Using Aspects to Support the Software Process

Oren Mishali

Technion - Computer Science Department - Ph.D. Thesis PHD-2010-05 - 2010

Using Aspects to Support the Software Process

Research Thesis

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

Oren Mishali

Submitted to the Senate of
the Technion — Israel Institute of Technology
Shvat 5770 Haifa February 2010

"For the LORD giveth wisdom, out of His mouth cometh knowledge and discernment." (Proverbs 2, 6)

The research thesis was done under the supervision of Prof. Shmuel Katz in the Computer Science Department.

I would like to thank my advisor, Prof. Shmuel Katz, for his professional assistance and guidance along the way. For consistently coordinating highly productive meetings, and for taking my (sometimes excessive) enthusiasm to safer and more practical places. I also appreciate his constant care for various administrative issues.

I would like to thank Dr. Yael Dubinsky for her contribution to several parts of my research, in particular those involved the user experiments, and for many fruitful discussions. Thanks goes also to Dr. Itay Maman and Prof. Orit Hazzan for their involvement in the abstract thinking activity. I would like to thank Shahar Dag and the SSDL lab for the technical support given and for hosting the user experiments, and of course thanks to all the students from our faculty that took part in the experiments and in the different activities.

I would like to express my deepest gratitude to my parents, my father Joseph, and my mother Mazal, for always emphasizing the importance of studies and higher education, and for their endless support, both moral and financial.

And finally, special thanks goes to my dear wife Meital for being a true partner, for always reminding me the right balance in life and what is really important, and for her devotion in taking care of our sweet new born daughter Tamar during the past six months.

The generous financial support of the Technion and the AOSD Europe Network of Excellence is gratefully acknowledged.

Contents

Abstract **1**

Abbreviations and Notations **2**

1 Introduction **3**

2 Related Work **6**
2.1 Process Centered Engineering Environments 6
2.2 Eclipse Extension-Points and IBM Jazz 8
2.3 AOP and the Software Process 10

3 The HighspectJ Framework **12**
3.1 Background 12
3.1.1 AOP and AspectJ 12
3.1.2 The Need for High-Level Events 14
3.1.3 AspectJ Limitations 15
3.2 Overview of the Framework 16
3.2.1 Event and Response Aspects 17
3.2.2 Event Repository 18
3.2.3 Code Generation Based On a Specification 20
3.2.4 Deploying the Support 21
3.3 Example Process Support Implementation 23
3.3.1 Planning Phase 24
3.3.2 Code Generation Phase 26
3.4 Discussion 39
3.4.1 Complex Event Processing 39
3.4.2 Fragile Pointcut Problem and Obliviousness 40

3.4.3 Supporting Other Domains 42

4 How To Define Adequate Process Support? 44
4.1 Agile Definition of Process Support 45
4.2 Test-Driven Development Case-Study 46
4.2.1 The TDD-Guide Tool 47
4.2.2 User Evaluation Setup 50
4.2.3 Is TDD-Guide Effective? 52
4.2.4 Refining TDD-Guide 58
4.3 Defining Support for Abstract Thinking 62
4.3.1 Lab Activity Setup 62
4.3.2 Activity Findings 64
4.3.3 Discussing the Findings 71
4.4 Defining Support for Code Refactoring 73
4.4.1 Classroom Activity 73

5 Implemented HighspectJ Support 78
5.1 Support for Code Integration 78
5.2 Support for Usability Evaluation 84
5.3 Support for Test-Driven Development 90
5.4 Summary 95

6 Conclusions 96

List of Figures

3.1 HighspectJ aspect organization at deployment 17
3.2 HighspectJ support definition workflow 23
3.3 Aspect and test projects generated by the framework 27
3.4 High-level event aspect finite state machine 32

4.1 TDD-Guide's user interface presented to the developer 47
4.2 Reflecting on the user evaluation activity 55

5.1 Eclipse commit dialog 82
5.2 Eclipse new plug-in project wizard 85

Abstract

Aspect-Oriented Programming (AOP) allows augmenting existing systems with additional functionality in a modular fashion by introducing new language types called *aspects*. In this research aspects are used to support the software process. Aspects encapsulating support for process methods and practices are defined, and integrated into a development environment to achieve better conformance with the desired way of work. The aspects identify significant events during the development process, and then take an appropriate action, e.g., provide real-time process guidance and training, enforce policies and standards, and automate procedures.

We found that many of the development events of interest are expressed in high-level domain terms, and often depend on several other low-level events. Since AspectJ, the most popular aspect-oriented language, is not capable of naturally treating such events, a dedicated software framework that is based on AspectJ was created to treat high-level events – the HiSpectJ framework. High-level events are identified by specially structured *event aspects*, and other *response aspects* take an appropriate action. In addition, The framework provides extensive code generation facilities and reuse, and facilitates a layered definition of events, which allows viewing a system at multiple levels of abstraction.

The second research part is concerned with how to define adequate process support of this kind, i.e., support that is effective, correct, and adopted by the developers. The method we use is inspired from agile methodologies that advocate iterative software development, and emphasize the human aspect. Several related activities were conducted, where in the main one support for the Test-Driven Development practice (TDD) was implemented in an agile fashion, starting from basic and simple support that was iteratively refined through several experiments done with student developers.

Abbreviations and Notations

AOP	—	Aspect-Oriented Programming
API	—	Application Programming Interface
CEP	—	Complex Event Processing
FSM	—	Finite State Machine
IDE	—	Integrated Development Environment
LTW	—	Load-Time Weaving
PCE	—	Process-Centered Engineering Environment
PML	—	Process Modeling Language
TDD	—	Test-Driven Development
XP	—	Extreme Programming

Chapter 1

Introduction

This research brings the relatively new aspect-oriented paradigm to bear on the software development process. Aspect-Oriented Programming (AOP) [45] allows augmenting existing systems with additional functionality in a modular fashion, instead of having it scattered across several system modules, and/or tangled with other concerns. The functionality is defined within modular units called *aspects*, which are then integrated (*woven*) into an underlying system using a dedicated compiler. Our basic hypothesis, which was introduced in [55], is that this capability of AOP can significantly contribute to the software development process field.

The software development process (or software process) is the set of tools, methods, and practices used to produce a software product [36]. In order to master the growing complexity of software systems, having a systematic and continuously improved software process is essential. Many methods exist for process improvement (e.g., [61, 37]), and one common method – which is also used here – involves the integration of automatic process support into the software development environment [59]. By that, the desired way of work is embedded in the tools themselves, hence the inevitable gap between the desired process and the actual behavior of the participants is likely to be reduced.

The main novelty of this work is in using AOP for the purpose of integrating automatic process support into a development environment. The idea is realized in the context of a popular Java development environment – the Eclipse IDE [5]. Aspects encapsulating support for process methods and practices are defined, and woven into Eclipse code itself thereby augmenting

the environment with process support which was not planned in advance. The aspects can monitor compliance with defined guidelines, provide process guidance and training by means of real-time notifications to the developers, enforce policies and standards, and even automate procedures.

Generally speaking, aspects follow an event-action model, where each aspect defines different events during the execution of the program called *join-points* (e.g., method calls, assignments to variables), and also an *advice* part that executes an action when one of the defined join-points occurs. The process support that we suggest is also based on identifying events and acting upon them, but here the events of interest are high-level events that occur during the development of the software product, and not code-level events. In consequence, the events need to be expressed using terms at a level of abstraction higher than the underlying program's code, and they often depend on several lower-level events.

For instance, for a process guideline that calls for keeping the design documents synchronized with the code, one significant event of interest may be 'a developer creates a new Java class, and then updates the related design document'. The event describes a developer's behavior that conforms with the guideline, and upon its identification, we may for instance log it for future process analysis, or provide the developer with an instant positive feedback. This particular event depends on two lower-level events namely the creation of a new Java class, and updating a design document. Note also the different level of abstraction between such software development events to code-level events, e.g., executing a particular method, or assigning a variable a new value.

High-level events of this kind are common in the software process domain as well as in other domains [51, 33]. Still, AspectJ [4], which is the most popular aspect-oriented language, is not capable of naturally treating them. This is a known limitation of AspectJ, and several suggestions were made to extend the language as appropriate, e.g., [15, 21]. Alternatively, we take a design approach that is based on standard AspectJ. A software framework was developed – called HighspectJ [57] – that facilitates the definition and implementation of high-level events on top of Java-based systems. High-level events are detected using specially structured *event aspects*. The detection of a high-level event is separated from a response to the event, and therefore a second type of aspect called a *response aspect* is defined, that reacts

upon activation of event aspects. The framework provides extensive code generation facilities which significantly reduce the coding effort, and utilizes a reusable repository of events. Moreover, a layered definition of events is facilitated, which as advocated in [51], is useful when the operation/usage of the system must be viewed and controlled at multiple levels of abstraction.

The HighspectJ framework provides the technological solution, yet a separate method is needed which dictates how to use the framework for defining *adequate* process support for a given practice. That is, how should support be defined that is effective, correct, and also adopted by the developers. The method we use is inspired from agile development methodologies such as Extreme Programming (XP) [17]. Several agile concepts and terms are borrowed, the human aspect is highly emphasized as advocated in the agile manifesto [1], and most importantly – the process support is defined in an iterative fashion, starting from basic and simple support that is refined through experiments made with real developers.

Several activities were made focusing on the definition of adequate support for several process practices. A major case-study was conducted where support for the Test-Driven Development practice (TDD) [18] was implemented in an agile fashion [53]. The support was implemented in several iterations, based on feedback collected in experiments made with dozens of student developers from the Computer Science Department at the Technion. In two additional activities, guidelines for supporting two other practices were formulated, namely Code Refactoring, and Abstract Thinking [54], also with the help of student developers.

The rest of the document is structured as follows: related work is surveyed in Chapter 2, and then in Chapter 3 the technological part of the research is discussed – the HighspectJ framework. Chapter 4 is concerned with the definition of adequate process support, and includes descriptions of the above-mentioned student activities. In Chapter 5, HighspectJ support that was implemented for several practices is presented, and in Chapter 6 we conclude.

Chapter 2

Related Work

In this chapter, our approach for adding process support to development environments using aspects is compared with other models for process support. The aspect-based approach is compared to process-centered environments, to the standard Eclipse extension mechanism, and to the relatively new IBM Jazz platform. Besides discussing related process support models, we also consider other works that connect between AOP and the software process.

2.1 Process Centered Engineering Environments

Since the early days of software development environments, they were implemented with some sort of process support. In many cases, the environment supports a hard-wired predefined process, where in some cases some sort of customization is possible. The major problem of such environments is their inflexible process support which is usually restricted to a small family of software processes [31].

The landmark paper "software processes are software too" [60] initiated the development of a different class of environments called process centered software engineering environments (PCEs). In that paper, Osterweil argues that software processes should be treated like software: they should be specified, designed, implemented, etc. Consequently, a PCE does not provide built-in support for a single fixed software process. Instead, it may be customized to support a variety of processes by considering a description of a process – a process model – which is described in a language supported by

the PCE called a process modeling language (PML). To provide such support, a PCE is typically composed of a *user interaction layer* encompassing the development tools available to the users, a *repository* holding the process artifacts, and a *process server* controlling and managing the other parts according to the process model. Research in this field has provided many important contributions, but unfortunately, despite their promising potential, PCEs are not popular, and only a few have been transferred into industrial practice [28].

A significant problem of most PCEs is that adopting them results in significant changes to the working environment of an organization, especially from the developers' perspective. This occurs because PCEs usually replace an existing development environment instead of seamlessly integrating into it. In practice, the developers, who are accustomed to their current environment and want to use their favorite tools, do not welcome such a change, especially if the new environment is not yet proven to be helpful. Significant exceptions are Process Weaver and Provence. Process Weaver [24], which is commercially available, does not itself constitute an environment platform, but it adds work-flow process support to UNIX-based environments. The developers are provided with a new tool (called Agenda) that controls their work on their favorite UNIX tools according to a workflow defined by the process model.

Provence [48] defines an architecture for process support that preserves the original working environment of the developers. As in our approach, Provence is based on monitoring events during the development and acting upon them. A key difference is that Provence is based on identifying low-level file system events such as file changes and tool invocations, and therefore the inference of meaningful high-level process events is limited. On the other hand, with aspects, one can potentially expose any event of interest including rich context data. Another important difference is that using Provence depends on several components such as a smart file system, an event manager, and a process server, which may hinder its adoption. This is as opposed to our approach where the only requirement from the base-system is for it to be "weavable", i.e., there should be a mechanism which enables the weaving of aspects into it.

To operate properly, a PCE (actually the process server part) should be able to control the tools in the user interaction layer. Usually, a PCE pro-

vides its own set of tools and an infrastructure to integrate external tools. Under this configuration, achieving effective control is not easy. For example, Marvel/Oz [41, 19] and Merlin [40] support invocation of external tools by encapsulating them in tool envelopes (an interface). Such integration results in limited control since the process server cannot control the tools *during* their execution. In SPADE-1 [16], the process server, which communicates with a message-based integration environment called DEC FUSE, may also take control during tool execution. However, in DEC FUSE, like other message-based integration environments (e.g. FIELD [63]), the level of integration of each tool is predefined by the tool.

The use of aspects, on the other hand, allows to capture events even during tool execution by augmenting their code. There is no need to anticipate desired events in advance and no cooperation or explicit hooks are required from the tools. However, with regard to the responding action carried out by the aspects, if the development environment does not provide some sort of API allowing to manipulate process artifacts and to carry out related operations, the level of automation support significantly decreases or even not possible at all.

In [28], Fuggetta presents the history and main achievements of software process research. At the end of the paper, while proposing directions for future research in the field, Fuggetta says among others that PCEs "must be non-intrusive, i.e., they should smoothly integrate and complement a 'traditional' development environment. Moreover, it must be possible to deploy them incrementally so that the transition to the new technology is facilitated and risks are reduced.". We believe that the use of aspects for software process as presented here, is a significant step toward that goal. Aspects are indeed able to complement a standard environment with process support which was not anticipated in advance, and as will be shown, their increased modularity facilitates an iterative and incremental definition and deployment of the support that increases their applicability.

2.2 Eclipse Extension-Points and IBM Jazz

The Eclipse platform [5] is the development environment that we use to test our hypothesis. Eclipse in itself is an extensible platform and thus a comparison of our aspect-based approach to the common Eclipse extension

mechanism is implied.

Eclipse is an open-source extensible development environment. Eclipse provides tools and frameworks that span the whole software development life-cycle. The basic functionality of Eclipse is very generic, and Eclipse becomes what it is, i.e., a full-functioning development platform, by being constantly extended by additional functionality. These extensions are comprised of *plug-ins*. A plug-in is a module that adds functionality to Eclipse by extending well defined points called *extension-points*. In addition, a plug-in may also introduce new extension-points to be extended by other plug-ins. A plug-in consists of several resources usually including a library that contains its Java byte-code (a.k.a. JAR library). Usually, a complex extension is composed from several plug-ins whereas a simple one is written as a single plug-in. JDT [9] (Java Development Tooling), which provides Eclipse with the capability to develop Java applications, is an example of a complex extension composed from several plug-ins.

Potentially, a plug-in may introduce extension points that correspond to important events during its execution and these may be used for implementing process support. For instance, JUnit [10], which is a popular unit-testing framework for Java, introduces an extension point that corresponds to an execution of a unit-test, and other plug-ins, by extending it, may monitor test executions and take a supportive action. In addition, the Eclipse core and JDT offer an expressive API to be used by other plug-ins. For example, they offer an API to register change-events on different resources. Basically, these APIs may also be used to provide some sort of process support.

In our implementation, basic development events are exposed from the development environment using aspects, and not using extension-points or via listeners. This aspect solution is general and is not tied to a specific platform. Furthermore, we find the aspect-oriented approach much more flexible; in practice, extension-points of the type described are rare, and most of the plug-ins do not provide an API at all. Even if plug-ins provided extension-points, it is impossible to foresee all the places where process support is needed. New extension-points can be added, but since the addition and maintenance of extension-points is done by those who developed the relevant component, it may take time until a desired extension-point is actually added (if ever). In our approach, on the other hand, anyone who has basic AOP skills may define new aspects that expose additional events from

the development environment, even during releases.

The notion of software process is gaining more and more importance. IBM Jazz [6] is a platform for the full software life-cycle that leverages Eclipse architecture with team collaboration and process awareness. Jazz provides process support in a variety of forms. It allows to define process support in the form of rules that may guide, enforce, and automate the process. The Jazz platform is process neutral, that is, there is no built-in process support for a particular methodology, and it is left to the team to define a process support that best fits their project. This flexibility is achieved by designing the different components that constitute the Jazz platform to be *process enabled*. A process enabled Jazz component is one that is opened up so that it can be guided by process rules. It is up to the developers of the component to decide how to open it up, where the goal is to make available those things that could be useful in enacting any kind of process[1].

This approach resembles the Eclipse extension-point mechanism; the operation of Jazz components and tools may be controlled, but via pre-defined hooks. Like extension-points, these hooks may be extended if needed, but this entails negotiation with the component/tool developers. We argue that due to the highly dynamic nature of software development, software process support requires an increased level of flexibility, which we believe that aspects provide.

2.3 AOP and the Software Process

There are several other works that connect AOP and the software process. In [62], aspect-oriented concepts are proposed to complement and to assist the design of an existing process modeling language called APSEE (which is a language used to formally describe a software development process). Their hypothesis is that aspect-oriented programming concepts may enhance existing approaches for *describing* software processes. The use of AOP for enhancing process modeling languages is also mentioned in [49] and [39], where in the latter work the potential contribution of AOP to the software development process is thoroughly discussed. The way we use aspects differs

[1] Jazz Platform Technical Overview:
http://jazz.net/library/LearnItem.jsp?href=content/docs/platform-overview/index.html

from those approaches in that aspects are used as a mechanism to *integrate* process support into a development environment and not for software process description.

In [67], Shomrat and Yehudai address the possibility of using AOP to solve the problem of design enforcement. They investigate whether AOP in general and AspectJ in particular, are adequate. To do that, they define aspects that are intended to be woven into the software product under development. They find that AOP in general seems to be adequate but AspectJ is only partially adequate. They show, for example, that although enforcing coding standards would seem to be naturally treated by AOP, AspectJ cannot handle it. By attacking this problem from the side of the development environment, as we do, some of the problems they identify can be solved. For instance, while AspectJ cannot be used to enforce static naming conventions for classes, this need may be tackled by disallowing the developer to create classes not conforming with the convention.

Chapter 3

The HighspectJ Framework

In this chapter the technological part of the research is discussed, i.e., how aspect-oriented programming is utilized to augment a development environment with process support. As noted, the process support is based on identifying significant events that occur during the development process and acting upon them. We found that the software process domain requires a high-level event treatment. That is, typical development events of interest need to be expressed using terms at a level of abstraction higher than underlying program's code, and the events often have a complex nature. Such event treatment is not naturally supported by AspectJ, and therefore we developed a software framework called HighspectJ that facilitates the definition and implementation of high-level events on top of Java-based systems.

The chapter starts with background on AOP, and then explains the need for the framework; afterwards, the framework is presented and its usage for defining software process support is demonstrated. At the end of the chapter, some of the technical issues raised are further discussed.

3.1 Background

3.1.1 AOP and AspectJ

Aspect-Oriented Programming (AOP) [45] is a paradigm suggested due to the inability of object-oriented constructs to modularize cross-cutting concerns (i.e., concerns whose treatment spans multiple modules). This inability causes these concerns to be scattered across several system modules

and/or tangled with other concerns, which makes program comprehension and maintenance harder. AOP attempts to solve this problem by introducing new language constructs (*aspects*) that modularize and separate the cross-cutting concerns from the object-oriented system and also provide a mechanism to compose (*weave*) these aspects into the underlying system.

Generally speaking, an aspect contains an action (code segment) and a definition of different events during the execution of the program where the action should be executed. This action is called an *advice*, and the events are called *join-points*. Any AOP language defines its own set of "interesting" join-points. AspectJ [44, 4], the most popular AOP language which is an extension to Java, defines join-points such as method executions, method calls, and assignments to variables. An AspectJ aspect defines relevant join-points using *pointcut* descriptors; in addition, like a regular class type, the aspect may declare methods, member fields, etc. With AspectJ, it is possible to define, for instance, an aspect that logs each method call during the execution of the program:

```
1  public aspect LogAllMethodCalls {
2    pointcut allMethodCalls(): call(* *.*(..)) && !within(LogAllMethodCalls);
3      before(): allMethodCalls() {
4        System.out.println(thisJoinPoint);
5      }
6  }
```

The pointcut defined in line 2 uses a *call* primitive that defines a set of method calls, where in this case the provided regular expression corresponds to the set of all method calls in the system. The negated *within* primitive says that we are not interested in capturing method calls within the aspect itself (i.e., the print call in line 4) to prevent an infinite recursive loop. The advice in lines 3-5 specifies that *before* each method call defined by the pointcut takes place, its signature is printed to the console (the AspectJ object *thisJoinPoint* represents the active join-point).

An AspectJ aspect may also define an *after* advice that is executed after the advised join-point executes. Moreover, an advice can be of type *around* meaning that it is executed instead of the advised join-point where the advice can then include a *proceed* statement that executes the join-point.

