to run his actions can use the session configuration information to check the execution history and make sure his actions are run in the right places.

Sessions are configured using XML files. We resolved that these files be send along with the logs if they are transmitted from a remote system. This enables us to add the session identifiers to the engine trace and use it to find the configuration information active at any given point in the execution history.

Our parser architecture allows us to easily add grammar rules merely by implementing a new class. We added an optional suffix after the parameter hash which contains the session name:

```
(SESSION: sessionName, objectPointer)
```

Although this possibly adds redundancy, including the session information e.g. at the beginning of the trace is not a viable alternative: session configurations may be changed dynamically at runtime so the active session can change during the execution of a sequence.

We have also included the object pointer to the active session in the output. Though that might seem odd at first, we reason that, using our previously covered **RuntimeRescue** and **SessionView** plugins, we are able analyze an engine trace when the program is *still running* [?]. Using the object pointer, the active session in the engine can easily be determined.

6.4 Exposing Details for Visualization

In the following part [?] we will show how we chose to visualize the trace analysis conducted here. Our research into visualization techniques influenced the data structures we chose to offer to some extent.

Node and Leaf Representation We have differentiated levels and actions from the other rules to some extent and made them "nodes" in a tree. This arose from our the decision to map the stack on a tree and use tree-visualization techniques for display [?].

The `StackNode` class contains methods to establish a parent-child relationship between nodes. In addition, it provides getters for action text details for easy access by the frontend. We introduced this abstraction for two reasons: first to hide the different types of nodes in our constructed tree from the frontend and secondly to express the sub-sequence relationship a level has with the included actions as its items.

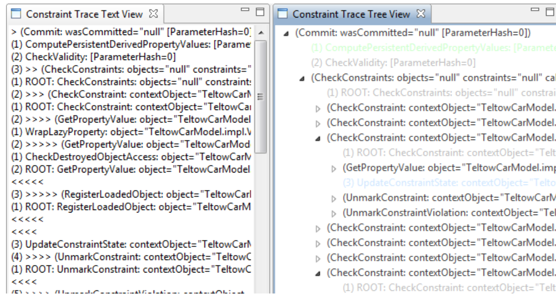

Fig. 5: A simple frontend for testing the parser

Parsing Thread When working with software, users have come to expect interactivity from their tools [85]. Even if a reaction to user input, like starting to parse an orchideo trace, takes longer than a few seconds, users expect to be able to interact with other parts of their program in the meantime. In order to relieve the frontend, which might change, from the burden of moving the parsing into a separate thread explicitly, we offer a callback interface and manage the thread for parsing the trace in the analyzer. When the analysis is started from the frontend, an implementation of the `IParserThreadCallback` interface may be passed into the backend to run code once the parsing is finished.

Monitor Interface Another usability consideration was to inform the user of progress when parsing. The Eclipse libraries offer so-called *ProgressMonitors*: objects, which provide callbacks to announce progress in one way or another and relay that information to the user—graphically or otherwise. To be able to offer such functionality in the frontend we applied the observer pattern to our `Lexer` class, offering notifications on read bytes of the input stream and the number of bytes left to read.

7 Evaluation

Using our parser implementation we were able to parse any stack trace we have at our disposal supporting old logs as well as extended traces with more information. To harden the parser against random errors we have built a test suite which mutates traces and tries to parse them. This way we found some additional problems mostly related to whitespace which we fixed. Our implementation can be used to safely parse and extract all unambiguous and even most ambiguous information correctly.

To check parsing of traces we have built a simple frontend (figure 5) which we used to check parsing on various stacks.

7.1 Requirements Fulfillment

One of our goals was to aid developers who are *users* of the orchideo framework when debugging their application. Before they had to use textual search in the

engine logs to find constraint violations, provided they knew what they were looking for. But even *if* a constraint violation was found this left the action text to interpret which, as we have shown, is not always easy if one does not know what to look for. Additionally, if not a constraint violation caused the failure, the user was at a loss as to what to look for in a trace. The problem degraded to a fuzzy search over the log where the human mind is prone to missing important details. We have diminished this problem for users of the orchideo framework and our plug-in provides support in the context of the initial requirement to help application developers *using* the orchideo suite solve their problems *fast* and without having to understand inner workings of the engine or even the aspects.

The requirement not to change the current log format has been addressed with only encoding optional additional information. The changes to the core are minimal and the impact of including it with new versions of the orchideo suite is neglectable. Finally, our plug-in can be deployed and used with existing systems and customers *now*: We are confident that this will lead to fast adoption among orchideo users. There is no need to deploy new versions of the orchideo|engine on each remote installation to benefit from the analysis we provide.

7.2 Discussion of the Implementation

The implementation we have devised—using a recursive parser simulating a state machine—provided a working solution quickly and is easy to extend and build upon. Representing states in our parser with grammar rules allowed us to include more information without sacrificing backwards compatibility. Even though the recursive nature of the parsing process might lead to difficulties if more aspects create much deeper nesting, we do not think this will be a problem for the following reason: The way aspects are used in orchideo they can be seen more as assets which make MDSD possible instead of a programming model in and of itself. Aspect oriented programming in orchideo alone, without the powerful weaving conditions as, for example, in AspectJ [17], would not unfold the real benefit the orchideo suite provides; the combination of MDSD and AOP.

Our implementation of the analyser as a separate OSGi bundle further allowed us to ship the analyser separately from visualization. This way, if at any time the evaluation criteria for problems change or a different visualization is required, it will be easy to write a plug-in against our API to fulfill that purpose.

Our solution may be considered less than ideal if the requirements had been known and considered in the design of the initial implementation of orchideo. As we have argued in section 2.2, a different logging method with a completely restructured way of generating debug output, separating errors from other information from the start might be a viable alternative if the current system is going to be refactored at some point to break backwards compatibility. Yet, new and changing requirements for growing software systems are inherent to the software development process [86]. The problems arising thereof and how to solve them are not part of this work. Extensions to existing software must strike a balance between the cost of changing the underlying system and the cost of

working around some of its implementation specific restrictions [87]. Our analyser works with the existing system and has minimal impact on orchideo and existing orchideo applications. Integration and usefullness with existing systems were requirements which are thus satisfied.

A different issue with our parser which, according to user feedback, arose sporadically were out-of-memory errors in the PermGen heap of the JVM. The PermGen is a special heap in some JVMs which contains the *permanent generation* of objects that should never be garbage collected. This usually includes things like class metadata, bytecode, interned strings and so on, however, there is no clear documentation on this as it is certainly also dependent on the virtual machine. We were unable to reproduce the problem through random testing nor could the application developers provide us with a way to reproduce the problem. Our current workaround includes increasing the heap space using a JVM parameter.

Yet another implementation specific problem with the current implementation is its performance. There are sections of the code where input is read byte-wise in order to ensure the best match. This slows down execution considerably; an average stack of about 15k lines takes something between 90 seconds and 2 minutes to parse. This depends largely on the distribution of constructs —some rules take considerably longer to parse than others. The extra care we have to take for action parameter fields makes this rule much slower than parsing a Java stack trace, as in a stack we can read line-wise.

7.3 Future Work

As with any large software project, several areas of work still remain which we may yet target.

We already mentioned an issue with logging object state in section 6.1. In this case, the actual object cannot be retrieved at the point where the trace is created due to restrictions in the EMF architecture applied. This problem may only be solved with some substantial changes to the orchideo|engine architecture and has in agreement with ex|xcellent been postponed for the time being.

Regarding the session information which we have added to the trace (section 6.3) we have plans to make this more useful by integrating it with our other tools. Right now, a developer will still have to manually compare the session configuration to the recorded execution history manually. This is especially difficult as the execution order is not entirely deterministic and may differ from the orchideo supplied session configuration view [?]. This can be enhanced to show actual discrepancies between configured aspects and execution history automatically. This will further reduce the time needed to find errors related to faulty session configurations.

Additionally, while the runtime rescue and the session view plugins allow the recorded session history to be inspected at runtime the object ids recorded in the trace still have to be matched manually to the sessions in the session view [?]. Better integration of both plugins should be addressed, too.

Finally, while we have been able to diminish the problem of ambiguous snippets leading to false matching, it might still present a problem. In the beginning the recursive parsing led to some problems when rules “misbehaved” for ambiguous matches and consumed more or less than they were supposed to. We solved this problem by using different lexers on substrings of the input. This restrains problems to affect only the action which contains the ambigious match. If the action in which the parse failed is the actual interesting one, however,the failure possibly leads to confusing information and might prompt a user to suspect an entirely different problem. We need to find a way around this, possibly by marking such problematic rules to communicate to the user that extra care should be taken.

8 Additional Related Work

Apart from the different logging solutions we have presented in section 2 there are other solutions similar in scope to our work. Those solutions, too, deal not only with the problem of providing analyzed output, but also try to help developers find bugs in deployed software.

Log file analysis as a formal method is used to analyse software event logs in order to evaluate whether the program has behaved as expected. This technique has been used for verification and testing of programs by Andrews and Zhang [88]. Their work includes a formalization of a state machine approach to log file analysis which is very similar to the changing state our parser implements by means of passing control to grammar rules. However, our focus was not on using that log for testing, but on structuring the output we are given.

The orchideo setup of logging on customer machines and sending log files for automated analysis is a common technique in the computer industry [89][90][91], easily the best known example is the crash dialog of the Windows NT operating system [92]. The binary crash report created by this tool is automatically analyzed to prioritize and categorize the crash reports.

In the mobile sector, the Mobile Crash debug system for Symbian OS provides the means to both send automated crash reports in binary form and analyse them [3]. This analysis' intention is similar to ours as it, too, tries to determine the type of problem and provide the developer with hints about were the actual problem lies.

9 Conclusion

For this project we have designed and implemented an OSGi bundle for Eclipse available for use by other plug-ins. The plugin is tailored to log format of the orchideo suite. In conjunction with our visualization solution (see [?]) we have optimized the workflow of the average orchideo developer, who no longer needs to learn about the interna of his framework to understand simple errors. Additionally, we have extended the log to provide more information to the developers

of aspects and the orchideo|engine. This may be built upon to provide automated validation of the session configuration as well.

We have seen how changing requirements and time restrictions can cause design decisions and in turn code which stay with a software project for considerable time. Debug code written early during the development of a program was extended and has been put to use in other ways than initially planned with the advent of new requirements.

We have also seen how working with the circumstances in a system, instead of changing them, might sometimes be worthwhile. It can save time and costs, new requirements may be integrated into existing structures without too much disruption and legacy system support may be easier because of it.

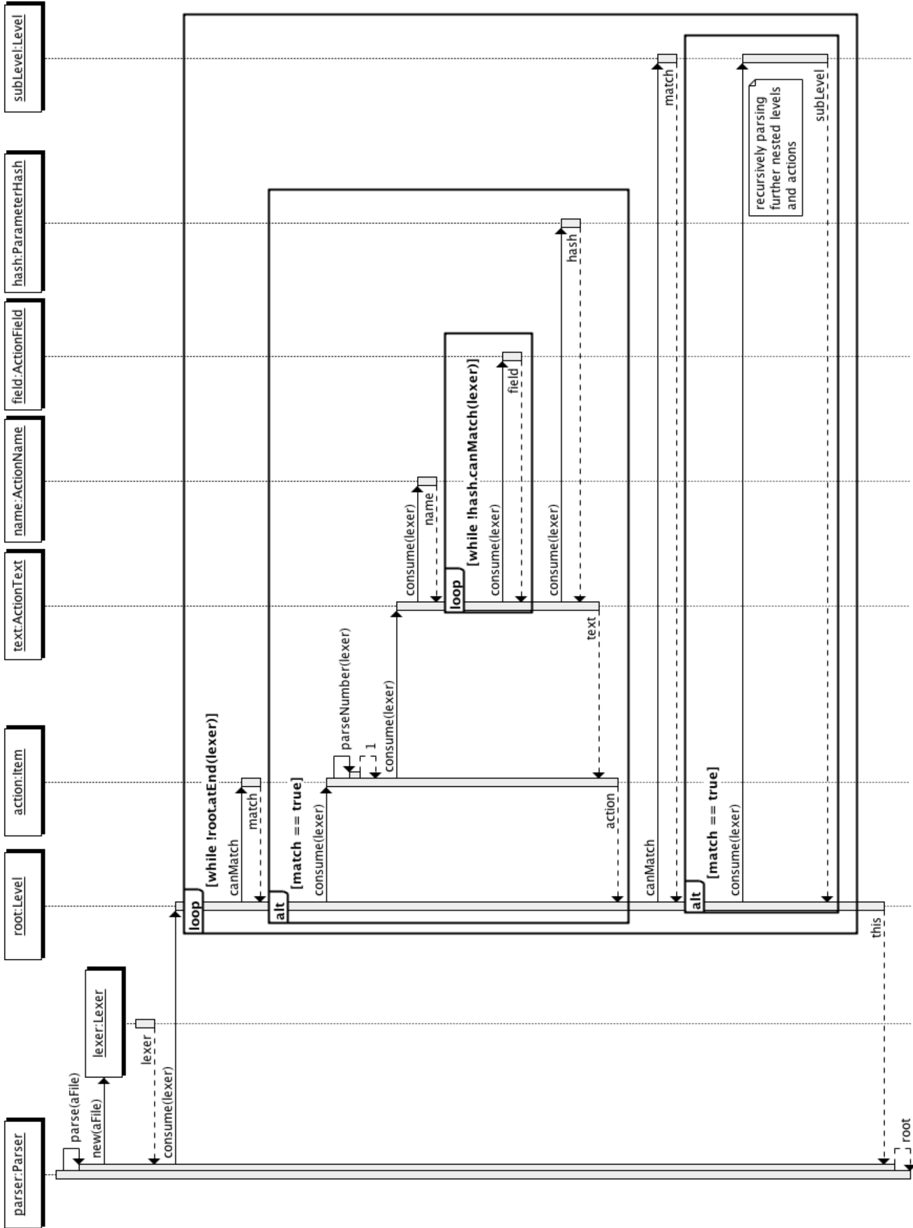

Fig. 6: Simplified parsing process

Bachelor Thesis

Exception Visualization

Philipp Tessenow

Supervisors:

Dr. Michael Haupt, Malte Appeltauer
Prof. Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute,
Potsdam, Germany

June 25, 2010

Exception Visualization

Philipp Tessenow

Hasso Plattner Institute
Potsdam, Germany
philipp.tessenow@student.hpi.uni-potsdam.de

Abstract. Current software development frameworks provide good support for the software creation process. Unfortunately they often give unexpected, incomprehensible or long error messages when things go wrong. The paper describes our research on error visualizations of tree-structured error messages. Furthermore we present our implementation of different error visualizations for the orchideo suite. Using our extensions for orchideo the error identification process of orchideo|engine error traces is much faster.

1 Introduction

Software developers spend a large part of their effort on debugging [93]. Studies differ on concrete numbers, but for the debugging process—which includes comprehending error messages of a compiler, a programming framework or another entity in the workflow—a developer uses around 40% of his time[94]. When development tools are growing, for example through adapting to new software development principles, like model-driven and aspect-oriented development, their error messages are harder to understand. This is because a developer has to understand which part of the tool-chain went wrong and for what reason.

As the inner structure of most software development frameworks grows increasingly complex, it is difficult to produce simple and easy to understand error messages. Error visualization techniques are meant to support a developer in understanding and fixing the error.

The orchideo suite [4] is a hybrid of an aspect framework and a model-driven development environment which is highly integrated in Eclipse [1]. orchideo provides excellent tool-support for the model-driven creation of application code. Furthermore the orchideo|engine enables the software developer to intuitively integrate aspect-oriented principles. While we experienced this during our research, we also recognized that debugging-support in orchideo is lacking.

The lacking debugging-support of the orchideo suite becomes noticeable when looking at the following scenario: An orchideo object model was built describing a customer who has some attributes, such as firstName and lastName. Using the code orchideo created out of this model, we instantiate a customer object as seen in Figure 1. Here we set the customers first name but not his last name, which is required for a valid customer object. When trying to add the customer to a database through the HibernateAspects `commit()` method, the orchideo|engine

```
Customer customer = (Customer)
    objectAspect.createObject(Customer.CLASS);
customer.setFirstName("Jane");
// customer.setLastName("Doe");
hibernateAspect.commit(); //throws an exception
```

```
  at org.eclipse.equinox.launcher.Main.run(Main.java:1311)
  at org.eclipse.equinox.launcher.Main.main(Main.java:1287)
[MainExceptionTrace:[Thread.java:java.lang.Thread#getStackT
> (Commit: wasCommitted="null" [ParameterHash=0] (SESSION:
(1) CheckValidity: [ParameterHash=0] (SESSION: TeltowCarMod
ParameterValue: [de.exxcellent.orchideo.objects.aspect.core
[no stacktrace available]
(2) >> (CheckConstraints: objects="null" constraints="null"
(1) ROOT: CheckConstraints: objects="null" constraints="nul
```

Fig. 1: In the Java code example at the top of this figure we create a Customer object. As the customer needs a first name *and* a last name, the `commit()` at line 4 fails. An `ExecutionInterruptedException` printing an orchideo|engine trace will be thrown. A small part of this trace is shown in the bottom part of this figure.

throws an `ExecutionInterruptedException` which dumps the current engine state to a string, which is shown in Figure 1. Besides the fact that the exception is thrown in an unexpected context (usually within some SWT-, Eclipse-plug-in-, or JUnit- code), the exception trace is very long and not easy to understand.

orchideo traces with sizes of about 3 MB—when saved to an ASCII-encoded file—are not unusual and contain several thousand lines of cryptic code. We attached a shortened version of an example trace in the Appendix C. Neither us nor most of the every-day orchideo developers were able to read and fully understand such a trace at first glance. Unfortunately the reason for the engine crash, for example a missing last name attribute of a customer, is hidden somewhere in this trace of many similar looking lines of text. To spot the error within the trace feels like the proverbial search for a needle in a haystack. The current situation at ex|xcellent is such that most of these traces are sent by mail to orchideo|engine developers asking them to find out what went wrong. Those engine developers are busy most of their time. Obviously this situation has some potential for optimization.

This paper discusses error visualization techniques taking orchideo as an example, as it produces very long and cryptic error messages. We present an orchideo plug-in, that is able to read and analyze orchideo traces. Our plug-in visualizes those traces using different methods to help the orchideo developer to understand them and find the hidden error.

In the following section we discuss several visualization techniques and the requirements they have to fulfill. After deciding which techniques suit our task, we present our implementation. This includes a detailed description of how these visualizations were implemented as well as how we integrated them in the de-

velopment workflow. Thereafter we evaluate our contribution to the orchideo suite on given user feedback. Finally we give a brief summary and present our conclusions.

2 Tree Visualization Techniques

The process of understanding orchideo|engine traces currently does not scale as there is only one engine developer that fully understands those traces. We want to enable each application developer to find out and understand what went wrong when an engine trace was thrown.

An orchideo trace has a special tree-like structure which can be parsed programmatically [**?**]. Our *Trace View* plug-in offers the ability to parse and display such a tree taking into account difficulties like an undecidable grammar and unexpected white-spaces. In addition to the ability to parse an orchideo trace, our goal is to visualize it in a way that overcomes the shortcomings of the current solution.

We have to keep in mind that we need a solution that suits different users: The application developer, who develops applications using orchideo, usually is confronted with orchideo traces because of a failure in his application code. The orchideo|engine developer however develops the orchideo|engine itself and usually gets orchideo traces when running the engine tests. In contrast to application developers, orchideo|engine developers are not interested which code called the engine, but in the engine code itself.

This led to different requirements that have to be fulfilled for a successful implementation. After we discuss the requirements, we give an overview to several tree visualization techniques. Finally we compare them and choose which technique to use in our implementation.

2.1 Requirements for Error Visualizations

To be able to fulfill our task and to evaluate what we implemented, we need to define which requirements our implementation should fulfill. After talking to orchideo|engine developers and application developers that use orchideo at ex|xcellent solutions, six main requirements have emerged.

