The HighspectJ Framework

Oren Mishali
Department of Computer Science
Technion – Israel Institute of Technology
omishali@cs.technion.ac.il

Shmuel Katz
Department of Computer Science
Technion – Israel Institute of Technology
katz@cs.technion.ac.il

ABSTRACT

AspectJ pointcuts relate to a specific execution point in the program and thus AspectJ is not capable of naturally expressing high-level events that are the culmination of a series of more basic events. Yet, there is a real need for expressing such events when dealing with domains having terminology at a level of abstraction higher than the program's code, e.g., software process support, usability evaluation of user interfaces, or detecting illegal banking practices. Here, we present a framework called HighspectJ that provides a structured AspectJ-based solution for defining and utilizing high-level events. The framework treats an event as a first-class object, separates between the identification and the treatment of the event, and facilitates definition of events in layers, where higher level events are defined in terms of lower level ones. In addition, definition and reuse of high-level events using aspects is facilitated by an event *repository* which contains event building blocks. The framework is described and an example for its utilization for defining software process support is given.

Categories and Subject Descriptors

D.2.10 [**SOFTWARE ENGINEERING**]: Design—*Methodologies*; D.2.11 [**SOFTWARE ENGINEERING**]: Software Architectures—*Patterns*; K.6.3 [**MANAGEMENT OF COMPUTING AND INFORMATION SYSTEMS**]: Software Management—*Software development,Software process*

General Terms

Design, Management

1. INTRODUCTION

In this paper we propose an aspect-oriented event infrastructure framework to facilitate application development. The need for such a framework arose in our research on using the aspect-oriented paradigm to support the software development process [11, 9, 10]. Generally speaking, we define AspectJ aspects which encapsulate the needed process support, and then are woven into the Eclipse IDE itself to achieve the desired effect. Our experience shows that using AspectJ as-is for this purpose is problematic. The root of the problem, which is a known limitation of AspectJ [1, 3], is that AspectJ enables one to define events – in terms of pointcuts – relating to a specific execution point in the program, e.g., a method call or a field assignment. On the other hand, there are many cases where we need to express higher-level events that are the culmination of a series of more basic events.

For instance, one software process practice which we support is Test-Driven Development (TDD) [9]. 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. In user experiments, we found that after creating a JUnit test class, novice TDD developers 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 class and then modifies the test in the editor for $time == T$. No modification of a Java class occurs in the middle.

AspectJ pointcuts are not capable of directly expressing such an event, though this sort of high-level event exists naturally when dealing with domains such as the software process, where the terminology used is at a level of abstraction higher than the base-system's code. Other domains where such high-level events are common are usability evaluation of user interfaces [4, 5], the treatment of non-functional requirements [6], and the management of distributed information systems [8]. The latter work defines a general non-aspect architecture for treating such events in a distributed context known as Complex Event Processing (CEP).

Several proposals have been made to extend AspectJ-like languages with history-based constructs that enable the definition of high-level events (elaborated in [1]). In this paper, rather than suggesting new language features, the problem is tackled using a design approach. Below, a framework called *HighspectJ* is presented that provides a structured AspectJ-based solution for defining and utilizing high-level events. The framework's main design concepts, inspired from the CEP paradigm, treat an event as a first-class object, sep-

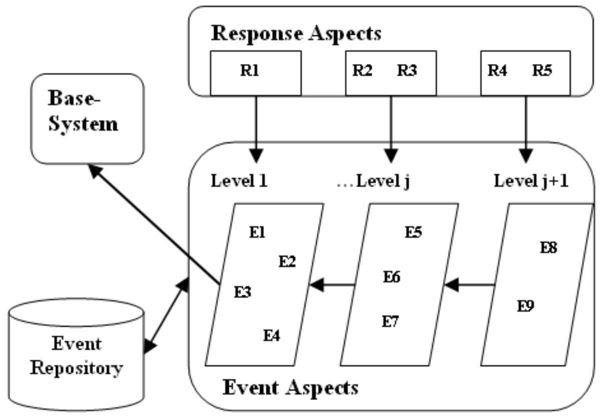


Figure 1: Overview of the framework's architecture

Listing 1: The IEventAspect interface

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

arate between the identification and the treatment of the event, and facilitate a layered definition of events that is natural in many domains [8, 7]. Moreover, definition and reuse of high-level events is facilitated by a key component of the framework – an *event repository* containing event building blocks contributed by different parties.

Providing this framework in an aspect-oriented context adds flexibility, variability, and greater modularity to event-based processing. The framework supports and codifies a high-level design pattern for aspect systems, and we view it as an example of how useful patterns for aspects can be provided with infrastructural support for appropriate applications.

In the following section an overview of the framework is presented. Then, in Section 3 an example is given for the utilization of the framework to define support for the software process domain, and finally we briefly describe some additional examples.