3.1.2 The Need for High-Level Events

Typically, the level of abstraction at which the aspects operate is the code of the system itself, where the aspects are expressed using terms taken from the code's design and structure. The software process domain has a terminology at a level of abstraction higher than the program's code. Therefore, aspects for process support are most naturally expressed using abstract high-level events and attributes corresponding to key software development activities and entities. Example key events are a modification of a Java class using the editor, and integrating (aka *committing*) a new version of code to a shared repository, probably while exposing relevant context variables, e.g., a commit comment written by the developer. Note the difference in abstraction level between such events, and Eclipse code-level events, e.g., calling a particular method that interprets a string from a GUI page.

Furthermore, there are many cases where there is a need to express high-level events that are the culmination of a series of more basic events. For instance, consider the TDD practice. TDD calls for development in cycles, where in each cycle a test is first written and only then the portion of code that makes the test pass is developed. One key TDD event is an attempt to write code when no test has been previously written. Another more complex TDD event was found in user experiments that were conducted for the TDD process support we have developed. We found that after creating a JUnit test class, some of the TDD developers (who were novices) spent a long time on editing the test class before moving to the corresponding Java class, hence violating TDD practice which advocates small initial cycles. To help in preventing such cases, we were interested in notifying the developers in real-time when the violation is about to occur. For that purpose, we needed to express an event which can be described as follows:

- The developer creates a JUnit test case and then modifies the test in the editor for $time == T$. No modification of a Java class occurs in the middle.

This high-level event depends on three lower-level events namely creating a JUnit test-case, modifying that test-case, and modifying a Java class. Identifying and treating such events in a system is a known system organization known as Complex Event Processing (CEP) [51], and is becoming popular for distributed information systems. In some cases, several levels

of terminology and abstractions exist, built one on the other. Additional domains where high-level events are common are usability evaluation of user interfaces [23, 33] (see Section 5.2), and the treatment of non-functional requirements such as fault tolerance and quality of service [34] (see Section 3.4.3).

3.1.3 AspectJ Limitations

Using AspectJ as-is for the purpose of defining high-level events is problematic. The root of the problem, which is a known limitation of AspectJ [15, 21], is that AspectJ pointcuts are not expressive enough to define events that are the culmination of a series of previous operations. The only AspectJ primitive that can somehow relate to a previous calculation is *cflow* (and its variation *cflowbelow*). With *cflow*, one may define a set of join-points that are in the control flow of another join-point that happened earlier. For instance, the pointcut *cflow(call(void MyApp.main()))* picks out each join-point in the control flow of a call to MyApp's main method. All other AspectJ primitive pointcuts (e.g., *execution*, *call*, *set*) relate to the current execution point, without being able to refer to previous events. This limitation finds its expression also here where many of the events of interest have such a form.

Another limitation of AspectJ relates to its available mechanism to expose context data from the base system. AspectJ pointcuts can make available context variables of the executing program, to be used in the action part of the advice. AspectJ provides three primitives namely *this*, *target*, and *args*, that allow exposing the caller object, the callee, and method parameters, respectively. For instance, consider the following pointcut:

```
pointcut callsToSum(Object caller, Object callee, int x, int y):
  call(* MyMath.sum(..)) && this(caller) && target(callee) && args(x, y);
```

MyMath.sum() is a simple method that returns the addition of its two integer parameters. The pointcut defines all the calls to *sum()* made in the program. In the pointcut's signature in the first line, it is indicated that four variables are exposed: the method's caller object, the callee, and the two integer parameters. In the second line, these variables are actually bound to the appropriate types in the program using the available primitives. The advice which uses the pointcut may save their values, and even manipulate their referenced objects.

The inability of AspectJ pointcuts to treat previously occurring events finds its expression also here. Its pointcuts are able to expose only certain types from the current execution point of the program, where in our case it is often natural for the events to expose additional context data which belongs to previous operations, and also types that best model the domain, not necessarily those defined in the base-system. For instance, we may want the event mentioned earlier, that signals a commit operation, to expose also the identifier of the development task that the developer worked on. This information may not be available to the pointcut at the time of the commit request, yet it *was* available when the developer updated the task's status before committing. Another significant obstacle is that events in AspectJ are not first-class entities, which is a fundamental requirement for treating complex event sequences [51]. In consequence, one cannot naturally save and process occurring events.

Due to these limitations, in order to implement a high-level event and response with AspectJ, one needs to define an aspect that has internal variables and advice segments that record a sequence of events that can culminate in announcing a high-level event. Several proposals have been made to extend AspectJ-like languages with history-based constructs that enable the definition of high-level events, e.g., [15, 21, 71]. Alternatively, we tackle this problem using a design approach – facilitated by the HighspectJ framework we have created – that improves and structures the naïve AspectJ implementation.

3.2 Overview of the Framework

The HighspectJ framework [57, 56] consists of Java/AspectJ types, components, coding guidelines, and code generation facilities, all of which enable defining, testing, and utilizing high-level events on top of Java-based systems. The framework's main design concepts treat an event as a first-class object, separate between the identification and the treatment of the event, and facilitate a layered definition of events that is natural in many domains. Definition and reuse of high-level events is facilitated by a key component of the framework – a repository containing event building blocks contributed by different parties. In addition, the framework provides code generation facilities which significantly reduce the coding effort. Below, the different

Listing 3.1: An interface that each event aspect implements

```
public interface IEventAspect {
    public void event(HJEvent event);
    public String getEventId();
    public void init();
}
```

features of the framework are elaborated.

3.2.1 Event and Response Aspects

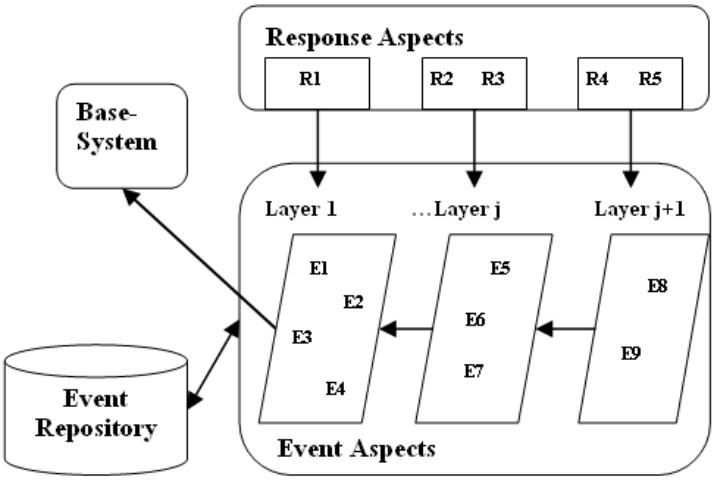

Figure 3.1: HighspectJ aspect organization at deployment

The aspect functionality defined using the framework is organized and connected to a base-system as described in Figure 3.1. To implement the needed high-level functionality, two kinds of aspects are defined, *event* and *response* aspects. Event aspects expose high-level events and context, and response aspects react upon their event identifications. An event aspect adheres to the *IEventAspect* interface shown in Listing 3.1. It maintains an internal state based on lower-level events, and at a particular point, based on its state, implicitly notifies others on the occurrence of the event that it represents by calling a special *event(HJEvent)* method which is part of its interface.

Each event aspect declares a public inner class *Event* that extends the provided *HJEvent* class. The class *Event* represents the event that is identified by the event aspect. It is passed to the *event(..)* method when it is activated, and holds event information such as an id, a time stamp denoting the time when the activity that it represents took place, and event context data. By introducing a dedicated event type, context data can be exposed from the base-system in a much more flexible fashion than in regular AspectJ including context data belonging to past events, and new types created and initialized by the aspect.

During its operation, the event aspect uses a class called *Events* which provides a variety of static utilities mostly to determine timing relationships among events (e.g., whether a set of events occurred in a certain order, getting the latest/earliest event in a given set). Such usage is demonstrated in the example in the following section, as are other types and methods not elaborated here.

Response aspects have pointcuts matched by calls to event methods from within event aspects. Using that information, they analyze and react to high-level event activations. Their functionality varies from monitoring of activities, to providing real-time notifications to the developers, to automation and enforcement of development operations. As shown in the figure, the framework supports a hierarchical organization of events, where event aspects in layer j+1 use event aspects in layer j and thereby represent system activities at a higher level of abstraction. Such a multi-layered definition is appropriate for systems and requirements with several levels of terminology and abstraction, and is a natural organization in many domains [51, 38].

3.2.2 Event Repository

Implementing the event aspects in the first layer may be a tedious task. As seen in the figure, these aspects, as opposed to higher-level event aspects, depend directly on base-system code. Thus implementing them requires familiarity with the system's internals in order to find the appropriate joinpoints and also to know how to expose the desired context. This is a problem since those interested in defining HighspectJ support are not necessarily those who implemented the base-system, and thus they usually lack this kind of knowledge.

Listing 3.2: DTD of an event aspect

```
<?xml version="1.0" ?>
<!DOCTYPE eventAspect [
<!ELEMENT eventAspect (id,documentation,lowerEvents?,
baseSystem?, requireBundle?,supplementBundle,code)>
        <!ELEMENT id (#PCDATA)>
        <!ELEMENT documentation (#PCDATA)>
        <!ELEMENT lowerEvents (eventAspect+)>
        <!ELEMENT baseSystem (id,version+)>
        <!ELEMENT version (#PCDATA)>
        <!ELEMENT requireBundle (bundle+)>
        <!ELEMENT bundle (#PCDATA)>
        <!ELEMENT supplementBundle (bundle+)>
        <!ELEMENT code (#PCDATA)>
]>
```

To overcome this obstacle, a key component of the framework is utilized – the *Event Repository*. The repository allows the contribution of event aspects, and those who are familiar with the base-system's internals may contribute first-layer event aspects to be reused by users who are interested in defining response aspects or higher-level event aspects. As opposed to first-layer events, the implementation of event aspects in the upper layers using the framework is relatively straightforward, since it is not based on the base-system's code and it is driven by code generation facilities. Nevertheless, their contribution to the repository is also possible, and promotes additional higher-level reuse.

The event aspects are stored within the repository in an XML format whose DTD is shown in Listing 3.2. For each event aspect, its id, a short documentation, and a list of lower-level event aspects on which it depends are stored (except for first-layer event aspects that operate directly on the base-system). The element *baseSystem* is only relevant for first-layer event aspects and denotes the id of the base-system to which they are targeted (e.g., Eclipse3.5.1). Higher-level event aspects do not depend on a particular base-system but on lower-level event aspects. Therefore, they are platform-independent and may be reused across several platforms. The framework, which currently supports the deployment of the aspects on top of the Eclipse IDE, utilizes a dedicated weaving mechanism which supports the weaving of aspects into Eclipse code in load-time and thus no compilation of Eclipse is

required. The DTD elements *requireBundle* and *supplementBundle* tell the weaving mechanism the Eclipse plug-ins (aka bundles) on which the event aspect depends, and the bundles to which it is woven, respectively. The actual AspectJ code of the event aspect is included within the *code* element.

3.2.3 Code Generation Based On a Specification

Even if the needed low-level events exist in the repository, a manual implementation and validation of a high-level event and response may be tiresome and time consuming. Fortunately, there are many systematic and repeatable parts within the implementation that can be automated. The framework provides code generation facilities which significantly reduce the coding effort. In essence, a user interested in implementing a high-level event aspect and a corresponding response, provides the framework with a scenario specification for the event aspect. The scenarios are expressed in a simple event-based notation, and each scenario provides an *example* for the functionality of the event aspect, hence may be considered as a high-level unit test for it.

The basic elements within a scenario are a sequence of low-level events which together cause a higher-level event (internally, all the events within a scenario are mapped to event aspects). As an example, consider the following scenario:

 # Example scenario
e1
e2
e3
→ E

It is specified that one cause of the high-level event E is the event sequence e1–e2–e3. That is, when such a sequence is observed during the operation of the base system, event E is expected to be activated. An event within a scenario may be augmented with a '+' sign (e.g., e2+) meaning that event sequences with more than one sequential occurrences of the augmented event are also considered as causes of the high-level event (e.g., e1–e2–e2–e2–e3). A scenario event may also include context variables, a feature that

is demonstrated in the example implementation in Section 3.3.

Usually, a high-level event is specified by several scenarios, each describing a single cause for its activation. By default, all low-level events, defined in any scenario of event E, affect each other's identification. For instance, suppose that an additional scenario for event E uses also an e4 event (which is not used by our example scenario). This means, that the example scenario specifies that event E should be activated when the sequence e1–e2–e3 is observed, and event e4 has *not* occurred in the middle. This default semantics may be modified as explained in Section 3.3.2. In any case, low-level events that cause other high-level events in the system are considered irrelevant, and are ignored while the sequence is being identified.

The framework, when provided with a set of scenarios which cause a particular high-level event, automatically produces corresponding code; relevant low-level event aspects are imported from the repository, and code for the high-level event and the response aspect is produced automatically. In addition, unit tests for the high-level event aspect, which are a one-to-one mapping of the scenarios, are also generated. The generated high-level event's code makes all the unit tests pass, yet in some cases the generated code needs to be generalized or to be augmented with timing constraints, hence manual intervention is needed. As demonstrated below, this manual coding is relatively simple and straightforward.

3.2.4 Deploying the Support

The generated aspect code is packed, and when added to a development environment, the environment is customized to support a certain process practice. In general, for a base system to benefit from the aspect-based support, it can be written in any language as long as aspects can be woven into it. In other words, its language should have a corresponding aspect-oriented language (e.g., AspectJ for Java or AspectC++ [3] for C/C++). The current implementation of the HighspectJ framework generates support written in Java and AspectJ and thus is applicable for base systems written in Java. Clearly, the implementation can be extended to also generate non-Java code and by that to be applicable for more base systems.

Eclipse is an open-source platform. In general, if the mechanism that weaves aspects into the base system allows weaving to code given in a bi-

nary form, then there is no need to have the source code of the base system available. The AspectJ compiler, for instance, allows such weaving and thus a Java system given in bytecode can also be augmented with HighspectJ support. Nevertheless, having the source code available is a significant advantage, since in that way first-layer event aspects may be contributed by anyone. Otherwise, their contribution can only be made by the base system's vendors, who have access to the source code.

Usually, weaving aspects into a base system requires its compilation together with the aspects. That is a serious drawback, since for large systems each removal or addition of aspects requires a long compilation process. As noted, we use another method for weaving – load-time weaving (LTW) – which does not require the compilation of the base system in advance. A weaving mechanism supporting LTW enables the augmentation of aspects when classes are loaded. As a result, instead of compiling the whole system with the aspects in advance, the aspects are stored in a dedicated location and are woven into relevant system classes when the classes are loaded. In such a case, the addition, modification, or removal of the aspects is straightforward – the aspects should simply be added/updated/removed from the dedicated location and the base system should be initialized. As noted, AspectJ supports load-time weaving, and the implementation of the HighspectJ support over Eclipse is based on the Equinox Aspects weaving mechanism, an Eclipse extension allowing the weaving of aspects into it during load-time. Equinox Aspects is part of the official distribution of AJDT [2], which is an Eclipse extension for AspectJ development.

3.3 Example Process Support Implementation

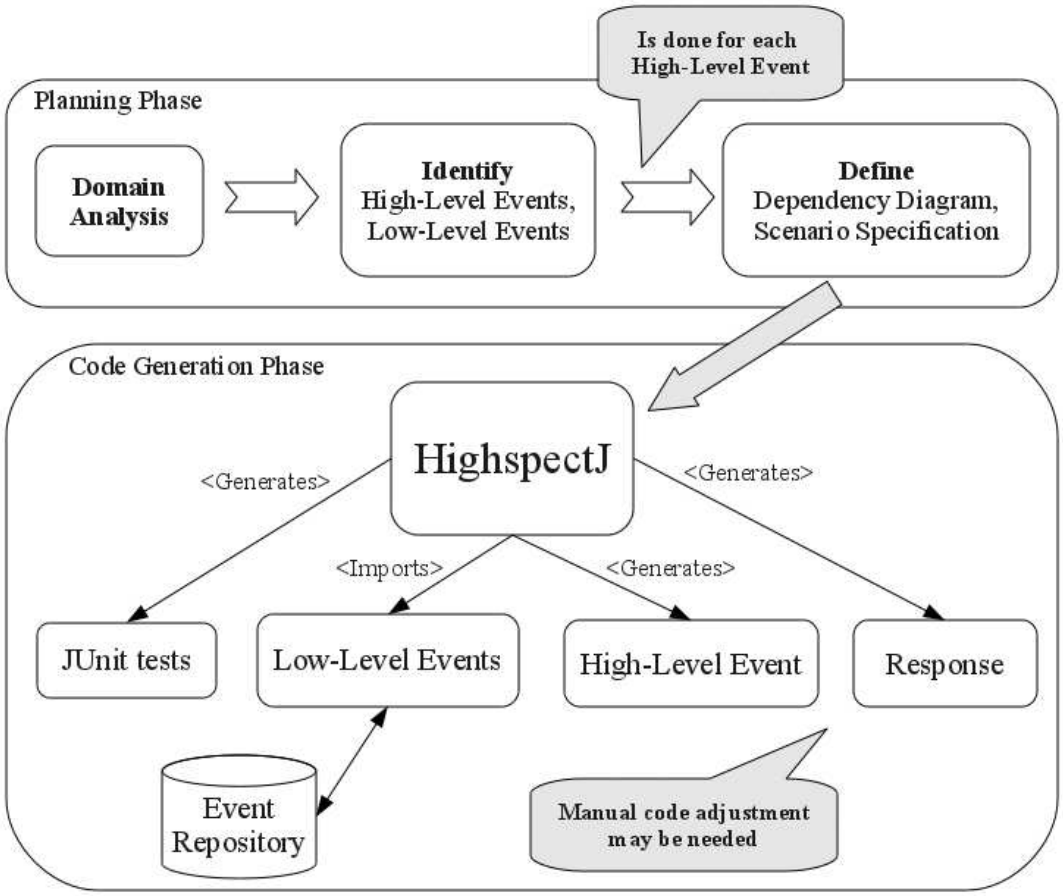

Figure 3.2: HighspectJ support definition workflow

In this section, we demonstrate using the framework for implementing process support for the TDD practice. The general workflow of defining support using the HighspectJ framework is shown in Figure 3.2. The definition of the support starts with a planning phase, where the problem domain is analyzed, and the desired high-level events are identified, as well as the low-level events on which they depend. The planning ends when two artifacts are defined for each high-level event of interest: a *dependency diagram*, and a *scenario specification*. In the code generation phase, both artifacts are

loaded into the framework which automatically produces a variety of types:

- Low-level event aspects – correspond to the scenarios' low-level events; are imported from the event repository.
- A High-level event aspect – fully implements the scenario specification.
- Junit tests – validate the functionality of the generated high-level event aspect.
- A Response aspect – carries a default notification response.

As indicated in the figure, the generated high-level event and response aspects may undergo several manual code adjustments. Below, implementation of TDD support according to workflow is presented, and each of the mentioned steps is elaborated.

3.3.1 Planning Phase

As noted, TDD calls for development in cycles, where in each cycle a unit test is first written and only then the developer writes the minimal portion of code that makes it pass that test. This ordering is considered to have many advantages such as forcing simplicity, leading to a simple design, and providing a "safety net" from changes in the code. In Chapter 5 the complete specification of the defined TDD process support is presented. Here, the steps in the development of aspects for a specific TDD guideline called 'One test at a time' are given in detail. The guideline calls for the development of one unit test in each TDD cycle thereby focusing on a single aspect of the desired functionality. This is as opposed to creating several unit tests in advance, and then implementing the whole (or most of the) functionality at once.

As with any HighspectJ process support, we are interested in identifying events during the development where the guideline is followed, and in events that represent deviations from it. Therefore, a high-level event called *OneTestAtaTime* is desired, that makes notice of such cases of conformance and deviation. The event is specified using three scenarios, one provides an example for a positive behavior of a developer relative to the practice, and the other two are negative counterexamples:

 # Developer starts coding with one failing test
TestCaseModified
TestCaseExecuted[result = "failure", failures = 1]
CodeModified
→ OneTestAtaTime[type == "Conformance", cause == "To_Code_One_Failure"]

 # Developer starts coding with several failing tests
TestCaseModified
TestCaseExecuted[result = "failure", failures = 2]
CodeModified
→ OneTestAtaTime[type == "Deviation", cause == "To_Code_Several_Failures"]

 # Developer creates the second consecutive unit test
UnitTestCreated
TestCaseModified+
UnitTestCreated
→ OneTestAtaTime[type == "Deviation", cause == "Second_Consecutive_Test"]

Within each scenario, a certain developer's behavior is simulated, either positive or negative. A scenario event may have variables that expose underlying context data, and these should be set as appropriate. The first scenario demonstrates a behavior that conforms with the guideline, where a developer starts coding while having *one* failing test. In the first line of the scenario a short description in words is given. In lines 2-4 the developer's positive behavior is simulated: after working on test code (*TestCaseModified*[1] event), an event corresponding to the execution of the test is triggered (*TestCaseExecuted*) where its *result* context variable is set to *failure* to indicate a failing execution, and also the number of failing tests is set to one; eventually, the event *CodeModified* is activated, which indicates that the developer is moving to develop the functional code.

As explained, the last line means that when such an event sequence occurs, this is one cause of the high-level event *OneTestAtaTime* that identifies cases of conformance or deviation from the guideline we are considering. The event is expected to have specific context data: a *type* field which reports conformance or deviation (here conformance), and a *cause* field which holds

[1]In JUnit, unit tests are defined within a dedicated Java class called a test-case.

the reason for the activation.

Note the difference between the assignment operator '=' used in the second line, and the equality operator '==' used in the last line of the scenario. Within the event sequence, the events are *assigned* context data according to the specific behavior which is simulated. In contrast, in the last line of the scenario the expected result is tested and thus the context variables of the high-level event are checked for equality.