Integration into the Current Workflow. It is a common case that an application developer using orchideo has to handle orchideo|engine traces. Therefore he should be able to handle the trace as conveniently as possible without leaving his usual workflow. User interviews showed that orchideo traces usually exist either in the form of log files from production systems, or plain text copied from mails or other resources. It should be possible to use *drag and drop* or *copy and paste* actions as an input method for the visualization. Furthermore the visualization itself should be placed within the developers usual environment—orchideo.

Give Hints as to which Part of the Application Code may have Produced the Error. When an application developer who uses orchideo is confronted with an engine trace, it is hard for him to see which part of his application code caused the orchideo|engine crash. This is unexpected at first glance, because the `ExecutionInterruptedException` contains the usual Java execution stack that shows where the exception was thrown. We have to keep in mind that orchideo|engine code is often called within several environments like SWT [95], JUnit [96], or the Eclipse plug-in framework [45]. Each of these frameworks enlarge the printed execution stack, which makes it hard to find the erroneous code. Even worse, orchideo itself adds some entries to the execution stack too.

Therefore we need a mechanism that can handle a given execution stack to find the part of the code causing the error that the application developer has modified.

Show the Necessary Information in as Little Space as Possible. An application developer should focus on writing code and designing the application. When an engine trace is thrown it should be possible to analyze it alongside as this is not a developers main task. Therefore the visualization should occupy as small screen space as possible without limiting the quality of the trace analysis.

Find and Highlight Common Errors. There are two common reasons for an orchideo|engine crash causing a trace. The most common causes are *constraint violations* which appear when a constraint, which was defined in the orchideo model [**?**], was violated. This happens when the engine tries to handle invalid orchideo objects. The other cause is an erroneous aspect implementation that the engine tries to execute. We can not exclude other reasons causing an engine crash, but we have not found any in all of the traces we analyzed.

Our project partners at ex|xcellent solutions confirmed that the large majority of reasons for an engine crash can be traced back to the two cases described above. Therefore the error visualization has to provide excellent support for these cases.

Hide Unimportant Information. The orchideo trace potentially contains several thousand lines of code representing the state of the orchideo|engine at the moment its execution stopped. Usually only a few of these lines are important to understand what went wrong. In consideration of the previous requirement to occupy only a small amount of space on the screen, we need to hide unimportant information.

Enable Engine Developers to Find Uncommon Problems in the Engine Trace. In some cases it is important to have all the information at hand. This happens when the structure of the trace is important as it decodes which actions were executed in what order. So we need to hide unimportant information but should be able to provide all the necessary information on request.

Keeping the tree structure of engine traces in mind, we want to present different

```
{"Level": {
  "name": "Commit",
  "wasCommited": null,
  "ParameterHash": 0,
  "items": [
    {"Item": {"name": "CheckValidity", "parameterHash": 0,
        "javaError": ...}},
    {"Level": {"name": "CheckConstraints",
               "objects": null,
               ...,
               "items": [ ... ]
    }},
    ...
  ]
}}
```

Fig. 2: An example orchideo trace written in JSON. The trace is shortened considerably using dots (. . .), which are not part of the trace. There are two different JSON objects: Items, which are leafs in the tree structure, and Levels, which contain further Levels and Items in their items field.

tree visualization techniques. Afterwards we compare each of these techniques on the basis of our requirements.

2.2 Text Representation

orchideo applications which are used in production usually catch any engine crash and try to recover their state so that the user can continue their work [7]. All orchideo traces are written to log files for later analysis. The trace is a textual representation of the current engine state. As discussed earlier in this section this representation is hard to understand and space consuming [**?**].

We discuss JSON and XML as possible representatives for text output in general.

JSON The JavaScript Object Notation (JSON) is defined as a "lightweight, text-based, language-independent data interchange format" [79]. As its name suggests, JSON is defined as a subset of JavaScript. JSON uses six different types, four primitive and two structured ones, to represent data. The primitive types are strings, numbers, booleans, and null. Objects and arrays are the structured types used in JSON.

Whereas the notation of the primitive types is trivial, we have to follow some syntax for the two structured types. Objects are unordered collections of name-value pairs, where name is always a string and value can be of any type. The following is a JSON object: `{"name": "commit", "parameterHash", null}`. However Arrays are ordered sequences of values which may look like the following:

[{"name":"commit"},{"name":"WrapLazyProperty"}], which is an array of two simple objects.

In our example trace in Figure 2 we have two different kinds of objects: levels and items. Each level has an attribute called items, which is an array of further sub-levels or items. Items may have other attributes like parameterHash (integer), root (boolean), and name (string). A lot of other optional attributes, like wasCommited, or contextObject, may be defined for an item as a lot of different types of items (each of them representing a different orchideo action) exist.

The orchideo|engine trace consists of different actions, which are organized in a tree structure. With JSON being a general purpose object notation format, it is not optimized in representing tree structures. This results in a more verbose engine trace as shown in Figure 2. However, a verbose engine trace, having the disadvantage of generating large files,would be acceptable if it aided the locating of errors in the trace. Unfortunately it is hard to separate different levels and items as they are always—with the exception of the very first level—written within just another level.

This makes it hard to spot the error within a JSON formatted orchideo trace. Consequently, we would need another visualization on top of JSON to satisfy the requirements we defined previously.

```
<level name="Commit" wasCommited="null" parameterHash=0>
  <item name="CheckValidity" action="before"
     parameterHash=0/> ... </item>
  <level name="CheckConstraints" objects="null" ...>
    ...
  </level>
</level>
```

Fig. 3: An example orchideo trace expressed in XML. The trace is shortened using dots (. . .), which are not part of the trace. Again, we see two different kinds of entities: levels and items.

XML The extended markup language (XML) specification defines XML as a language that describes a class of data objects called *XML documents* [25]. Those documents consist of storage units called *entities*, which themselves refer to other entities. Different kinds of XML documents can be specified through Document Type Definitions (DTD). DTD's define which entities are valid within an XML document and which attributes they may have.

XML documents may be well-formed and valid. According to the XML specification they are well-formed if they follow a list of rules. Some key rules are:

– it contains legal properly-encoded Unicode characters only
– there is only one single *root* element containing all the other elements

– element tags are case-sensitive; beginning and end tags must match exactly

Besides being well-formed, XML documents can be valid. To be valid an XML document has to contain a reference to a Document Type Definition [25]. A DTD defines which entities and attributes are allowed within an XML document.

Figure 3 shows a shortened orchideo trace which is written as an XML document. As we are not discussing if the document is valid or well-formed, the DTD is not regarded and we focus on the document content itself. The document contains two kinds of entities: levels and items. Levels, in contrast to items, may contain further levels or items as child entities. Both may have attributes like *name*, or *parameterHash* as indicated in Figure 3.

The XML notations makes it easy to separate different levels and items. Furthermore it enables us to see the hierarchical structure of the trace.

Comparison Both representations could be used as direct replacements for the current trace as they can be used within the same workflow. They are both a lot more readable and processable than the current solution as they have a well defined and documented structure with existing parsers available [97][98]. Generating XML or JSON output with the given engine state is possible as there are existing libraries and tools available for that case [99][100].

When it comes to readability both formats have their drawbacks as they produce potentially long traces (with XML using two lines per level and one line per item, which is as long as the current trace, and JSON using even more lines). So both text representations do not fulfill the requirements stated above, but nevertheless provide a slightly better handling than the current trace. For example it is possible to use such traces as input for existing visualization tools [101][102], because they follow widely accepted standards.

2.3 Tree View

Traditional windows explorer like tree views are commonly known because most people use them in their file browser or other programs. This view shows levels as a label with a preceding ⊞-icon. A click on this icon unfolds the tree to show sub-levels. Items appear in the same way as levels, except for their icon.

The traditional tree view is known as a very competitive visualization technique when it comes to user experience [73]. Most users know this kind of tree representation. Therefore it is hard for other visualizations to have a better user experience, even if they perform slightly better in some cases.

Figure 4 shows a Tree View containing an orchideo trace. Level have their usual ⊞-icon. Both, levels and icons, have additional icons that indicate to which kind of orchideo-action they belong: root, before, after, or around. Text, which gives further details about a level or an item, is available in a separate tool-tip window if the mouse hovers over the item. The tool-tip's text is used as a label for each level/item in a truncated form.

Each item and each level consumes a configurable number of lines (the preference is set to three lines in Figure 4). The old orchideo trace needs a minimum

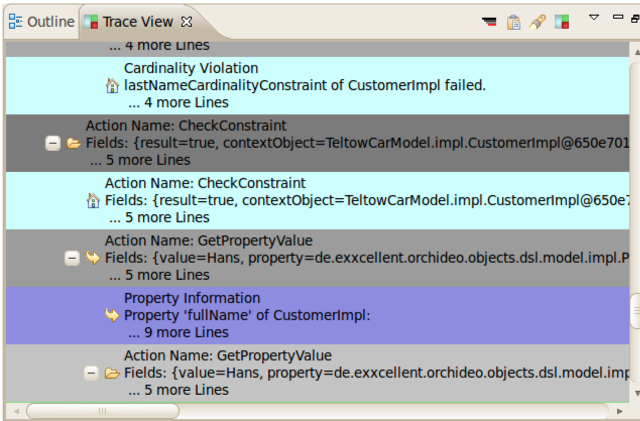

Fig. 4: An orchideo trace shown in a standard explorer-like Tree View. Levels are displayed as nodes and Items as leafs. A tool-tip shows further information for a certain item or levels when the mouse hovers over it.

of two lines per level. With the Tree View only using up to one line per level and the ability of the tree to collapse unused or uninteresting branches the Tree View shortens the displayed trace a lot. This helps in gaining an overview of the trace.

Another benefit is that it is easy to separate different items because of their background color. Levels get a random gray color (having approximately equal brightness at the same level within the hierarchy). The item's color depends on which action they represent. A `MarkConstraintViolation` for example has a light blue color. This makes the whole trace more readable.

Overall, we are able to hide parts of unimportant information (collapse branches), highlight common errors (e.g. `MarkConstraintViolations` have their own color), and we are able to show the information within a smaller space. This fulfills some key requirements for a good error visualization of orchideo traces. However other requirements, such as the integration of the visualization in a developers workflow, still need improvement.

Despite the advantages of the Tree View, there are some restrictions. An orchideo trace containing fourteen thousand levels and items is common. To display such a trace with the Tree View requires less space compared to the former solutions, therefore it increases readability. Nevertheless, we still need to scroll to see the whole trace. Furthermore finding a specific item through folding and unfolding the tree is not convenient. This makes it hard to get an overview over the orchideo trace at a glance.

2.4 Linear Trace

The Linear Trace is a mutant of the former discussed Tree View. We designed it to combine advantages of the Tree View with the ability to hide unimportant

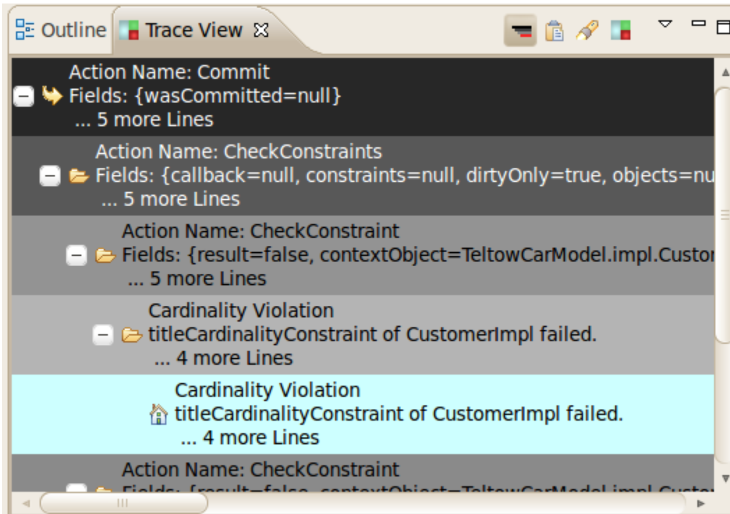

Fig. 5: This is an orchideo trace shown in a Linear Trace view. The Linear Trace view is a mutant of the Tree View, which shows only important items and their path to the root level.

information. It shows only important items of the orchideo trace and their path to the root element. Important items are defined via a search pattern, which can be modified by the user.

This modification, in contrast to the Tree View, hides information while maintaining the ability to highlight important parts of the trace. This modification improves the hiding of information compared to the Tree View while maintaining the ability to highlight important parts of the trace. We lose information as we specialize on displaying only a subset of the trace. Therefore the Linear Trace support enables the finding of errors, for example by searching for `MarkConstraintViolations`. On the other hand it hinders the user to explore the full trace, which makes it harder to locate specific problems, such as incorrect orders of levels and items.

2.5 Tree-Map

A Tree-Map is another tree visualization technique, which was originally presented by Ben Schneidermann et.al. in 1991 [103]. It is a two-dimensional space-filling approach which is capable of displaying the whole tree in a given space without scrolling. It makes use of the humans ability to recognize different elements in a picture and the relationship between the elements. This allows people to gain information much faster from a picture compared to scanning and understanding text [104]. Some of the main objectives of the Tree-Map's design, which match our requirements, are:

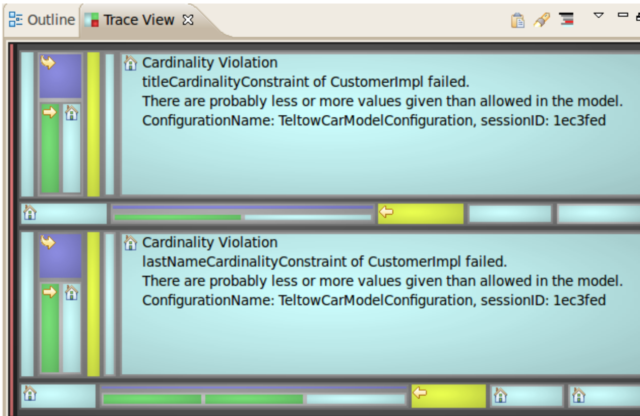

Fig. 6: This is a Tree-Map showing an orchideo trace. Levels and items are drawn as rectangles. Their size depends on their importance (which is whether they fit certain search criteria). The color is gray for levels and colored for items.

- the screen space is used effectively by using 100% of the available display space
- interactivity by offering responsive control over the presentation of information
- facilitate the rapid extraction of information with low perceptual and cognitive loads.

To understand how a Tree-Map decodes a tree, further explanation is necessary. Each node of a tree is a rectangle whose size is proportional to some attribute of the node. Its color also depends on an attribute of the node. This enables the Tree-Map to visualize two different criteria in the same space.

The rectangle representing the root-level of our orchideo trace gets all of the available space. Each sub-level or item of this level is drawn within that rectangle using an amount of space proportional to one attribute and a color depending on another. Then the drawing process continues recursively for each sub-level.

Figure 6 shows a Tree-Map which visualizes an orchideo trace. The size of each item is calculated on their importance. We calculated the importance depending on whether an item matches a certain search expression. In Figure 6 we search for `MarkConstraintViolation` as those violations are the most common reasons for an orchideo trace. The items and levels are colored analogous to the Tree View. Levels get a random gray color with nearly equal brightness for levels of the same depth in the trees hierarchy and a larger difference in their brightness if their depth in the tree is different. This enables the user to see the depth of different levels and items through their gray surrounding. Equal to the Tree View, items are colored differently if they represent different kinds of actions of the orchideo trace.

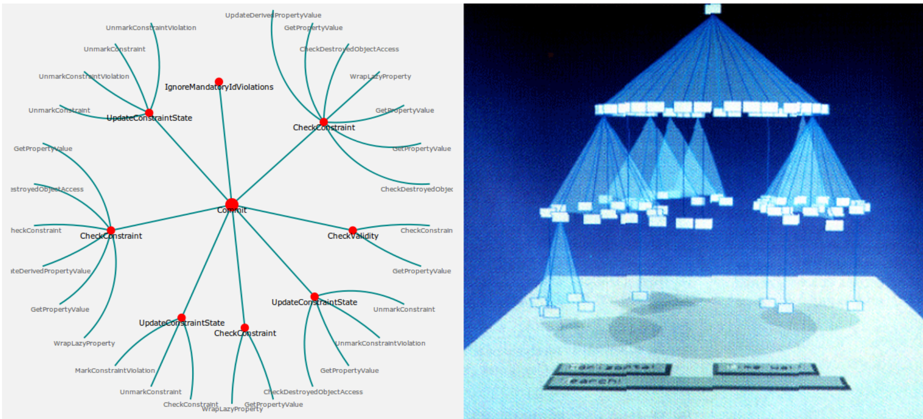

Fig. 7: The left image shows an orchideo trace using hyperbolic space. The root-level is placed in the center. Sub and sub-sub-level are placed in circles around the root-level. The right image shows a Cone Tree, a three-dimensional visualization, as it was presented in the paper of Robertson et.al. [106]. The root-level is the topmost cuboid. Sub-levels are placed below their parent level in a circular manner.

A disadvantage of Tree-Maps is that it feels uncommon to most developers to read Tree-Maps. Furthermore it is hard to see the detailed structure of the trace. Despite the possibility to enlarge certain items through adapting the search pattern, it is not possible to see all items at the same time (with the uncommon exception of small traces displayed on a large screen).

Nevertheless, Tree-Maps are a promising technique for displaying orchideo traces. They enable us to get a fast overview of the whole trace while making it easy to spot the error through giving certain items more space on the screen. As less important items and levels get a relatively small space—going as far as hiding items which would be too small—this visualization naturally hides unimportant information.

2.6 Tree Visualization using a Hyperbolic Space

The traditional way to draw trees is a rooted, directed graph with the root element at the top of the figure and sub-nodes below the parent nodes. Each node is connected with a line to its parent node [105]. The hyperbolic space approach modifies this visualization by laying out the tree on a hyperbolic plane. This plane is mapped on a circular two-dimensional space to be displayed.

This special layout has several interesting consequences. Because the hyperbolic plane is a non-Euclidian geometry, parallel lines diverge from one another. Furthermore this causes the available space to grow exponentially with the distance from the center [107], thus causing the visualizations to resemble a fisheye view [108].

Initially the root node is displayed at the center, which conveys an overview over the first levels of hierarchy and the general structure of the tree. Navigation through the tree is done by clicking on a node to move it towards the center.

Figure 7 shows an orchideo trace on a hyperbolic space, which is visualized using the JavaScript InfoVis Toolkit [101]. Levels and items look similar with the only difference that levels are linked to sub-levels or other items, whereas items are always leaf nodes.

As none of the nodes are highlighted in a special way, it is hard to find a specific action which may have caused the orchideo trace. It feels easy and natural to browse through the trace, but as a trace contains several thousand levels and items it is difficult to find any particular item. Due to this, unimportant information should be hidden in anyways. The hyperbolic space layout consumes only a well-defined amount of space and displays its information without the need to scroll. However, this visualization does not give any hint on the error location. This leads to a situation where a programmer manually searches the tree without having any hint about which branch has already been visited.

2.7 Cone Tree

Cone Trees are three-dimensional tree visualizations to maximize effective use of the available screen space. Furthermore it uses animations to shift some of the users cognitive load to the human perceptional system [106].

The root-node is drawn near the ceiling of the visible three-dimensional room. Sub-nodes are drawn in a layer below their parent nodes layer. Each node which is in the same depth of the trees hierarchy is placed on the same layer. The sub-nodes are placed in a way that they build a cone with their common parent node as the apex. The whole tree is scaled to fit into the available room. Each cone is partly translucent so that each cone is visible, but does not block the view on cones behind it.

If the user clicks on a node, the Cone Tree rotates the target node to the front. Simultaneously all nodes on the path from the targeted node to the root-node are moved to the front, too. The animation reliefs the user from rescanning the image as it uses their perceptual system to track the rotation [106].

Figure 7 shows a Cone Tree as it was originally presented by Robertson et.al. [106]. Due to its tree structure the orchideo trace would fit naturally in a 3D-space using this visualization. Cone Trees have the advantage that we get a quick overview over the whole structure of the tree, as it always scales into the available screen space. This however makes it hard to understand trees with a high level of nesting. Fortunately orchideo traces have a relatively low level of nesting—in all likelihood the depth is lower than twelve [?]. Cone Trees use their depth to store the several thousand levels and items orchideo traces have. Therefore Cone Trees excel in showing the trace in a relatively small space through hiding less important nodes in the depth of the image. Common errors like `MarkConstraintViolations` could be drawn in a similar color as in the Tree-Map. In contrast to the Tree-Map, it is not possible to show further information giving details for each node. In general placing labels for each node is a problem

in Cone Trees because of the amount of items being displayed at the same time overlapping each other. This is solved by only displaying labels for the currently selected path. Unfortunately this may result in wild clicking through the tree to get information for different nodes.

