Using Aspects to Support the Software Process: XP over Eclipse

Oren Mishali and Shmuel Katz
Department of Computer Science
Technion, Israel Institute of Technology
{omishali, katz} @cs.technion.ac.il

ABSTRACT

Usually, aspects enhance a software product by being composed - or woven - into it. Here, on the other hand, we use aspects to support the software development process itself. The underlying system, i.e., the system to which the aspects are woven, is not the software product but the environment where it is developed. We define aspects to support both software process management and software process modeling. As we show, the aspects can monitor, enforce, or even partially implement compliance with desired development practices. They also provide a basis for a precise description of a software development process. As a case-study, we consider Extreme Programming (XP) and the Eclipse platform. XP is a software development methodology described by underlying values, principles and practices. We present examples of AspectJ aspects that support XP guidelines such as "compose tests before coding" or "provide rapid feedback". Their abstract definitions are shown to be platform independent and correspond to the XP ontology. Their concrete implementation and weaving is connected to Eclipse, an open-source development environment. The design and a prototype implementation of aspects for XP over Eclipse is described.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*Integrated environments*; D.2.9 [**Software Engineering**]: Management—*Software process models, Programming teams, Life cycle*; K.6.3 [**Management of Computing and Information Systems**]: Software Management—*Software development, Software process*

General Terms

Languages, Management

Keywords

Aspects, Software Process, Extreme Programming, Eclipse

1. INTRODUCTION

1.1 Aspects for software process support

Aspect-Oriented Programming (AOP) languages provide constructs to represent cross-cutting concerns as modular units (aspects) and then enable their composition (weaving) into an underlying system. Usually, aspects are used as direct support to the software product under development by being woven into it. Here, on the other hand, aspects are used to support the *Software Process* (SP) and are intended to be woven into the environment where the product is developed. The class of aspects considered, which we call *SP-aspects*, can support both software process management and software process modeling.

The software process is the set of tools, methods, and practices used to produce a software product [13]. In order to master the growing complexity of software development, an improved software process is crucial. For an organization with a defined software process, there is always a gap between the defined process and the actual behavior of the participants, and it is the main objective of software process management to reduce this gap as much as possible. A common way to do this is by integrating software process support into software development environments.

SP-aspects can augment development environments with a variety of management strategies such as measurement of the actual behavior, enforcement of policies and standards, and automation of procedures. Such an integration, if done without AOP, results in less flexible tools and other undesired impacts like scattering and tangling. Moreover, the SP-concerns are especially difficult to implement integrally because of their broader interpretation of crosscutting: multiple development tools and/or process participants are spanned.

SP-aspects are also useful for software process modeling [6]. That is, these aspects can be a basis for an abstract representation of the software process. Such a model, beyond facilitating automated process support, is also intended to promote process understanding and communication among process participants, and to serve as a vehicle for training and education. This novel non-functional role has several implications with respect to the design of SP-aspects, as shown in the rest of the paper.

Since the early days of software development environments, they were implemented with some sort of process support. With most environments, the emphasis was (and is) to provide improved coordination and integration among the different tools in order to facilitate partial automation of tasks, and to maintain consistency among artifacts through different development phases [21]. The major problem of such environments is their inflexible process support which is usually restricted to a small family of software processes [12].

Consequently, process-centered software engineering environments (PCEs) were developed. PCEs are environments that support a wide variety of software processes by considering a description of a software process as a parameter and behaving accordingly. As in SP-aspects, they can support both software process management and modeling. Unfortunately, despite their promising potential, PCEs are not popular, and only a few have been transferred into industrial practice [12, 10]. A detailed description and comparison of such environments is given in Section 4.

1.2 SP-aspects repository

We envision the development of a reusable SP-aspects repository which provides support for different methodologies. The repository keeps its own set of SP-aspects for each methodology and each aspect in the set supports one or more facets of the methodology. SP-aspects are platform-independent and have general abstract definitions that can be realized in a variety of concrete forms. Therefore, the repository provides its users with a mechanism to generate concrete implementations of SP-aspects suitable for their needs.

SP-aspects model a software development methodology based on its *ontology*. Each methodology has its own descriptions and terminology which define an ontology of significant events during system development, as well as entities (artifacts or human participants), activities to be done, and meaningful predicates. In practice, abstract pointcuts and advices represent key events and activities respectively, and they use an independent set of types which represent the entities during the development. The predicates are represented by abstract boolean methods.