The second scenario, which is negative, resembles the positive scenario except for that the developer starts coding with *two* failing tests and not with one, and thus contradicts the guideline. Here, the high-level event is expected to have a Deviation type, and a matching cause. Note that in this scenario the value set within the *failures* variable only exemplify a more general behavior where several (more than one) failing tests are indicated. This is quite common, and the resulting aspect implementation should be generalized accordingly, as demonstrated later on. The last scenario is also negative, and aims at capturing the developer's deviation at the moment it occurs, that is, when an additional unit test is about to be created. The scenario is further discussed in the next section, and in Section 4.2.4.

Based on the three scenarios, a second planning artifact called a dependency diagram is defined with all of the events in the scenarios:

TestCaseModified, UnitTestCreated, TestCaseExecuted[result(String), failures(Integer)], CodeModified $\rightarrow$ *OneTestAtaTime[type(String), cause(String)]*

In the diagram, the different events which appear in the scenarios are declared, where each declaration consists of a name, and context variables (if they exist). The dependency between the low-level events and the high-level event is also specified.

3.3.2 Code Generation Phase

Project Setup and Low-Level Event Aspects

As shown in Figure 3.2, the two planning artifacts are loaded into the framework, which in turn produces a variety of AspectJ/Java types. The types are generated within two dedicated Eclipse plug-in projects that are created in advance. The first project *hj.tdd.onetestatatime* contains the aspect

functionality, and the second project *hj.tdd.onetestatatime.tests* the related unit-tests.

The framework, based on the events declared in the dependency diagram, generates content within the two projects as shown in Figure 3.3. Three packages are created within the aspect project on the left to host the generated aspects, one for the low-level event aspects (*hj.tdd.onetestatatime.events1*), another for the high-level event aspect (*hj.tdd.onetestatatime.events2*), and a third for the response aspect (*hj.tdd.onetestatatime.response*). One JUnit test-case is generated within the test project on the right for testing the high-level event aspect. The test-case has initial content derived from the dependency file and later on JUnit test methods will be added to it. Automatic testing for the low-level event aspects is not supported.

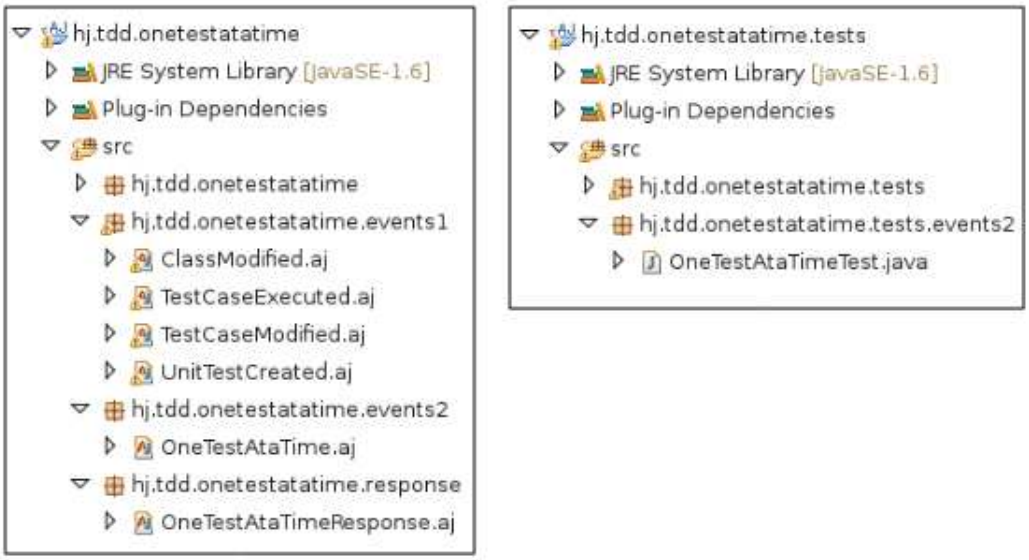

Figure 3.3: Aspect and test projects generated by the framework

During the generation process, the framework consults the event repository to check whether corresponding event aspects for the low-level events defined in the dependency file are available. In case they are, the aspects are imported into the relevant package in the project; otherwise, their skeleton is generated, but they need to be manually implemented (and then probably

exported to the repository to facilitate reuse). In our case, related event aspects exist in the repository and thus are imported. Following is an example of one such aspect:

```
1  public aspect TestCaseModified implements IEventAspect {
2    ...
3    ...
4
5    after(AbstractTextEditor editor):
6      execution(* AbstractTextEditor.setFocus()) && this(editor) {
7        fileName = ((IEditorPart)editor).getEditorInput().getName();
8    }
9    after(org.eclipse.swt.widgets.Event e):
10     execution(* StyledText.modifyContent(..)) && args(e,..){
11       if(e.type == SWT.Modify){
12         if(fileName.endsWith("Test.java"))
13           event(event);
14       }
15   }
16 }
```

The aspect corresponds to the low-level event *TestCaseModified* and reports an event each time the developer modifies a JUnit test-case type using the editor (i.e., inserts new characters). The main advice in lines 9-15 monitors an Eclipse method called *modifyContent(..)* that is executed upon each insertion of new text to the editor (*StyledText* is a primitive editor type which belongs to the Eclipse API). The method has a parameter that carries event information, and the parameter is made available to the body of the advice by binding it using the *args(..)* descriptor in line 10. Within the advice it is checked whether the event is indeed a modification event (line 11), and in case it is, if the name of the edited file matches a test-case type according to a naming convention, the event aspect reports its own event by calling its *event(..)* method (line 13).

An additional auxiliary advice is defined (lines 5-8) that prepares the ground for the main advice by saving a single value from a previous calculation. The advice is activated each time the window with the Eclipse editor becomes active ('is given a focus' in Eclipse terms), and is responsible to extract the name of the edited file to a private field defined in the aspect. This advice is essential since the type *StyledText*, which is used in the main advice, does not expose this kind of information. Note that always the editor is given a focus before new text is inserted to it and thus the main advice is guaranteed to use a valid file name. This demonstrates the added

flexibility of using an aspect type to signal an event; the aspect – as opposed to a pointcut – can collect context data from different execution points and therefore the reported event may be richer in data. Furthermore, using this technique the search for a candidate Eclipse's join-point is made easier (otherwise we should have searched for an Eclipse type that both signals the event of interest and exposes the desired context data).

Alternatively, the need for a secondary advice could be eliminated if the aspect was implemented using two lower-level event aspects namely *TextEditorModified* and *EditorGivenFocus*. However, since the aspect is very simple and has a single auxiliary variable, we find it not necessary to add an additional level of indirection.

Generating the Response Aspect

A response aspect is also generated and is provided with default content that presents a notification to the developer upon each activation of the high-level event:

```
1  public aspect OneTestAtaTimeResponse {
2    after(OneTestAtaTime.Event e): execution(* OneTestAtaTime.event(..))
3          && args(e) {
4      HJNotification notification = new HJNotification();
5      if(e.getType().equals("Deviation")){
6        notification.setType(HJNotification.Type.DEVIATION);
7      } else {
8        notification.setType(HJNotification.Type.CONFORMANCE);
9      }
10     notification.setText(e.getCause());
11     notification.setEvent(e);
12     HJUI.presentHJNotification(notification);
13   }
14 }
```

In lines 2-3, the advice monitors each execution of the *event(..)* method of the high-level event aspect. In consequence, the response aspect creates a new notification object (line 4) which is initialized according to the event info (lines 5-11), and then sent for presentation (line 12). A response aspect of this kind is transparent to the base-system (but not to its users), and is classified in [42] as a Spectative aspect. The responding advice may exhibit different modes of operation with regard to their effect on the underlying base system. A common operation mode is for a response aspect to collect real-time event data made available by event aspects, and then log it. As-

pects of this kind are also Spectative, and are transparent both to the base system and to its users.

A response aspect may also prevent the continuation of the current operation of the system. In Eclipse, that may be done by implementing the response aspect to present the developer with an Error Dialog, and then throwing a runtime exception that terminates the current operation. Preventive aspects of this kind are classified as Regulative aspects [42]. Such an enforcing behavior should be done with special care to make sure that the aspect does not leave the system in an unstable state. For instance, suppose that the operation in question is the creation of a new Java project, and we want the response aspect to prevent the developer from creating the project. We clearly do not want the aspect to stop the operation when some of the project's artifacts were already created in the workspace. This means that it should be validated that the specific join-point on which the response aspect operates indeed executes before any internal operation related to the creation of the project. Another point to consider is whether the enforcement is reasonable from the user (developer) perspective. For instance, preventing the insertion of new text into the Java editor might appear to some developers as an excessively invasive intervention.

Depending on the API provided by the base system, a response aspect may also automate user operations, and potentially modify the underlying program's state, hence being classified in [42] as an Invasive aspect. As noted, by default, the generated response aspect is a Spectative notifier; if other modes of response are desired, the code within the advice should be modified accordingly.

Generating the Unit-Tests

The scenarios specifying the high-level event aspect are loaded to a dedicated text file in the aspect project, and in consequence, the framework generates corresponding unit-tests within the test-case, where a separate JUnit test method is generated for each scenario. The test simulates the behavior defined in the scenario by activating the appropriate low-level event aspects in the defined order, and then checks whether the high-level event was activated as expected. Note that for a scenario event with an E+ semantics, the corresponding low-level event aspect is activated *once* within the test method. Following is the JUnit method generated for the nega-

tive scenario 'Developer starts coding with several failing tests' which was presented earlier:

```
1  @Test
2  public void developerStartsCodingWithSeveralFailingTests()
3          throws InterruptedException {
4    testCaseModified.event(testCaseModified.new Event());
5    Thread.sleep(1000);
6
7    TestCaseExecuted.Event testCaseExecutedEvent = testCaseExecuted.new Event();
8    testCaseExecutedEvent.setResult("failure");
9    testCaseExecutedEvent.setFailures(2);
10   testCaseExecuted.event(testCaseExecutedEvent);
11   Thread.sleep(1000);
12
13   codeModified.event(codeModified.new Event());
14   Thread.sleep(1000);
15
16   assertNotNull(logger.getEvent());
17   OneTestAtaTime.Event oneTestAtaTimeEvent =
18         (OneTestAtaTime.Event) logger.getEvent();
19   assertEquals("Deviation", oneTestAtaTimeEvent.getType());
20   assertEquals("TDD_OneTestAtaTime", oneTestAtaTimeEvent.getCause());
21 }
```

Each event in the scenario is mapped to an activation of the corresponding event aspect (the test-case keeps instances of the different event aspects in private field members). The first low-level event token *TestCaseModified* of the scenario is translated to the activation command in line 4 followed by a default time delay of one second (which appears in the code in milli-seconds). Lines 7-11 match the second scenario's event token *TestCaseExecuted* which is specified to have a particular context data and thus before calling its *event(..)* method, the context is set as appropriate (lines 8-9). The activation of the last low-level event in the sequence *CodeModified* is simulated in lines 13-14, and in the rest of the code the expected post-condition is checked. First, in line 16, it is checked that the high-level event was indeed activated. The checking is facilitated by a logger aspect provided by the framework. If the event was activated, it is retrieved in lines 17-18, and in lines 19-20 it is asserted to have context data as specified in the last line of the scenario.

Generating the High-Level Event Aspect

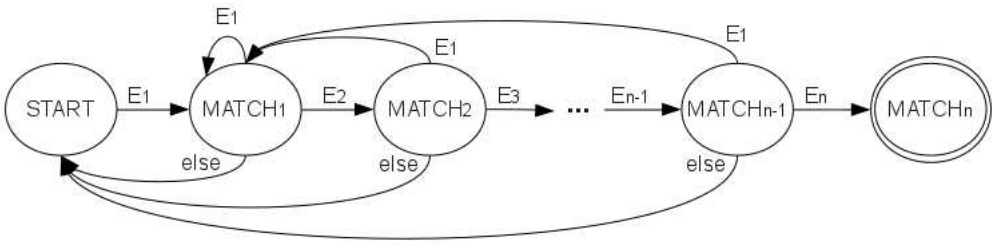

Figure 3.4: High-level event aspect finite state machine

The framework generates a high-level event aspect which behaves according to the scenario specification. The event aspect contains repeated code units, each identifying a sequence of low-level events belonging to a specific scenario. To identify an event sequence, each code unit implements a Finite State Machine (FSM) whose general behavior is shown in Figure 3.4. This general FSM identifies the event sequence E_1,E_2,...,E_n. Being in a state $MATCH_i$ means that the sequence E_1,...,E_i is currently matched; accordingly, being in $MATCH_n$ means that the whole sequence was identified. A transition from state $MATCH_i$ to $MATCH_{i+1}$ is made when event E_{i+1} occurs. In addition, each state has a transition to state $MATCH_1$ if the first event in the sequence E_1 is seen, and then a new matching process is starting. Note that this transition to $MATCH_1$ may conflict with a possible transition caused by the same event E_1 to a next state; the current implementation gives a higher priority to a next state transition thus optimistically continues the matching rather than beginning a new sequence identification. Eventually, for state $MATCH_i$, if a different event than E_{i+1} or E_1 occurs, a transition is made to START state, to begin a new matching process (recall that by default all low-level events defined in any scenario of the high-level event aspect are considered). As noted, a scenario event E_i may be added an E+ semantics (i.e., E_{i+}) meaning that it may appear sequentially one or more times. In such a case, a self transition E_i is added to the state $MATCH_i$ in the generated FSM.

For each scenario, a FSM identifying the scenario's low-level event sequence is generated. In addition, the FSM's generated code may be manually adjusted, and by that the behavior of the presented FSM may be refined.

One type of refinement concerns the transition to the next state. As noted, by default, a transition from state MATCH_i to MATCH_{i+1} is made when the occurring event is equal to event E_{i+1} in the scenario. Two events are equal if they have the same id and the same values for each of their context variables. Since a scenario is an *example* for a specific developer's behavior, often we would like to replace the equality check between the context variables with a weaker condition such as 'greater than' or 'lower than'. For instance, in the scenario 'Developer starts coding with several failing tests' the developer has two failing tests, yet this is just an example for the general case where several (more than one) tests are created.

Another type of refinement involves the addition of timing constraints to the next state transition. By default, a transition from state MATCH_i to state MATCH_{i+1} occurs whenever a E_{i+1} event is activated. The transition may be restricted to occur, for instance, only if no longer than ten seconds have passed since event E_i was noticed.

A last refinement involves reducing the set of events that affect the FSM. By default, the FSM is notified about the occurrence of each low-level event that is defined in the dependency diagram of the high-level event. In practice, it means that the matching process starts over when one of such events occurs (except for E1 and next state transition). However, sometimes we do not care if certain events not in the sequence are activated, and thus even if they occur we would like to continue the matching process. For instance, we may be interested in an event denoting a class modification which was preceded by a failing test, regardless of whether the test-case was modified during that time or not. Also here, the default generated code may be easily adjusted to filter the set of affecting events.

In Listing 3.3 and 3.4, the generated high-level event aspect *OneTestAtaTime* is shown (due to space limitations, code is divided into two parts). As noted, a similar code unit is generated for each scenario and therefore we present the code generated for the following scenario:

 # Developer starts coding with several failing tests
TestCaseModified
TestCaseExecuted[result = "failure", failures = 2]
CodeModified
→ OneTestAtaTime[type == "Deviation", cause == "To_Code_Several_Failure"]

Listing 3.3: First part of high-level event aspect *OneTestAtaTime*

```
1  public aspect OneTestAtaTime implements IEventAspect {
2    public class Event extends org.highspectj.HJEvent {
3      private String type;
4      private String cause;
5      ...
6    }
7    ...
8    private HJEvent[] expectedScenario2 = createExpectedScenario2();
9    private HJEvent[] actualScenario2 = new HJEvent[3];
10   private enum StatesScenario2 {START, MATCH1, MATCH2, MATCH3};
11   private StatesScenario2 stateScenario2 = StatesScenario2.START;
12   ...
13
14   // Scenario developerStartsCodingWithSeveralFailingTests
15   private HJEvent[] createExpectedScenario2() {
16     HJEvent[] expected = new HJEvent[3];
17     TestCaseModified.Event testCaseModifiedEvent;
18     TestCaseExecuted.Event testCaseExecutedEvent;
19     CodeModified.Event codeModifiedEvent;
20
21     testCaseModifiedEvent = testCaseModified.new Event();
22     expected[0] = testCaseModifiedEvent;
23
24     testCaseExecutedEvent = testCaseExecuted.new Event();
25     testCaseExecutedEvent.setResult("failure");
26     testCaseExecutedEvent.setFailures(2);
27     expected[1] = testCaseExecutedEvent;
28
29     codeModifiedEvent = codeModified.new Event();
30     expected[2] = codeModifiedEvent;
31
32     return expected;
33   }
34   private boolean notAffectingScenario2(HJEvent e){
35     return false;
36   }
37   ...
38 }
```

Listing 3.4: Second part of high-level event aspect *OneTestAtaTime*

```
1  public aspect OneTestAtaTime implements IEventAspect {
2    ...
3    // Scenario developerStartsCodingWithSeveralFailingTests
4    after(HJEvent e): execution(* IEventAspect.event(..))
5     && args(e) && within(hj.tdd.onetestatatime.events1.*){
6      if(notAffectingScenario2(e))
7        return;
8
9      switch(stateScenario2) {
10     case START:
11       // Transition to next state
12       if(e.equals(expectedScenario2[0])){
13         actualScenario2[0] = e;
14         stateScenario2 = StatesScenario2.MATCH1;
15         break;
16       }
17       // Otherwise, stay in START
18       break;
19     ...
20     case MATCH2:
21       // Transition to next state
22       if(e.equals(expectedScenario2[2])){
23         actualScenario2[2] = e;
24         stateScenario2 = StatesScenario2.MATCH3;
25         break;
26       }
27       // Transition to state MATCH1
28       if(e.equals(expectedScenario2[0])){
29         stateScenario2 = StatesScenario2.MATCH1;
30         break;
31       }
32       // Otherwise, transition to START
33       stateScenario2 = StatesScenario2.START;
34       break;
35     }
36
37     if(stateScenario2 == StatesScenario2.MATCH3){
38       // Start over
39       if(e.equals(expectedScenario2[0]))
40         stateScenario2 = StatesScenario2.MATCH1;
41       else
42         stateScenario2 = StatesScenario2.START;
43
44       // Trigger the event
45       Event event = new Event();
46       event.setType("Deviation");
47       event.setCause("TDD_OneTestAtaTime");
48       event(event);
49     }
50   }
51 }
```

In Listing 3.3, auxiliary methods and types are defined, that are used by the generated FSM (Listing 3.3). In lines 2-6, an internal *Event* class is defined. This class represents the event that is identified by the aspect, and is generated within each event aspect. In line 8, the array *expectedScenario2* is defined and initialized using the method *createExpectedScenario2()* in lines 15-33 (the scenario in question is defined second in the scenario specification, and in the code it is referred to as *Scenario2*). The method returns a sequence of events that is similar to the low-level event sequence of the scenario. This array is used by the FSM to know which event sequence to identify.

In line 9, another array of events is defined yet is left empty at this stage. The array holds the events in the scenario's sequence that *actually* occur during the development (as opposed to the expected array that holds dummy events). The actual occurring events are saved to allow querying for event relations against past events (e.g., timing constraints). Our scenario has three low-level events and accordingly a START state and three additional MATCH states are defined (line 10), where the initial state is initialized to START in line 11. The method *notAffectingScenario2(HJEvent)* in lines 34-36 determines which of the events that are defined in the different scenarios is considered by the FSM. By default, the method returns *false* for all such events meaning that they all affect the matching process. The set of affecting events may be reduced by refining the method.

The FSM that identifies the scenario's event sequence is presented in Listing 3.4. The FSM is implemented using the AspectJ advice in lines 4-50. The advice is executed when one of the events defined in the low-level event package (line 5) occurs, unless filtered by *notAffectingScenario2(HJEvent)* method (lines 6-7). Within the advice, the low-level event that has occurred is referred to as *e*. A *switch* statement (lines 9-35) directs to a relevant *case* block depending on the current state, which is initialized to START. Note the *case* block in line 20; being in a MATCH2 state means that a sequence of two events is currently identified (i.e., *TestCaseModified – TestCaseExecuted*). Therefore, if the occurring event *e* is identical (line 22) to the third event in the scenario, *CodeModified*, the FSM transits to state MATCH3 (line 24). If there is no match, transitions to states MATCH1 and START are handled (lines 27-35) as previously explained.

After leaving the *switch* statement, the condition in line 37 checks whether we reached the MATCH3 state, and if so, reports an event while setting the next state as appropriate (back to START or to MATCH1 state). To report an event, a new instance of the internal class *Event* is created, and initialized with appropriate *type* and *cause* values (lines 45-47). Eventually, in line 48, the event is sent as a parameter to the *event(..)* method that provides it with a time-stamp and signals the activation of the high-level event. The control is now passed to the response aspect, which by default presents the developer with an appropriate notification.

Implementation Notes

The generated high-level event aspect identifies event sequences *exactly* as specified in the scenarios. However, in many cases, a scenario only provides an example for a more general desired event sequence, and thus the generated code needs to be generalized (the possible code adjustments were discussed earlier).