2.8 Which Visualization to Choose?

We presented multiple ways to visualize the orchideo trace: text representations like JSON or XML, the Tree View,the Tree-Map, hyperbolic space representations,the Cone Tree, and the Linear Trace. As we need to find the best visualization technique for our purpose, we briefly compare them with focus on the requirements listed in Section 2.1.

Integration into the Current Workflow. All the presented visualizations can be realized as an Eclipse plug-in within a new view. The actual implementation of the plug-in defines the quality of the integration into a developers workflow. Therefore *drag and drop* or similar methods for handling incoming traces are features that do not depend on the actual visualization technique.

Give a Hint which Part of the Application Code may have Produced the Error. Hints for the cause of an engine crash are hidden at different places within the orchideo trace. When the `ExecutionInterruptedException` is thrown, it prints its execution stack. An engine developer can use that stack dump to find at which point within the engine the error was thrown. An application developer on the other hand needs to skip the engine part of the stack dump (as well as other parts of the dump added by third party frameworks like SWT) to find where his code calls the failing engine. As our visualizations do not provide support in showing and analyzing the execution stack, we implemented the feature to *jump to the failing code* in our Eclipse plug-in.

As an application developer often finds enough information in the execution stack of the engine to see where the crash originated from, an engine developer needs more information. It is possible that Java error-traces are attached to levels and items [**?**]. These traces need further investigation as they, if they occur, encode probably the reason for the engine crash. Here Tree-Maps and the Linear Trace provide good support in highlighting Java error-traces through giving them an important level and therefore hiding other items.

If a developer finds the error causing item, he needs to know the path from that item to the root level. An example path could look like that:
`Commit >> CheckConstraints >> CheckConstraint >> CardinalityViolation`
We see that the path to the root level gives important meta information about the item. It says that there was a failure during the commit action caused by a failing constraint due to a wrong cardinality of some attribute. Apart from the path from the item to its root level, we need further information on the item itself like which constraint failed on what object. Here all our graphical visualizations except for the Tree-Map and the hyperbolic tree provide good support as they show the path from an item to the root level at a glance. Our textual trace

representations fail, because they require too much scrolling to see the path or do not provide enough separation between different levels and items.

Show the Necessary Information in as Small Space as Possible and Hide Unimportant Information We wanted to integrate the error visualization into the orchideo development workspace. Therefore the required space should be as small as possible while simultaneously ensuring usability while looking at the visualized orchideo trace. The Tree-Map, Cone Tree, Linear Trace and hyperbolic tree layout provide good support for that scenario. They hide unimportant information and set focus to important items in the trace.

Unfortunately the Cone Tree has some problems when it needs to display a lot of information as it scales the tree. This makes it hard to separate particular items and levels. As all the items and levels overlap each other, it is difficult to find a specific node because of the sheer mass of nodes being displayed.

As discussed above, hyperbolic trees have the problem that it is uncomfortable to navigate through large trees, especially since most users are not used to this kind of visualization. The Linear Trace and the Tree-Map suit this requirement very well but have some usability issues. Using the Linear Trace the user still needs to scroll vertically and horizontally in most cases. They do not need to scroll using the Tree-Map, but most users told us that they find it hard to read. Apart from that the Tree-Map gives an overview of the trace even if it is displayed on very small spaces.

Find and Highlight Common Errors. It is possible in all graphical visualizations to give some nodes a specific color that indicates that this item may be of interest. Furthermore some search functionality can be implemented for all of them. Again the Linear Trace and the Tree-Map shine, because they highlight and display common errors even at first glance.

Enable Engine Developers to Find Uncommon Problems in the Engine Trace. It is not possible to foresee every possible kind of error that is hidden in an engine trace. Most visualizations that support good information hiding are specialized on common errors as they can highlight important items for that error and hide others.

To enable engine developers to find uncommon problems, like a wrong order of executed actions or wrong sessions used in different actions, we needed to provide a visualization that displays the whole trace without hiding information. Furthermore the order of actions should be the same as in the original trace and all possible information available for any specific item or level should be reachable. The Tree View provides the best support for that case, as most developers are able to use a Tree View intuitively. Furthermore the Tree View allows us to reach every node of the tree while giving the option to fold and therefore hide certain branches.

Because none of the visualizations fully suits all requirements, we decided to implement a plug-in that combines three techniques: the Tree View, the Linear

Trace, and the Tree-Map. Once a trace is loaded using one of these visualizations, it should be possible to switch to every other visualization we offer without loosing context. That enables us to benefit from the advantages of each of these visualizations.

3 Implementation of the Trace View Plug-in

We presented several methods to visualize the orchideo trace and discussed which combination of visualization techniques serves us best. Furthermore we decided to use the Tree View as it is a general purpose tree visualization with very good performance in terms of task completion and user satisfaction[109]. The Linear Trace was taken as a mutation of the Tree View, because it offers better information hiding while having the same interface as the Tree View. Finally we decided to include a Tree-Map visualization into our trace view plug-in as it gives an overview of the trace with minimal space requirements.

In this section we present how we integrated those visualizations into the orchideo suite. We discuss the overall architecture of our plug-in as shown in Figure 8 and how it is integrated in an orchideo developers workflow. Afterwards we explain the implementation details of each of the different visualizations and common features among them.

3.1 Trace View as an Eclipse Plug-in

To support a developers workflow we directly integrate our visualizations into their development environment—orchideo. As orchideo is based on Eclipse [4] we implemented an Eclipse plug-in that can read and analyze an orchideo trace [?] and is able to visualize it. Our plug-in adds a new view to the orchideo suite called trace view. Initially it opens the Linear Trace view we presented earlier. As there is no initial orchideo trace to be displayed, it shows a text explaining that *copy and paste* or *drag and drop* may be used to give a new trace to the view.

Technically we distinguish two views only: the Tree-Map and the Tree View. This is because the Linear Trace is equivalent to the Tree View, except that it applies a filter to the tree which hides unimportant items as described in Section 2.4. Both views use the same input which is given by the orchideo trace parser described by Tim Felgentreff [?]. The parser generates a tree containing `Level` and `Item` objects, which have the same semantics as the levels and items we described earlier. Both `Level` and `Item` are subclasses of the abstract class `StackNode`, which implements some common behavior. This includes for example the following methods:

`getActionName()` returns the name of the action this `StackNode` represents. If the `StackNode` is a `Level` it returns the value for the levels root action for this method and all methods listed below.

`getFields()` returns all optional name-value pairs which are given by the represented action as a `HashMap`.

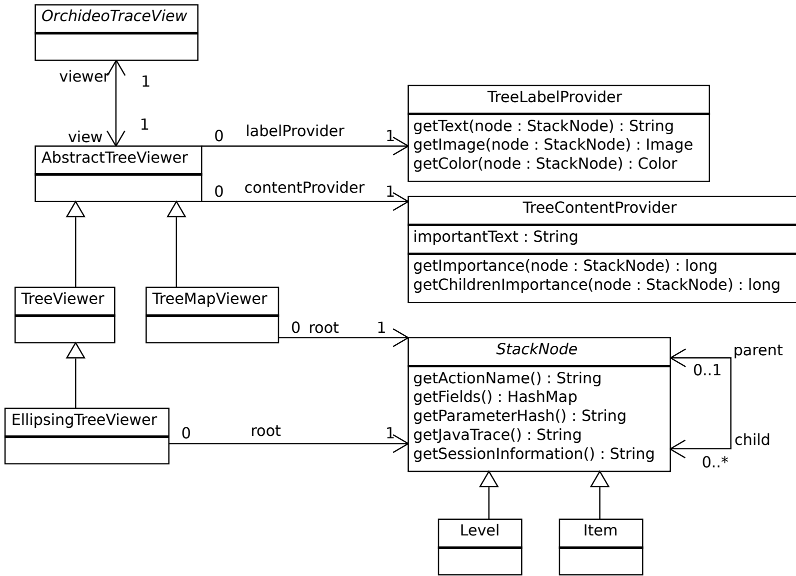

Fig. 8: A class diagram showing the relation between our plug-in (`OrchideoTraceView`) and our viewers. Each of our viewers knows the root node of the tree it displays. A viewer may get further information for a specific node through their content provider and label provider.

`getParameterHash()` returns the parameter hash which is given by each item of the orchideo trace.

`getJavaTrace()` returns any text which was placed within or after an item, but does not belong to the item itself. Usually this is a Java error-trace of an exception that was thrown within the orchideo|engine.

`getSessionInformation()` returns an object that holds the information which session was used by the orchideo|engine while executing the action this item represents. The information includes the session name and an ID identifying the session.

Further methods like `getChildren()` and `getParent()` have an obvious meaning and are used by both the Tree View and the Tree-Map to gather the required information to display the tree. The standard SWT tree view requires to get information by a `ContentProvider`, which offers information about the tree structure, and a `LabelProvider`, which gives information about the appearance of a specific node[45]. We added a `TreeLabelProvider` and a `TreeContentProvider`, which are used by the Tree View and our Tree-Map.

Our `TreeContentProvider`, additionally supports retrieving the importance of an item or a level. The importance is used by Linear Trace and Tree-Map to decide which items to show. We define the importance of levels or items as a

non-negative number, which is the sum of the importance of the object itself (local importance) and the importance of all its children, where 0 indicates the lowest importance. The content provider provides methods to get the importance of an object (`getImportance()`) or the importance of the objects children only (`getChildrenImportance()`).

Levels contain information about the structure of the orchideo trace. The Tree View, which we proposed to analyze details of the structure of the trace, ignores importances. But the Tree-Map and the Linear Trace need levels to be important if they contain important items. Therefore we give each level the local importance of zero, which means its importance is defined by its children only. This lets a level's importance increase with the importance of its items. The Tree-Map and the Linear Trace view can thus enlarge/display levels if their items are important.

To boost the importance of certain items, the user may define a text pattern. If an item matches this pattern it will be given a higher importance. The user may use patterns analogous to the class search functionality of Eclipse. If the user wants to highlight items which contain the text `MarkConstraintViolation` (the default search pattern as most errors are hidden in items with those violations), he may search e.g. for "MarkConstraintViolation", "MaCoVi", or "ma*on*ion". We built a `RegexHelper` class that is able to generate a regular expression when given such a pattern as input. Additionally items and levels get a higher importance if a Java error trace is attached to them. This is useful for orchideo|engine developers, because sometimes—when there is an error in the implementation of the orchideo|engine—the Java error is the reason for the trace. The implementation of the orchideo|engine makes certain Java errors appear often in the trace. We give only one of these errors a higher importance, to avoid flooding the visualization with copies of one and the same Java error.

3.2 Implementation of the Tree View

So far we have discussed the theoretical background of our plug-in. Now we want to present parts of our implementation that are actually involved in displaying the tree visualization techniques discussed in Section 2.

We reused the JFace TreeView, which uses an SWT tree widget to display tree structures [110]. It is the Tree View that handles the details of expanding and collapsing items as shown in Figure 9. We extended the standard Tree View implementation to support highlighting items. This should be done by ensuring that the item is visible (ensure that all branches up to the item are expanded and the view is scrolled to a position where the item is visible) and selected. Furthermore we had issues concerning the non-uniform way item labels are displayed on different operation systems. The labels of our items usually have more than one line of text. On Windows machines this was not a problem as the windows SWT tree implementation only displays the first line. However the Linux implementation displays all lines of the text, which made the view difficult to understand due to each item taking up a lot of space on the screen. Since Linux is the main development platform at ex|xcellent solutions, we made

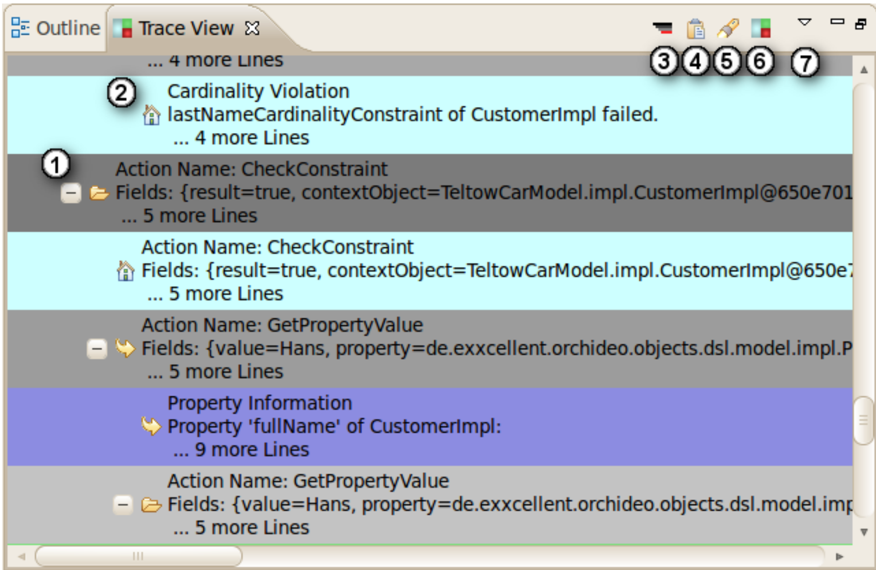

Fig. 9: This is a Tree View showing an orchideo trace. The numbers ① and ② are referring to different parts of the visualization, while the ③ to ⑦ refer to the tool bar of our view.

the maximum number of lines an item may consume a preference to enable the user to decide which implementation suits him most.

We subclassed the JFace `TreeViewer` to extend it with the new functionality. Unfortunately the tree viewer comes with some restrictions which are stated in the Eclipse documentation [111]:

> ”This class is not intended to be subclassed outside the viewer framework. It is designed to be instantiated with a pre-existing SWT tree control and configured with a domain-specific content provider, label provider, element filter (optional), and element sorter (optional)“.

Therefore we used custom tree and content provider, as discussed in Section 3.1. We ignored the restriction and subclassed the viewer, to extend it as intended. This lead to some less elegant parts in our code (e.g. some `instanceof`-checks to ensure we have our custom viewer). However, it made our Tree View implementation support highlighting items and added support for adjustable multi-line labels for items on Linux systems.

Our label and content provider make the tree structure that the trace parser produces accessible to our modified `TreeViewer`. Finally our plug-in supports the Tree View visualization as presented in Section 2.3: The line of the Tree View which is marked as number ① in Figure 9 shows a currently expanded level. Through collapsing and expanding levels we are able to explore the orchideo trace in detail to find items which point to errors in the application code. The item

shown near ② refers to a cardinality violation, which is a common error hidden in orchideo traces. We instantly see that the last name cardinality constraint failed on a customer. To get further information we can hover over this item to see a tool-tip with a more detailed description. Numbers ③ to ⑦ refer to the views tool bar, which we discuss in Section 3.5. The Tree View is the first visualization technique for which we discussed the implementation. We want to present the implementation of the Linear Trace view and the Tree-Map in the following sections.

3.3 The Linear Trace as a Simplification of the Tree View

The Linear Trace we presented in Figure 5 basically has the same functionality as the Tree View but only shows a subset of the items. It shows only important items and their levels up to the root-level. The JFace `TreeViewer`, which we subclassed as described in Section 3.2, supports hiding elements through filters [111]. This, along with our content provider offering information about the importance of items, enabled us to hide less important items.

We added an action to our `TraceView` which provides support to enable and disable a `LinearImportanceViewerFilter`. The action is accessible through an iconic button and a menu entry in our plug-in view. If the filter is applied, our tree viewer asks the filter for every item whether it should be displayed or not by calling the `select()` function. This function is intended to return `true` if a given object should be displayed on the viewer. The filter selects an item if a node (or one of its subnodes) has a higher importance than a specific *base importance* or matches a search pattern, which can be set in the preferences.

Developers using orchideo may now load traces into the Linear Trace view and are not confronted with thousands of items anymore. They can see items they are looking for by defining a search pattern. This pattern is predefined to show the most common errors hidden in orchideo traces, which are cardinality violations. They can explore the actions showing the error and actions that caused the execution of erroneous actions, because orchideo-actions are expressed as items in the visualized trace. If it is necessary to get more information about a specific item, again the information can be accessed through a tool-tip that appears when hovering over the item.

Furthermore the user may decide to switch to the Tree View to get additional information about surrounding (yet hidden) items. Alternatively he can switch to the Tree-Map view if he wants to continue his work but simultaneously wants to have the orchideo trace at hand displayed on a small part of his screen.

3.4 Implementation of the Tree-Map

There was no predefined view for the Tree-Map visualization, which is why we decided to implement it ourselves. The Tree-Map view, which is shown in Figure 10, is a graphical visualization of the orchideo trace. As explained in Section 2.5, a Tree-Map consists of a rectangle for every node in the tree. Each rectangle contains the rectangles for its child-nodes within its bounds. Furthermore we

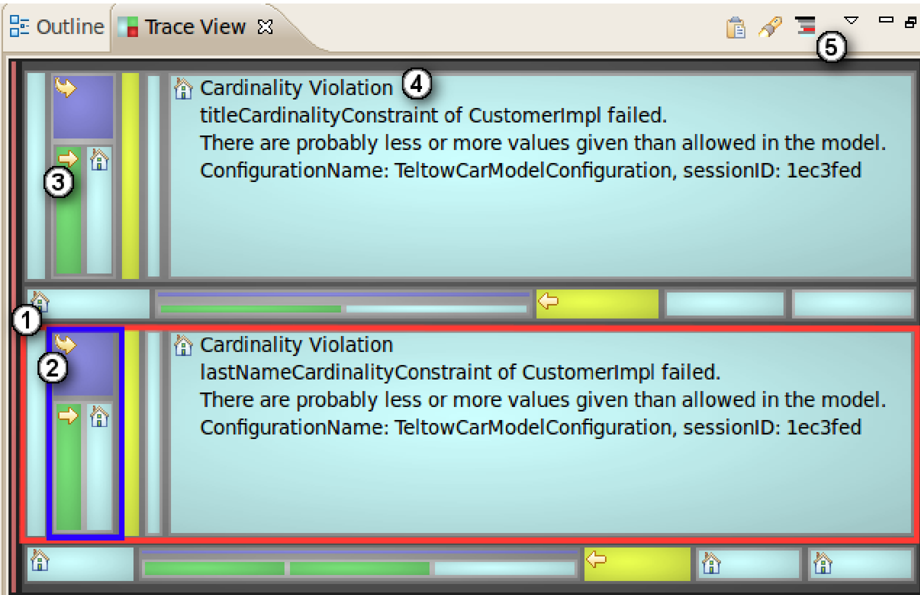

Fig. 10: This is a Tree-Map showing an orchideo trace. The numbers refer to different parts of the visualization.

wanted to display an icon and a description within large items of the tree. The icon indicates whether the item represents a before-, after-, around-, or root-action. The description is a text as seen in Figure 10, ④ for cardinality violations (the large light blue rectangles).

We implemented a class `TreeMap`, which is a subclass of the SWT canvas widget. Objects of that class, when given an appropriate root-node, a content provider, and a label provider, are able to draw Tree-Maps. To that end they instantiate `TreeMapRectangles` and pass them the `StackNode` (which is a level or an item, as discussed in Section 3) the rectangle shall represent. The sequence diagram in Figure 14 shows the Tree-Map drawing process.

The `TreeMap` creates a new `TreeMapRectangle` and gives it the root `StackNode` to be displayed. After setting the label and content provider, it calls the `drawTree()` function. The newly created rectangle attempts to draw itself within the given bounds. Therefore it asks the label provider for the color in which it will draw itself on the screen. To give it a better visual experience, a slight highlight is added to the plain color of the rectangle. In the second step the rectangle asks the content provider for the children of the current `StackNode`. The rectangle calculates the space each of its children will be given depending on their importance. The more important a child is, the more space may it have to draw itself. For each child the rectangle creates new `TreeMapRectangles` which recursively continues the drawing process.