2. OVERVIEW OF THE FRAMEWORK

The HighspectJ framework[1] consists of Java/AspectJ types, components, and coding guidelines, all of which facilitate defining, testing, and utilizing high-level events on top of Java-based systems. This version of the framework is not distributed, since many of the issues of distribution are orthogonal to the concepts presented here.

An overview of the framework's architecture is presented in Figure 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 each event aspect adheres to the *IEventAspect* interface shown in Listing 1. A typical event aspect maintains an internal state (usually fields that aggregate information based on lower-level event aspects), and at a particular point, based on its state, implicitly notifies others on the occurrence of the event that it represents by calling its *event(HJEvent)* method. The type *HJEvent* holds event information such as an id, a time stamp denoting the time when the activity that it represents took place, and event context. Response aspects react to those event activations, and their functionality varies from monitoring of activities to providing real-time notifications. Other types in the framework are presented in the example in the following section.

The framework supports a hierarchical definition of event aspects. As shown in the figure, event aspects can be defined in several layers where the aspects in layer j+1 use aspects in layer j, and thereby represent system activities at a higher level of abstraction. The lowest (first) layer is defined over the base system, and the event aspects in this layer are the only ones that operate directly on system join-points. This hierarchical organization of events is emphasized in [8], and, as explained there, is useful when the operation/usage of the system is naturally viewed at multiple levels of abstraction.

The task of implementing first-layer event aspects is time consuming since it requires expert knowledge of the base system in order to find the appropriate system join-points and to know how to expose the desired context. The framework simplifies the task by providing a central location where such lower-level knowledge can be shared: the *event repository.* The repository provides an interface for the contribution of first-layer event aspects mainly by those who are familiar with the base-system's internals (contribution of higher-level event aspects is also facilitated). The repository can be queried for specific events by *implementers* of higher-level events, which makes the implementation of event aspects in the upper layers relatively straightforward. The implementer does the actual coding of the event aspects according to a specification given by his/her *manager*, and loads them to the repository. Then, the manager also uses the repository to define the response aspects operating on the existing event aspects.

3. SOFTWARE PROCESS EXAMPLE

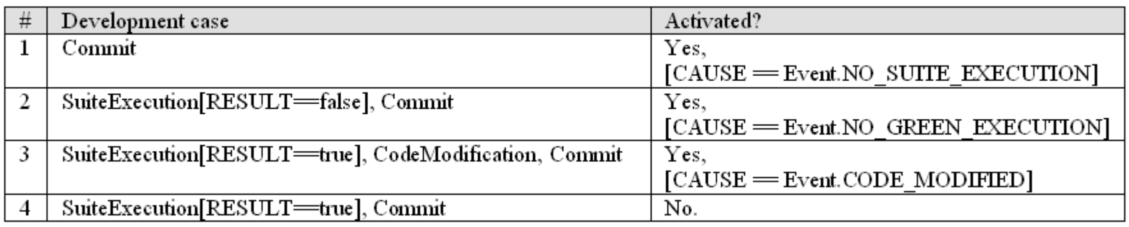
In this section, the HighspectJ framework is utilized to define event-based support for a particular facet of the software development process – the code-integration phase – which involves the integration of code developed locally to a shared code database. After analyzing the problem domain, we focus on the specification and implementation of an event aspect which identifies a particular deviation from the desired integration process.

3.1 Domain Analysis

In a typical software development process, each developer adds or modifies code in a local workspace and once in a while integrates (commits) the code into a shared code database. Usually, the developer cannot just commit the code but should carry out additional operations. Those operations together with the actual commit are denoted as the integration phase.

Following is a typical integration scenario that will serve as a basis for the definition of our integration support:

[1]The framework and an example are available at
`http://www.cs.technion.ac.il/~omishali/HighspectJ`



#	Development case	Activated?
1	Commit	Yes, [CAUSE == Event.NO_SUITE_EXECUTION]
2	SuiteExecution[RESULT==false], Commit	Yes, [CAUSE == Event.NO_GREEN_EXECUTION]
3	SuiteExecution[RESULT==true], CodeModification, Commit	Yes, [CAUSE == Event.CODE_MODIFIED]
4	SuiteExecution[RESULT==true], Commit	No.