Our TDD guideline is specified using three scenarios (Section 3.3.1). The FSM that is generated for the first positive scenario should not be changed. This is because the behavior that is demonstrated in the scenario, of a developer who modifies test code, executes a *single* failing test, and then modifies the functional code, describes exactly the positive case, and thus no generalization of the scenario is needed. As opposed to that, the negative behavior presented in the second scenario, *TestCaseModified – TestCaseExecuted[failures = 2] – CodeModified*, should be generalized, since it is only an example for a more general case where any number of failing tests greater than one is noticed. Following is the relevant code snippet of the FSM that should be modified:

```
1  public aspect OneTestAtaTime implements IEventAspect {
2    ...
3    // Scenario developerStartsCodingWithSeveralFailingTests
4    after(HJEvent e): execution(* IEventAspect.event(..))
5      && args(e) && within(hj.tdd.onetestatatime.events1.*){
6      ...
7      case MATCH1:
8        // Transition to next state
9        if(e.equals(expectedScenario2[1])){
10         actualScenario2[1] = e;
11         stateScenario2 = StatesScenario2.MATCH2;
12         break;
13       }
14     ...
15   }
16 }
```

Being in a MATCH1 state (line 7) means that the first event in the sequence already occurred (in our case it is the event *TestCaseModified*). The *if* condition in line 9, determines the circumstances in which a transition to the next MATCH2 state is made. As seen in line 9, by default a transition is made if the occurring event e is equal to the next (second) event in the expected event array. That is, that event e is also a *TestCaseExecuted* event with exactly two failures. The equality check should be generalized as follows:

```
1        // Transition to next state
2        if(eventsSameId(e, expectedScenario2[1]) &&
3          ((TestCaseExecuted.Event)e).getFailures() > 1){
4          ...
5        }
```

As opposed to the previous default condition, here the only check that is made against the expected event is that they share the same id, i.e., that event e is indeed a *TestCaseExecuted* event (line 2). Then, in the next line, it is also checked whether event e has more than one failures, and if so, a state transition is made. The method *eventsSameId(..)* that is used in line 2 is defined in the class *Events* of the framework, which provides a variety of static utilities to determine different relations between events. Examples for other *Events* utilities are *eventsOrdered(..)* that checks whether the given event parameter are sorted in a chronological order, and *eventsDiffLowerThan(..)* that checks whether the time difference between the two provided events is lower than a given amount of seconds.

The FSM that is generated for the third scenario needs no refinement. Recall the scenario's event sequence: *UnitTestCreated – TestCaseModified+ – UnitTestCreated*. Since we would like to allow the insertion of multiple characters between the two test creations, the second event in the sequence, *TestCaseModified*, has an E+ semantics indicating that it may occur one or more times sequentially. In consequence, the generated FSM has slightly different code:

```
1  public aspect OneTestAtaTime implements IEventAspect {
2    ...
3    // Scenario developerCreatesTheSecondConsecutiveUnitTest
4    after(HJEvent e): execution(* IEventAspect.event(..))
5      && args(e) && within(hj.tdd.onetestatatime.events1.*){
6      ...
7      case MATCH2:
8        ...
9        // Stay in current state
10       if(e.equals(expectedScenario3[1])){
11         actualScenario3[1] = e;
12         break;
13       }
14     ...
15   }
16 }
```

Within the MATCH2 state, it is first checked whether a transition to the next state should occur (omitted from the listing). If there is no match, an additional *if* block is met (lines 10-13), instead of immediately returning to states START or MATCH1 (which is the case in Listing 3.4). Within the *if* condition in line 10, it is checked whether the occurring event e is equals to the second expected event *TestCaseModified*. This check was already made once and now the case of multiple sequential occurrences is handled. In case of a match, the switch statement is left (*break* command in line 12), hence leaving the current state as it is – in state MATCH2.

3.4 Discussion

3.4.1 Complex Event Processing

Identifying and treating high-level events in a system is a known system organization known as Complex Event Processing (CEP) [51], and is becoming popular for distributed information systems. CEP is useful in a variety of applications including examples such as stock trading, credit card fraud

detection, business activity tracking, medical monitoring and the like. Additional domains where high-level events are common are usability evaluation of user interfaces [23, 33], and the treatment of non-functional requirements [34]. Use of such events is appropriate and helpful when the domains treated have terminology at level of abstraction higher than individual events in the code.

There are several similarities between the CEP architecture presented in [51] and the system organization described here, in particular, the treatment of an event as a first-class object and having layers of events built hierarchically. Yet, while in CEP the proposed architecture is general and no specific technology is advocated, we bring the power of AOP to bear on this emerging area. By utilizing aspect-oriented technology, the identification of the events from the base system is much more flexible and straightforward, and is easier to reuse in multiple systems. Using aspects also removes the need, seen in other approaches to complex event processors, for a dedicated runtime environment: the aspects are ultimately compiled into Java bytecode, so only a regular Java runtime environment is needed.

3.4.2 Fragile Pointcut Problem and Obliviousness

Over time, changes to the aspects are likely to occur. The first class of changes to consider is due to updates in the software process itself. Here, changes to response aspects and event aspects at all layers are possible. This sort of change is made easier by the code generation facilities provided by the framework, and by the event repository. Another class of changes is related to the inevitable modifications in the implementation of the underlying development environment. Since the aspects depend on the development environment's code (not necessarily only in its public API), the link between the aspects and the code might become unstable in subsequent releases; with respect to the original intended semantics of the pointcuts, they may capture unintended join-points and also miss intended ones.

This problem is well known and called *the fragile pointcut problem*. Several approaches were suggested to cope with this problem [30, 14]. Aldrich introduces *open modules* [14] where it is suggested to extend the familiar API of a system with pointcuts that represent internal events that are semantically important. The aspects depend on these API pointcuts hence

a contract is made. This approach is indeed stable but restricts the power of AOP since aspects can only operate on a limited set of events defined in advance by the developers of the base system. Moreover, the approach violates the *obliviousness* property [25], which in general is considered desirable for AOP systems, which means that the base system should not be designed with AOP in mind.

In our system organization the fragility problem is mostly relevant for the first-layer events aspects that are defined directly on the base-system. These aspects extend the system with well-defined events, and the upper layer aspects depend solely on them. The model we suggest is less limited than open modules in the sense that first-layer event aspects can be contributed by anyone anytime even during releases. Furthermore, the obliviousness property is kept since the base system's developers need not be aware of the contributions. The obliviousness property is also preserved in upper layers, where the event aspects are not aware of other event/response aspects using them; the aspect simply calls a method notifying about the event and those interested in the event define pointcuts above the method, as opposed to, e.g., the Observer pattern where the subject should know about its dependents.

Regarding the first-layer event aspects themselves, some sort of review and validation is required for the links to the relevant unstable parts of the aspects, in each major release. This validation should check that they preserve the original event semantics and that they expose the original context. A somewhat similar approach for the fragility problem is presented in [43] where the concept of model-based pointcuts is introduced. Model-based pointcuts do not operate directly on the base-system but on an abstraction of it, and thus the aspects which use these pointcuts are kept stable. Upon system evolution only the connection between the base-system and the model should be rechecked.

3.4.3 Supporting Other Domains

The HighspectJ framework was created and used with software process support in mind. However, the framework has a general architecture [56] that may be used to support additional high-level domains as well. Several examples are provided below, mostly in a distributed setup.

The current version of the framework is not distributed. Of course, distributed systems raise additional issues such as remote identification of pointcuts or application of advice, but these are orthogonal to our suggested architecture. Several languages such as JBoss AOP [8] or Spring AOP [11] use AspectJ-like constructs over a fixed distributed middleware framework, exploiting the features of the distributed framework. The AWED language [58], implemented over Jasco [68] has explicit constructs for detecting remote pointcuts and for executing advice remotely. No matter how such issues are treated, the layered architecture, with separate event and response aspects, can profitably be used with any of these (and the precise language constructs needed will not be further considered here).

In [34], aspects are shown appropriate for nonfunctional distributed concerns such as fault tolerance, security, and quality of service, previously treated by configurable protocols such as in the Cactus system. However, they list obstacles to such use, such as the lack of expressibility of pointcuts, the need for more flexible context information and access, the need for abstract events, and the need to introduce time-related events and terminology not in the base system. Clearly, the HighspectJ framework is designed to overcome these problems.

For instance, consider the typical distributed concern of quality of service (QOS). The lower-level events of remote method calls can be grouped at the next event level to categories such as initiating a new request, querying the status of a request, a positive response to a request, denial of a request, etc., each with its own context. A higher-level event aspect incorporates time events, delays, etc., to define an event of a successful request-response pair (where the response is within required time constraints after the request). Above these are events that indicate meeting standards (counting successful/unsuccessful request-response pairs and periodically computing percentages), entering a mode of heavy-loads with exceptional delays, identifying denial-of-service attacks as abstract events, etc. At each level, response

aspects can both monitor what is happening and change the responses to requests based on the load. The separation of the high-level events and any desired response is especially appropriate because of changing policies: the reaction to entering a heavy-load situation could vary over time, depending on the perceived urgency of high QOS and/or available resources to increase capacity.

The auditing concern of a banking system is another example. In particular, a layered approach to treating the *money laundering* concern[2] can be imposed over such a system, motivated by changes in legislation and tax law. Such a concern has complex terminology and events at several levels of abstraction. For example, an intermediate level of a collection of suspicious *red-flag* events is natural. Thus, the creation of multiple on-line bank accounts of a similar type from the same IP address could be identified as such an event. Red-flag events can trigger a deeper analysis to identify, e.g., events signaling the practices of *smurfing* or *kiting*. The former involves creating numerous small entities (accounts or enterprizes) to avoid reporting currency exchanges, while the latter involves moving among multiple domain names in financial transactions to avoid detection. Identifying such events involves gathering and exposing information irrelevant to a banking system that has no direct treatment of this concern.

[2]See Wikipedia for explanations of terminology.

Chapter 4

How To Define Adequate Process Support?

The HighspectJ framework facilitates the definition of event-based support for software process practices. However, while providing the technological solution, the question of how to come up with *adequate* process support is not treated by the framework. This chapter is concerned with the definition of such process support, i.e., support that is effective, correct, and also adopted by the developers. To achieve that goal, the process support is defined in an agile fashion. In particular, the support is defined and implemented in an iterative fashion, and the human aspect is highly emphasized in several dimensions.

The chapter starts with an overview of our approach for process support definition, while highlighting the agile concepts and principles which are borrowed. We then describe a major case-study where support for the TDD practice was implemented in an agile fashion, starting from basic and simple support that is iteratively refined through user experiments. The TDD support was refined based on user feedback yet its initial specification was given by us. As opposed to that, two additional activities are described where the developers themselves are involved in identifying possible first-iteration support for two other practices: Abstract Thinking, and Code Refactoring.

4.1 Agile Definition of Process Support

Agile development methodologies, e.g., Extreme Programming (XP) [17] and Scrum [65], call for an adaptive software development approach rather than an overly planned one, encourage incremental development in short iterations, value lightweight solutions and simplicity, and also emphasize the human aspect [1].

The kind of process support that we suggest is lightweight, and is based on limited interference during the development rather than tight and intrusive one – which we believe is more appropriate for highly creative processes such as software development. We metaphorically view a software development process (or any other process) as a "trail" defined by the process methods and practices; the developers are considered as "hikers" who are supposed to follow the trail but, for various reasons, they once in a while deviate from it. The process support is aimed at helping the developers to follow the trail, and accordingly we attempt to identify cases of *Conformance*, and of *Deviations* from the trail. Upon such identification, the developers are usually provided with notifications intended to encourage the current behavior, or correct the deviation, respectively.

The process trail is informally described by a set of *development stories*, each describing something that the developer should do relative to a given process practice. The term is borrowed from XP, where *user stories* are used to describe desired system features. User stories are written by the customer, and are usually less detailed and formal than traditional requirements specifications. For each user story, a set of acceptance tests is developed, usually within a test harness such as JUnit. The story is considered to be correctly implemented if all these tests pass, and thus the tests may be considered as its actual specification [17]. As in XP, each development story is related with a set of acceptance tests (i.e., development scenarios), that capture typical cases of conformance or deviation from the story.

The scenarios, besides validating the support for the story (recall that they are translated into concrete JUnit tests), may serve as a major source for the documentation of the practice, i.e., as a process model, which is an important vehicle for process understanding and communication between process participants [20]. This approach, of using the tests as documentation, is strongly advocated by the agile community [17, 52].

Anticipating places where process support is needed, or knowing in advance the adequate set of scenarios for a given story, is difficult. Therefore, a major agile principle which we adopt is iterative development. We argue that in order to define effective and practical process support, one should define the support in an agile fashion, starting from simple support that is iteratively refined through user experiments [53]. The iterative definition of the support is facilitated by the increased modularity of AOP and the fact that new aspects can be easily incorporated into the development environment.

Agile methods also emphasize close relationships with customers and users during the development, in order to develop a system that actually meets user needs. This approach finds its expression here in the iterative development of the support that involves user feedback, and also in additional activities that were made where the first version of the support was formulated with the help of users (developers).

4.2 Test-Driven Development Case-Study

In this section we present a case-study conducted in the Computer Science Department at the Technion where support for the TDD practice was developed in an agile iterative fashion. TDD (also referred to as 'test first') is widely considered both one of the central contributions of Extreme Programming to general agile techniques [18], and one of the most difficult practices to internalize [22, 29]. In XP, at the beginning of each development iteration, each developer is assigned several tasks derived from user stories. Each task is implemented in a series of steps; working TDD means that in each step a test (unit-test) is first written and only then the developer writes the minimal portion of code that makes it pass that test. TDD means that testing drives coding and not vice versa. This ordering is considered to have many advantages such as forcing simplicity, leading to a simple design, and providing a "safety net" from changes in the code.

The process support which was developed for the TDD practice is referred to as the TDD-Guide tool. The tool can detect conformance or deviation from test-driven practice as coding or testing steps are being developed, and provide valuable notifications to the developer. Some of the notifications provide the developer with positive feedback when the practice is followed,

while others identify deviations from TDD. When deviations are detected, the tool can guide the developer to correct the deviation or even strictly enforce TDD by not allowing the developer to perform an operation deviating from the practice.

The tool was developed in three iterations, and three user experiments were conducted. In each experiment, student developers with novice TDD skills were given a Java development task, and were asked to develop the task using TDD. TDD-Guide was integrated in advance into the users' development environment (Eclipse), and significant development steps were logged, also using aspects. Based on the logs and post-questionnaires given to the students, we searched for and developed possible support refinement towards the next iteration. We were also interested in examining the reaction of the developers to this kind of on-line guidance.

We present results showing that TDD-Guide is in general perceived by the users to be helpful and that the tool indeed is effective in guiding TDD. More importantly, we show how the user experiments provide important user feedback that helps to refine the process support, and how the HighspectJ framework supports such refinement.

4.2.1 The TDD-Guide Tool

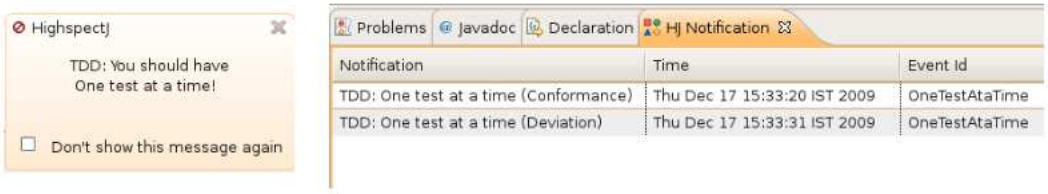

Figure 4.1: TDD-Guide's user interface presented to the developer

TDD-Guide provides the developer with positive/negative notifications during the course of development. TDD-Guide is an application of the HighspectJ framework and thus the notifications are presented within the framework's dedicated UI elements as shown in Figure 4.1. A new notification is presented within a colored notification bar (on the left), and all the notifications are grouped within an Eclipse view (on the right). In addition, the developer can click on a notification and provide textual feedback that

is saved for further analysis.

In the first and second development iterations of TDD-Guide, the aspect functionality was defined in a rule-based fashion using an earlier version of the framework, and not in a scenario-based fashion which is now used. Nevertheless, where we relate here to a previous definition of TDD-Guide, the newer scenario-based notation is used for the sake of uniformity and understandability. As of the third development iteration, four development stories constitute the TDD-Guide tool, each specified by several scenarios that provide concrete development examples for the story. The full specification of TDD-Guide with all the stories and scenarios is presented in Chapter 5, and here an overview is given.

The first development story is called 'Basic TDD cycle', and describes the most fundamental TDD guideline which is to write a test before code, and also how to conduct a TDD cycle:

Story #1: Basic TDD cycle. *The developer should write a failing test before the desired unit of functionality is developed. More specifically, the development should progress in cycles where in each cycle a test is first written, the test is then executed and shown to fail, the desired functionality is coded, and eventually the test is successfully executed.*

The story is exemplified by five different scenarios, among them the following main positive scenario:

 # Complete TDD cycle
UnitTestCreated
TestCaseModified+
TestCaseExecuted[result = "failure", failures = 1]
CodeModified+
TestCaseExecuted[result = "success", failures = 0]
→ BasicTDDCycle[type == "Conformance", cause == "Complete_TDD_Cycle"]

The scenario describes a complete TDD cycle. The cycle starts with a developer creating a unit-test (*UnitTestCreated*). In JUnit, a unit-test is a Java method marked with a @Test annotation. The unit-tests are defined within a dedicated Java class denoted as a JUnit test-case. After creating

the unit-tests, test logic is filled-in using the Eclipse Java editor (*TestCase-Modified*); the event is activated when a single character is inserted, hence its multiple sequential activations are allowed. Afterwards, the test case is executed and shown to fail (*TestCaseExecuted*), followed by an implementation of the missing functionality (multiple *CodeModified* events); eventually, the test is executed again and this time passes. When such developer's behavior is observed, the high-level event *BasicTDDCycle* is expected to be activated while reporting conformance with the story, which causes a corresponding response aspect to present the developer with a positive (encouraging) feedback aimed at increasing the developer's confidence that he/she is doing well, and clarifying the task ahead.

TDD-Guide is currently configured to support novices in learning the TDD practice and thus additional (more focused) positive feedback is provided. This configuration is flexible and may be modified (perhaps by leaving mainly the negative scenarios) if the tool is to be used by developers more experienced with the practice. A positive scenario titled 'Developer continues with testing after a cycle is ended' describes a developer who starts a new TDD cycle (i.e., defining a new test) immediately after the previous cycle has been completed. As shown in the user experiments, developers have a tendency to continue with coding after completing a cycle and thus for novices the act of moving to the tests is worth providing a positive feedback. Another positive feedback, facilitated by another scenario, can be provided when the developer moves to the code right after defining and executing a failing test, which is a point where a positive remark can be made about progress made during the TDD cycle.

The last two scenarios are typical counterexamples. In the first one, which is presented below, the developer starts a new TDD cycle by defining a new test, and then moves directly to the code without executing the test and seeing it fail. Another scenario reports a deviation which was previously mentioned, where a developer continues with coding after a TDD cycle is ended, instead of starting a new cycle, i.e, defining a new test.

 # Developer moves to code without executing the test
UnitTestCreated
TestCaseModified+
CodeModified
→ BasicTddCycle[type == "Deviation", cause == "To_Code_Without_Execution"]

The second development story is called 'One test at a time', and denotes a recommended TDD guideline[1] of not trying to fix several things at a time (support for the guideline was implemented in Section 3.3). Its related event *OneTestAtaTime* detects and announces cases of deviation from the story, e.g., when a developer starts coding while having several failing tests, and also cases of conformance, e.g., when coding starts with a single failing test as expected. The event *SimpleInitialCycle* which corresponds to the third development story 'Simple initial cycle', identifies cases where a developer spends too much time on implementing the first TDD cycle which is supposed to be simple and rapid. We elaborate on these two events in Section 4.2.4 where the refinement of the TDD support through the iterations is discussed.

The last development story is called 'TDD coding standards', and its high-level event helps to enforce a simple coding convention which states that the name of a test-case should end with the string "Test". The event reports a deviation when a test-case not following the convention is about to be created, and also when a developer attempts to create a regular Java class with the "Test" suffix. In such cases, a corresponding response aspect presents an error and stops further development until the developer follows the naming convention.

4.2.2 User Evaluation Setup

A support tool for software development can be best validated through user evaluation. The validation should involve user experience that is rigorously planned and executed aiming at refining the tool to be more effective [13, 66]. The users in our case are developers in software development teams who work to produce code according to some predefined functionality and need to produce unit tests to support the code. Three user experiments were conducted, for the first two development iterations of TDD-Guide. The experiments began from an in-depth qualitative approach [26] to aid in formulating refinements, and progressed towards more quantitative experiments with statistical analysis and finally a control group, in order to validate the effectiveness of a stable version of the tool.

[1] http://c2.com/cgi/wiki?OneUnitTestAtaTime

The purpose of the first experiment was to examine the initial process support of the first iteration in a real development setup. In XP terms, the experiment can be considered as a 'spike solution' where a simple program is written to explore potential issues. Six experienced developers familiar with Java and Eclipse, and less familiar with TDD, were given a simple development task and asked to develop the task while using TDD. TDD-Guide was integrated in advance into their development environments, and significant development steps were logged by using the framework's logging facilities. After the activity, personal reflections were filled-in by the developers; additional informal feedback, e.g., in the form of mails, was also treated.

The second version of the tool was experimented with in a larger setting, more focused on support refinement, namely examining the effectiveness of the tool in supporting TDD, and searching for unanticipated development states. The participants in the second experiment were 34 CS-major fourth-year students in an advanced Software Engineering project course. The experiment had three phases:

- A pre-experiment phase in which participants filled in a questionnaire about the level of their familiarity and experience with programming concepts and tools in general and with TDD in particular. Then, they heard a one-hour lecture about unit testing and specifically about TDD.