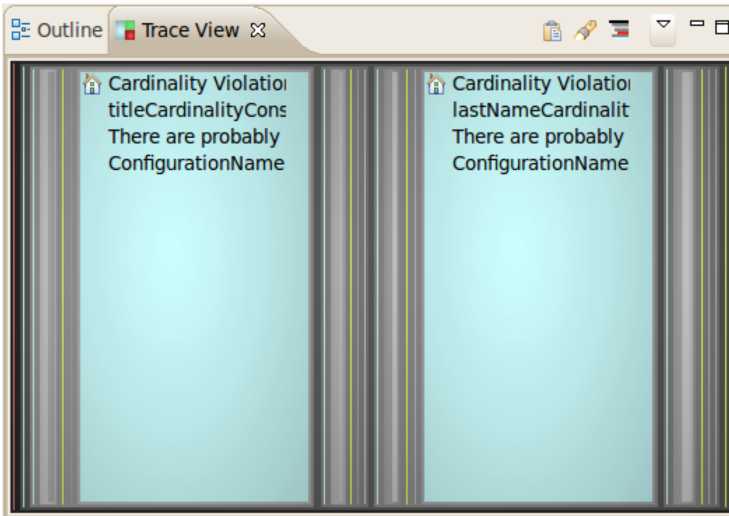

Fig. 11: An orchideo trace displayed in a modified Tree-Map. This Tree-Map does not switch between vertical and horizontal orientation while drawing its rectangles.

Each `TreeMapRectangle` draws its sub-rectangles with a different orientation. When a horizontal orientation is applied, the child rectangles will be drawn from left to right, whereas they are drawn from top to bottom on a vertical orientation. The rectangle with the red border near ① in Figure 10 draws its children with a horizontal orientation applied. It is crucial for the Tree-Map that the orientation switches between vertical and horizontal orientation with each recursive step. We do the orientation switch within the `calculateBounds()` method seen in Figure 14. What happens if the orientation switch is not done is shown in Figure 11, where we applied a horizontal orientation for all rectangles. Here the two constraint violations dominate the picture occupying the same space as before, but leaving the rest of the picture as an undefined gray area. This visualization does not display any structure of the trace, and leaves valuable space unused.

The recursion will stop when items with no further sub-rectangles are drawn. When there is enough space available, the rectangle representing the item contains additional information about that item. As seen in Figure 10 near ③ we print a small icon that indicates the kind of action (before, after, around, or root) along with the background color of the rectangle. If there is even more space available, we print a description of the item, which we parsed and analyzed out of the orchideo trace. An exemplary description can be seen near ④. Those descriptions are different for every kind of action that is parsed. For `MarkConstraintViolations`, we are able to say which kind of constraint violation triggered the violation. In Figure 10 the violation was triggered by a cardinality

violation. Another possible constraint violation is the OCL violation. In addition to the kind of violation we can print the reason for the violation, which is for example a title cardinality violation of a customer object. Furthermore we print a hint, which helps to find the causing error in the application code.

Even with limited screen space, the Tree-Map offers a visualization which shows the structure of the trace, highlights errors that the users is searching for, and gives a detailed description of the error. Nevertheless it has some weaknesses. The Tree-Map hides some items with low importance, which would be too small to be displayed. This makes the Tree-Map less useful to explore the orchideo trace in detail. We noticed that the Tree View serves this scenario best, so we implemented a handy shortcut to switch to the Tree View while keeping the focus on a specific item. If the user double-clicks an item (or a level) in the Tree-Map it switches to the Tree View and highlights this specific item. This enables the user to rapidly locate an item or a level in the Tree-Map and then switch to the Tree View to examine the items neighborhood.

3.5 Shared Features of the Visualizations

The visualizations we discussed earlier are distinct, but nevertheless share some functionality. As we wanted to integrate our plug-ins into the workflow of the orchideo developer, we provide *drag and drop* and *copy and paste* support, which we briefly discuss in this section. Furthermore we want to discuss some details of the previously mentioned tool-tip and search support.

Our *TraceView* plug-in adds a new view to the orchideo suite, which is implemented by the class *OrchideoTraceView*. This class is responsible for switching between our visualizations. Furthermore it shows a tool bar and implements other visualization independent behavior. It uses the Eclipse plug-in API to register paste support and drop support for files. Both the drop handler and the paste action forward their input to the content provider. In turn the content provider starts a new parser thread [**?**] to retrieve our tree structure to display. The parser however is able to handle files as well as text from the clipboard. That way the content provider only needs to take the parsed output or handle errors like parsing errors or file not found errors. As the content provider knows the view it is connected to, it can call `refresh()` on the view to update it.

To make the clipboard paste-support explorable, we additionally added an icon to the views tool bar to paste orchideo traces into the view. This icon is shown in Figure 9 near ④. Other icons on the tool bar are: an icon for toggle between the linear trace and the Tree View ③, set the search pattern ⑤, toggle to Tree-Map ⑥, and a text based menu offering the same options ⑦. The toggle to Tree-Map icon is replaced by another icon when the Tree-Map is displayed (Figure 10, ⑤), which switches back to the Tree View.

Along with the other icons, we added an icon for search support to the tool bar. A click on this icon, as well as the keyboard shortcut CTRL-F, opens a small input window as seen in Figure 12 near ①. Changes in this window change the search pattern which is used to calculate important items. As changes are applied while typing, the user gets a fast response on his search for the visualizations that

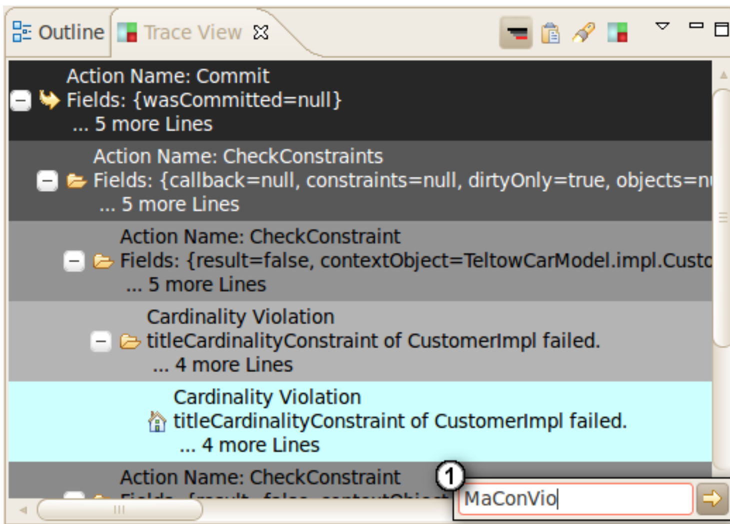

Fig. 12: An orchideo trace displayed using the Linear Trace view. The user currently edits the search pattern ①.

apply information hiding based on the importance of nodes. When the changes are applied by hitting the return-key or a click on the nearby button, the next item or level that matches the search pattern will be selected and scrolled into view. When doing the same search again or pressing the keyboard shortcut CTRL-K, the following item matching the pattern will be highlighted. Of course, this only works in the Tree View and the Linear Trace, as the Tree-Map does not support scrolling and separate highlighting of items.

In addition to the previously discussed functionality we extended the orchideo|engine to print some additional information into the trace [?]. When an `ExecutionInterruptedException` is dumped into a log file or the Eclipse error console, the engine prints the code location where the error was thrown just before the original trace. Our trace parser is able to read this optional part of the trace. We visualize this execution stack, that was printed when the `ExecutionInterruptedException` was constructed, in our *Exception Constructor Stack* view as seen in Figure 13. With the Exception Constructor Stack it is possible to see where the exception causing the orchideo stack was thrown without browsing through SWT, JUnit or other framework code. The view explicitly shows the orchideo|engine part of the stack to support orchideo|engine developers who want to see where the error occurred in their engine. To support application developers using orchideo each line of application code in the execution stack is marked with a red background color and an arrow. This enables the application developer to immediately see where in his code the error was caused. In Figure 13 we can see that the error was thrown in the `Activator` class of the TeltowCar [?] application.

Exception Constructor Stack

Method	Source
java.lang.Thread.getStackTrace	null:-1
de.exxcellent.orchideo.engine.ExecutionInterrupte	ExecutionInterruptedExceptior
de.exxcellent.orchideo.engine.impl.ExecutionCont	ExecutionContextImpl.java:93
de.exxcellent.orchideo.engine.impl.ExecutionCont	ExecutionContextImpl.java:57
de.exxcellent.orchideo.engine.impl.SessionImpl.e	SessionImpl.java:197
de.exxcellent.orchideo.objects.aspect.persistence	HibernateAspectImplBase.java
de.uni_potsdam.hpi.bp2009h1.teltowcar.applicatio	Activator.java:52
org.eclipse.osgi.framework.internal.core.BundleCo	BundleContextImpl.java:782
java.security.AccessController.doPrivileged	null:-2
org.eclipse.osgi.framework.internal.core.BundleCo	BundleContextImpl.java:773
org.eclipse.osgi.framework.internal.core.BundleCo	BundleContextImpl.java:754
org.eclipse.osgi.framework.internal.core.BundleHo	BundleHost.java:352
org.eclipse.osgi.framework.internal.core.AbstractB	AbstractBundle.java:280
org.eclipse.osgi.framework.util.SecureAction.start	SecureAction.java:408
org.eclipse.core.runtime.internal.adaptor.EclipseLa	EclipseLazyStarter.java:111
org.eclipse.osgi.baseadaptor.loader.ClasspathMan	ClasspathManager.java:449
org.eclipse.osgi.internal.baseadaptor.DefaultClass	DefaultClassLoader.java:211

Fig. 13: This information window shows where the orchideo trace was originally thrown. Our plug-in marks the part of the potentially long execution stack where application code was executed.

4 Evaluation

We discussed our approach to visualize the orchideo trace in detail in the previous sections. Our goal was to extend the orchideo suite with an error visualization technique, that enables orchideo developers to better read and understand orchideo traces. In this section we discuss the applicability of our implementation in consideration to the requirements we defined in Section 2.1. Furthermore we discuss future work to improve the performance of the trace view plug-in.

4.1 Our Implementation in Consideration to the Requirements

In order to evaluate our implementation, the requirements that were defined in Section 2.1 are reviewed. For each requirement we discuss whether our plug-in suits the users needs. Subsequent to the discussion, we evaluate if we reached our goals as intended.

Integration into the Current Workflow. Our extension to the orchideo suite integrates seamlessly into the programming environment. An additional view can be added to the developers workbench, when our plug-in is loaded. The Trace View could be placed for example at the place where the Outline view is located. In this way it occupies a minimal amount of space while still being able to give

valuable information about the trace. For further investigation, a developer may switch to the Tree View in full-screen mode, which Eclipse provides for every view.

orchideo traces usually exist in the form of log files on production systems, or as text such as in the orchideo error console. Our plug-in supports both cases as it is possible to drag files into the Trace View from the users file manager, or just paste a trace from the clipboard into the view. The latter case could be optimized, however, as the trace is printed in the orchideo error console and a developer needs to manually feed our view with that trace. A better solution would be that the plug-in discovers and visualizes the trace in the error console without any interaction of a developer.

To solve that problem, we implemented an additional plug-in called *Runtime Rescue* [**?**], which is able to automatically detect `ExecutionInterruptedExceptions`. If the exception is thrown, the plug-in halts the execution of the program just before the exception would be thrown and automatically jumps to the application code that was executed just before the orchideo|engine was called. Simultaneously the trace that would have been thrown by the exception, is displayed in our Trace View. With the help of the Runtime Rescue plug-in a developer can skip several manual steps which were required previously to debug orchideo applications. In combination, both plug-ins enable real-time debugging [75] of orchideo traces, allowing a developer to see the error in the visualization *and* still check the live objects.

Give Hints as to which Part of the Application Code may have Produced the Error. With the old orchideo trace it was hard to find the part of the developers source code that was responsible for the trace. We implemented a mechanism that handles the given execution stack to find the code that caused the trace.

In order to do so, we slightly modified the part of the orchideo|engine that dumps the trace to add the current execution stack to the top of the trace. This execution stack can be viewed by the Exception Constructor Stack window shown in Figure 13, which points to the code that may have caused the trace. When debugging an orchideo application, the functionality provided by the Runtime Rescue plug-in is faster as it automatically jumps to the code in question instead of indirectly referring to it. Nevertheless the Exception Constructor Stack window is useful in post-mortem situations, especially when the trace comes in the form of log files from a remote production system.

Show the Necessary Information in as Little Space as Possible. The Linear Trace view and especially the Tree-Map view requires only little space on a developers screen. Both views are able to display several thousand lines of orchideo trace within the space used by the Eclipse Outline View. The views accomplish this by hiding unimportant information. As information hiding always comes with a loss of information, we added the functionality to switch to the Tree View without loosing track of a specific item. This allows the Trace View to show all the necessary information while using as small space as possible.

Find and Highlight Common Errors. Referring to our project partners at ex|xcellent solutions, constraint violations are the most common cause of an orchideo trace. Our default search pattern, which is applied when opening the view, highlights constraint violations. This enables a developer to find such violations easily.

Another kind of error in the orchideo|engine that occurred sporadically is related to a lot of Java exceptions being included in the trace. As we give nodes with those Java exceptions attached a higher importance, they can be spotted with the Tree-Map or the Linear Trace view. If there are multiple nodes containing the same Java exception, we only raise the importance of one of these nodes. This prevents us from flooding our views with too many highlighted items.

If the order of executed actions is important, the error can be found using the Tree Views search functionality. Consequently all common errors, which are the errors listed above, can be found using our visualization with far less effort needed than reading through the original orchideo trace.

Hide Unimportant Information. Through calculating the importance of every item and level, the Linear Trace view and the Tree-Map are able to hide unimportant items. The importance of items can be changed by setting a flexible search pattern, which enables a developer to only see items he wants to see.

Enable Engine Developers to Find Uncommon Problems in the Engine Trace. As the Tree View does not hide any items of the trace, every possible information which is encoded in the trace can be found. This enables a developer to even find errors in the trace that we have never thought of. However we can not evaluate if it is convenient to spot those errors.

Overall, our implementation suits the requirements listed above in most cases. But especially when orchideo developers want to live-debug their application, our implementation does not provide optimal support to their workflow. We have implemented some more plug-ins during our project, which are not presented in this paper, such as the Runtime Rescue or the Session View plug-in [**?**]. As these plug-ins specialize more on live-debugging, they can provide optimal support, where the Trace view plug-in flags.

4.2 Future Work

Even though our implementation suits the requirements, we see potential to improve certain details. For example we currently show session information for every level and item in the trace. This session information includes the name of the active aspect configuration at the execution time of the represented action. Furthermore it includes an ID referring to the actual configuration object stored in the orchideo|engine. During a live-debug session, it would be useful to link the aspect configuration part displayed in the Trace View to the actual object in the Session View [**?**]. Unfortunately the performance of the trace parser is not sufficient [**?**], which possibly leads to undesirable latency when re-parsing the

trace is needed. On the other hand there are often few different configurations active, which makes it an easy task to instantly see which configuration was used. Because of the poor cost-benefit ratio we actually do not concentrate on further optimizations on this issue.

Another detail concerns the Tree-Map visualization, as the majority of developers we interviewed were not instantly able to understand the Tree-Map. They were confused due to the unusual visualization. But after an explanation most of them were able to use it—some of our interviewees were even enthusiastic about the higher information density. We can support the process of understanding the Tree-Map by implementing some further functionality: we want the Tree-Map to highlight those rectangles that the mouse is hovering over, which would possibly help the user to explore and understand the tree structure displayed in the Tree-Map. With that modification it would be easier to trace which sub-levels and items belong to the level at the mouse location.

4.3 Influence on the orchideo Developers Workflow

When we initiated the project, we interviewed orchideo developers about their current workflow. A widespread problem was that developers are often confronted with an orchideo trace. These traces occur during the development process—while debugging—or when the software is deployed and orchideo|engine crashes are found in the log files. The usual workflow to handle orchideo traces was to forward those traces to an orchideo|engine developer, who is able to understand the traces. This led to a major slowdown of the development process, as an engine developer's time is highly occupied.

We interviewed orchideo developers again, after they tested our plug-in and got throughout positive feedback. The feedback proved that the orchideo developers workflow has already changed. Using the plug-in we developed to visualize orchideo traces, developers are able to identify the problem that caused the trace. As it does not matter if the trace was created with or without our modifications, or if the trace was created on a developers machine or remotely, our plug-in offers a post-mortem analysis of traces which even works with older versions of orchideo.

orchideo developers now handle traces by themselves. The error finding process is much faster, since the application developers can work independently of the orchideo|engine developer. The plug-ins are in active use at ex|xcellent for developing orchideo applications.

5 Conclusion

We presented several tree visualization techniques and discussed whether they provide good support for error visualization based on the example of orchideo traces. Furthermore we implemented a plug-in for Eclipse containing a combination of three visualizations we considered optimal for our case: The Tree View, the Linear Trace view, and the Tree-Map. The visualizations are displayed within

a new view our plug-in provides for the orchideo workbench. Additionally we discussed the implementation of the plug-in and the visualizations in detail. The evaluation showed that the combination of these three visualization techniques suits our requirements. Finally we have seen that our implementation is already used in a production environment by our project partner ex|xcellent solutions.

We conclude that the error visualizations we provide actually helps in understanding orchideo traces. In combination with other plug-ins and enhancements we implemented for orchideo [?][?][?], we believe that the trace View plug-in effectively supports orchideo developers writing their applications and orchideo|engine developers who extend orchideo.

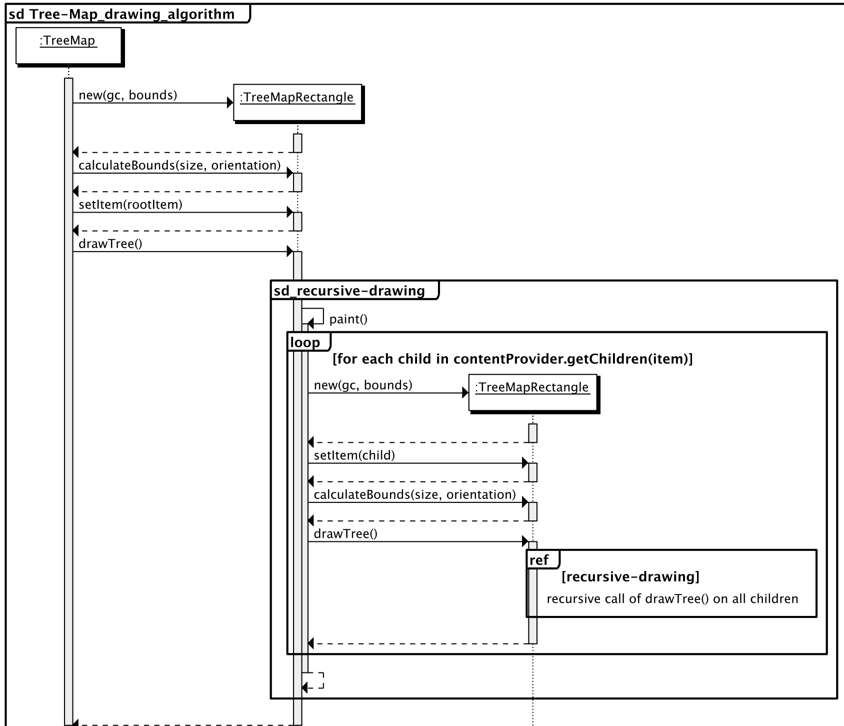

Fig. 14: The basic drawing algorithm of `TreeMapRectangles`. First the rectangle draws itself and then recursively each of its child rectangles.