Figure 2: Specification of SuiteExecutionProblem event aspect

Listing 2: JUnit test for SuiteExecutionProblem event aspect

```
public class SuiteExecutionProblemTest extends TestCase {
    ...
    public void testDevelopmentCase3() throws InterruptedException {
        SuiteExecution.Event suiteExecutionEvent = suiteExecution.new Event();
        suiteExecutionEvent.RESULT = true;
        suiteExecution.event(suiteExecutionEvent); Thread.sleep(1000);

        codeModification.event(codeModification.new Event()); Thread.sleep(1000);

        commit.event(commit.new Event());

        SuiteExecutionProblem.Event e = (SuiteExecutionProblem.Event)Logger.getEvent();
        assertEquals(SuiteExecutionProblem.Event.CODE_MODIFIED, e.CAUSE);
    }
}
```

- (1) Coding (2) GetToken (3) SynchQuery (4) [Synchronize] (5) MakeTheSuitePass (6) Commit (7) ReturnToken

The scenario is inspired from agile development approaches, such as Extreme Programming [2], that call for the existence of a comprehensive test suite containing all the unit tests, for the execution of the suite prior to the commit, and for the maintenance of an integration token, a kind of a mutex mechanism ensuring that only one developer integrates at a particular time[2].

After performing the needed code changes for completing the task (step 1 in the scenario), the developer requests the integration token; this step can be done in many ways as agreed by the team, for instance using a simple mail protocol. The developer then submits a synchronization query to the code database (step 3), to check whether the local code needs to be updated due to modifications made by other developers. The fourth step is optional and involves the actual synchronization in case code updates exist. Finally, a commit operation is performed (step 6), and the integration token is returned to the team.

3.2 The Suggested HighspectJ Support

The scenario dictates the desired integration process. However as with any software process practice, there is an inevitable gap between the desired process to be followed and the actual behavior of the participants. For instance, the developer may try to commit without first executing the test suite or getting the integration token, or might forget to return the token after the commit takes place. These and other deviations may be identified by event aspects that notify of their occurrence in real-time. Then, corresponding response aspects, operating on the event aspects, may bring the deviations to the attention of the developer, or log them for further analysis and reflection, depending on the management strategy used.

We focus on the specification and implementation of a specific event aspect – *SuiteExecutionProblem* – which identifies an integration problem related to test suite execution. The event aspect is based on the underlying events *SuiteExecution*, *CodeModification*, and *Commit*. These events are themselves event aspects representing basic software development activities of executing the test suite, modifying code within the editor, and checking-in code into the code database, respectively. The events are stored within the event repository and thus relatively easy definition of the higher-level event aspect is facilitated.

An event aspect is specified by a set of lower-level event sequences; each sequence denotes a specific ordering of underlying events on which the event aspect depends. For each event sequence, the specific state of its context variables is described, as well as whether the *event* method should be activated. The specification of *SuiteExecutionProblem* is presented in Figure 2. In our particular software process context, the event sequences are denoted as *development cases*. In the first development case, the developer attempts to commit without a prior execution of the test suite. When such a case is identified, the aspect activates its event while exposing the cause of the problem (via the CAUSE context variable). In the second development case, the suite is indeed executed before the commit but contains failing tests; also here, a problem is reported carrying a different CAUSE. In the third problematic case, the suite is executed

[2]`http://www.xprogramming.com/xpmag/unofficial_faq.htm`

Listing 3: SuiteExecutionProblem event aspect

```
 1 public aspect SuiteExecutionProblem implements IEventAspect {
 2     public class Event extends HJEvent {
 3         ...
 4         public String CAUSE;
 5         public Event(){
 6             setId("org.highspectj.events.SuiteExecutionProblem");
 7         }
 8     }
 9     private Event event = new Event();
10     private SuiteExecution.Event suiteExecution;
11     private CodeModification.Event codeModification;
12 
13     public void event(HJEvent event){
14         event.setTime(new Date());
15         init();
16     }
17     ...
18 
19     after(): execution(* Commit.event(..)){
20         if(suiteExecution == null){
21             event.CAUSE = Event.NO_SUITE_EXECUTION;
22             event(event);
23         } else if(suiteExecution.RESULT == false){
24             event.CAUSE = Event.NO_GREEN_EXECUTION;
25             event(event);
26         } else if (Events.isOrdered(suiteExecution, codeModification)){
27             event.CAUSE = Event.CODE_MODIFIED;
28             event(event);
29         }
30     }
31 }
```

successfully but code modification takes place afterwards, which may indicate a need for an additional suite execution. The fourth development case relates to the expected behavior where the developer conducts a successful suite execution and immediately afterwards commits the code. In that case, no event is triggered.

The HighspectJ framework is designed to facilitate Test-Driven Development (TDD) of event aspects based on their specification, where in each TDD step a JUnit method for a specific development case is first written and then the code within the event aspect that passes the test is developed. The JUnit method for the third development case is presented in Listing 2. The development case is simulated by activating each of its events; it is done by calling the corresponding event aspect's *event(..)* method (e.g., line 6). The event is specified to have a particular context and thus before calling the *event(..)* method, the context is set as appropriate (line 5). Note the default time delay of one second between the events (lines 6,8) which is required in order to query for timing relations between the events, as shown below. After simulating the event sequence, the post-condition is checked. The checking is facilitated by a *Logger* aspect provided by the framework. At the end of the simulation, the logged event is retrieved and checked for the expected context value (lines 12,13). If the event was not activated, the *Logger* returns *null* and the test method fails.