- An experiment experience phase in which participants moved to the computer lab where they were guided in groups of 2-4 to perform a specific programming task. After completing the task they filled in a personal reflection.

- A post-experiment phase in which participants were asked one week after the activity to indicate two features of TDD-Guide that they perceive as most significant and two possible improvements or extensions. This feedback was obtained using a web-based feedback mechanism familiar to the students.

Twenty seven of the participants filled in the questionnaire of the pre-experiment phase. The results show that participants felt knowledgeable with Java programming and object-oriented design, less knowledgeable with

Eclipse IDE and unit testing, and beginners in JUnit and TDD. Regarding the development process, participants were less experienced with measuring the development process and product, but felt knowledgeable and even expert with working in pairs.

After the second experiment took place, a third experiment with a control group was conducted with the same version of TDD-Guide except for minor adjustments. In the experiment, 35 student developers were divided into two groups, and as in the previous experiments, were asked to develop a Java programming task in TDD. The first group had 19 participants who were working on 10 workstations (9 pair developers, 1 single), and the second group had 16 students and 8 workstations (8 pairs). TDD-Guide was only installed in the workstations belonging to participants of the first group. Before the development phase of the experiment, a short introduction to JUnit and TDD was given, and participants filled-in a knowledge questionnaire. Also here, participants were novices in TDD and JUnit, and felt knowledgeable/expert with pair programming. This experiment was not focused on support refinement but on assessing the effectiveness of TDD-Guide, by comparing the performance of the two groups with regard to the practice. As in the first and second experiments, aspects were also added to Eclipse to log actual behavior and timing information.

The same Java development task was handed in all the experiments. Given a project named *money.conversions* that contains classes and conversion utilities, participants were asked to define a class *Money* that represents a certain amount of money in a specific currency[2]. In addition, the class should have the method *Money add(Money money, String currency)* where the returned *Money* represents the addition of the called *Money* object and the given *money* argument, in the given *currency* argument. Participants were asked to develop according to the TDD technique (within 35 minutes) and to take notice of TDD-Guide messages.

4.2.3 Is TDD-Guide Effective?

The initial feedback from the first user experiment was encouraging: all of the six participants showed positive reactions to the idea of accompanying

[2]The given task is a simplified version of a well-known TDD example by Kent Beck and Erich Gamma (http://junit.sourceforge.net/doc/testinfected/testing.htm).

the development with notifications and alerts, and four participants reported that the notifications helped them to develop test-first. No change was noticed in Eclipse performance due to the addition of aspects into it. The feedback led to changes in the user-interface; the notifications were presented to the developers in a dedicated Eclipse view, and two participants reported that paying attention to that view was sometimes a burden and disrupted their concentration. Accordingly, we decided to add a colored notification bar hoping that colored feedback would be more intuitive than a purely textual one. The logging was also improved by adding time-stamps to the logs to facilitate better reasoning, and minor changes were introduced to the process support.

In the second user experiment, the logs of twelve groups that completed their task were analyzed. The example findings below, which relate to the first TDD story 'Basic TDD cycle', demonstrate the effectiveness of TDD-Guide in supporting test-driven development:

- The first finding deals with the intuitive tendency of developers to start programming with coding rather than with testing. An expected behavior in the beginning of the log is *Test – TestFailed – Code* where *Test* means writing test lines, *TestFailed* means that running JUnit causes a failure, and *Code* means writing code lines. Four logs out of twelve include *Code – DeviationMessage – Test* at the beginning of the log (meaning that they started directly with code as they used to, noticed the TDD-Guide deviation message that appeared and responded by starting to test). This tendency to start with coding was also found in the middle of the task when instead of the expected *Code – TestSucceeded – Test* we found *Code – TestSucceeded – Code – DeviationMessage – Test*. These cases show that novice developers can benefit from TDD-Guide notifications and by following them they overcome their tendency to start coding without testing, and thus adhere to the TDD practice. We found two cases of *Code – DeviationMessage – Code* meaning the deviation message was ignored by the developers who continued to work on the code although there was no failed test. This can be also explained as a refactoring activity and was marked by us for further investigation.

- The second finding concerns getting used to actually run the tests before moving to code. We found eight cases in six logs where developers did *Test – Code – DeviationMessage – TestFailed – Code* meaning they worked on the test and switched to code without receiving the feedback of running JUnit, and thus not knowing for sure that the test fails. Following the deviation message they ran the test, causing a test failure, and went back to code.

- The third finding relates to the learning curve that can be observed especially when adding the time measure of the different activities. The following series of events was found starting at the beginning of a specific log:

  – *Code – DeviationMessage* for 1 minute; no work for 2 minutes;

  – *Test* for 15 seconds;

  – *Code – DeviationMessage* for 4 seconds; no work for 7 seconds;

  – *Test* for 8 minutes;

  – *Code – DeviationMessage* for 1 second;

  – *TestFailed – Code*

  The deviation message was used three times to correct the development in this trace. We observed here and elsewhere that the time to respond to the deviation messages decreased while the time invested in testing increased.

- The third finding reveals strong emotions against testing and can be seen as anecdotal: one group used "i dont want to test" as part of their test file name.

After completing the development task, participants filled in their level of agreement with related statements. The table in Figure 4.2 summarizes their answers; a clear majority is marked in grey. As can be observed most participants felt that the Eclipse IDE works as usual (statement 1) and that TDD-Guide helped them in working according to the TDD technique (e.g., statement 6). However, statements for which no clear majority exists reveal issues that may suggest UI and/or support refinement. For instance, statements 2 and 4 reveal usability issues, and statements 8 and 13 disagreement

with the TDD guidance. Statements 7 and 16 uncover resistance to the TDD concept; we believe this only emphasizes the necessity of the guidance, in particular for novices who are not yet familiar with the advantages of TDD.

#	Statement	Agree	Tend to agree	Tend to disagree	Dis-agree	No answer
1	During development, I felt that the Eclipse interface responded as usual	9	15	8	2	
2	Paying attention to the TDD messages was a burden	3	14	14	3	
3	I have hardly had any TDD Deviation (Error) messages	1	12	11	8	2
4	Some of the TDD messages were not comprehensible	5	12	11	5	1
5	Sometimes I didn't agree with what a TDD message was saying	2	4	15	12	1
6	The messages helped me to develop test first (TDD)	11	13	8	1	1
7	I find TDD an annoying technique	5	14	13	2	
8	Sometimes, I just ignored a TDD message	9	9	8	8	
9	Sometimes, I felt that a TDD message was needed but it didn't show up	3	5	18	8	
10	I think that accompanying the development with messages and alerts is not a good idea and just interferes with the fluent work	1	8	17	8	
11	Several times, TDD messages led to a change in my behavior	4	16	10	4	
12	I looked several times at the reference page to figure out how to develop test-first	3	6	10	15	
13	When a failing test does not exist, TDD-Guide should always disallow any coding	5	13	13	3	
14	I got several "false alarms" (incorrect TDD messages)	2	3	14	14	1
15	The message view was more useful than the message bar	2	18	11	3	
16	I will definitely develop test-first in the future	2	15	10	6	1
17	Most of the TDD Warning messages were justified	6	24	5	-	

Figure 4.2: Reflecting on the user evaluation activity

Control group experiment

As noted, an additional experiment with a control group was conducted, where 35 student developers were divided into two groups:

- *TDD-Guide group.* 19 participants working on 10 workstations (9 pairs, 1 single).
- *Control group.* 16 participants working on 8 workstations (8 pairs).

We analyzed the development logs of the two groups, where the idea was to compare between them w.r.t. the number of TDD violations that are derived from TDD-Guide's development stories. The following three TDD violations were considered:

1. **Code before test.** The developer is developing (functional) code, while according to the correct TDD ordering he/she is supposed to develop test code.
2. **Moving to code without test execution.** The developer is moving from test to code without executing JUnit and verifying that the test actually fails.
3. **Developing several tests at a time.** The developer is working on several tests at once instead of developing a single test in a cycle.

For each TDD potential violation, we looked at related deviations and missteps. The term *deviation* refers to an actual and continuous violation of the TDD practice, e.g., a developer who creates a Java class and implements a certain unit of functionality before the related test is defined. The term *misstep*, on the other hand, denotes a case where the developer starts a violation course, but due to a TDD-Guide notification, immediately corrects his/her behavior and thus an actual deviation does not take place. Missteps are only relevant in the first group with TDD-Guide installed.

The number of deviations and missteps in the two groups, for each of the defined violation categories, is presented in Table 4.1. Table 4.2 summarizes the results and calculates an average deviation/misstep rate in each workstation.

Violation category	TDD-Guide group	Control group
Code before test	7 missteps, 1 deviation	6 deviations
Moving to code without test execution	8 missteps, 1 deviation	8 deviations
Developing several tests at a time	1 deviation	1 deviations

Table 4.1: Deviations and missteps according to violation category

Criteria	TDD-Guide group	Control group
Number of deviations	3 (0.3 per station)	15 (1.875)
Number of missteps	15 (1.5)	N/A

Table 4.2: Summary of the results including average rate

Out of 18 potential deviations in the group with the support, TDD-Guide helped in turning 15 of them to missteps (hence correction rate of 83.3%). As a result, it may be seen that the *actual* deviation rate in the TDD-Guide group (0.3 deviations per workstation), is significantly low compared to the one in the control group (1.875 deviations per workstation). Note that the potential deviation rate in the TDD-Guide group is 1.8 (deviation rate + misstep rate), which is quite similar to the deviation rate in the control group (1.875). TDD-Guide has dramatically prevented this potential from being exploited.

We would like to emphasize another finding. At first glance, the violation 'Moving to code without test execution' may not be considered as severe, because we may think that the developer only forgot to execute the test. However, in practice we observed that this kind of violation is not that innocent. If it is only a matter of forgetfulness, when a TDD-Guide notification of this kind is introduced we would expect the developer to simply execute the test and then return to the code. Nevertheless, only two such cases were observed in the group with the support. In six other cases, the developer when provided with a notification, before executing the test, first went to modify the test code (for 5 minutes, 12 minutes, 1 minute, 1 minute, and 20 seconds). We learn, that in these cases TDD-Guide caused the developers not only to execute the test, but probably to define it in a more meaningful manner.

4.2.4 Refining TDD-Guide

The feedback that is collected during a user experiment, and in particular the analysis of the related development logs, facilitates the refinement of the process support in its next development iteration. The refinement involves adjustments of existing scenarios, introduction of new scenarios, and even of new development stories. Below, examples are provided for refinements made to TDD-Guide in its third development iteration, based on feedback received in the second user experiment.

Refining the Story 'One test at a time'

In the second iteration, the high-level event *OneTestAtaTime* was defined that whenever the developer modifies code while having more than one failing test, a deviation notification is provided. Such behavior can be described by the following scenarios:

 # Developer modifies code with several failing tests
TestCaseExecuted[result = "failure", failures = 2]
CodeModified
→ OneTestAtaTime[type == "Deviation", cause == "Code_Several_Failures"]

 # Developer modifies code with one failing test
TestCaseExecuted[result = "failure", failures = 1]
CodeModified
→ OneTestAtaTime[type = "Conformance", cause = "Code_One_Failure"]

The first scenario describes a developer who modifies a Java class after executing a test-case with two failing unit-tests (which is a deviation), and the second scenario describes a positive behavior of a developer who executes a test-case having one failing test and then moves to code. (Recall that the two failing tests in the first negative scenario are only an example for a more general case where any number of failings tests greater than one is disallowed.)

One of the logs revealed that coding while having several failing tests is not always a deviation; that may happen when coding indeed starts with

one failing test however changes made in the code cause the failure of others. Although it may indicate tests that are not reasonably independent, it should not be classified as a deviation. In consequence, it was decided that a deviation should not be reported whenever code is modified after executing several failing tests, but only if coding *starts* in that way. Accordingly, the first negative scenario was modified as follows (changed parts are underlined):

 # Developer starts coding with several failing tests
TestCaseModified
TestCaseExecuted[result = "failure", failures = 2]
CodeModified
→ OneTestAtaTime[type = "Deviation", cause = "To_Code_Several_Failures"]

The addition of a *TestCaseModified* event ensures that a negative notification is only provided when the developer moves from testing to coding after executing several failing tests. Similarly, the positive scenario was also modified to report conformance only if coding starts with one failing test. The logs revealed another potential problem; we observed three occurrences of a deviation message "Do not code with several failing tests" belonging to that story that were ignored, i.e., the developers continued to code although having more than one test failed. The reason may be that the guidance was applied in retrospect, i.e., when the developers already had the tests written. The lesson learned was that a deviation should be reported as early as possible, when its correction is practical. Indeed, in the third development iteration a new negative scenario was added, that captures the deviation at the moment it occurs:

 # Developer creates the second consecutive unit test
UnitTestCreated
TestCaseModified+
UnitTestCreated
→ OneTestAtaTime[type == "Deviation", cause == "Second_Consecutive_Test"]

In the scenario, the developer creates a unit-test method (*UnitTestCreated*) using the editor, modifies the test's code, and then creates an addi-

tional unit-test. At this particular moment a deviation is expected to be reported, aimed at causing the developer to stop creating the additional test and move to the code (after executing the single failing test of course).

Adding the Story 'Simple initial cycle'

The addition of time-stamps to the logs revealed that a significant aspect of a correct TDD trail is related to time. For instance, we found several cases where tests were developed (for the first time) for more than ten minutes before moving to the code, and cases where the first successful test execution took place only after fifteen minutes. Considering the relatively simple programming task given, both findings indicate that the initial TDD steps are too complex. Therefore, in the third development iteration of TDD-Guide a new story was defined 'Simple initial cycle' to help in avoiding such behavior. The story has two scenarios, one negative and one positive:

 # Developer spends too much time on the first test
TestCaseCreated
TestCaseModified+
wait(310)
TestCaseModified
→ SimpleInitialCycle[type == "Deviation", cause == "TDD_SimpleInitialCycle"]

 # Developer finishes the first cycle on time
TestCaseCreated
wait(410)
TestCaseExecuted[result = "success", failures = 0]
→ SimpleInitialCycle[type == "Conformance", cause == "First_Cycle_On_Time"]

The first scenario is negative, and provides an example of a developer who spends too much time on modifying the first cycle's test. Here, 'too much time' is defined as more than five minutes, however it may be adjusted as agreed in each development team. The general behavior which we would like to exemplify is of a developer who creates a test-case, and then modifies the test-case for more than five minutes. The first *TestCaseModified* event has an E+ semantics, yet as noted, in the example that appears in

the corresponding JUnit test a single occurrence of that event is simulated. In this example, a test-case is created (*TestCaseCreated*) and modified once (*TestCaseModified*), afterwards a minimal period of time that exceeds the defined five minutes threshold is waited. The *wait(..)* construct causes the scenario's simulation to pause for the period of time indicated (given in seconds), where the next event in the sequence – in our case *TestCaseModified* – is assumed to occur immediately afterwards. This second activation of *TestCaseModified* is expected to trigger the high-level event *SimpleInitialCycle* with a deviation type, and an appropriate cause.

This example relates to a developer's behavior which is not that natural, but it is still a valid example for the intended general case. The scenario may be better understood while relating to the corresponding FSM; the FSM stays in the second state as long as a *TestCaseModified* event is observed and no longer than five minutes have passed since the initial test creation. A *TestCaseModified* event that occurs afterwards transits the FSM to the final state. Defining an example scenario for a behavior that involves timing constraints is more challenging and the scenario should be probably augmented with a short textual documentation to make it more comprehensible.

The second scenario is positive, and describes a developer whose first successful test execution occurs within seven minutes since the test-case was created (which we find a reasonable time to complete the first TDD cycle). Here, the low-level events of interest are *TestCaseCreated* and *TestCaseExecuted*. If a test-case is created, and then within seven minutes a successful test execution occurs, an activation of type conformance of the event *SimpleInitialCycle* is expected. Also here the amount of time specified of six minutes and fifty seconds is only an example (note that in both cases the time given hints at the general case).

The automatic code generation of the framework ignores *wait()* tokens, and timing constraints should added to the generated FSM manually (see Section 3.3). In addition, the group of events that affects each scenario should be refined. In the first scenario, we do not care if a *TestCaseExecuted* event occurs in the middle of the defined event sequence, hence this event should be manually removed from the group of affecting events (which is by default all events in all scenarios). Also in the second scenario, other events (i.e., *TestCaseModified*, *TestCaseExecuted[result = "failure"]*) are allowed (and are expected) to occur in the middle of the sequence.

4.3 Defining Support for Abstract Thinking

Abstract thinking is defined as the ability to generalize, or focus on the significant details while ignoring the less significant ones. It is considered to be a key skill for software engineers as reported in some preliminary educational initiatives that enables a comprehensive understanding of a specific concept or a problem using different levels of detailing [47, 46].

Defining concrete IDE support for abstract thinking by means of on-line notifications and alerts is challenging. Unlike TDD, abstract thinking is an informal practice, which cannot be easily captured by a set of well-defined guidelines. While coming up with the first version of TDD-Guide was quite straightforward, based on existing literature and our own experience, for such an abstract practice it is not even clear where to start. Therefore, in order to understand the nature of the support needed, we conducted a three-part lab activity. In the first part, thirty-three participants studied the notion of abstraction, namely abstract thinking. Then they worked on real-life development tasks, where their development steps as well as their visible thinking processes were logged by observers. Finally, a reflection on the activity was collected [64, 32].

Analyzing the data from the logs and reflection, we conclude that concrete HighspectJ support for abstract thinking within the IDE is practical, and suggest two kinds of such support. The first is concerned with providing a positive feedback in cases where abstraction is used, and the second with cases where the developer is encouraged to move to a different level of detailing, that is, promoted to use abstract thinking. We believe that such support may increase the awareness of abstraction and encourage its further utilization. Below, we elaborate on the activity, its findings, and conclusions.

4.3.1 Lab Activity Setup

The participants in the activity were thirty three fourth-year students who were computer science majors in an advanced software engineering project course. The activity had three parts:

- Lecture – in which participants learned and discussed the notion of abstraction and how it is related to software engineering.

- Development activity – in which participants heard about the pair programming practice and were distributed into groups of two or three developers in order to perform a development task in solo or pairs, respectively, where the additional person in each group served as an observer. This part was performed in the lab where the course was taught.

- Reflection – in which participants reflected on the complete activity using a web-based feedback mechanism familiar to the students. The feedback was given during the week after the activity.

As part of the work in the course, about two months before the activity we asked the participants to fill in a questionnaire about their level of familiarity and experience with programming concepts and tools. Twenty seven participants answered. The results showed that participants felt knowledgeable with Java programming and object-oriented design, and less knowledgeable with Eclipse IDE and unit testing. Regarding the development process, participants felt they were less experienced with measuring the development process and product, but felt knowledgeable and even expert with working in pairs.

In the lab, participants were distributed into thirteen groups: seven groups of three participants (a pair of developers and an observer) and six groups of two participants (a developer and an observer). All developers were asked to pick a development task from their on-going project and to develop it during this phase of the activity. Solo developers were asked to think aloud – i.e., talk while working – about whatever they were doing. The observers were asked to be non-participatory observers, i.e., not to talk with the solo/pair or interfere in their work. Their task was to document the development activity. Table 4.3 presents the kind of pages observers were asked to fill during the activity.

Time stamp	Conversation	Activity	Comment

Table 4.3: Observer page

Once the lab activity started, all developers began working on their tasks and observers began to record their observations. This lasted for exactly one

hour, after which we stopped the participants asking them to upload the code and test files that were changed to the course's web site. The observers pages were collected and an open discussion was guided in order to receive an initial feedback.

4.3.2 Activity Findings

Overall we collected thirteen activity logs each spanning two to five observation pages; Observers of solo developers filled a total of fourteen pages (2,2,2,2,3,3 average of 2.3) where observers of pairs filled a total of twenty-five pages (2,2,2,4,5,5,5 average of 3.6). We analyzed the logs and found cases of *abstraction usage* (i.e., cases in which abstract thinking is used) and cases of *abstraction necessity* (i.e., cases where abstract thinking may help).

Abstraction usage

We found four categories of abstraction usage that are characterized by the use of generalization or by the process of ignoring details in order to advance the development work. In what follows we describe each category and illustrate it using the data from the activity log.

Searching. To perform a search operation, one needs to select keywords that adequately represent the problem domain. This selection of searching keywords is an abstraction of type removing details. Here, in the process of solving a problem, participants selected keywords from the problem domain and searched for these in Google or in the API documentation in order to find a solution. Tables 4.4 and 4.5 present such a case as reported by observers of a solo and a pair, respectively. When no data appears in the comment column it is not presented.

Time stamp	Conversation	Activity
11:04	Open meta-inf in order to add an extension point	Looking in Google, studying how to add the required action
11:05	Use popupMenu as extension → view contribution – add a unique ...	Continue to move between Eclipse and Google in order to build the extension

Table 4.4: Searching in solo programming

Time stamp	Conversation	Activity
11:12	Discussion about how to implement the server	
11:16		Adding Jars to the client project
11:20		Searching Google for permission in RMI
11:24		First trial to solve the permissions problem
11:27		Second search in Google for a solution
11:28	One suggests solving the problem using a policy file. Also suggests refining the search according to the Eclipse error	

Table 4.5: Searching in pair programming

We can see that in both cases there is a move between coding and searching activities. In the pair observation there is also a conversation on which keywords to use and about refining these keywords.