Bachelor Thesis

Continuous Integration For Eclipse Plug-ins

Frank Schlegel

Supervisors:

Dr. Michael Haupt, Malte Appeltauer
Prof. Robert Hirschfeld
Software Architecture Group
Hasso Plattner Institute,
Potsdam, Germany

June 25, 2010

Continuous Integration For Eclipse Plug-ins

Frank Schlegel

Hasso Plattner Institute
Potsdam, Germany
frank.schlegel@student.hpi.uni-potsdam.de

Abstract. In our project work we developed plug-ins that extend the Eclipse IDE in various ways. To reduce integration risks we wrote unit and UI tests and set up a continuous integration system. Unfortunately existing CI solutions for Eclipse plug-ins do not meet all of the requirements we identified especially in term of UI testing. Therefore we developed an own CI solution that meets these requirements and we suggest it to everyone who develops Eclipse plug-ins. In this paper we describe this solution in detail.

1 Motivation

The orchideo|suite [4] is an extension to the Eclipse IDE [112] which brings support for aspect-oriented programming (AOP) and model-driven software development (MDSD) as described in paper [?]. The purpose of our project work was to extend orchideo with plug-ins to improve the debugging support for the AOP and MDSD implementation of orchideo. These are mainly extensions that contribute to the Eclipse IDE user interface including views, preference panes and buttons. The plug-ins are discussed in detail in [?], [?] and [?].

The use of *continuous integration* (CI) [113,114] in other projects convinced us of its benefits [115,116]. Hence we decided to setup a CI infrastructure for the Eclipse plug-ins we wanted to develop as well. After we became acquainted with the Eclipse Plug-in Development Environment (PDE)[1] and gathered some experience with orchideo while developing the demo application *TeltowCar* [?], we defined requirements on a CI system for Eclipse plug-ins.

As we tried to setup a CI server that meets our requirements we soon reached a point where existing tools were too restricted for our purposes. Therefore we decided to use the extensible *Hudson* CI server and customize it by writing own plug-ins for it. As we furthermore needed Eclipse to run automated on the CI server to test our plug-ins, we also decided to write plug-ins that enables Eclipse to be controlled remotely by Hudson.

As the requirements we defined are strongly driven by the principles for a continuous integration system laid down by Martin Fowler [117], Section 2 will give a brief introduction in these principles and the benefits of continuous integration. The requirements we defined are introduced in Section 3 followed

[1] http://www.eclipse.org/pde/

by an introduction on existing continuous integration tools with a discussion on their advantages and drawbacks in Section 4. In Section 5 we present the tools we developed for our CI setup to meet our requirements and demonstrate an example work flow in Section 6. In Section 7 we evaluate our solution followed by a comparison with related work in Section 8. Section 9 forecasts some possible extensions to the proposed setup and on future work. In Section 10 we finally give a short conclusion of our work.

2 An Introduction to Continuous Integration

The term *"Continuous Integration"* has emerged in the Extreme Programming community and was established by Martin Fowler [117] and Kent Beck [118] in 2000. Continuous Integration stands for quality improvement, rapid bug detection and decrease of integration time. Since Extreme Programming becomes more and more popular in software development, the use of continuous integration systems becomes more self-evident. Along with a source code management system it is an essential part of most development setups.

This section gives a brief introduction to the key practices of continuous integration introduced by Martin Fowler [113] and points out the benefits of using continuous integration systems.

2.1 Practices of Continuous Integration

Maintain a single source repository. The basis of every continuous integration system is a single source repository, in most cases managed by a Source Code Management (SCM) tool like Subversion [119] or Git [120]. This repository must be a well known place for everyone involved in the project. It should contain everything needed to perform a build, but nothing actually built to avoid conflicts of system specific binaries and to reduce the amount of data in the repository. It should be possible to do a checkout on a clean machine and fully build the system.

Automate the build. The build process of the project should be fully automated. There should be one single command to build and launch the project on a clean system. There are several common automated environments for builds, like GNU Make [121] and Apache Ant [122], that are designated to that job. They also offer the possibility to build alternative targets for different cases, for example different binaries for different platforms. This is especially useful for performing *staged builds* (see below).

Make the build self-testing. A program may run after successfully building the executable, but that does not mean it fulfills its requirements. To ensure the correctness of the code, automated tests should be performed with every single build [123]. The failure of a test should cause the build to fail. Suites for automated tests, like the xUnit family [124], provide the possibility to run automated tests with a simple command. They also provide detailed reports of the test results and integration in a lot of IDEs like Eclipse.

Everyone commits to the repository every day. The hardest problems are those which are detected too late. Frequent commits to the repository allow to find and fix problems quickly. It is important that a developer can correctly build his code, including passed build tests on his local machine, before committing. The developer first updates their working copy from the central repository, resolves any conflicts and builds on his local machine. If the build passes, the developer may commit to the repository. It is a good idea to commit at least once per day. So every developer gets changes of the other developers frequently and errors can be detected almost as soon as they emerge.

Every commit should build the mainline on an integration machine. The mainline, meaning the main development branch in the repository, should stay in a healthy state to ensure that each developer can operate on a stable base. Therefore it is necessary to find and fix problems as quickly as possible. This can be achieved by building the mainline with every commit. Since the developers are responsible for their commits, they must be able to monitor the mainline build. The most comfortable way for achieving good visibility is by using a continuous integration server that acts as a monitor to the repository. Most of them provide a web interface with visual feedback of the project's state which should be accessible to every developer committing to the project.

Keep the build fast. The purpose of continuous integration is to provide rapid feedback. A build that takes a long time is very counterproductive. The goal should be to have a build that takes at most ten minutes. Because tests may take a long time to run, it is technically not always possible to run a build within ten minutes.

The common way to bypass long waiting time is the usage of a *staged build*. The idea of a staged build is to perform multiple builds in sequence. A commit to the mainline triggers the first, so-called *commit build*. It performs all the actions that are needed to ensure that the build is stable enough for other people to work on [123]. If the commit build passes, the developer is free to resume his work; if not, he has to fix all errors occurred. In the meantime, the CI system performs all the actions that take too long to be performed within the commit build. If something went wrong in the second build phase, it is a good idea to add a test for that error to the commit build.

Test in a clone of the production environment. A running system on the integration server is not always a guaranty for a running system in the customer's environment. To eliminate as much error sources as possible it is advisable to set up the test environment to be as exact a mimic of the production environment as possible. An emulated test environment, for example provided by virtualization, can slow down the test process. Therefore it is common to have an artificial test environment for fast commit tests and a production clone for secondary testing.

Make it easy for anyone to get the latest executable. Validating the final product is a difficult process. Therefore it is necessary that anyone involved in the project

should be able to get the latest executable and be able to run it. This can be easily achieved by providing a well known place where people can find the latest executables to pass the commit tests. It is also advisable to put executables of several versions in such a store to always have a backup.

Everyone can see the result of the last build. Since continuous integration lives on communication, it is essential that everyone can easily see the state of the system and the changes that have been made to it. Most CI systems provide a web interface that provide all the important information like the state of the builds, or the last changes in the repository. Some teams prefer to hook up a more recognizable display to the build system, like a traffic light signalizing the current state of the mainline build.

Automate deployment. Since deployment is a part of integration, there should also be scripts that will easily allow automatic deployment into any environment. Each time the application is deployed, the infrastructure will be reset to as close to the base state as possible. After the application has been deployed to that clean environment, a suite of simple deployment tests will prove the success of the deployment process.

2.2 Benefits of Continuous Integration

Continuous integration is primary about reducing risks [115]. By integrating frequently defects can be found nearly as soon as they emerge and it is much easier to get rid of them. It turns out that projects using continuous integration tend to have considerably less defects [125].

It is hard to estimate the time spent for integration if deferred until after the actual development. By integrating with every commit there is no need to take care of it. There is always a deliverable, working version of the product built automatically on the CI server [123].

Modern continuous integration servers provide a clearly arranged web interface, and are quickly installed and configured. The amount of time needed to install a CI system is probably smaller than the time it takes to integrate *after* the actual development. They also deliver an insight into the current state of development to the stakeholders—which changes were made to the repository, which features are covered with test cases and which are not, and the quality of the code.

3 Requirements on a CI System for Eclipse Plug-ins

In our selection of components for our CI setup we specifically regarded the following requirements. These requirements arise from the principles discussed in Section 2.1 and from the experiences we made while developing Eclipse plug-ins.

From scratch. After every test run, the CI system should be reset to its original state. This way the testing environment for the plug-ins should be simulated as realistic as possible. Our plug-ins should be able to run in a untouched orchideo and thus need to be tested in this environment.

Simulated production environment. The test environment on the CI server should simulate the actual production environment of the customer as good as possible. In our case this means running an Eclipse IDE on the server that is a copy of the customers Eclipse instance; in our case it is a virgin orchideo copy. Running the plug-ins in any artificial Eclipse environment adapted for continuous integration purposes can not guarantee that they will work in the actual production environment.

Automated and headless UI *tests.* The CI system should be able to run the test suite and log the results. Furthermore the continuous integration server needs to run the test environment headlessly, that means without a developer controlling the process. This is not only true for unit tests but also for UI tests. Especially in our case the plug-ins are strongly anchored in the Eclipse IDE and therefore need tests to prove their correct integration into the IDE. These tests also need to be executed automatically and headlessly on the CI server.

Visual feedback. The CI system need to be able to run UI tests headless, that means without the observation of a user. Since anyhow problems may occur during that process, there needs to be an ability to observe the process. The user should also be able to interrupt the test run, inspect the current state of the system to debug the test setup and continue the process.

We faced situations where UI integration tests caused errors that force blocking popups to appear in the IDE which causes the server to break. Although there is an option to ignore them, we do not use it because we want to be aware of these errors. Therefore we need a possibility to debug UI tests.

Furthermore, in spite of running headless, the test results should be easy to access for the developer and displayed in a well-arranged manner without observing the test process.

Deployment. For distributing plug-ins and updates, Eclipse provides the ability to generate *update sites*. These sites can be used to install plug-ins using a wizard in the Eclipse IDE. Eclipse furthermore automatically detects new versions of plug-ins on an update site and notifies the user about it.

We want to use this feature to deliver new versions of our plug-ins to our customer. The CI therefore needs to be able to automatically bundle the plug-ins and generate an update site with the newest bundles.

Configuration. Some continuous integration tools are based on XML files for configuration. As we do not like to write cryptic XML to configure our CI system, we want a clean, easy-to-use interface for defining the build steps on the CI server.

4 Existing Build and Integration Solutions

In this section, we introduce the tools we use in our CI setup and point out, why these tools alone do not adequately meet our requirements.

4.1 Apache Ant

Apache Ant [122] is a build management tool written in Java. In XML files a sequence of tasks can be defined, which Ant then launches automatically. Ant ships with more than 150 predefined tasks, for example for building Java projects, but also custom tasks can be defined. Many tools that use Ant come with predefined Ant tasks that provide an interface to the tool. This also applies to some continuous integration servers.

4.2 Buckminster

Many software projects tend to be complex and feature many dependencies to other projects. This is also true for Eclipse plug-in projects.

The XML based tool *Buckminster* [126] helps to resolve these dependencies and creates so called *virtual distributions*. Those distributions combine components which may be located in different repositories and therefore enable to share components of complex systems. The action Buckminster performs when localizing the required components, recursively resolving dependencies and building and installing the product is called *materialization*.

4.3 JUnit

JUnit [127] is a popular Java unit testing framework. It is part of the *xUnit* family that originated with *SUnit* proposed by Kent Beck [128]. JUnit allows to define test cases with assertions and to bundle them in test suites.

JUnit ships with the Eclipse IDE as a plug-in in version 4. It is also integrated into the IDE with its own view that provides visual feedback about the results of the test runs. Furthermore the plug-in provides the ability to create *launch configurations* that specify the tests to run. This is especially handy for sharing configurations with other developers or the CI server.

4.4 SWTBot

SWTBot [129] is an open source Java-based tool for acceptance testing of SWT[2] and Eclipse-based applications. SWTBot provides a simple API that facilitates testing of the user interface of Eclipse applications. Hence it is well qualified to test the integration of Eclipse plug-ins into the IDE. The tool comes with several Ant tasks and is able to be combined with JUnit and is therefore suitable for integration into a CI system. Unfortunately SWTBot needs a *normal* Eclipse to run UI tests and is therefore incompatible with most existing headless Eclipse solutions.

[2] http://www.eclipse.org/swt/

4.5 Eclipse Headless Support

The Eclipse Plugin Development Environment (PDE) [130] is capable of running in a headless-mode to build Eclipse features and plug-ins. The PDE build system is part of the Eclipse PDE Build plug-in. It provides several Ant tasks for building Eclipse plug-ins without running the Eclipse UI and publishing the results on an Eclipse update site.

In order to build plug-ins, bundles or fragments in a headless way, PDE build requires the creation of a Eclipse feature listing all the elements to be built. Furthermore a build configuration file describing the build parameters, like the build directory or the ID if the feature to built, is necessary. A template for such a property file can be found in the PDE plug-in itself.

The property file then needs to be passed to the *AntRunner* application [131] together with a build script provided by the PDE build plug-in. AntRunner is part of the *Ant Eclipse plug-in*. It provides the entry point for running Ant tasks inside Eclipse.

The PDE also provides more advanced tasks, for example for fetching projects from a source repository. See [132] for more information.

4.6 Hudson

Hudson [133] is one of the most popular continuous integration tools. It was primarily developed by Kohsuke Kawaguchi, a former employee of Sun Microsystems/Oracle [3] who started a company in 2010 named InfraDNA [4] aimed at supporting Hudson commercially.

Hudson is an open-source Java-based continuous integration tool that supports a lot of source configuration management tools and is able to execute the most common build tools. Hudson also provides an easy-to-use but customizable and therefore powerful web interface with comprehensive monitoring facilities.

Hudson can be extended using plug-ins. They for example allow to introduce additional SCM implementations to Hudson. Hudson provide several extension points for integrating custom build steps into the build system. A lot of existing plug-ins can be installed out of the Hudson web interface. Hudson plug-ins are written in Java and based on Maven[5].

Hudson's strengths lie in its easy installation and configuration. Hudson runs in any servlet container that supports Servlet 2.4/JSP 2.0 or later, such as Glassfish or Apache Tomcat. To configure Hudson, there is no need to edit XML-files—it can be configured entirely from the web interface with on-the-fly error checks and inline help.

4.7 Evaluation of Existing Solutions

All the tools introduced above meet their demands in excellent fashion, but none of them—even in combination—meets our requirements.

---

[3] http://www.oracle.com/
[4] http://www.infradna.com/
[5] http://maven.apache.org/

One of our most important requirements was to launch tests on the CI server automatically and headlessly. This is a self-evident requirement to a CI system, but it can cause many problems when it comes to integration and UI testing. The most serious problem is running acceptance tests in a headless Eclipse environment. The Eclipse PDE and Ant plug-ins indeed provide abilities for running Eclipse headlessly, but SWTBot needs a real UI environment to run its tasks. It is therefore incompatible with the normal headless Eclipse approach.

An other argument against the usage of the PDE headless Eclipse is the additional expenses spent for configuration. The headless Eclipse works with Ant tasks, but the developer just uses launch configurations for defining the test runs on his local machine. The developer does not want to keep two different configuration files in sync just because the CI system does not understand launch configurations. This would be a typical causation for a *"Works on my machine"* problem, where the tests run on the local machine but not on the CI server.

An alternative for this is to solely use Ant tasks for configuring and launching test runs. But this is not that convenient for the developer and has the drawback that JUnit, when invoked from Ant, provides no visual feedback in the IDE. Since our goal was an easy and seamless integration and configuration of the CI, these options are unsatisfactory.

Furthermore this headless solution does not reflect the actual production environment. It was designed only for building purposes and therefore does not load all the plug-ins loaded in the production Eclipse IDE. Dependency errors may occur on the CI server, but not in the production environment or vice versa.

To summarize, we want a CI server that acts exactly the same way as the developer does when writing and testing Eclipse plug-ins. We want a real production Eclipse *controlled* and *observed* remotely by the CI server as proposed in the next section.

5 Realized CI Setup for Eclipse Plug-ins

As stated above, the existing integration tools alone are not suitable for our purpose. As we develop plug-ins for the Eclipse IDE, we depend on Eclipse as testing and integration environment. Therefore it stands to reason that we extend Eclipse in a manner that it can be integrated into a continuous integration system.

We decided to use Hudson as CI server. We did not want to spend much time for the establishment of the CI server and therefore searched for a fast and flexible solution. With its easy installation and good extendability, Hudson appears to be the best solution to meet our requirements.

In this section, we describe our CI setup in detail. We especially lay emphases on the plug-ins we developed to customize Hudson and Eclipse and describe their necessity and their roll in the overall picture.

5.1 Eclipse Robot

Our solution for running an Eclipse headlessly is the *Eclipse Robot* plug-in. This plug-in allows to remotely control the Eclipse instance it is installed in via sending predefined commands to a socket. We implemented commands that are necessary to simulate a real developer using the Eclipse IDE. These commands allow to perform actions in the same manner as the developer by using the same tools and mechanisms the developer would use on his local machine. We implemented the following commands:

`IMPORT <path to projects>` When receiving this command, the Eclipse Robot tries to import and open all projects that it will find recursively in the given *path* into the IDE. The path can be relative to the current workspace directory and may contain the wildcard character * as placeholder for any number of characters. Eclipse will import the projects the same way the developer would do using the *Import Wizard*.

`OPEN/CLOSE <project pattern>` This command causes Eclipse to open or close the given projects in the workspace. The *project pattern* may also contain the wildcard character *. This allows matching more than one project with one command. Opening and closing projects is particularly useful for testing dependencies of plug-ins.

`RUN <launch configuration> IN <project>` This command runs a *launch configuration* in the specified *project*. It is useful to create different launch configurations for different test runs (useful for *staged builds 2.1*) and run them sequentially using this command.

`BUNDLE <project pattern>` This command bundles projects into `jar` files using the default Eclipse plug-in *Export Wizard* facilities. The *project pattern* may be project names with the wildcard character * for matching more than one project with one command.

`CREATE P2 SITE <name>` This causes Eclipse to take the bundles created with the `BUNDLE` command and create an Eclipse update site out of them. Update sites can be used to distribute Eclipse plug-ins via network or Internet. This is useful for deploying the products.

`DEBUG` The Eclipse Robot runs sequential through all commands without stopping. The `DEBUG` command causes the robot to hold the execution and wait for user input to proceed (in form of an input dialog). While the robot is waiting, the Eclipse instance is fully usable. This enables the developer to debug the processes on the CI server via VNC access if something unexpected happens. When finished debugging, the developer closes the input dialog and the robot proceeds as normal.

`(NO) SCREENSHOTS ON <JUnit result>` This command causes Eclipse to take screenshots every time JUnit produces the given *result*. A leading `NO` causes Eclipse to stop taking screenshots on the given results. JUnit results are `OK`, `FAILURE`, `ERROR`, `IGNORED` and `UNDEFINED`. By giving `ALL` as parameter, Eclipse will take screenshots on every JUnit result. Taking screenshots is especially useful for UI integration tests with SWTBot to reconstruct the test run and

```
CommandHandler chain = new LaunchConfigurationCommand().
                  append(new ImportCommand()).
                  append(new ExitCommand()).
                  append(new BuildBundleCommand()).
                  append(new CreateP2SiteCommand()).
                  append(new OpenCloseProjectCommand()).
                  append(new DebugCommand());
```

Fig. 1: The chain of `CommandHandlers`

find possible sources of errors. Another use case is to automatically take screenshots for documentation.

`EXIT` This typically is the last command in the execution session. It closes the Eclipse instance.

If the robot receives an unknown or misspelled command, the robot will post an error massage on the socket. The same is true if invalid arguments were given with the commands.

Implementation Details The Eclipse Robot plug-in is composed of three main components: The `LaunchServer`, the `EclipseRobot` itself and several `CommandHandler`. Their roles are now discussed in detail.

The `LaunchServer` is a server that runs in its own thread. It is responsible for establishing a socket connection on port `4243`. It than listens to commands arriving at a socket and forwards them to the `EclipseRobot`. The server also provides the possibility to post back status feedback with the title `ERROR`, `INFO` or `WARN` on the same socket. `CommandHandlers` may use this feature to inform the user. The `LaunchServer` is started when the Eclipse Robot plug-in is activated which happens with the start of the Eclipse IDE. The server signals whether the plug-in activation was successful or not on the standard output stream.