The *SuiteExecutionProblem* event aspect satisfying the above-mentioned specification is presented in Listing 3. As previously noted, like any other event aspect, this one implements the interface *IEventAspect.* It also contains a public inner class called *Event* representing the event that is identified by the event aspect, extending the *HJEvent* class provided by the framework. Any event context exposed by the event aspect should be declared within the *Event* class as public fields. In our example, a single context field is defined, representing the *CAUSE* of the problem and additional corresponding constants, not shown in the listing. Note that this technique allows an event aspect to expose context data that is not defined in the underlying base system or in lower-level events, but which is needed for the task at hand.

In line 13, we see the *event(..)* method which is part of the *IEventAspect* interface, and is called by the event aspect when an occurrence of the event is identified. It calls the *init()* method, also part of the interface, that flushes the event aspect's state. In that way, the event aspect is prepared for a new event cycle; this initialization is required since the event aspect is a singleton and thus the same member fields are used in subsequent event cycles.

The core functionality of the event aspect, which is to monitor underlying events and decide when to call its *event(..)* method is implemented by several advices. The first two advices (not shown in the listing) simply save the lower-level events reported by *SuiteExecution* and *CodeModification.* The third advice (line 19) handles the logic; upon a *Commit* event, and depending on the state of the saved underlying events, the *event(..)* method is called while passing it the *event* field with the appropriate context. Note the use of the *isOrdered(..)* method in line 26; this static utility, defined in the *Events* class of the framework, gets a set of events and returns *true* if the given events are in their chronological order and *false* otherwise. Here it is used to verify whether code modification took place after the test suite execution. The class *Events* provides additional static utilities mostly to determine timing relationships between events.

Note that both the event aspect and its JUnit test contain repeatable and systematic code segments, most of them derived from the specification. In the next release of the HighspectJ framework, it is planned to facilitate automatic code generation of these segments, which will reduce the coding effort and increase code reliability.

Above, an event aspect identifying a common deviation from the described integration process was defined. As shown, the event aspect has a structured specification which is transformed into concrete test cases. The *SuiteExecutionProblem* event aspect may be contributed to the event repository and thus enable its utilization by others. For instance, a manager may use it to define a corresponding response aspect that will monitor activations of the *event(..)* method and take appropriate action according to the cause of the problem. A typical action would be to provide the developer with a notification in real-time, whenever the deviation occurs. For that, the response aspect creates an object of type *Message*, initializes it with a proper textual message and management strategy (e.g., ERROR, WARNING), and uses the *EventViewer* to present it. Besides utilization by response aspects, a contributed event aspect may be used by implementers to define higher-level event aspects. For instance, assuming that we have several event aspects indicating different problems in the process, an event aspect called *CongestedProcessProblems* may be defined. This aspect identifies cases where several process problems happen in a short time interval and helps to recognize sensitive stages in the development process.

4. ADDITIONAL EXAMPLES

As noted, the layered event architecture is appropriate for situations where the terminology of the concern treated by the aspect is far from that of the underlying system. Although we have identified many applications, including so-called *nonfunctional* concerns, where such a design is appropriate, below we describe just two, for reasons of space.

As one nonfunctional concern, we have used the framework to treat usability evaluation of user interfaces. Usability is defined as the extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency, and satisfaction. One common method to evaluate the usability of a given system is automatic evaluation [7], where the usage of the UI by real users is automatically monitored, analyzed, and searched for usability problems. The potential of AOP for automatic usability evaluation is known [13, 12], but an event-based version provides a reusable collection of usability events (both positive and negative), using terminology not relevant to the application itself. For example, using a complex series of buttons and GUI elements instead of a simpler direct possibility for the same task defines a potential *visibility problem* event (the simple solution is hard to find).

Another example is the auditing concern of a banking system. In particular, a layered approach to treating the *money laundering* concern 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 signalling 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.

5. REFERENCES

[1] 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.

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

[3] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In *AOSD*, 2004.

[4] Kurt D. Fenstermacher and Mark Ginsburg. A lightweight framework for cross-application user monitoring. *Computer*, 35(3):51–59, March 2002.

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

[6] Matti A. Hiltunen, François Taïani, and Richard D. Schlichting. Reflections on aspects and configurable protocols. In *AOSD*. ACM, 2006.

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

[8] 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.

[9] Oren Mishali, Yael Dubinsky, and Shmuel Katz. The TDD-guide training and guidance tool for test-driven development. In *XP*, volume 9 of *Lecture Notes in Business Information Processing*, pages 63–72. Springer, 2008.

[10] 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.

[11] Oren Mishali and Shmuel Katz. Using aspects to support the software process: XP over eclipse. In *AOSD*. ACM, 2006.

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

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