Naming. When choosing a proper name for an underlying coding element (e.g., classes, methods), one captures the essence of the element in a short

yet meaningful term. The process of choosing a name involves abstract thinking of type removing details. In Tables 4.6 and 4.7 we show naming-related excerpts from two different pairs: naming a class and renaming class members.

Time stamp	Conversation	Activity
11:00	How to name the class?	Creating LoginDialog Class

Table 4.6: Naming a class

Time stamp	Conversation	Activity
11:35	Performing refactoring to member names that are more correct in IOAutoX	Updating gui_integration_objects.IOAutoX

Table 4.7: Renaming class members

Testing. We claim that writing a test for a specific unit requires a different level of abstraction than writing the code of the unit itself. Instead of focusing on the implementation details of the unit, writing the test triggers thinking about how to check the unit functionality. Table 4.8 presents such a case.

Time stamp	Conversation	Activity
11:14	A slight argument on the right location of classes under the different packages. Talk about testing the AutoXlog class using JUnit	Move AutoXlog to Model_Experiments package. Adding a test for AutoXlog named AutoXlogTest
11:20	Making a decision on the correct protocol between the GUI and the model	Add GenerateAutoXStatisticsEvent in the Event Enum

Table 4.8: Testing a class

In this case the test code was written only after the application class was written. Once the testing class was finished, the developers moved back to the application code to develop a different functionality.

In JUnit, the *setUp()* method holds initialization code common to all test methods. Therefore, the time when *setUp()* is introduced (see Table 4.9) is an example of abstraction of type generalization since common initialization code for all existing test methods should be identified.

Time stamp	Conversation	Activity
11:31	Consulting with each other about the way of testing	Implement setUp(), add a new test, write test function named testGetExperimentID()

Table 4.9: Implementing setUp()

Using diagrams and views. The phrase "a picture is worth a thousand words" hints at why moving from writing lines of code to looking at a diagram can actually change one's level of abstraction, gaining meaningful insight while doing so. The diagram can be a UML-like representation of a class (Table 4.10) or an Eclipse view (Table 4.11) depicting the packages in a program.

Time stamp	Conversation	Activity
10:54	A conversation among the pair developers with respect to the requirements. What one object needs from the model and vice-versa. The conversation involved sketching and designing an additional class to represent the AutoX object	
11:01	Talking about code that is automatically generated as a result of an inheritance... Talking about performing refactoring to part of the code as a result of moving from iteration 1. Using a previous diagram in order to define the class members	Adding classes to the code...

Table 4.10: Using a diagram

Time stamp	Conversation	Activity
11:03	One refers to some packages with errors and discusses with the other how to handle them, i.e., define new tasks	

Table 4.11: Using an Eclipse view

Table 4.10 (10:54) denotes a conversation that led to the drawing of a class diagram and to a subsequent design discussion. This diagram was then used (Table 4.10, 11:01) by the developers when they coded in the members of that class. In Table 9 the developers readily evaluate their project's status, from Eclipse's Package-Explorer view, in terms of amount of pending errors. This evaluation leads to the definition of a yet another development task.

Abstraction necessity

We identified two categories of abstraction necessity in the activity logs. A more elaborate discussion of concrete support for abstraction necessity is presented in Section 4.3.3.

Searching. We identified events in which the developers searched for types in their code in order to perform their task (see Table 4.12).

Time stamp	Conversation	Activity	Comment
11:02	A conversation on where to add the solution in the code	Erasing existing code and a discussion on the solution implementation	Search for the solution location
11:06	How to implement the solution and using which existing classes in the project or in Java	Begin to write the new code and search for a specific method	Begin implementing and search classes that can help

Table 4.12: Searching for types

In Eclipse, searching for a type can be performed using several methods. It can be done by navigating through the project's file tree or by using higher-level search facilities such as the Search-Type-Dialog. We therefore suggest that a lengthy browsing through the project's file-tree should serve as a key-event that will open a notification recommending higher-level searching aids.

Testing. A testing-related need for abstraxtion is characterized by the creation of several test methods in a test class. We suggest that this key-event trigger a notification suggesting the creation of *setUp()* and *tearDown()* methods to enable a common initialization and finalization of the test fixture, thus encouraging abstract thinking of type generalization.

Another abstraction necessity stems from the fact that we have activity logs where not even a single testing-related activity was recorded. This obviously implies that the developers are not looking at their program from

a test-centric standpoint. Such situations can be identified by a timer-based key-event that is associated with a testing-oriented recommendation dialog.

Reflection

In this phase, participants were asked to reflect on the activity. Specifically, they were asked what they had learned, how they planned to use abstraction, and to give an example of a pair conversation that includes abstraction. Twenty six such reflections were examined.

The following are some answers by participants with respect to how they planned to use abstraction.

- "When I explain the project goal to my friends I use a level of abstraction that can give them a concept without delving into the details. At work, when we plan a project we go over all abstraction levels in the UML and do them all."

- "I will use abstraction in the following case: there is a need to implement classes for different kinds of messages that are transferred in the network. All messages need the functionality of converting the message from stream class to bytes and vice versa. In this case I will create an abstract class that includes the above-mentioned functionality and common definitions where every message will inherit it and implement the functionality according to its needs."

- "In the design phase of iteration 2 we used abstraction of the code for a static diagram (classes) of the software in order to be able to talk about what will be in the code from a high level perspective."

- "Working in a large team, in order to reach the biggest common denominator I will use abstraction to explain the project essence and its tasks."

Most participants answered this question by referring to high-level design activities. In the last example, there is a use of abstraction for simplifying communication. In reply to this question, only four out of twenty six participants mentioned IDE-level activities, such as the ones mentioned in the second example above.

Eighteen participants out of twenty six answered the question to describe a pair protocol that includes the use of abstraction. Among them fourteen indicated coding activities. We present two specific answers in which participants used a metaphor as the abstraction usage in the conversation. Metaphor [50] can be viewed as a kind of an abstraction since it uses one set of terms in order to better explain concepts expressed in another set of terms. The following are parts of the answers of participants.

- Developer A: We need to develop software to implement the communication between two stations.
  Developer B: Lets think as if we have here two people that talk with each other. How does the conversation begins?
  Developer A: One says hello and the other answers.

- Person A: I would like to implement a communication protocol between server and clients.
  Person B: What do you mean?
  Person A: I would like to develop a program that implements a connection that is similar to a father who distributes roles among his sons; he can give a role only to one son simultaneously and cannot answer questions of more than one son simultaneously.

4.3.3 Discussing the Findings

We believe that abstract thinking is an important cognitive tool that is required in the field of software engineering. In order to use this tool, a developer must be aware of its existence and of its applicability to the task at hand. Examining the issue of awareness we see that the majority of participants in our activity associated abstract thinking with activities that are external to the use of the IDE. The answers in the reflection part of the activity primarily mentioned design activities and sentences regarding the nature of the project or the initial definition of large modules thereof. Despite the fact that our activity logs show many events where abstract thinking was used during coding, the participants largely ignored events of this kind in the reflection.

This suggests that our participants awareness regarding the role of abstract thinking during actual coding activities is low. Therefore, in what

follows we suggest guidelines for concrete IDE support that will improve the developers awareness to coding-time abstract thinking.

- **Conformance notifications.** The idea is to identify certain events where the developer moves between different levels of abstraction and provide a positive feedback that may encourage further usage of abstract thinking. Moreover, bringing into awareness the use of abstraction may lead to a more professional usage of this cognitive tool.

  One example, derived from the findings, is to define a high-level event that detects cases where after a long editing session the developer consults a related design diagram and then goes back to coding. We see this as a good use of abstract thinking since the developer decided to move from a detailed view of the program to a less detailed one and vice versa.

- **Deviation notifications.** Such notifications relate to abstraction necessity, where based on monitoring the developers activity, a recommendation may be provided whenever a different level of detailing may be helpful.

  For example, we can define an event that detects cases where several tests have failed together. As a result, a notification may be presented saying: "try to find what these tests have in common". Further assistance can be provided by showing the intersection of the code coverage of these failing tests. Naturally, the developer does not have to follow this recommendation. However, if he/she chooses to accept the recommendation he/she will be able to think about the task from a different point of view (with a different level of detailing).

  We conclude with one subtle point that is related to the interplay of pair programming and abstract thinking. Observers of pairs recorded significantly more observations than observers of a solo developer. This is clearly indicated by the average number of pages filled by the observers (2.3 vs. 3.6). We attribute this to the fact that the pair-programming practice promotes verbal communication, which in turn promotes abstract thinking. The developers had to explain their intentions instead of simply writing them in code. This process of explanation requires abstraction, as well as self reflection, and is expected

to enhance the pair's quality of work. Implementing support for verbal communication within the IDE is left as food for thought.

4.4 Defining Support for Code Refactoring

Code Refactoring [27] is about improving the structure of the program without modifying its behavior. It is one of the Extreme Programming core practices, and a common practice in other agile methodologies as well. A developer who is following the practice is supposed to notice – during the development – *code smells*, that is, indications in the code that it needs to be better organized and structured, and then take an appropriate refactoring operation. For instance, a common code smell is the existence of a long method; the corresponding refactoring operation is called *extract method*, and it involves the division of the long method to several smaller methods. A comprehensive catalog of code smells and related refactoring operations was introduced by Martin Fowler in [27].

We are interested in defining initial process support for the refactoring practice. Defining first-version support for this practice is perhaps less straightforward than TDD, but definitely more obvious than abstract thinking. Still, like abstract thinking, refactoring support was specified with the help of the developers. By that, we may come up with different and unexpected directions for support. Furthermore, the developers may show greater sympathy to process support initially defined by them and not solely by the management.

4.4.1 Classroom Activity

Thirty-four student developers from a yearly project course in our faculty took part in the activity. The activity's main goal was to teach the students the refactoring practice, where one of its major parts was to define concrete development scenarios for the practice. The activity was carried out during two course meetings (the course has weekly meetings). In the first meeting, the students learned the basics of the practice. Following a short introduction, the developers were divided into pairs, and each was handed an activity sheet. The sheet introduced main refactoring concepts and contained code snippets, where the students were asked to suggest appropriate refactoring

operations. In addition, it included the following open questions:

1. Give (at least two) examples for possible refactoring operations in your project's code.

2. In which circumstances would you recommend to perform a refactoring operation?

3. In your opinion, how are unit test related to refactoring operations?

4. What is a reasonable frequency for conducting refactoring?

Towards the second meeting in the next week, the students' answers for these questions were analyzed, and selected quotes that may hint at general directions for process support were extracted, and organized into so-called "Refactoring Issues". The issues capture repeating and/or significant ideas raised by the students with regard to the practice. The issues and example quotes are presented in Table 4.13.

Already at this stage it may be observed that some of the students' quotes are concrete enough to be translated into development scenarios. For instance, we may identify a developer who is conducting a sequence of copy & paste operations (issue #2), or one that executes the unit-tests after conducting a refactoring operation (issue # 6). Yet, many of the quotes are still vague, and thus we wanted the students to continue with materializing the practice.

After presenting the students with the refactoring issues and related quotes, they were introduced to the main concepts of the HighspectJ framework, and in particular to the concept of development scenarios. Then, the students were divided into their original groups (around six teammates each), and each group was given a task; based on the presented issues, each group had to define a set of development scenarios while indicating the issues that they are derived from. The students were asked to define two positive scenarios and two negative ones, and to present them to the entire class at the end of the activity. They were asked to define the scenarios in a free natural language and not in an event-based notation. In Table 4.14, selected scenarios as defined by the groups are presented.

#	Refactoring issue	Example quotes
1	Code smells that indicate a need for refactoring	"in a case where a class or a method is too long, we may split their code"; "using constants (numbers, strings, etc.)"
2	Certain usage of development tools may hint at a need to refactoring	"doing plenty of copy & paste operations"; "a lot of accesses to methods of other class"
3	Refactoring – not only for the functional code	"we may refactor the unit-tests to make them more organized and readable"; "after code refactoring, we may need to refactor also the tests"
4	When to refactor?	"in case of minor changes then anytime is appropriate"; "during the development"; "at the end of the day"; "before adding new code to existing old code"; "at the end of a milestone/iteration"
5	Refactoring improves code testability	"refactoring results in smaller code units which can be better tested"; "refactoring simplifies the code and thus it is easier to write tests"
6	Refactoring is supported by testing	"after a refactoring operation, unit-tests should be executed to make sure the system's functionality is preserved"
7	Testing encourages refactoring	"while working on the unit-tests you go over the code and may recognize where refactoring is needed"; "you may identify a need to refactoring during work on the unit-tests"

Table 4.13: Refactoring issues based on students' quotes

#	Scenario	Type	Issue	Discussion
1	Having methods with more than 20 lines	Negative		
2	The developer splits a long method to several methods	Positive	1	
3	The developer copies&pastes several lines in several different places	Negative	2	
4	The developer writes a public method and does not write a test for it	Negative	3	
5	The developer copies&pastes more than one line	Negative	2	Isn't it too rigid?
6	Before committing, the developer compares changes in the current version with the previous version (a kind of a reflection)	Positive	4	How will you implement it? answered that upon usage of the 'compare' feature
7	The developer deviates from the maximal method length which was set	Negative	1	
8	The developer executes the unit tests	Positive	6,7	
9	The developer refactors a certain class, updates the unit tests, then executes them and get a green [JUnit] bar	Positive	6	
10	The developer is doing refactoring while the bar is green	Positive	6	
11	The developer is doing refactoring while the bar is red	Negative	6	
12	The developer defines a new class whose code significantly resembles an existing class	Negative	1	How can we detect resemblance?

Table 4.14: Selected Refactoring Scenarios

As seen in the table, many of the scenarios' descriptions are quite concrete and can be easily expressed in an event-based notation. For instance, for scenario #7 that mentions a deviation from a maximal method length that was set, an event called *MethodModified* may be defined, that exposes the length of the modified method. Upon activation of the event, if the reported length is higher than a defined threshold, a deviation is reported. Scenario #3 mentions copy&paste operations that appear in several different places. Here, a *CopyPaste* event may be defined, and upon identification of several occurrences of the event within a short time period, a high-level event called *RefactoringSmell* may be reported. Note also scenario #9 that describes a high-level event which is composed of three different low-level events namely a class refactoring operation, a modification of a unit test, and an execution of a unit test.

During the week after the activity, the students were asked to answer a web-based feedback. Among others, each of the the students was asked to choose one development scenario from the ones defined in class (or to come up with a new one), and to elaborate on its possible implementation in the IDE. This question was given since some of the scenarios presented in class were still vague and general (e.g., #12, #6), and we hoped that additional individual work might increase the level of detailing and feasibility of the scenarios, that is needed in order to actually implement them.

This activity demonstrated how to materialize a process practice in the form of concrete scenarios with the help of the developers (recall that here and also in the previous abstract thinking activity the scenarios were not implemented). The activity started with general open questions and a related presentation, and progressed towards more closed stages where development scenarios were defined in groups and eventually in individuals. The first version of TDD-Guide was defined solely by us; in the abstract thinking activity student developers were involved but were passive, and here they took an active part in defining the scenarios. There is probably room for these three approaches, depending on the context and circumstances.

Chapter 5

Implemented HighspectJ Support

In this chapter, the implemented HighspectJ support is surveyed – support for code integration guidelines, usability evaluation of user interfaces, and of course for the TDD practice. For each supported practice, background and motivation are given, and the complete specification of the support is presented and explained including notes on the implementation. At the end of the chapter, the overall implementation is summarized, and links for downloading and installing the support are provided.

5.1 Support for Code Integration

In this section, HighspectJ support that was developed to help in internalizing code integration guidelines is presented. In a typical software development setup, each developer adds or modifies code in a local workspace, and once in a while integrates the code into a shared repository. The coordination between the different developers and the code repository is usually managed by a Version Control System such as CVS or SVN. These systems allow each developer to perform several operations; during a *commit* operation (or check-in), the developer loads new code that was developed in his/her local workspace to the shared repository. Another operation is *synchronize*, where the developer submits a query to the repository to ask whether new code changes made by other developers exist. If changes exist,

the developer may take an additional operation called *update*, where the local code is automatically augmented with the new changes.

During the development, and depending on the development methodology used, several code integration guidelines should be followed to allow a fluent collaboration between the developers and hence a more stable developed system. For instance, one common guideline dictates that a developer, prior to a commit operation, should synchronize the local code from the repository, and update the local code if changes exists. By that, it is guaranteed that the developers, before committing, take into consideration changes made by others and not override them. HighspectJ integration support for this and other integration guidelines is presented below.

Specification of the Support

Usually, the developer cannot just commit the code but should carry out additional operations. Such operations, including the commit itself, are denoted as the *commit process*, which is described by the following development story:

Commit process. *Prior to a commit operation, the developer should synchronize the local code with the code in the shared repository, and update if needed. Then, the test suite containing all the unit tests should be executed in the local workspace. The test suite should pass (otherwise code should be fixed), and no further local code modifications should be made until the commit operation.*

The story mentions two additional activities that should be conducted prior to a commit: synchronize/update operations, as explained above, and a successful execution of the test suite. XP, as well as other agile methodologies, calls for the maintenance of a comprehensive suite of unit tests that should be executed periodically. One such occasion is prior to a commit operation, in order to increase the confidence that changes introduced by the developers to the shared code repository keep the developed system in a stable state. Note that the update operation should occur *before* the test suite is executed thereby ensure that the most updated version of the system is tested.

In addition, the developers are asked not to modify the local code after the test suite passes. If such a modification is required (e.g., when the test suite fails), then the commit process should be conducted again, i.e., additional synchronization/update and test suite execution should be carried out. This management policy is quite strict, and other more flexible policies may allow minor code modifications without requiring an additional commit process to take place. For such policies, the story and the scenarios should be refined accordingly.

A high-level event called *CommitProcess* implements support for the story and is specified by four different scenarios. The first scenario describes a typical positive developer's behavior in relation to the story:

 # Developer conducts a successful commit process
CodeModified
CodeSynchronized
TestSuiteExecuted[result = "success"]
CommitCode
→ CommitProcess[type == "Conformance", cause == "Successful_Commit_Process"]

The scenario simulates a developer who modifies the local code (*CodeModified*), and then asks the repository whether new code changes were committed during that time (*CodeSynchronized*). Afterwards, the test suite is successfully executed (*TestSuiteExecuted*), and eventually a commit operation takes place (*CommitCode*). In consequence, the high-level event is expected to be activated with a Conformance type and an appropriate cause.

Note that a *CodeUpdated* event is missing from the positive scenario, although the developer, depending on the code synchronization result, may conduct an update operation. It is because we are interested in a more general scenario that causes an activation of the high-level event regardless of whether the code was updated or not. As a consequence, if the developer follows the scenario but ignores a need for an update either unintentionally or deliberately, the high-level event is still activated with a Conformance type.

We estimate that it is not reasonable for a developer who just initiated a synchronization request to miss a need for an update. In addition, the process support we define is in general not intended to cope with deliberate

deviations but with cooperative developers that are interested in following the practices. Clearly, such an assumption is not trivial and requires intensive process education and training in the organization. Assuming that, the type of support we suggest is highly efficient in case of unaware or unintentional deviation; upon deviation the notifications serve as an important training tool or simply bring the deviation to the developer's attention. In case of a deliberate deviation, the level of support might encourage the developer to rethink whether the deviation is desirable. However, in this case the support is less effective unless an enforcement strategy is used with very detailed policies.

The three additional scenarios are counterexamples, that is, describe a developer who deviates from the story:

 # Developer attempts to commit without any prior operation
CodeModified
CommitCode
→ CommitProcess[type == "Deviation", cause == "Missing_Commit_Process"]

 # Developer attempts to commit without executing the test suite
CodeModified
CodeSynchronized
CommitCode
→ CommitProcess[type == "Deviation", cause == "No_Suite_Execution"]

 # Developer attempts to commit without updating code
CodeModified
TestSuiteExecuted[result = "success"]
CommitCode
→ CommitProcess[type == "Deviation", cause == "No_Update"]

In these three examples, the developer does not perform one or more of the required operations prior to committing. In the first negative scenario, the developer tries to commit without conducting any of the required operations. Alternatively, the scenario may relate to a developer who followed the guideline once, but then modified the code due to test failures, and is now trying to commit without conducting the required operations again as

mentioned in the development story.

Note the *CommitCode* event which is used in this (and other) scenario; in Eclipse, a commit operation is performed using the dialog window which is presented in Figure 5.1. The event *CommitCode* follows a naming convention indicating that the event is activated *before* the actual commit operation takes place. In our particular implementation it is triggered when the commit dialog is opened. This is as opposed to a possible *CodeCommitted* event (note the different name) that occurs after the 'OK' button of the dialog is pressed, when the new code changes were already committed. In all of the three negative scenarios, it is essential to use the *CommitCode* event since we want the developer to be notified (or alternatively enforced) before the code is committed so he/she has a chance to correct the deviation.