The `EclipseRobot` is responsible for handling the incoming commands. It holds a *chain* of different `CommandHandlers` in the form of a singly-linked list as seen in Figure 5.1. It is a typical chain of responsibility: When the `EclipseRobot` is told to execute a command, it asks the first `CommandHandler` in the chain to handle the command. If the first `CommandHandler` cannot handle the command, it asks its successor to handle the command and so on.

The `CommandHandler` is the abstract base class for handlers that act on commands sent to the robot's socket. It holds a reference to the next `CommandHandler` in the linked list as well as the method `append` to append new `CommandHandler` to the list.

It also provides the method `handle` to handle the given command passed in form of a string. `handle` checks the pattern of the command it can handle

against the given input string. If it matches, it invokes the method `run`. Else, it tells the next `CommandHandler` in the list to handle the command.
Specific commands have to subclass `CommandHandler` and override two methods:

`getPattern()` Should return a regular expression pattern that should match the input line the command will understand and react on. It is possible to define groups within the pattern that are later accessible via the `group()` method of the `CommandHandler`. This is useful to get parameters given with the command. For example the `IMPORT` command described above has the pattern `"^IMPORT (.*)"`. The name of the project to import given with the command is then accessible by calling `group(1)`.

`run()` The method that is executed if the pattern matches. It is possible to access parameters (regular expression groups) given with the input line using the `group()` method as mentioned above. The actual functionality of the specific command will be implemented here.

We already implemented special `CommandHandlers` to handle the commands listed above.

5.2 JUnit Logger

The JUnit Logger is a plug-in which supplies the ability to provide the results of the JUnit test runs to the Hudson CI server. Our goal is to run an Eclipse IDE, control it remotely from Hudson and also *monitor* the results of the run in Hudson.

Hudson comes with a JUnit interpreter plug-in (5.5) able to read JUnit test reports stored as XML files. Unfortunately JUnit does not provide the possibility to write test results in XML files. The Ant tasks for JUnit enables to log the test results, but as motivated in Section 4.7 we do not want to use Ant to trigger the JUnit tests. So we are forced to write our own logger for the JUnit results. We accomplish this by writing a plug-in for Eclipse that logs the JUnit test results into a file that will later be evaluated by Hudson.

Implementation Details JUnit provides the ability to register a `TestRunListener` that gets notified about JUnit events. The events we are interested in are listed below.

- `sessionStarted` is triggered when JUnit starts a new test run. We then initialize the XML writer.
- `testCaseFinished` is signaled when one test case within a session has finished. We then tell the writer to log the results of the test case.
- `sessionFinished` is triggered when JUnit finishes the current test session. We use it to log a summary of the test run containing the elapsed time and write the output file furnished with a time stamp.

The plug-in can write output as XML, but also in Json or Yaml format and is therefore flexible to be modified for other purposes.

5.3 Traffic Light Plug-in

As both, Kent Beck [118] and Martin Fowler [113] state, it is useful to have a direct display to the CI server, like a light signal or a lava lamp. So we decided to install a traffic light in our office that shows the current project state. We also take that as an occasion to learn about the Eclipse Plug-in Development Environment (PDE). We therefore wrote a plug-in to control the traffic light out of Eclipse to be used by our CI server.

The traffic light is connected to a server which reacts on simple control sequences of one Byte size incoming on a socket. We wrote an Eclipse plug-in that registers a JUnit `TestRunListener`. Every time JUnit produces a result, our plug-in sends a command according to the JUnit status to the traffic light server.

So every time JUnit produces new results, we see it directly on the traffic light without watching the CI process on the server. If the light is red, the build failed; if it is green, everything is fine. It also strengthens the motivation to keep the build green, because everybody can immediately see the test results.

5.4 Hudson Builder for Eclipse Robot

We decided to use *Hudson* as CI server. There are a lot of plug-ins which make Hudson flexible and future-proof. Hudson is also customizable because it provides the ability to write own plug-ins for it. Furthermore we like Hudson's ability to execute downstream job builds, that means the ability to define jobs to run after the current job. This is for example useful for staged builds or when developing multiple plug-ins in one project. We configured jobs for building and testing each of them and one job for building and testing all together. The all-in-one job is triggered after each single plug-in job. In this way we guarantee the correctness of each plug-in alone and in cooperation with the other plug-ins.

As stated above (4.6), Hudson brings the capability to introduce custom build steps. This is accomplished by writing own builders as plug-ins. We developed a builder that starts an Eclipse IDE instance on the CI server and allows to send commands to the Eclipse Robot plug-in (5.1) running in that instance. The builder contributes a simple interface to the project's configuration page in the Hudson web front-end as presented in Figure 2.

It provides a `Command line` field where commands to the Eclipse Robot can be entered—one command per line. `Path to eclipse executable` should point to the Eclipse installation that should be launched.

5.5 JUnit Interpreter

Hudson comes with native support for reading JUnit reports and displaying them in the web front-end. All we need to specify is the location of the file our *JUnit Logger* plug-in (5.2) writes. Hudson can provide useful information about test results, such as historical test result trends (see Fig. 5), a web UI for viewing test reports, failure tracking and so on. So the developer can see the progress of the project at a glance.

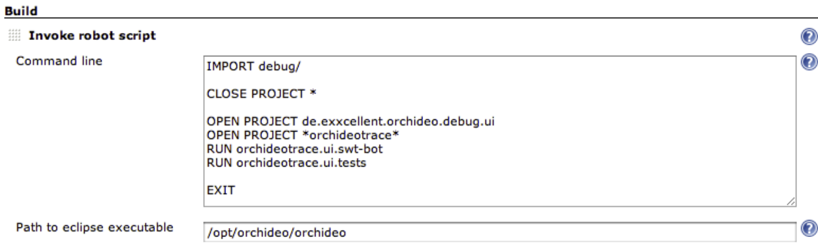

Fig. 2: Screenshot of the Hudson Builder user interface

The JUnit report evaluation can be enabled in the *Post-build Actions* of a job by enabling the `Publish JUnit test result report` switch and specifying the directory of the test report XML files written by the JUnit Logger.

5.6 VNC Access

As defined in our requirements, we wanted to enable the developer to follow the test and integration process live on the CI server. Therefore the CI server should provide a VNC display. Fortunately there is already the Hudson plug-in *Xvnc* [6] that meets this demands. For Xvnc to work the CI server must run a configured VNC server.

6 Workflow

In this section we describe the installation, configuration and testing process we have in mind for the continuous integration system mentioned above that is able to run a project as described below. Furthermore we describe how the different parts of the system as described in Section 5 interact.

Initial Situation Given there is an *Eclipse Plug-in Project*. This project has several test-cases, which can be run as simple JUnit tests, as well as some front-end tests using *SWTBot* in a separate test project. To run all tests at once, a launch configuration has been created that calls the functional test-suite as well as the front-end tests. The results can be seen directly within the Eclipse *JUnit View*. This launch configuration has been saved to a `.launch` file and checked into the repository to make sure every developer is running the tests with the same settings.

The project depends on a few other packages available from the web. *Buckminster* tasks have been created to download all dependencies to the local workspace and set things up so the dependencies can be bundled into the `lib` folder in the deployment JAR.

[6] http://wiki.hudson-ci.org/display/HUDSON/Xvnc+Plugin

Typically, when setting up a new workspace, a developer would checkout the sources, import the existing projects (the plug-in and the test project) into the workspace and run the Buckminster task to retrieve all dependencies. Then the developer can make his changes and later run the test-suite to make sure no regressions were introduced. Afterwards, the new feature is deployed in a JAR.

Installation To setup a continuous integration server using the proposed CI system, not much has to be done. On the CI server, *Hudson* needs to be installed and running. As Hudson is deployed in a single JAR, this should not be much of a problem. Additionally our *Hudson Builder* plug-in for the Eclipse Robot (5.4) needs to be installed.

The next step is to install a copy of the Eclipse IDE on the CI server. This Eclipse instance will then be used for unit and UI testing on the CI server. To simulate the production environment as realistic as possible, we simply take a copy of the Eclipse IDE the customer will finally use and add the *Eclipse Robot* (5.1) and the *JUnit Logger* plug-ins (5.2) to it. As for our CI setup, we also installed our *Traffic Light* plug-in as described in Section 5.3.

Finally, to support the observation of the test process, the Hudson Xvnc plug-in 5.6 needs to be installed and an *Xvnc* server should run on the CI server, for example by installing the *tightvnc* package on the server.

Configuration Introducing the project described above to Hudson is now easy: We create a new *job* in Hudson, of type *free-style software project*. In this job, we insert the repository URL to check out. After that, we add a new build step to run Buckminster tasks which will download the required dependencies into the workspace.

As we want to be able to observe the Eclipse instance while running, we use Hudson's capabilities to provide a VNC display by checking the `Run Xvnc during build` switch. Now every developer can observe the build process by using a local VNC client that connects via network to the CI server.

The next step is to configure the Eclipse Robot in the web interface. Therefore we add a new build step to the project and choose *Invoke robot script*. In this interface to our Hudson Builder, we can specify the path to the Eclipse executable and the commands that the Eclipse Robot plug-in in that Eclipse instance should execute (see Fig. 2). In this case, we set the robot to import our project and run the *launch configuration* for us. Optionally Hudson can be configured to send out e-mails to all developers or just the developer who did the last check-in after running a job.

After running the tests, we want Eclipse to bundle the plug-in into a JAR-file to have a deployable version of the plug-in after every run. To finally deploy the latest binaries, we tell the robot to cause Eclipse to create a new Eclipse update site out of the latest created bundles. This update site can then be shared with all stakeholders to keep them up-to-date. A final command tells the Eclipse IDE to close and finalizes the build job on Hudson.

```
IMPORT directory/exampleProject/
RUN exampleLauncheConfiguration IN de.example.project
BUNDLE *
CREATE P2 SITE example
EXIT
```

Fig. 3: A sample command line to the Eclipse Robot

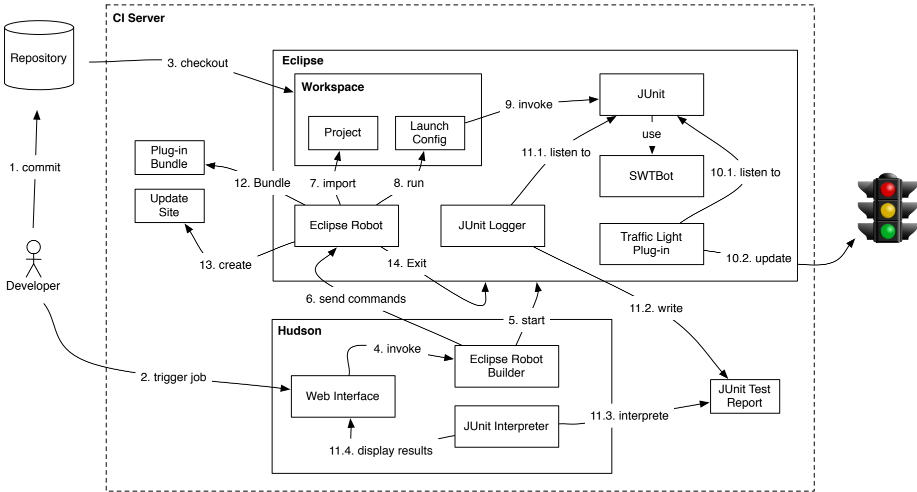

Fig. 4: Operation of the CI system

Figure 3 shows a sample command line for the project mentioned above. It instructs the Eclipse Robot to import the project in the `directory/exampleProject/` directory, and run the launch configuration named *exampleLaunchConfiguration* in the *de.example.project* project that includes the tests for the project. After that, the robot should bundle all imported projects and create an Eclipse update site named *example* out of the created bundles. Finally, the robot should exit the Eclipse instance.

Operation Figure 4 gives an overview of the activities taking place on the CI server. This section refers to this figure and explains these activities in detail.

Given a developer has made changes to the projects and he commits them to the repository (1.). Then he opens the Hudson web interface, navigates to the corresponding job's page and triggers a build (2.). Hudson then checks out the new sources from the repository into the Eclipse workspace (3.).

Hudson now starts the Buckminster task to give Buckminster the chance to download new dependencies of the project to the local platform. Furthermore Hudson starts the Xvnc server and provides a display the developer can use to

observe the test run. For the sake of clearness these steps are not mentioned in Figure 4.

When finished with the first build step, Hudson will trigger the next step—in this case the Eclipse Robot task (4.). Our Hudson builder will therefore start the Eclipse installation specified in the interface (5.) and send the given commands over socket 4243. The Eclipse Robot plug-in running in that Eclipse instance receives these commands (6.).

The commands will cause Eclipse to execute the actions described above. It will first import the specified project into the Eclipse IDE (7.) and run the specified launch configuration (8.). This will cause JUnit to launch the tests specified in the launch configuration (9.). For the UI tests it uses SWTBot.

During the execution of the launch configuration our Traffic Light plug-in will listen to the JUnit results (10.1) and update the state of the traffic light with every new test result (10.2.). At the same time our JUnit Logger will also listen to JUnit (11.1.), logs the test results and writes them as XML file to the hard drive (11.2.). This XML file's content will be interpreted by Hudson's JUnit interpreter (11.3.) and displayed as graphs with different levels of detail on the job's web page (11.4.). The JUnit results will also be used to determine the status of the build displayed on Hudson's main page.

After the run of the launch configuration the Eclipse Robot tells Eclipse to bundle the plug-in (12.) and to create an update site for it (13.). Finally the robot processes the last command which causes the Eclipse IDE to exit (14.).

The developer can now see the results of the job run on the web interface and act according to them. He can also use the created bundle and update site to deploy the plug-in to the customer.

7 Evaluation

As seen in the previous section, the introduction of a new project to the CI system only comprises defining test runs in a launch configuration, creating a new job in Hudson and configuring the Hudson Builder with, in the example, five short commands. Precondition is only the non-recurring installation of Hudson and the desired Eclipse IDE on the CI server. As for Eclipse this is only a copy and paste action.

Beside the fast installation and configuration, this CI approach has other advantages. By remotely controlling an Eclipse instance that is a copy of the actual production system, we are able to eliminate as much potential causes of risk as possible. Our *Eclipse Robot* virtually simulates a real customer working with the plug-ins under development in the Eclipse IDE he uses every day. This dramatically reduces final deployment problems because we virtually deploy with every commit.

The use of *Hudson* as CI server entails some more advantages. One of our goals was to always have a clean build, that means a build on a clean system. Using Hudson we do not need to take care of that. Hudson checks out all the sources with every build and cleans the workspace after every build. This assures that

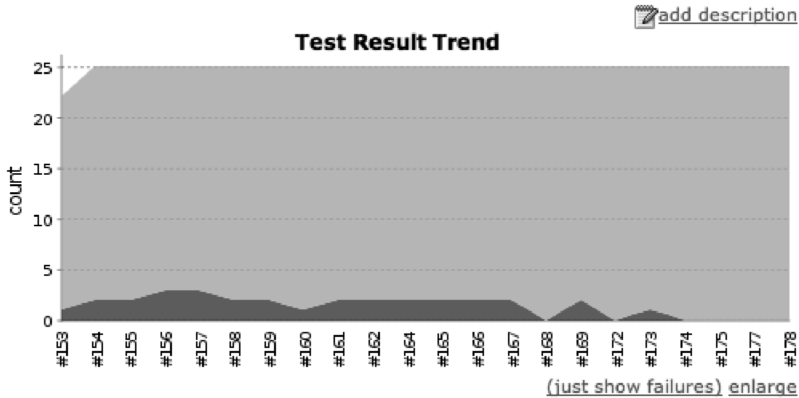

Fig. 5: Screenshot of the Hudson test result trend

the customer is able to work with our plug-ins without the need of dependencies we are not aware of.

Hudson has also capabilities for visual feedback of the build process. With Hudson's web *Dashboard* the developer is always able to monitor the build process and trigger new jobs. Thanks to our *JUnit Logger* plug-in for Eclipse, Hudson is able to evaluate the JUnit results and provide visual feedback for test results and trends as displayed in Figure 5. It shows a trend of test results for the last builds. The x-axis shows the number of the build and the y-axis the count of tests run with that build. Using colors the tests are distinguished in passed and failed tests.

As for *deployment* the Eclipse Robot provides a command to automatically create an update site—the easiest method for providing the newly created plug-ins to the customers. They can use their Eclipse to browse the update site and install the plug-ins and their dependencies from there. Furthermore Eclipse will automatically check for updates on the site and inform the customer about that. So the customer will always be able to test the latest plug-ins and give fast feedback on them.

Beside all the advantages and features stated above there are of course some drawbacks at this point of development. At the moment the Eclipse Robot is limited to the given commands and the update site is not automatically deployed on the desired server. But we have plans for improving our CI solution. See Section 9 for more details on future work.

8 Related Work

This section introduces tools we also regarded while planing our CI setup, but which do not meet our requirements for some reason. Nevertheless they are worse to be mentioned.

CruiseControl Auxiliary to Hudson, there exist similar continuous integration systems that are suitable for our purpose. One of them is *CruiseControl* [134,135].

CruiseControl was the first continuous integration system. It was originally developed by employees of *ThoughtWorks* [7] to support continuous integration for projects they were working on. Today it is a open-source Java-based extensible continuous integration tool which is also ported for *.NET* and *Ruby*.

CruiseControl supports the practices of continuous integration as stated in Section 2.1. It automatically builds projects on code change using a given configuration, has support for all popular test frameworks and automatically deploys the final product if the build succeeded. As there are ports for different platforms, the server can run on a wide range of different production environments. It provides various abilities to publish the build results and artifacts and show the build status on a web interface.

We decided to use Hudson instead of CruiseControl because it persuades us with its easy installation and configuration and with its ability for downstream project builds. But it is also imaginable to write a similar builder for CruiseControl as we did for Hudson and use CruiseControl to drive the CI system.

Cruise Cruise [136] is a commercial release management tool developed by ThoughtWorks independently of CruiseControl. Cruise pays special attentions on so called *Pipeline Workflows*—a concept for easy to configure staged builds to provide fast feedback for the developer. To speed up the build process, Cruise supports grid computing with multiple agents on any machine. This also allows to setup multiple production environments that can be tested simultaneously.

Like CruiseControl and Hudson, Cruise provides a configurable web interface with a dashboard and access to the central artifact repository. Unlike CruiseControl and Hudson, Cruise is not able to be extended with plug-ins and therefore hard to customize to specific project environments. Beside the financial factor this is the main reason for us to not use Cruise to drive our CI solution.

Headless Eclipse Builder The *Headless Eclipse Builder* [137] is an Eclipse plug-in that allows to start Eclipse headlessly and let it run several build tasks. The plug-in is invoked from the command line with following command:

```
eclipse -nosplash -data <workspace_dir> -application
com.ind.eclipse.headlessworkspace.Application [parameters]
```

Examples for `parameters` are:

`import` imports the project found in the current directory.
`build` for building the imported project.
`exportjars` for bundling the imported project into a JAR file according to `.jardesc` files found in the projects root directory.

[7] http://www.thoughtworks.com/

This plug-in fits the purpose of headless building a single project according to configuration files in the project's directory. But it does not support building multiple projects with dependencies, running tests with JUnit and SWTBot and creating and publishing an update site. Furthermore there is no integration in Hudson or other CI servers. In summary, it is a good solution for headless builds similar to the abilities of the PDE, but does not meet our requirements.

9 Future Work

9.1 Extensibility

As already mentioned, the *Eclipse Robot* is limited to the given commands. At the moment there is no possibility to add new commands without changing our source code. But we plan to provide and have partially implemented *extension points* to our plug-in witch allow to define custom commands.

Eclipse provides the concept of *extension points* and *extensions* to allow that functionality can be contributed to plug-ins by other plug-ins. Plug-ins which define extension points provide an interface other plug-ins can implement to contribute functionality. An extension point is basically a *contract* about how other plug-ins can add functionality. The plug-in which defines the extension point is also responsible for evaluating the contributions made by plug-ins implementing the extension. Beside the definition of the extension point it therefore needs some code to evaluate contributions of other plug-ins. Any other plug-in which defines an appropriate *extension* contributes the defined *extension point*. Extensions and extension points are defined in the `plugin.xml` of the corresponding plug-in. [45]

We plan to add an extension point to the Eclipse Robot plug-in, that allows other plug-ins to define their own robot commands. We already defined an abstract `CommandHandler` base class which the new command have to subclass including two methods that need to be implemented (see Sec. 5.1).

The problem that we are facing at this time of development is the definition of the command order. That means how plug-ins implementing our extension point can define the place the new command should be positioned in the chain (see Sec. 5.1). Our current approach is that a command extension can specify another command it should be positioned before, after or instead of in the chain. If nothing is specified, the command will be placed at the end of the chain.

9.2 Parallel Testing

Testing, especially UI testing with SWTBot, can take a long time in which developers can not work because they wait for the results. One recommendation of Martin Fowler [113] was to use a *staged build*, that means splitting the test run into different working units with different duration to get fast feedback (see Sec. 2.1). With our current setup, we are able to perform staged builds by running different launch configurations with different amount of tests in sequence. So the

server can run the fast commit tests (see Sec. 2.1) at first, display the results to the developer and afterwards run the time-consuming UI tests. So the developers only have to wait for the results of the fast run and may continue working while the slow run keeps on running on the CI server.

Nevertheless this process can be speed up by parallelizing tasks using multiple Eclipse instances simultaneously. For example if there is one instance running the commit tests and another one the full test suite including UI tests. It is also imaginable to test different parts of the projects, maybe from different branches, at the same time. This will also prevent waiting on project builds of other developers.

9.3 Convenience

Optimized Hudson Builder Interface. At the moment, the user interface of our *Hudson Builder* is rather rudimentary. The user needs to enter the Eclipse Robot command manually into a multi-line text field. So he needs to know what commands are available and what string pattern they listen to. To improve the usability, it is imaginable to refine the web interface to the builder. We propose to display the commands in a queue-like list with *Add* and *Remove* buttons at the bottom of it. When clicking the *Add* button, a dialog will appear where the user can choose a command from a drop-down list and can fill in parameters according to the chosen command. This will reduce errors due to misspelling and point out all capabilities of the Eclipse Robot.

Preconditions for Commands. At the current state of development, it is not possible to define preconditions for commands. Preconditions would be especially useful for commands such as `BUNDLE` or `CREATE P2 SITE`. These commands bundle and publish the current sources—with all errors and malfunctions. Since it is advisable to always have a stable build, it would be useful to be able to define preconditions such as *"Bundle only if at least 95% of all tests pass"* or *"Bundle only if commit tests pass"*.

One possibility to implement this is to analyze the results of the JUnit run before bundling/deploying the plug-in and proceed only if they fulfill the precondition. Another possibility is to define commands like *"Proceed only if at least 95% of all tests pass"* and launch them prior to the bundle/deploy commands.

Define an Update Site Location. At the moment, the Eclipse update site created with the `CREATE P2 SITE` command is published into a default directory inside the Eclipse workspace. To publish the site in a location which is reachable for all stakeholders, we currently use a second build step after the robot script to copy the update site to an online accessible location via command line.

For more convenience, we plan to add an input field to the Hudson builder user interface for the user to define the desired location the update sites should be published in.

10 Conclusion

In our project work we developed plug-ins that extend the Eclipse IDE in various ways. We wrote unit and UI tests to ensure their quality and set up a continuous integration system for our project.

In this paper we described the requirements we defined for a CI system for Eclipse plug-ins and the challenges we were faced implementing it. Furthermore we explained the solutions we found to finally set up a CI system that meet our requirements.

We developed an Eclipse headless solution that is as close to reality as no other. By remotely controlling a *real* Eclipse IDE we can be more convinced for plug-ins to run in there to also run in the final production environment.

As we are very satisfied we suggest our continuous integration system to everyone who develops Eclipse plug-ins, especially those requiring CI testing.

Summary

In the first phase of our project we developed the application TeltowCar to get to know the orchideo framework and how the two approaches model-driven software development and aspect-oriented programming are realized with orchideo. We found that orchideo provides good tool-support for model-driven software development. The aspect model of orchideo is simple and the development of aspects with the graphical editor feels natural after a short time. But during this research phase we found some problems as well. Most of them are typical problems orchideo developers have to deal with.

With a closer look to the AOP realization in orchideo we can assess that some general AOP problems apply to orchideo. The impact of aspects were not visible to the developer. We therefore analyze statically the advice weaving of the orchideo|enging to provide the developer with information about woven code while developing object-oriented code. With our plug-in the developer can stay focus on writing code and must not switching back and forth between files to get to know aspect impacts.

During our work with the orchideosuite we noticed that it is in the same situation as other MDSD and AOP frameworks: Developers are supported well when defining their models and program behavior. When it comes to debugging, satisfactory tool support is missing. We built debug tools that help the programmer to understand the complex processes in the orchideo framework. We also provide information that would else be not easily accessible, but that is crucial when searching for defects in the software.

During our research we have seen that the orchideo|engine throws orchideo traces when an error occurs. These traces are long and cryptic, which led to a situation where every trace was sent to the few people who know how to read them. We built a solution to analyze these traces and convert them to a structured in-memory representation which can be flexibly queried. In addition we have enhanced the express ability of the traces and now provide more information to developers working on the engine, on aspects and on applications. We have done this without sacrificing backwards compatibility and our tools are used with existing applications, today.

Structured data representing a system is difficult to access without knowledge about the underlying system. Our goal was to visualize the orchideo trace to present it in a way that every developer is able to see the cause of the trace for himself. Therefore we presented several visualization techniques and discussed whether they provide good support for error visualization based on the example of orchideo traces. We implemented a plug-in for Eclipse containing a combination of three visualizations we considered optimal for our case: the Tree View, the Linear Trace view, and the Tree-Map. The evaluation showed that the com-

bination of these three visualization techniques suits our requirements. Finally we have seen that our implementation is already used in a production environment by our project partner ex|xcellent solutions. Considering this we conclude that the error visualizations we provide actually helps in understanding orchideo traces.

In our project work we developed plug-ins that extend the Eclipse IDE in various ways. We wrote unit and UI tests to ensure their quality and set up a continuous integration system for our project. We described the requirements we defined to a CI system for Eclipse plug-ins and the problems we were faced implementing it. Furthermore we explained the solutions we found to finally set up a CI system that meet our requirements. We developed an Eclipse headless solution that is as close to reality as no other. By remotely controlling a *real* Eclipse IDE we can ensure for plug-ins to run in there to also run in the final production environment. As we are very satisfied we suggest our continuous integration system to everyone who develops Eclipse plug-ins, especially those requiring CI testing.

In this paper we have summarized our experiences with orchideo-based and aspect-oriented development. orchideo-based applications have major benefits: in particular they require minimal boiler-plate code to get persistent and decoupled applications. orchideo is able to cope with the rapid changes in software development. Our plug-ins in particular allow application, aspect and engine developers to change, test and debug code quickly and spend less time fixing error, more time creating value.

References

1. des Rivieres, J., Beaton, W.: Eclipse platform technical overview. Whitepaper, IBM/Eclipse Foundation, April (i) (2006)
2. Zeller, A.: Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann (2006)
3. Wang, K.: Post-mortem debug and software failure analysis on Symbian OS. (2007)
4. exxcellent solutions: orchideo die effiziente art, software zu entwickeln
5. Budinsky, F., Brodsky, S., Merks, E.: Eclipse modeling framework. Pearson Education (2003)
6. The Eclipse Foundation: Gmf documentation. http://wiki.eclipse.org/index.php/GMF_Documentation (June 2010)
7. exxcellent solutions: orchideo documentation
8. Stahl, T., V
"olter, M., Efftinge, S., Haase, A.: Modellgetriebene Softwareentwicklung. Techniken. Engineering, Management **2** (2007)
9. Miller, J., Mukerji, J.: Mda guide version 1.0.1. Technical report, Object Management Group (OMG) (2003)
10. Object Management Group: Object management group. http://www.omg.org/ (2010)
11. Filman, R., Elrad, T., Clarke, S., Ak?it, M.: Aspect-oriented software development. Addison-Wesley Professional (2004)
12. Stachowiak, H.: Allgemeinen Modelltheorie. Springer, Wien (1973)
13. Vlissides, J.: Generation Gap. C++ Report **8**(10) (1996) 12–18
14. Voelter, M.: Patterns for Handling Cross-cutting Concerns in Model-Driven Software Development. In: 10th European Conference on Pattern Languages of Programs (EuroPLoP). (2005)
15. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, The (2nd Edition). Pearson Higher Education (2004)
16. Dijkstra, E., Dijkstra, E.: A discipline of programming. prentice-hall Englewood Cliffs, NJ (1976)
17. Laddad, R.: ApectJ in Action. Manning (2003)
18. Eaddy, M., Aho, A., Hu, W., McDonald, P., Burger, J.: Debugging aspect-enabled programs. In: Software Composition, Springer (2007) 200–215
19. Concerns, C.: Patterns for Handling Cross-Cutting Concerns in Model-Driven Software Development. (2005)
20. Group, O.M.: Object constraint language, version 2.2. Technical report, Object Management Group (OMG)
21. Arthorne, J.: Project builders and natures. http://www.eclipse.org/articles/Article-Builders/builders.html (2003)
22. Deugo, D.: Foundation Patterns. In: Proceedings of the 1998 Pattern Languages of Programming Conference, Citeseer (1998)
23. The Eclipse Foundation: Swt documentation. http://www.eclipse.org/swt/docs.php (June 2010)
24. Bauer, C., King, G.: Hibernate in action. Manning (2004)
25. Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E., Yergeau, F.: Extensible markup language (XML) 1.0. W3C recommendation **6** (2000)
26. Gamma, E., Beck, K.: JUnit. At http://www. junit. org

27. Irwin, J., Kickzales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.: Aspect-oriented programming. Proceedings of ECOOP, IEEE, Finland (1997) 220–242
28. Rothlisberger, D., Harry, M., Villazón, A., Ansaloni, D., Binder, W., Nierstrasz, O., Moret, P.: Augmenting static source views in ides with dynamic metrics. In Proceedings of the International Conference on Software Maintenance (ICSM) (2009)
29. Tischler, R., Schaufler, R., Payne, C.: Static analysis of programs as an aid to debugging. In: Proceedings of the symposium on High-level debugging, ACM (1983) 158
30. Laddad, R.: ApectJ in Action 2nd Edition. Manning (2003)
31. Gosling, J., Joy, B., Steele, G., Bracha, G.: Java (TM) Language Specification, The (Java (Addison-Wesley)). Addison-Wesley Professional (2005)
32. Foundation, T.E.: Ajdt: Aspectj development tools. http://www.eclipse.org/ajdt/ (June 2010)
33. Laddad, R.: Aop@work: Aop myths and realities (February 2006)
34. Kellens, A., Gybels, K., Brichau, J., Mens, K.: A model-driven pointcut language for more robust pointcuts. Proceedings of Software engineering Properties of Languages for Aspect Technologies (SPLAT'06), Bonn, Germany (2006)
35. Stoerzer, M., Graf, J.: Using pointcut delta analysis to support evolution of aspect-oriented software. In: Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on. (2005) 653–656
36. Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (2002)
37. Hanenberg, S., Oberschulte, C., Unland, R.: Refactoring of aspect-oriented software. In: Proceedings of the International Conference on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays), Springer-Verlag (2003) 19–35
38. Parnas, D.: On the criteria to be used in decomposing systems into modules. Communications of the ACM **15**(12) (1972) 1058
39. Parnas, D.: Information distribution aspects of design methodology (1971)
40. Steimann, F.: The paradoxical success of aspect-oriented programming. ACM SIGPLAN Notices **41**(10) (2006) 497
41. Nagy, I., Bergmans, L., Aksit, M.: Composing aspects at shared join points. In: Proceedings of International Conference NetObjectDays, NODe2005. Volume 69., Citeseer (2005)
42. Zhang, D., Duala-Ekoko, E., Hendren, L.: Impact analysis and visualization toolkit for static crosscutting in aspectj. Proceedings of the 17th IEEE International Conference on Program Comprehension (ICPC) (May 2009)
43. Clement, A., Colyer, A., Kersten, M.: Aspect-oriented programming with ajdt. In: ECOOP Workshop on Analysis of Aspect-Oriented Software, Citeseer (2003)
44. The Eclipse Foundation: Java Development Tools. http://www.eclipse.org/jdt/ (June 2010)
45. Eric Clayberg, D.R.: eclipse Plug-ins 3rd edition. Addison Wesley (2008)
46. McDowell, C., Helmbold, D.: Debugging concurrent programs. ACM Computing Surveys (CSUR) **21**(4) (1989) 622
47. Kersten, M.: Tool requirements for commercial development with aspectj. In: AOSD Workshop on Commercialization of AOSD Technology, Citeseer (2003)
48. Zhang, D., Hendren, L.: Static Aspect Impact Analysis. Technical report, Citeseer (2007)

49. Voelter, M.: Best practices for dsls and model- driven development
50. Rosenberg, J.: How debuggers work: algorithms, data structures, and architecture. John Wiley & Sons, Inc. New York, NY, USA (1996)
51. Buxton, J.N., Randell, B., eds.: Software Engineering Techniques: Report of a conference sponsored by the NATO Science Committee, Rome, Italy, 27-31 Oct. 1969, Brussels, Scientific Affairs Division, NATO. (1970)
52. Zeller, A.: Isolating cause-effect chains from computer programs. In: ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE-10), Charleston, South Carolina (November 2002)
53. Gugerty, L., Olson, G.: Debugging by skilled and novice programmers. In: Proceedings of the SIGCHI conference on Human factors in computing systems, ACM (1986) 174
54. Stallman, R., Pesch, R., Shebs, S., et al.: Debugging with GDB. Free Software Foundation (1993)
55. Sun/Oracle: The Java Debugger documentation. http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html (June 2010)
56. Sun/Oracle: Java Debug Interface documentation. http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/ (June 2010)
57. Gray, J.: Why do computers stop and what can be done about it? In: Symposium on reliability in distributed software and database systems. Volume 3. (1986)
58. Musuvathi, M., Qadeer, S., Ball, T., Basler, G., Nainar, P., Neamtiu, I.: Finding and reproducing heisenbugs in concurrent programs. In: Proceedings of the Eighth Symposium on Operating Systems Design and Implementation (OSDI'08). (2008) 267–280
59. George, B., Bohner, S., He, N.: Towards a Model Level Debugger for the Cougaar Model Driven Architecture System. Innovative Concepts for Autonomic and Agent-Based Systems (2006) 86–97
60. Selic, B.: The pragmatics of model-driven development. IEEE software **20**(5) (2003) 19–25
61. Raistrick, C., Francis, P.: Model driven architecture with executable UML. Cambridge Univ Pr (2004)
62. Alexander, R., Bieman, J., Andrews, A.: Towards the systematic testing of aspect-oriented programs. In: Proc. 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC 2003), Dallas, Texas. Volume 54. (2003)
63. Ceccato, M., Tonella, P., Ricca, F.: Is AOP code easier or harder to test than OOP code. In: On-line Proceedings of the First Workshop on Testing Aspect-Oriented Programs (WTAOP 2005). (2005)
64. Sun Developer Network: JDI StackFrame Class Reference. http://java.sun.com/j2se/1.5.0/docs/guide/jpda/jdi/com/sun/jdi/StackFrame.html (April 2004)
65. Sun/Oracle: The Mirror interface documentation. http://java.sun.com/j2se/1.4.2/docs/guide/jpda/jdi/com/sun/jdi/class-use/Mirror.html (June 2010)
66. Helsinger, A., Thome, M., Wright, T.: Cougaar: a scalable, distributed multi-agent architecture. In: 2004 IEEE International Conference on Systems, Man and Cybernetics. (2004) 1910–1917
67. The Eclipse Foundation: Eclipse Platform Debug documentation. http://www.eclipse.org/eclipse/debug/platform/ (June 2010)
68. Jacobs, T., Musial, B.: Interactive visual debugging with UML. In: Proceedings of the 2003 ACM symposium on Software visualization, ACM (2003) 122

69. Ko, A.: Debugging by asking questions about program output. In: Proceedings of the 28th international conference on Software engineering, ACM (2006) 992
70. Mathis, R.: Teaching debugging. ACM SIGCSE Bulletin **6**(1) (1974) 63
71. Ritchie, D.: C reference manual. Unpublished memorandum, Bell Telephone Laboratories (1973)
72. Kortenkamp, D., Milam, T., Simmons, R., Fernandez, J.: Collecting and analyzing data from distributed control programs. Electronic Notes in Theoretical Computer Science **55**(2) (2001) 236–254
73. Andrews, J., Zhang, Y.: Broad-spectrum studies of log file analysis. (2000)
74. Netzer, R.: Optimal tracing and replay for debugging shared-memory parallel programs. In: Proceedings of the 1993 ACM/ONR workshop on Parallel and distributed debugging, ACM (1993) 1–11
75. Glass, R.: Real-time: The "lost world" of software debugging and testing. Communications of the ACM **23**(5) (1980) 271
76. Livshits, B.: Turning Eclipse Against Itself: Improving the Quality of Eclipse Plugins. month (2005)
77. Odekirk-Hash, E., Zachary, J.: Automated feedback on programs means students need less help from teachers. ACM SIGCSE Bulletin **33**(1) (2001) 55–59
78. Gülcü, C.: Short introduction to log4j. http://logging.apache.org/log4j/docs/manual (04 2007)
79. Crockford, D.: The application/json media type for javascript object notation (json) - rfc 4627 (2006)
80. Apache Foundation: Apache ant. http://ant.apache.org (2000)
81. Semicomplete.com: Logstash - centralized log storage, indexing, and searching. http://code.google.com/p/logstash/ (04 2010)
82. Semicomplete.com: Grok - a powerful pattern-matching/reacting tool. http://code.google.com/p/semicomplete/wiki/Grok (04 2010)
83. Parr, T.: The Definitive ANTLR Reference: Building Domain-Specific Languages. The Pragmatic Bookshelf (2007)
84. Grimm, R.: Rats!–an easily extensible parser generator. http://cs.nyu.edu/rgrimm/xtc/rats.html (06 2010)
85. Dix, A., Ellis, G.: Starting simple: adding value to static visualisation through simple interaction. In: Proceedings of the working conference on Advanced Visual Interfaces, ACM (1998) 134
86. Beck, K., Fowler, M.: Planning extreme programming. Volume ipse. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (2000)
87. Boehm, B.: Software engineering economics. Englewood Cliffs (1981)
88. Andrews, J.: Testing using log file analysis: tools, methods, and issues. In: Proceedings of the 13th IEEE international conference on Automated software engineering, Citeseer (1998) 157
89. Apple Inc.: Macos x crash reporter. http://developer.apple.com/mac/library/technotes/tn2004/tn2123.html (June 2010)
90. Pitt, M., Zimmerman, M.: Ubuntu crash handler. https://launchpad.net/ubuntu/+spec/automated-problem-reports (June 2010)
91. Thomas, J., Sitter, H.: Kubunut debug installer. https://code.edge.launchpad.net/~kubuntu-members/kubuntu-debug-installer/trunk
92. Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., Grant, D., Loihle, G., Hunt, G.: Debugging in the (very) large: ten years of implementation and experience. In: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles, ACM (2009) 103–116