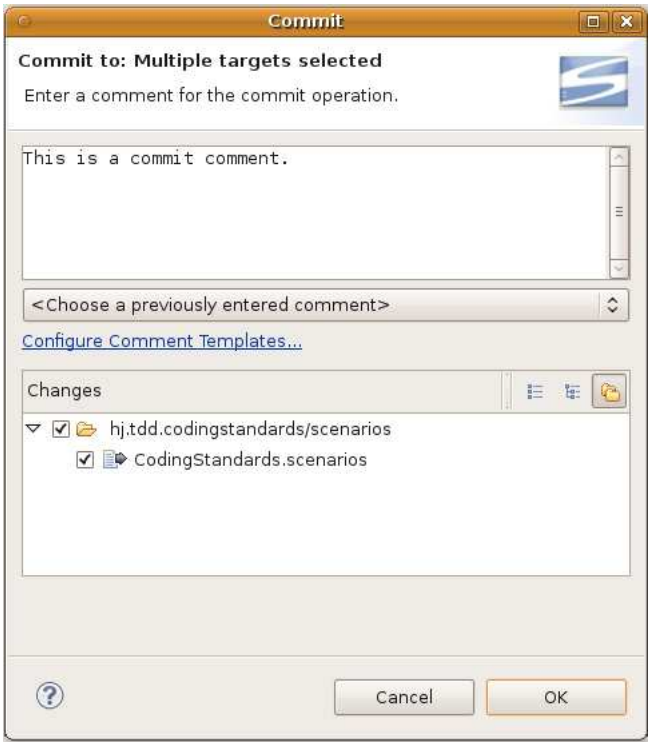

Figure 5.1: Eclipse commit dialog

In the second negative scenario, code was synchronized (and perhaps updated) but the test suite was not executed at all. In the last scenario, the

suite was executed, but the code was not synchronized. Usually, a version control system throws an exception in a case where the developer attempts to commit while new changes made by others exist in the repository, and thus an additional support may seem to be needless. This case if still supported because 1) by that, the developer is presented with a uniform set of process notifications and alerts 2) a scenario, besides supporting the practice, is also a source for process documentation 3) we do not want to depend on a particular behavior of a version control system.

Implementing the Support

Based on the four scenarios, the following dependency diagram was loaded into the HighspectJ framework:

CodeModified, CommitCode, CodeSynchronized, TestSuiteExecuted[result(String)]
→ CommitProcess[type(String), cause(String)]

The framework was also provided with the presented scenario specification, and in turn a variety of AspectJ/Java types were generated within two Eclipse plug-in projects: *hj.integration.commitprocess*, and *hj.integration.commitprocess.tests*. The first aspect project was provided with the following content:

- Four low-level event aspects – were imported from the event repository.
- One high-level event aspect – containing four generated FSMs, one for each scenario.
- One response aspect – having a default notification response.

Within the second test project, a JUnit test-case for the high-level event aspect was generated. The test-case has four JUnit test methods, one for each scenario. The response aspect was manually adjusted to take a regulative action (enforcement) when one of the negative scenarios is noticed. The default notification response for the positive scenario was left unchanged, and no adjustments were made to the generated FSMs of the high-level event aspect.

5.2 Support for Usability Evaluation

Usability is defined as the extent to which a product can by used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction [12]. Recently, the use of usability evaluation activities as an integral part of the development process, and their management from within the development environment is promoted [35]. In this section the HighspectJ framework is used to define usability evaluation support. In particular, we describe HighspectJ support aimed to assess the usability of particular UI components – Eclipse Wizards[1].

One common method to evaluate the usability of a given system is automatic evaluation [38], where during the usage of the UI by real users, data is automatically collected, and then analyzed and searched for usability problems. The potential of AOP for automatic usability evaluation is known [70, 69]. The benefits of using AOP are obvious: increased modularity without the need to modify the base-system's code, or for explicit hooks or defined API's in the base-system. Furthermore, using AOP one can perform the analysis of the events during the user-UI interaction, and thus the gathered usability data, which is usually voluminous and rich in detail, is more focused.

The UI components that we consider are Eclipse Wizards, which are components that guide the user through the specification of a complex task. Typically, a wizard consists of several pages where in each page the user is asked to fill-in input data required for the accomplishment of the task, e.g., as in Figure 5.2. When the user finishes specifying the input for the task, he/she presses the 'Finish' button and the task is carried out by the system. Usually, the user is not required to walk through all the pages, and some are optional or contain default input data.

[1]Note that Eclipse is treated here as a software product more than as a development environment.

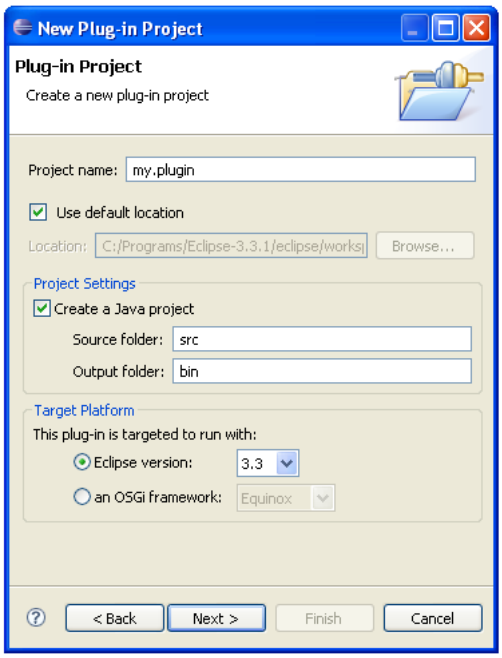

Figure 5.2: Eclipse new plug-in project wizard

Although the usability evaluation we suggest is relevant to most Eclipse wizards, we concentrate on a specific New-Plug-in-Project wizard shown in the figure. As explained in Section 2.2, Eclipse has an extensible plug-in architecture, and new plug-ins can be developed and added to it, thereby augmenting Eclipse with new functionality. A plug-in is developed within a special-purpose Java project whose creation is specified by the above-mentioned wizard. In the figure, only the first page of the plug-in wizard is shown; for Eclipse 3.5, an additional elementary page exists, and if the user is interested in creating the project using one of the existing templates, up to five pages are needed to specify the input data for the task.

In assessing the usability of the plug-in wizard, we focus on a specific usability guideline: the existence of a comprehensible help and documentation mechanism. According to Jacob Nielsen [7], "Any such [help and documentation] information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large." In principle, Eclipse provides two ways to access help pages. The first is simply to open the main Eclipse Help through the main menu of Eclipse. When using

that option, the user is presented with a help menu common to all Eclipse plug-ins and needs to search for the pages relevant to the problem. Another way to access help pages, which conforms more closely to Nielsen's guideline of providing a task-focused help, is the context-sensitive help mechanism. Using that option, the user can easily access help pages which are relevant to the task at hand; usually the help is accessed by pressing the F1 key and/or by a dedicated button located within the specific environment where the user is working. Such a context-sensitive help button exists for most Eclipse wizards and also for our plug-in wizard, as shown in the figure, in the lower left corner of the wizard (a circled question mark).

Our goal is to collect usage data about the wizard which is particularly focused on evaluating the usability of the local help mechanism in terms of *visibility* of the button, and *comprehensibility* of the provided help content. Based on the gathered data, several observations and conclusions may be drawn. For instance, if Eclipse users use the main Eclipse help while working with the wizard, it may indicate that the local help was not sufficiently comprehensible, or not visible if it was not used at all. Furthermore, we may find that the local help is barely used, which may hint at a visibility problem (or alternatively a wizard that is easy to use ad needs no explanation). Below, the specification of the usability support is presented and explained along with implementation notes.

Specification of the Support

A high-level event called *PluginWizardEvaluation* was implemented, that identifies significant usability event sequences that occur during the operation of the wizard. The event is specified using six scenarios, hence detects six different usage patterns.

The following two scenarios describe a user who opens the wizard but then aborts it, i.e., presses the 'Cancel' button, without completing the task:

 # User cancels without using the local help
PluginWizardOpened
PluginWizardCanceled
$\rightarrow$ PluginWizardEvaluation[type == "Default", cause == "Canceled_LocalHelpNotUsed"]

 # User cancels after using the local help
PluginWizardOpened
LocalHelpOpened+
PluginWizardCanceled
→ PluginWizardEvaluation[type == "Default", cause == "Canceled_LocalHelpUsed"]

Normally, a user is expected to complete the task, i.e., hit the 'Finish' button of the wizard. The abnormal behavior that is described by the scenarios may occur for several reasons, where one of which may be a usability problem. We distinguish between two cases; in the first case – represented by the first scenario – the user opens the wizard (*PluginWizardOpened*), and then, probably after navigating through it, hits the 'Cancel' button (*PluginWizardCanceled*). In this scenario, the local help facility is *not* used. This is as opposed to the second scenario that also identifies cancellation of the wizard, but where the local help was opened during the navigation (*LocalHelpOpened*). These two cases require the attention of the wizard's UI designers; the first scenario may indicate that the local help is not adequately visible, and in the second scenario, since the local help was used and still the task was aborted, it may be suspected that the local help was not comprehensible enough.

Both scenarios cause the activation of the high-level event *PluginWizardEvaluation*. The event has a default *type*, which may be used by events that carry information which is not classified as deviation or conformance, and also a *cause* variable which briefly describes the usage pattern that was detected. A corresponding response aspect saves these notifications (and others) for later observation and analysis made by usability experts. In some cases, automatic data collection of this kind may not be sufficient to draw certain conclusions, but still it may initiate and facilitate a further more focused investigation, e.g., interviewing the relevant users.

The FSMs generated for the two scenario need no manual adjustments and are ready to use. Note the E+ semantics which is added to the *LocalHelpOpened* event in the second scenario (and also in additional scenarios below). Such semantics is essential since we are interested in identifying also event sequences where the local help is used multiple times (and not only once).

While the two above-mentioned scenarios describe an abnormal usage,

the following two scenarios denote a normal expected operation of the wizard where the 'Finish' button is eventually pressed (event *PluginWizardFinished*), meaning that the user is interested in completing the task that is carried out by the wizard.

 # User finishes without using the local help
PluginWizardOpened
PluginWizardFinished
→ PluginWizardEvaluation[type == "Default", cause == "Finished_LocalHelpNotUsed"]

 # User finishes after using the local help
PluginWizardOpened
LocalHelpOpened+
PluginWizardFinished
→ PluginWizardEvaluation[type == "Default", cause == "Finished_LocalHelpUsed"]

Also here, the scenarios distinguish between the case where the local help is not used during the wizard's operation (first scenario), and the case where the local help is accessed (second scenario). If the results show that the local help is intensively used while navigating through the wizard, it may hint at a possible problem in the design or appearance of a particular wizard's page. On the other hand, if the local help is barely used, it may indicate that the wizard's operation is straightforward, or alternatively a problem in the visibility of the local help button; by comparing the actual average operation time of the wizard to an estimated normal operation time, a conclusion may be drawn of whether it is a matter of a simple-to-use wizard, or a complex one (and thus not visible local help). As with the previous scenarios, also here the FSMs generated within the high-level event aspect need no change.

The two last scenarios take into consideration the usage of the main Eclipse help. As explained, Eclipse has a central help system, that is common to all installed plug-ins. As opposed to the local help facility, which is focused on the task at hand, in the main help system the user needs to search for the desired information. An Eclipse user who wishes to open the main help window while working on a particular wizard, must first *cancel* the wizard's modal window (and hence lose any information that was filled-in).

Such behavior may reveal a usability problem of the local help; if the

user, before opening the main help, did not use the local help, it is probably a visibility problem since a reasonable user would at least take a glance at the more accessible help facility, before starting a more complex lookup process, and probably losing previously entered data. This usage pattern is captured by the first scenario which is presented below. Note that the FSM code generated for the scenario is adjusted to only consider activations of *MainHelpOpened* event that occur immediately after the wizard is aborted (less than 20 seconds), in order to neutralize other activations not related to the current usage of the wizard.

Alternatively, if the user uses the main help after using the local help, it is probably a problem of comprehensibility of the local help pages, since additional help is needed. Such a case is identified by the second scenario given below, where an activation of the *LocalHelpOpened* event is simulated during the operation of the wizard.

 # User uses the main help without using the local help
PluginWizardOpened
PluginWizardCanceled
wait(5)
MainHelpOpened
→ PluginWizardEvaluation[type == "Default", cause == "MainHelp_LocalHelpNotUsed"]

 # User uses the main help after using the local help
PluginWizardOpened
LocalHelpOpened+
PluginWizardCanceled
wait(5)
MainHelpOpened
→ PluginWizardEvaluation[type == "Default", cause == "MainHelp_LocalHelpNotUsed"]

Implementing the Support

To implement the described usability evaluation support, the six scenarios and the following dependency diagram were loaded into the framework:

PluginWizardOpened, PluginWizardCanceled, LocalHelpOpened, PluginWizardFinished, MainHelpOpened → PluginWizardEvaluation[type(String), cause(String)]

In consequence, the aspect project *hj.ue.wizards* was provided with five corresponding low-level event aspects (from the repository), one high-level event aspect, and one response aspect. A JUnit test-case for the high-level event aspect was also generated – within the test project *hj.ue.wizards.tests*. Minor manual code adjustments have been made to some of the generated FSMs as previously described (addition of timing constraints). In addition, the response aspect was modified to log the different event activations instead of notifying the users of their occurrence.

5.3 Support for Test-Driven Development

The definition and evaluation process of the TDD support (TDD-Guide tool) was thoroughly discussed in the previous chapter. In addition, an example TDD implementation was presented in Section 3.3. Here, the full specification of the tool is presented, while highlighting parts in the specification and implementation that were not previously elaborated.

Specification of the Support

The specification includes four development stories which are given below, each with its own set of scenarios. Short explanation and implementation notes are provided where applicable.

Story #1: Basic TDD cycle. *The developer should write a failing test before the desired unit of functionality is developed. More specifically, the development should progress in cycles where in each cycle a test is first written, the test is then executed and shown to fail, the desired functionality is coded, and eventually the test is successfully executed.*

The story describes the most fundamental TDD guideline which is to write a test before code, and also how to conduct a TDD cycle. The story has five development scenarios which are presented below. The first positive scenario describes a complete successful TDD cycle and was explained in Section 4.2.1. After completing a TDD cycle, the developer is supposed to start a new cycle, i.e., define a new test. The second scenario captures a typical related deviation where the developer continues with coding instead, and the third scenario describes the expected positive behavior. A common TDD deviation is moving from test to code without executing the test and see that it actually fails (see Section 4.2.3). Such deviation is indicated in the fourth scenario, and the positive behavior of executing the test first is demonstrated in the last (fifth) scenario. Note the E+ semantics which is given to event *TestCaseModified* in scenarios 1,4,5, which means that sequences with multiple sequential activations of the event are also considered. The aspect code generated for these five scenarios needs no manual adjustment.

 # Complete TDD cycle
UnitTestCreated
TestCaseModified+
TestCaseExecuted[result = "failure", failures = 1]
CodeModified+
TestCaseExecuted[result = "success", failures = 0]
→ BasicTDDCycle[type == "Conformance", cause == "Complete_TDD_Cycle"]

 # Developer continues with coding after a cycle is ended
CodeModified
TestCaseExecuted[result = "success", failures = 0]
CodeModified
→ BasicTDDCycle[type == "Deviation", cause == "Coding_After_Cycle_Ended"]

 # Developer continues with testing after a cycle is ended
CodeModified
TestCaseExecuted[result = "success", failures = 0]
TestCaseModified
→ BasicTDDCycle[type == "Conformance", cause == "Testing_After_Cycle_Ended"]

 # Developer moves to code without executing the test
UnitTestCreated
TestCaseModified+
CodeModified
→ BasicTDDCycle[type == "Deviation", cause == "To_Code_Without_Execution"]

 # Developer moves to code after executing the test
UnitTestCreated
TestCaseModified+
TestCaseExecuted[result = "failure", failures = 1]
CodeModified
→ BasicTDDCycle[type == "Conformance", cause == "To_Code_After_Execution"]

Story #2: One test at a time. *When wishing to add new functionality, the developer should focus on testing a single aspect of it. Consequently, the developer should write a single failing unit-test in each TDD cycle.*

The second story denotes a recommended TDD guideline of not trying to fix several things at a time. The three scenarios of the story were explained in Section 3.3, and their iterative refinement was discussed in 4.2.4.

 # Developer starts coding with one failing test
TestCaseModified
TestCaseExecuted[result = "failure", failures = 1]
CodeModified
→ OneTestAtaTime[type == "Conformance", cause == "To_Code_One_Failure"]

 # Developer starts coding with several failing tests
TestCaseModified
TestCaseExecuted[result = "failure", failures = 2]
CodeModified
→ OneTestAtaTime[type == "Deviation", cause == "To_Code_Several_Failures"]

 # Developer creates the second consecutive unit test
UnitTestCreated
TestCaseModified+
UnitTestCreated
→ OneTestAtaTime[type == "Deviation", cause == "Second_Consecutive_Test"]

Story #3: Simple initial cycle. *The first TDD cycle should be simple. The developer should not write a complicated test, just one that checks for the very elementary unit of functionality. In other words, the first successful test execution is expected within a short period of time.*

This third story was introduced in the third development iteration of TDD-Guide based on feedback gained in the second user experiment, and has two scenarios which involve timing constraints. An explanation for the story and the scenarios was given in Section 4.2.4.

 # Developer spends too much time on the first test
TestCaseCreated
TestCaseModified+
wait(310)
TestCaseModified
→ SimpleInitialCycle[type == "Deviation", cause == "Long_First_Test"]

 # Developer finishes the first cycle on time
TestCaseCreated
wait(410)
TestCaseExecuted[result = "success", failures = 0]
→ SimpleInitialCycle[type == "Conformance", cause == "First_Cycle_On_Time"]

Story #4: TDD coding standards. *A test case should have a name that distinguishes it from a regular class. In particular, the String "Test" should be part of a test's name (and not of a class' name).*

The high-level event of the last story helps to enforce a simple coding convention which states that the name of a test-case should end with the string "Test". The event reports a deviation when a test-case not following the

convention is about to be created (first scenario below), and also when a developer attempts to create a regular Java class with the "Test" suffix (second scenario). The default generated FSMs are refined to treat the general naming convention and not only the specific names defined in the example scenarios. In addition, the default action taken by the response aspect (notification) is modified to enforcement, i.e., the aspect presents an error and stops further development until the developer follows the naming convention. Note that the two low-level events used (*CreateTestCase*, *CreateClass*) follow the pre-action naming convention mentioned in Section 5.1, in order to facilitate an enforcement policy.

 # Developer violates test naming convention
CreateTestCase[name = "My.java"]
→ CodingStandards[type == "Deviation", cause == "Test_Naming_Convention"]

 # Developer violates code naming convention
CreateClass[name = "MyTest.java"]
→ CodingStandards[type == "Deviation", cause == "Code_Naming_Convention"]

Implementing the Support

Each development story is implemented in a separate Eclipse project, hence the following projects were created (each having a related test project):

- *hj.tdd.basictddcycle*
- *hj.tdd.onetestatatime*
- *hj.tdd.simpleinitialcycle*
- *hj.tdd.codingstandards*

In total, the four projects use seven low-level events, and within each project corresponding high-level event and response aspects were generated, including a JUnit test-case in the matching test project.

5.4 Summary

Six aspect projects and a similar number of test projects were defined to host the support for the three above-mentioned practices. In total, the different projects use sixteen low-level events that were contributed in advance to the event repository. In addition, the repository includes the six high-level events that were implemented, and eight additional low-level events that we defined, which describe other basic development operations (e.g., refactoring, and creation of additional types). Therefore, the event repository currently contains a total number of thirty event aspects.

The implemented support as well as additional material are available to download at the thesis Homepage [2].

[2] http://ssdl-wiki.cs.technion.ac.il/wiki/index.php/Using_Aspects_to_Support_the_Software_Process

Chapter 6

Conclusions

The HighspectJ framework provides flexible, lightweight, and effective process support integrated into an existing development environment. The framework facilitates automatic generation of classes and aspects from a scenario-based process description, which is based on a separation between event identification and response, and on using a repository of aspects that relate directly to concrete events from the code of the development environment.

Software process support is defined using the framework in an agile fashion. Informal development stories describe the desired way of work, and each story is realized using several example scenarios, either positive or negative. Development stories and scenarios are defined in an iterative fashion, where user feedback is used both to define the initial support, and to refine it during the iterations.

AOP is a powerful and flexible tool for augmenting systems with new functionality. In this work, aspect functionality for a specific domain – the software process – was structured and automated while taking a generative component-based implementation approach. We find this approach promising and encourage the definition of additional domain-specific aspect functionality in this manner.

Complex event sequences have been shown to be useful for software process support. Natural extensions for this work may continue the search for significant software process patterns, and perhaps organize and classify them in dedicated catalogs. The development scenarios were suggested as a vehicle for process understanding and communication, yet this direction was

not investigated enough. In a future work, this potential may be further exploited, perhaps by conducting user experiments that put more emphasis on this matter, e.g., investigate how developers understand the practice from the scenarios, encourage and pay attention to related conversations etc.

In some cases, the aspect code generated by the framework needs manual adjustments. One possible framework enhancement is to completely eliminate manual intervention, probably by making the scenario specification more expressive. In addition, the usability of the framework should be enhanced, for instance, by providing graphical user interface for scenario definition, and improved visualization aids. Despite the limitations of the implemented framework, this work has shown that effective guidance can be added to a development environment, and can be developed using an agile aspect-oriented approach.

Bibliography

[1] Agile manifesto: http://agilemanifesto.org.

[2] Ajdt homepage: http://www.eclipse.org/ajdt.

[3] AspectC++ homepage: http://www.aspectc.org/.

[4] AspectJ project: http://www.eclipse.org/aspectj.

[5] Eclipse homepage: http://www.eclipse.org.

[6] IBM Jazz hompage: http://jazz.net.

[7] Jakob Nielsen's website, http://www.useit.com/.

[8] JBoss AOP, http://jboss.com/products/aop.