93. Xu, S., Rajlich, V.: Cognitive process during program debugging. In: ICCI '04: Proceedings of the Third IEEE International Conference on Cognitive Informatics, Washington, DC, USA, IEEE Computer Society (2004) 176–182
94. Chmiel, R., Loui, M.C.: Debugging: from novice to expert. In: SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education, New York, NY, USA, ACM (2004) 17–21
95. Northover, S., Wilson, M.: Swt: the standard widget toolkit, volume 1. Addison-Wesley Professional (2004)
96. Massol, V., Husted, T.: JUnit in Action. Manning Publications Co., Greenwich, CT, USA (2003)
97. Megginson, D.: Sax (2010)
98. Codehaus: Jackson json processor. http://jackson.codehaus.org/ (June 2010)
99. McLaughlin, B.: Java and XML data binding. O'Reilly & Associates, Inc., Sebastopol, CA, USA (2002)
100. Grüneis, J.: Object–XML mapping with JAXB2
101. Belmonte, N.G.: Javascript infovis toolkit - interactive data visualizations for the web. http://thejit.org/home/ (June 2010)
102. Kyle Scholz, L.H.: jsviz. http://code.google.com/p/jsviz/ (June 2010)
103. Shneiderman, B.: Tree visualization with tree-maps: 2-d space-filling approach. ACM Trans. Graph. **11**(1) (1992) 92–99
104. Johnson, B., Shneiderman, B.: Tree-maps: a space-filling approach to the visualization ofhierarchical information structures. In: IEEE Conference on Visualization, 1991. Visualization'91, Proceedings. (1991) 284–291
105. Knuth, D.E.: The art of computer programming, volume 1 (3rd ed.): fundamental algorithms. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA (1997)
106. Robertson, G.G., Mackinlay, J.D., Card, S.K.: Cone trees: animated 3d visualizations of hierarchical information. In: CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems, New York, NY, USA, ACM (1991) 189–194
107. Lamping, J., Rao, R.: Laying out and visualizing large trees using a hyperbolic space. In: UIST '94: Proceedings of the 7th annual ACM symposium on User interface software and technology, New York, NY, USA, ACM (1994) 13–14
108. Sarkar, M., Brown, M.H.: Graphical fisheye views of graphs. In: CHI '92: Proceedings of the SIGCHI conference on Human factors in computing systems, New York, NY, USA, ACM (1992) 83–91
109. Kobsa, A.: User experiments with tree visualization systems. In: INFOVIS '04: Proceedings of the IEEE Symposium on Information Visualization, Washington, DC, USA, IEEE Computer Society (2004) 9–16
110. Harris, R.: The Definitive Guide to SWT and Jface. Apress, Berkely, CA, USA (2007)
111. The Eclipse Foundation: Eclipse documentation. http://help.eclipse.org/ (June 2010)
112. The Eclipse Foundation: The Eclipse Project. http://www.eclipse.org/ (June 2010)
113. Martin Fowler: Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html (May 2006)
114. Duvall, P., Matyas, S., Glover, A.: Continuous Integration (2007)
115. Holck, J., Jørgensen, N.: Continuous Integration and Quality Assurance: a case study of two open source projects. Australasian Journal of Information Systems **11**(1) (2007)

116. Farley, D.: Single-Click Software Release. In: The Thoughtworks Anthology: Essays on Software Technology and Innovation. The Pragmatic Bookshelf (2008) 172–182
117. Martin Fowler: Continuous Integration (original version). http://martinfowler.com/articles/originalContinuousIntegration.html (2002)
118. Beck, K.: Extreme programming explained: embrace change. Addison-Wesley Professional (2000)
119. Collins-Sussman, B., Fitzpatrick, B., Pilato, C.: Version control with subversion. O'Reilly Media, Inc. (2004)
120. Loeliger, J.: Version control with Git. O'Reilly Media (2009)
121. Stallman, R., McGrath, R., Smith, P.: GNU make: A program for directing recompilation (1999)
122. Apache Software Foundation: Apache Ant. http://ant.apache.org/ (2010)
123. Su, H., Jodis, S., Zhang, H.: Providing an integrated software development environment for undergraduate software engineering courses. Journal of Computing Sciences in Colleges **23**(2) (2007) 149
124. Meszaros, G.: XUnit Test Patterns: Refactoring Test Code. Prentice Hall PTR Upper Saddle River, NJ, USA (2006)
125. Ebert, C., Parro, C., Suttels, R., Kolarczyk, H.: Improving validation activities in a global software development. In: icse, Published by the IEEE Computer Society (2001) 0545
126. The Eclipse Foundation: Buckminster, Component Assembly Project. http://www.eclipse.org/buckminster/ (June 2010)
127. Beck et.al.: JUnit Project. http://junit.org/ (June 2010)
128. Beck, K.: Simple Smalltalk Testing: With Patterns. http://www.xprogramming.com/testfram.htm (1994)
129. The Eclipse Foundation: The SWTBot Project. http://eclipse.org/swtbot/ (June 2010)
130. The Eclipse Foundation: Eclipse Plugin Development Environment Website. http://www.eclipse.org/pde/ (June 2010)
131. The Eclipse Foundation: Platform Ant Project. http://www.eclipse.org/eclipse/ant/ (June 2010)
132. The Eclipse Foundation: Plug-in Development Environment Guide – Customizing a Headless Build. http://help.eclipse.org/helios/index.jsp?topic=/org.eclipse.pde.doc.user/tasks/pde_customization.htm (June 2010)
133. Kohsuke Kawaguchi: Hudson Extensible Continuous Integration Server Website. http://hudson-ci.org/ (June 2010)
134. ThoughtWorks, Inc.: CruiseControl. http://cruisecontrol.sourceforge.net/ (June 2010)
135. Schluff, S.: Continuous Integration mit CruiseControl. http://www.oio.de/cruisecontrol.pdf (Feb 2003)
136. ThoughtWorks, Inc.: Cruise – Release Management. http://www.thoughtworks-studios.com/cruise-release-management (June 2010)
137. Laszlo Varadi: Headless Eclipse Builder. http://code.google.com/p/headlesseclipse/ (June 2010)

Appendix

A Sort the Actions of a Specific Session Configuration

This method is called three times for every advice list (before, after, around) of a session configuration. After this the actions are sorted the `wovenActionsPerJoinPoint` were saved in our advice cache.

```
1
2   /**
3    * Divide all woven actions from a list of advice to their
4    * join points. Thereby transfer the order and kind of
5    * the advice to the actions.
6    */
7
8   private void getActionsPerJoinPoint(Advice[] adviceList,
        Map<Action, List<Action>[]> wovenActionsPerJoinPoint,
        SessionConfiguration sc) {
9
10    Action anyAction =
          EngineAspectDescriptor.INSTANCE.getAnyAction();
11    wovenActionsPerJoinPoint.put(anyAction, new List[] {
12         new ArrayList<Action>(adviceList.length),
13         new ArrayList<Action>(adviceList.length),
14         new ArrayList<Action>(adviceList.length) } );
15
16    for (Advice adviceModel : adviceList) {
17      boolean isAnyAction = false;
18      boolean previousBaseActionFound = false;
19      Action pointcut = adviceModel.getPointcut();
20      List<Action> wovenActions =
            adviceModel.getWovenActions();
21
22      //any action is woven to every other action
23      if (EcoreUtil.equals(pointcut, anyAction)) {
24        isAnyAction = true;
25        for (List<Action>[] actions :
              wovenActionsPerJoinPoint.values()) {
26          actions[adviceModel.getAdviceKind().getValue()].
27              addAll(wovenActions);
28        }
29      }
30
31      //if there is no entry for the current join point
32      //we have to create such entry
33      if (!wovenActionsPerJoinPoint.containsKey(pointcut)) {
34        wovenActionsPerJoinPoint.put(pointcut, new List[] {
```

```
35            new ArrayList<Action>(adviceList.length),
36            new ArrayList<Action>(adviceList.length),
37            new ArrayList<Action>(adviceList.length) } );
38
39          //check if base actions of the current join point
40          //exists as a join point in higher-precedence advice
41          //and remember them for the current join point
42          //(action woven to any action will be contained)
43          if (!isAnyAction && pointcut.getBaseAction()!= null){
44            Action currentAction = pointcut;
45            //check for more base actions until one is found
46            //as a join point (or no single one is found)
47            //and nothing happens
48            while (currentAction.getBaseAction() != null) {
49              if (wovenActionsPerJoinPoint.containsKey(
50                                currentAction.getBaseAction())) {
51                List<Action>[] wovenActionsFromBaseAction =
52                  wovenActionsPerJoinPoint.get(
53                                   currentAction.getBaseAction());
54                for (int i=0; i <
                     wovenActionsFromBaseAction.length; i++) {
55                  wovenActionsPerJoinPoint.get(pointcut)[i].
56                           addAll(wovenActionsFromBaseAction[i]);
57                }
58                //actions from higher base actions are already
59                //queued to this found one
60                previousBaseActionFound = true;
61                break;
62              }
63              currentAction = currentAction.getBaseAction();
64            }
65          }
66          //if the current join point is not the anyAction we
67          //have to check if there were actions woven to any
68          //action in a higher-precedence advice and remember
69          //them for the current join point
70          if (!isAnyAction &&
71            !previousBaseActionFound &&
72            wovenActionsPerJoinPoint.containsKey(anyAction)) {
73              List<Action>[] wovenActionFromAnyAction =
                   wovenActionsPerJoinPoint.get(anyAction);
74              for (int i=0; i <
                   wovenActionFromAnyAction.length; i++) {
75                wovenActionsPerJoinPoint.get(pointcut)[i].
76                  addAll(wovenActionFromAnyAction[i]);
77              }
78          }
79        }
80        //finally add the woven actions of the current advice
81        //(any actions is already added)
```

```
82        if (!EcoreUtil.equals(pointcut, anyAction)) {
83          wovenActionsPerJoinPoint.get(pointcut)[
84            adviceModel.getAdviceKind().getValue()].
85              addAll(wovenActions);
86
87          //check if the current join point action is the base
88          //action from any other existing join point action
89          //and add the woven actions
90          for (Action earlierJoinPoint :
                wovenActionsPerJoinPoint.keySet()) {
91            if (earlierJoinPoint.getBaseActions().
92              contains(pointcut)) {
93              wovenActionsPerJoinPoint.get(earlierJoinPoint)[
94                adviceModel.getAdviceKind().getValue()].
95                  addAll(wovenActions);
96            }
97          }
98        }
99
100     }
101   }
```

B The orchideo|engine Trace Format

This is the a grammar we have found through analysis of the existing orchideo logs and the orchideo|engine. We use an extended the BCNF in order to express some of the ambigutìes of orchideo traces.

- Terminals are within apostrophes (“ ’ ”) or as a character group (“ [0-9] ”).
- Whitespace and newlines are ignored unless explicitely specified.
- An exclamation mark (“ ! ”) is used to express a negative look-ahead assertion.
- The tilde symbol (“ $\sim$ ”) expresses a non-consuming look-behind condition.
- The forward slash (“ / ”) is used to express a look-ahead assertion.
- Brackets are used for precedence clarification

```
Level ::= ’>’ ( Level | LevelContent ) ’<’ JavaTrace?

LevelContent ::= ’(’ ActionText ’)’ ’\n’ LevelActions

LevelActions ::= ( Item+ RootAction Item* ) |
                 ( Item* RootAction Item+ )

Item ::= ( ActionCount !’ROOT:’ ActionText ) | Level

ActionText ::= ActionName ’:’ ActionParameter* ParameterHash

RootAction ::= ActionCount ’ROOT:’ ActionText

ActionParameter ::= ParameterName ’=’ ParameterValue

ParameterName ::= JavaIdentifier

ParameterValue ::= ’"’ .* NestedField* ’"’

NestedField ::= ’(’ NestedKeyValuePair ( ’,’
                NestedKeyValuePair )* ’)’

NestedKeyValuePair ::= FieldName ’:’ FieldValue

FieldName ::= JavaIdentifier

FieldValue ::= .* /(( ’,’ NestedKeyValuePair ) |
                   ( ’)’ ParameterHash )

ActionCount ::= ’(’ [1-9][0-9]* ’)’

ActionName ::= JavaIdentifier
```

```
ParameterHash ::= '[ParameterHash=' ([0-9]+ | ExceptionText) ']'

ExceptionText ::= '<caught exception>'

JavaTrace ::= ~'\n' ('<' .* '>' '\n')? JavaException? JavaAtTrace*

JavaException ::= JavaPackage '.' JavaClass ( 'Error:' |
                  'Exception:' )?

JavaAtTrace ::= ~'\n' ' '+ 'at' ' ' JavaPackage '.'
                JavaMethodDefinition '(' JavaMethodDefinition
                ':' [1-9][0-9]* ')'

JavaMethodDefinition ::= JavaClass '.' JavaMethod

JavaPackage ::= [a-z0-9]+ ( '.' [a-z0-9] )*

JavaClass ::= [A-Z|'$'|'_'] JavaIdentifierTrail

JavaMethod ::= [a-z|'$'|'_'] JavaIdentifierTrail

JavaIdentifier ::= [A-Za-z|'$'|'_'] JavaIdentifierTrail

JavaIdentifierTrail ::= [A-Za-z|'_'|'$'|0-9]*
```

B.1 Additions To Engine Trace

The following changes were made to the trace by us in the course of the project.

```
Level ::= ExecutionStackTrace? '>' ( Level | LevelContent ) '<'
          JavaTrace?

ExecutionStackTrace ::= '[MainExceptionTrace:[' TraceElement*
                        ']EndOfMainExceptionTrace]\n'

TraceElement ::= FileName ':' JavaPackage '.' JavaClass '#'
                 JavaMethod ':' [1-9][0-9]*

FileName ::= [^/\\:\*\?"<>|]+ '.java'

ActionText ::= ActionName ':' ActionParameter* ParameterHash
               SessionInformation?

SessionInformation ::= '(SESSION:' SessionName ', ' ObjectId ')'
```

C orchideo|engine trace example

This is a shortened trace the orchideo|engine, which includes our modifications, throws on an error. In our TeltowCar application we created a customer without defining his last name and tried to commit this customer to a database. As there is a cardinality constraint that checks for the customers last name the engine throws an ExcecutionInterruptedException with the following content:

```
de.exxcellent.orchideo.engine.ExecutionInterruptedException
  at
     de.exxcellent.orchideo.engine.impl.ExecutionContextImpl.init(Ex
  at
     de.exxcellent.orchideo.engine.impl.ExecutionContextImpl.<init>(
  at
     de.exxcellent.orchideo.engine.impl.SessionImpl.execute(SessionIn
  at
     de.exxcellent.orchideo.objects.aspect.persistence.hibernate.impl
  at
     de.uni_potsdam.hpi.bp2009h1.teltowcar.application.Activator.sta

[...]

  at org.eclipse.equinox.launcher.Main.main(Main.java:1287)
[MainExceptionTrace:[null:java.lang.Thread#getStackTrace:-1;Execution
ExecutionContextImpl.java:de.exxcellent.orchideo.engine.impl.Executio

[...]

Main.java:org.eclipse.equinox.launcher.Main#run:1311;Main.java:org.ec
> (Commit: wasCommitted="null" [ParameterHash=0] (SESSION:
    TeltowCarModelConfiguration, 1d7ce63))
(1) CheckValidity: [ParameterHash=0] (SESSION:
    TeltowCarModelConfiguration, 1d7ce63)
InvalidObjectNetworkError:
Parameter: constraintViolations
ParameterValue:
    [de.exxcellent.orchideo.objects.aspect.core.constraint.Constraint
    de.exxcellent.orchideo.objects.aspect.core.constraint.ConstraintI
[no stacktrace available]
(2) >> (CheckConstraints: objects="null" constraints="null"
    callback="null" dirtyOnly="true" [ParameterHash=1231]
    (SESSION: [...]))
(1) ROOT: CheckConstraints: objects="null"
    constraints="null" callback="null" dirtyOnly="true"
    [ParameterHash=1231] (SESSION: [...])

[...]

(3) >>> (CheckConstraint:
    contextObject="TeltowCarModel.impl.CustomerImpl@5984204e"
```

```
    constraint="de.exxcellent.orchideo.objects.dsl.model.impl.Cardina
    (description: null) (name:
    lastNameCardinalityConstraint) (oclExpression: null,
    type: Cardinality)" result="false"
    [ParameterHash=1733828838] (SESSION: [...]))
(1) ROOT: CheckConstraint:
    contextObject="TeltowCarModel.impl.CustomerImpl@5984204e"
    constraint="de.exxcellent.orchideo.objects.dsl.model.impl.Cardina
    (description: null) (name:
    lastNameCardinalityConstraint) (oclExpression: null,
    type: Cardinality)" result="false"
    [ParameterHash=1733828838] (SESSION: [...])
(2) >>>> (GetPropertyValue:
    object="TeltowCarModel.impl.CustomerImpl@5984204e"
    property="de.exxcellent.orchideo.objects.dsl.model.impl.PropertyI
    (description: null) (name: lastName) (ordered: false,
    unique: true, lowerBound: 1, upperBound: 1,
    defaultValue: null, aggregation: false, identity: false,
    transient: false, derived: false)" value="null"
    [ParameterHash=1775216870] (SESSION: [...]))
(1) WrapLazyProperty:
    object="TeltowCarModel.impl.CustomerImpl@5984204e"
    property="de.exxcellent.orchideo.objects.dsl.model.impl.PropertyI
    (description: null) (name: lastName) (ordered: false,
    unique: true, lowerBound: 1, upperBound: 1,
    defaultValue: null, aggregation: false, identity: false,
    transient: false, derived: false)" wrappedObject="null"
    [ParameterHash=1775216870] (SESSION: [...])

[...]

<<<<
(3) UpdateConstraintState:
    contextObject="TeltowCarModel.impl.CustomerImpl@5984204e"
    constraint="de.exxcellent.orchideo.objects.dsl.model.impl.Cardina
    (description: null) (name:
    lastNameCardinalityConstraint) (oclExpression: null,
    type: Cardinality)" status="false"
    [ParameterHash=1557591902] (SESSION: [...])
(4) >>>> (UnmarkConstraint:
    contextObject="TeltowCarModel.impl.CustomerImpl@5984204e"
    constraint="de.exxcellent.orchideo.objects.dsl.model.impl.Cardina
    (description: null) (name:
    lastNameCardinalityConstraint) (oclExpression: null,
    type: Cardinality)" wasUnmarked="true"
    [ParameterHash=1733828838] (SESSION: [...]))
(1) ROOT: UnmarkConstraint:
    contextObject="TeltowCarModel.impl.CustomerImpl@5984204e"
    constraint="de.exxcellent.orchideo.objects.dsl.model.impl.Cardina
    (description: null) (name:
```

```
    lastNameCardinalityConstraint) (oclExpression: null,
    type: Cardinality)" wasUnmarked="true"
    [ParameterHash=1733828838] (SESSION: [...])
<<<<
(5) >>>> (MarkConstraintViolation:
    contextObject="TeltowCarModel.impl.CustomerImpl@5984204e"
    constraint="de.exxcellent.orchideo.objects.dsl.model.impl.Cardina
    (description: null) (name:
    lastNameCardinalityConstraint) (oclExpression: null,
    type: Cardinality)" wasMarked="true"
    [ParameterHash=1733828838] (SESSION: [...]))
(1) ROOT: MarkConstraintViolation:
    contextObject="TeltowCarModel.impl.CustomerImpl@5984204e"
    constraint="de.exxcellent.orchideo.objects.dsl.model.impl.Cardina
    (description: null) (name:
    lastNameCardinalityConstraint) (oclExpression: null,
    type: Cardinality)" wasMarked="true"
    [ParameterHash=1733828838] (SESSION: [...])
<<<<
<<<

[...]

(6) IgnoreMandatoryIdViolations: [ParameterHash=0] (SESSION:
    [...])
<<
<
InvalidObjectNetworkError:
Parameter: constraintViolations
ParameterValue:
    [de.exxcellent.orchideo.objects.aspect.core.constraint.Constraint
    de.exxcellent.orchideo.objects.aspect.core.constraint.ConstraintI
[no stacktrace available]
```

Erklärungen

Erklärung von Tim Felgentreff

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Potsdam, den 25. Juni 2010

(Tim Felgentreff)

Erklärung von Lysann Kessler

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Potsdam, den 25. Juni 2010

(Lysann Kessler)

Erklärung von Philipp Tessenow

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Potsdam, den 25. Juni 2010

(Philipp Tessenow)

Erklärung von Stephanie Platz

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Potsdam, den 25. Juni 2010

(Stephanie Platz)

Erklärung von Frank Schlegel

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Potsdam, den 25. Juni 2010

(Frank Schlegel)

Erklärung von Christina Palm

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, alle Ausführungen, die anderen Schriften wörtlich oder sinngemäß entnommen wurden, kenntlich gemacht sind und die Arbeit in gleicher oder ähnlicher Fassung noch nicht Bestandteil einer Studien- oder Prüfungsleistung war.

Potsdam, den 25. Juni 2010

(Christina Palm)