[9] Jdt homepage: http://www.eclipse.org/jdt.

[10] JUnit homepage: http://www.junit.org.

[11] Spring AOP, http://www.springframework.org.

[12] ISO 9214-11, Ergonomic requirements for office work with visual display terminals (VDTs), Part 11: Guidance on usability, 1998.

[13] Gregory D. Abowd Alan Dix, Janet E. Finlay. *Human-Computer Interaction (3rd Edition)*. Prentice Hall, 2004.

[14] Jonathan Aldrich. Open modules: A proposal for modular reasoning in aspect-oriented programming. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL: Foundations Of Aspect-Oriented Languages*, pages 7–18, March 2004.

[15] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. *ACM SIGPLAN Notices*, 40(10):345–364, October 2005.

[16] S. Bandinelli, M. Braga, A. Fuggetta, and L. Lavazza. The architecture of SPADE-1 process-centered SEE. In Brian C. Warboys, editor, *Proceedings of the* 3^{rd} *European Workshop on Software Process Technology*, pages 15–30. Lecture Notes in Computer Science Nr. 772, Springer–Verlag, September 1994.

[17] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, Reading, MA, 2000.

[18] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[19] Israel Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the oz environment. In *ICSE*, pages 179–188, 1994.

[20] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, September 1992.

[21] Remi Douence, Pascal Fradet, and Mario Südholt. Trace-based aspects. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 201–217. Addison-Wesley, Boston, 2005.

[22] Yael Dubinsky and Orit Hazzan. Measured test-driven development: Using measures to monitor and control the unit development. *Journal of Computer Science*, (3):335–344, 2007.

[23] Kurt D. Fenstermacher and Mark Ginsburg. A lightweight framework for cross-application user monitoring. *Computer*, March 2002.

[24] Christer Fernström. Process WEAVER: Adding process support to UNIX. In *Proceedings of the Second International Conference on the Software Process*, pages 12–26. IEEE Computer Society Press, February 1993.

[25] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors, *Aspect-Oriented Software Development*, pages 21–35. Addison-Wesley, Boston, 2005.

[26] Uwe Flick. *An Introduction to Qualitative Research*. Sage Publications, Thousand Oaks, California, 1998.

[27] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Object Technology Series. Addison-Wesley, June 1999. With contributions by Kent Beck, John Brant, Willima Opdyke, and Don Roberts.

[28] Alfonso Fuggetta. Software process: a roadmap. In *ICSE - Future of SE Track*, pages 25–34, 2000.

[29] Boby George and Laurie A. Williams. A structured experiment of test-driven development. *Information & Software Technology*, 46(5):337–342, 2004.

[30] William G. Griswold, Kevin Sullivan, Yuanyuan Song, Macneil Shonle, Nishit Tewari, Yuanfang Cai, and Hridesh Rajan. Modular software design with crosscutting interfaces. *IEEE Software*, 23(1):51–60, 2006.

[31] Volker Gruhn. Process-centered software engineering environments, A brief history and future challenges. *Ann. Software Eng*, 14(1-4):363–382, 2002.

[32] Orit Hazzan. The reflective practitioner perspective in software engineering education. *Journal of Systems and Software*, 63(3):161–171, 2002.

[33] Hilbert and Redmiles. Extracting usability information from user interface events. *CSURV: Computing Surveys*, 32, 2000.

[34] Matti A. Hiltunen, François Taïani, and Richard D. Schlichting. Reflections on aspects and configurable protocols. In Robert E. Filman, editor, *AOSD*. ACM, 2006.

[35] Shah Rukh Humayoun, Yael Dubinsky, and Tiziana Catarci. UEMan: A tool to manage user evaluation in development environments. In *ICSE*, pages 551–554. IEEE, 2009.

[36] Humphrey. *Managing the Software Process*. Addison-Wesley, 1989.

[37] Watts S. Humphrey. Using a defined and measured personal software process. *IEEE Software*, 13(3):77–88, May 1996.

[38] Ivory and Hearst. The state of the art in automating usability evaluation of user interfaces. *CSURV: Computing Surveys*, 33, 2001.

[39] Stanley M. Sutton Jr. Aspect-oriented software development and software process. In Mingshu Li, Barry W. Boehm, and Leon J. Osterweil, editors, *ISPW*, volume 3840 of *Lecture Notes in Computer Science*, pages 177–191. Springer, 2005.

[40] G. Junkermann, B. Peuschel, W. Schäfer, and S. Wolf. MERLIN: Supporting cooperation in software development through a knowledge-based environment. In B. Nuseibeh A. Finkelstein, J. Kramer, editor, *Software Process Modelling and Technology*, pages 103–129. John Wiley and Sons, 1994.

[41] G. E. Kaiser. Experience with marvel. In D. E. Perry, editor, *Proceedings of the* 5^{th} *International Software Process Workshop*, pages 82–84, October 1989.

[42] Shmuel Katz. Aspect categories and classes of temporal properties. *Transactions on Aspect-Oriented Software Development*, 3880, 2006.

[43] Andy Kellens, Kim Mens, Johan Brichau, and Kris Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In Dave Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 501–525. Springer, 2006.

[44] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[45] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented

programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[46] Jeff Kramer. Is abstraction the key to computing? *CACM: Communications of the ACM*, 50, 2007.

[47] Jeff Kramer and Orit Hazzan. The role of abstraction in software engineering. *ACM SIGSOFT Software Engineering Notes*, 31(6):38–39, 2006.

[48] Balachander Krishnamurthy and Naser S. Barghouti. Provence: a process visualization and enactment environment. In Ian Sommerville and Manfred Paul, editors, *Proceedings of the Fourth European Software Engineering Conference*, pages 451–465. Lecture Notes in Computer Science Nr. 717, Springer–Verlag, 1993.

[49] Jia kuan Ma, Lei Shi, Ya sha Wang, and Hong Mei. Process aspect: Handling crosscutting concerns during software process improvement. In Qing Wang, Vahid Garousi, Raymond J. Madachy, and Dietmar Pfahl, editors, *ICSP*, volume 5543 of *Lecture Notes in Computer Science*, pages 124–135. Springer, 2009.

[50] George Lakoff and Mark Johnson. *Metaphors We Live By.* Chicago : University of Chicago Press, Chicago, 1980.

[51] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Addison-Wesley Longman Publishing Co., Boston, USA, 2001.

[52] Gerard Meszaros. *XUnit Test Patterns – Refactoring Test Code.* Addison Wesley, June 2007.

[53] Oren Mishali, Yael Dubinsky, and Shmuel Katz. The TDD-Guide training and guidance tool for test-driven development. In Pekka Abrahamsson, Richard Baskerville, Kieran Conboy, Brian Fitzgerald, Lorraine Morgan, and Xiaofeng Wang, editors, *Proceedings of the 9th International Conference on Agile Processes in Software Engineering and Ex-*

treme Programming, XP 2008, volume 9 of *Lecture Notes in Business Information Processing*, pages 63–72. Springer, 2008.

[54] Oren Mishali, Yael Dubinsky, and Itay Maman. Towards IDE support for abstract thinking. In *ROA '08: Proceedings of the 2nd international workshop on The role of abstraction in software engineering*, pages 9–13. ACM, 2008.

[55] Oren Mishali and Shmuel Katz. Using aspects to support the software process: XP over eclipse. In Robert E. Filman, editor, *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 169–179. ACM, March 2006.

[56] Oren Mishali and Shmuel Katz. *Complex Event Processing with Aspect Oriented Programming.* Patent, Submitted to the US patent office, 2009.

[57] Oren Mishali and Shmuel Katz. The HighspectJ framework. In *ACP4IS '09: Proceedings of the 8th workshop on Aspects, components, and patterns for infrastructure software*, pages 19–24, New York, NY, USA, 2009. ACM.

[58] Luis Daniel Benavides Navarro, Mario Südholt, Wim Vanderperren, Bruno De Fraine, and Davy Suvée. Explicitly distributed AOP using AWED. In Robert E. Filman, editor, *AOSD*, pages 51–62. ACM, 2006.

[59] Harold Ossher, William H. Harrison, and Peri L. Tarr. Software engineering tools and environments: a roadmap. In *ICSE - Future of SE Track*, pages 261–277, 2000.

[60] Leon Osterweil. Software processes are software too. In *Proceedings of the Ninth International Conference on Software Engineering*, pages 2–13. IEEE Computer Society Press, 1987.

[61] Martyn A. Ould. CMM and ISO 9001. *Software Process*, 2(4):281–289, December 1996.

[62] R. Q. Reis, C. A. Lima Reis, H. Schlebbe, and D. J. Nunes. Towards an aspect-oriented approach to improve the reusability of software process models. In *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, March 2002.

[63] Steven P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[64] D. A. Schon. *The reflective practitioner*. Harper Collins, 1983.

[65] Ken Schwaber and Mike Beedle. *Agile Software Development with Scrum*. Alan R. Apt, first edition, 2001.

[66] Helen Sharp, Yvonne Rogers, and Jenny Preece. *Interaction Design: Beyond Human-Computer Interaction*. John Wiley & Sons, 2007. 978-0470018668.

[67] Mati Shomrat and Amiram Yehudai. Obvious or not? Regulating architectural decisions using aspect-oriented programming. In Gregor Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 3–9. ACM Press, April 2002.

[68] Davy Suvée and Wim Vanderperren. JAsCo: An aspect-oriented approach tailored for component based software development. In Mehmet Akşit, editor, *Proc. 2nd Int' Conf. on Aspect-Oriented Software Development (AOSD-2003)*, pages 21–29. ACM Press, March 2003.

[69] Yonglei Tao. Capturing user interface events with aspects. In *HCI (4)*, pages 1170–1179, 2007.

[70] A.M. Tarta and G.S. Moldovan. Automatic usability evaluation using aop. *International Conference on Automation, Quality and Testing, Robotics*, 2:84–89, 2006.

[71] Robert Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *Foundations of Software Engineering (FSE)*, pages 159–169. ACM, October 2004.

המסגרת מספקת את הפתרון הטכנולוגי, אך לא מתייחסת לשאלה כיצד להגדיר תמיכת תהליך *איכותית*, כלומר, תמיכה יעילה, נכונה, וגם כזו המתקבלת באהדה יחסית בקרב המפתחים. אנו מציגים שיטה להגדרת תמיכה שכזו, המושפעת מגישה חדשה יחסית לפיתוח תוכנה הנקראת פיתוח אג'ילי (agile). גישת פיתוח זו מעודדת במיוחד פיתוח איטרטיבי (מחזורי), ומדגישה את המרכיב האנושי שבפיתוח בכל מיני מישורים. באופן דומה, אנו מגדירים את התמיכה בתהליך הפיתוח במספר איטרציות, החל מתמיכה בסיסית ופשוטה המשופרת לאורך האיטרציות בעקבות משוב המתקבל מניסויי משתמשים (מפתחי תוכנה).

נערכו מספר פעילויות אשר בהן הוגדרה ו/או מומשה תמיכה בתהליך פיתוח באופן אג'ילי. בפעילות הראשונה, העיקרית שערכנו, פותחה תמיכה לשיטת פיתוח מונחית בדיקות (test-driven development או בקיצור TDD), שיטה המעודדת כתיבה *של* בדיקות עבור הקוד *לפני* שפונקציונליות הקוד עצמה אשר אותה בודקים קיימת. התמיכה פותחה במשך שלוש איטרציות, על בסיס משוב שהתקבל משלושה ניסויי משתמשים בהם השתתפו עשרות רבות של מפתחים (סטודנטים בפקולטה למדעי המחשב בטכניון). בפעילויות נוספות שנערכו, הוגדרו (ולא מומשו) הנחיות לתמיכה בנוהל אשר מעודד חשיבה מופשטת (abstract thinking), ובנוהל אשר קורא לסידור וארגון הקוד תוך כדי הפיתוח (code refactoring). גם בשתי פעילויות אלו, התמיכה הוגדרה בעזרתם *של* סטודנטים בפקולטה.

המסמך מאורגן באופן הבא: לאחר הקדמה, בפרק 2 נסקרות עבודות דומות תוך השוואתן לעבודה שלנו. פרק 3 מציג ודן במסגרת שפיתחנו – HighspectJ. פרק 4 עוסק בהגדרה של תמיכת תהליך איכותית, ובו נסקרות הפעילויות השונות שנערכו. פרק 5 מציג את התמיכה שמומשה בפועל באמצעות המסגרת: בנוסף לתמיכה בשיטת הפיתוח TDD, פותחה תמיכה עבור הנחיות לשילוב קוד (code integration), ועבור הערכת שימושיות של מנשקי משתמש.

מוצר התוכנה ולא מאורעות ברמת קוד המערכת. מאורעות פיתוח אלו מיוחדים בזה שהם מבוטאים באמצעות מונחים ברמת הפשטה (אבסטרקציה) גבוהה יותר מרמת הקוד, וגם שלעתים קרובות הם תלויים בהתרחשות של *מספר* מאורעות בסיסיים יותר.

לדוגמא, נניח נוהל פיתוח הדורש תאימות (סינכרון) בין מסמכי התכן של המערכת לבין הקוד עצמו. עבור נוהל כזה, אחד ממאורעות הפיתוח אותם נרצה לזהות יהיה *'המפתח יוצר מחלקת Java חדשה, ולאחר מכן מעדכן את מסמך התכן המתאים'*. המאורע מתאר התנהגות של המפתח אשר עוקבת אחרי הנוהל (משמע התנהגות חיובית), וכאשר מאורע שכזה יתרחש נרצה למשל לתעד אותו לשם ניתוח עתידי של התהליך, או לספק למפתח משוב חיובי מיידי. המאורע המסוים הזה תלוי בשני מאורעות בסיסיים יותר: יצירה של מחלקת Java, ועדכון של מסמך התכן. כמו כן, שימו לב להבדל בין רמת ההפשטה של מאורע שכזה לבין מאורע קוד טיפוסי כגון הרצה של מתודה מסוימת או השמת ערך חדש למשתנה.

מאורעות-על מהסוג הזה נפוצים בתהליך פיתוח התוכנה, וגם בתחומים אחרים כגון הערכת שימושיות של מנשקי תוכנה, וניהול ובקרה של מערכות תוכנה מבוזרות. אך למרבה הצער, שפת הפיתוח AspectJ, שהיא השפה הפופולרית ביותר כיום לפיתוח מונחה היבטים, איננה מסוגלת לטפל במאורעות שכאלו באופן טבעי. זוהי מגבלה ידועה של השפה, שמנגנון ה-pointcut שלה, המאפשר להגדיר אוסף של מאורעות, איננו מאפשר להגדיר מאורעות אשר תלויים במאורעות אחרים בסיסיים יותר שהתרחשו קודם לכן. קיימות מספר עבודות המתמודדות עם הבעיה בהציען *הרחבה* של שפת AspectJ על ידי מנגנוני שפה חדשים המאפשרים להתייחס למאורעות מהעבר. אנו נותנים פתרון מסוג אחר המבוסס על שפת AspectJ הסטנדרטית ואיננו דורש הרחבה שלה.

הפתרון מתאפשר על ידי מסגרת תוכנה אשר פיתחנו הנקראת HighspectJ, המאפשרת להגדיר ולממש מאורעות-על מורכבים מעל מערכות Java. המאורעות מזוהים באמצעות היבטים מיוחדים הנקראים *היבטי-מאורע*, כאשר הזיהוי מופרד מהתגובה למאורעות, אשר מטופלת על ידי היבטים מסוג אחר הנקראים *היבטי-תגובה*. המסגרת מציעה שירותי חילול קוד (code generation) מקיפים אשר מקלים מאוד על מלאכת הקידוד של ההיבטים השונים, תוך שימוש חוזר בהיבטי-מאורע הנמצאים במחסן ייעודי. יתירה מזאת, המסגרת מאפשרת הגדרה רב-שכבתית של מאורעות, כאשר מאורעות בשכבה עליונה משתמשים במאורעות בשכבה שמתחת, מה שמאפשר לתאר את פעולות המערכת בכמה רמות של הפשטה.

תקציר מורחב

מחקר זה מחבר בין שיטת התכנות החדשה יחסית, תכנות מונחה היבטים, לבין תהליך פיתוח התוכנה. תכנות מונחה היבטים מאפשר להוסיף למערכת תוכנה קיימת יכולות פונקציונליות נוספות בצורה מודולרית, כלומר הפונקציונליות הנוספת מרוכזת במקום אחד במקום להיות מפוזרת בין רכיבים שונים במערכת – דבר המקשה על התחזוקה של המערכת ועל קריאות הקוד. הפונקציונליות מוגדרת בתוך טיפוסי שפה חדשים הנקראים **היבטים** (אספקטים), כאשר מנגנון מיוחד מאפשר את שילובם לתוך מערכת הבסיס. השערת המחקר הבסיסית שלנו היא ששיטת תכנות זו יכולה לתרום בצורה משמעותית לתהליך פיתוח התוכנה.

תהליך פיתוח התוכנה הוא אוסף הכלים, השיטות, והנהלים, המשמשים ליצירת מוצר התוכנה. על מנת להתמודד עם המורכבות ההולכת וגדלה של מערכות התוכנה המפותחות, על תהליך הפיתוח להיות מוגדר מסודר ושיטתי. שיטה נפוצה לשיפור תהליך הפיתוח – אשר גם אנו משתמשים בה – היא הטמעת תמיכה אוטומטית בתהליך בתוך סביבת הפיתוח. על ידי כך, צורת העבודה הרצויה משולבת בכלים עצמם, והפער הבלתי נמנע בין תהליך העבודה הרצוי לבין התהליך בפועל עשוי להצטמצם בצורה משמעותית.

החידוש העיקרי בעבודה זו הנו השימוש בהיבטים על מנת לשלב אל סביבת הפיתוח תמיכה אוטומטית בתהליך. הרעיון מיושם בהקשר של סביבת פיתוח מסוימת, פופולרית מאוד לפיתוח מערכות Java, הנקראת Eclipse. היבטים המכילים תמיכה בשיטות פיתוח שונות מוגדרים ומשולבים אל הקוד של Eclipse, ועל ידי כך מוסיפים לסביבה תמיכה בתהליך אשר לא תוכננה מראש. ההיבטים יכולים למדוד תאימות של התהליך בפועל ביחס לתהליך הרצוי, לספק למפתחי התוכנה הנחייה בצורה של הודעות הניתנות בזמן אמת, לאכוף ביצוע של נהלים ושיטות, ואף לבצע פעולות פיתוח מסוימות בצורה אוטומטית.

המודל לפיו היבטים פועלים על המערכת הוא של מאורע-תגובה (event-action). קיים מנגנון שפה הנקרא *pointcut* המאפשר להיבט להגדיר אוסף של מאורעות שונים בריצת התכנית (למשל, יצירה של אובייקטים שונים, קריאות למתודות). במהלך ריצת המערכת, כאשר אחד מהמאורעות שהוגדרו מתרחש, ההיבט מריץ קטע קוד הנקרא *עצה* (advice) כתגובה. התמיכה מונחית ההיבטים בתהליך שאנו מציעים, גם היא מבוססת על מודל זה של מאורע-תגובה. ברם, פה המאורעות המעניינים הם ברמת תהליך הפיתוח של

"כי ה' יתן חכמה; מפיו, דעת ותבונה" (משלי ב, ו)

המחקר נעשה בהנחיית פרופ' שמואל כ"ץ בפקולטה למדעי המחשב.

ברצוני להודות למנחה שלי, פרופ' שמואל כ"ץ, על התמיכה וההנחיה המקצועית לאורך הדרך. על זה שדאג באופן עקבי לתאם פגישות מועילות, ועל זה שידע תמיד לקחת את ההתלהבות (המופרזת לעיתים) שלי למקומות בטוחים ומעשיים יותר. כמו כן אני מעריך מאוד את דאגתו הכנה וטיפולו בעניינים מנהלתיים רבים.

תודה לדר' יעל דובינסקי על תרומתה למספר חלקים במחקר, במיוחד אלו הקשורים לניסויי המשתמשים שנערכו, ועל דיונים פוריים רבים. תודה גם לדר' איתי ממן ולפרופ' אורית חזן על מעורבותם בפעילות החשיבה המופשטת. ברצוני להודות לשחר דג ולמעבדה לפיתוח מערכות תוכנה (SSDL) על התמיכה הטכנית שניתנה ועל אירוח ניסויי המשתמשים, וכמובן תודה רבה לכל הסטודנטים מהפקולטה שלנו שלקחו חלק בניסויים ובפעילויות השונות.

ברצוני להביע את תודתי העמוקה להורי, אבי יוסף, ואמי מזל, על שתמיד דאגו להדגיש את חשיבות הלימודים ושל רכישת השכלה גבוהה, ועל תמיכתם חסרת הגבולות, הן הרוחנית והן החומרית.

לסיום, ברצוני להודות לאשתי היקרה מיטל, על היותה שותפה אמיתית – ממש עזר כנגדי, על שהזכירה לי תמיד את האיזון הנכון ומה באמת חשוב בחיים, ועל טיפולה המסור והאוהב בבתנו המתוקה תמר, שנולדה זה עתה, במהלך ששת החודשים האחרונים.

אני מודה לטכניון, ולרשת האירופאית למצוינות (AOSD Network of Excellence), על התמיכה הכספית הנדיבה בהשתלמותי.

שימוש בהיבטים לתמיכה בתהליך פיתוח התוכנה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

אורן משלי

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

שבט תש"ע חיפה פברואר 2010

שימוש בהיבטים לתמיכה בתהליך פיתוח התוכנה

אורן משלי