A general usage scenario of generating concrete SP-aspects starts with a user specifying the desired facets of the methodology to be supported and the target development platform. Then, relevant SP-aspects are presented, and the user is requested to refine their ontology elements. That is, the user must choose the desired refinements to relevant abstract key events, activities, and predicates. Finally, concrete SP-aspects, which are intended to be woven into the target development platform, are generated.

A key event, for example, an abstract pointcut describing creation of elements that belong to the coding phase, could be refined to (i.e., implemented as) the creation of certain Java packages and classes. Refining the activity part involves choosing a desired management strategy that should be taken when an event occurs. Various management strategies are represented in the repository. These vary from a 'non-interventionist' policy that only monitors and reports violations, to a stricter enforcement policy that forbids activities violating the desired SP rules, to an activist policy that puts conformance activities in the aspect itself.

1.3 Outline

In the rest of this paper, we describe a case-study which is a first step toward the proposed vision. We chose Extreme Programming (XP) as the software development methodology for the case-study and thus define *XP-aspects*, i.e., aspects to support XP. The concrete implementation of the aspects is connected to the Eclipse platform, an open-source extensible development environment. XP-aspects are developed using the AspectJ language [17] and we assume the reader is familiar with it. The following section begins with an overview of XP and then presents several definitions of XP-aspects. Section 3 introduces the Eclipse platform and presents a prototype implementation of XP-aspects over it, along with a discussion of the difficulties that have been encountered. In Section 4 related works are discussed and compared to SP-aspects. Finally, we provide some future research directions, briefly describe some user experience with XP-aspects over Eclipse, and conclude in Section 5.

2. DEFINITION OF XP-ASPECTS

2.1 Extreme Programming (XP)

XP [3], originated by Kent Beck, is supposed to be a fun, scientific, low-risk, and flexible way to develop software by small-to-medium-sized teams in the face of unclear and changing requirements. XP is based on four *values* which serve as its most fundamental guidelines: communication, simplicity, feedback and courage. XP encourages constant and honest communication between team members and customers, and calls for the simplest system that actually works. The approach also emphasizes the importance of concrete feedback to continuously know where the team is.

Since these values are too vague, several more concrete guidelines known as *basic-principles* are derived from them. These guidelines, such as "small initial investments", "rapid feedback", "incremental change", and "honest measurement", are meant to give the team much clearer guidance of how to behave, but they still lack practicability. Hence, XP defines *practices* which are the materialization of the values and principles in the actual development activities. According to Beck, none of these practices is either novel or considered to be perfect, but applying them all together is supposed to create an interplay where the practices complement each other's weaknesses.

XP defines twelve different practices. Most of them are directly related to the development activities, e.g., developing code in pairs (*pair programming*), integrating it often (*continuous integration*), and writing it according to agreed *coding standards*. Other practices have a more global view, e.g., using a *system metaphor* to improve understanding and communication, and delivering a working product each few months (*small releases*). Moreover, planning is also emphasized in a *planning game* where customers provide "user-stories" (descriptions of features), developers estimate their development time, and the scope of the release is determined.

All of the practices, including several not listed here, can be modelled by XP-aspects, explicitly or implicitly. Here, we concentrate on a detailed description of only a few of them. We define XP-aspects to support *test-first* and *collective ownership* practices and the *rapid feedback* principle. Generally, test-first means that testing should drive coding, and collective ownership calls for the involvement of the developers in the whole system and not only a small part of it. Rapid feedback emphasizes the importance of having feedback as soon as possible. More details are given in the relevant following definitions.

2.2 Requirements analysis

Following the repository vision in Subsection 1.2, XP-aspects should have general abstract definitions based on the XP ontology. Hence, their definitions are not restricted to a particular tool or development environment. As an example of a restrictive definition, consider the pointcut *creationOfJUnitTests()*. This pointcut defines the creation of tests that belong to JUnit, which is a common testing framework for Java developers. Of course, XP does not mention JUnit or any other specific testing framework. Hence, renaming the pointcut *creationOfUnitTests()* is much more reasonable since XP does mention the creation of unit-tests during the development.

XP-aspects are defined as abstract AspectJ aspects. As such, they provide only the general shape of the solution without getting into specific implementation details such as join-point specifications. Still, XP-aspects require the use of some context from the development environment where they are operating. To remain general, and to be based only on the XP ontology, XP-aspects do not use any underlying types such as classes or methods directly. Instead, XP-aspects use an independent set of types called *XP-Elements*, also derived from the XP ontology. These XP-Elements are a set of Java interfaces which represent the entities of an XP project. The elements are intended to be exposed by abstract pointcuts and then to be used by the advice part of XP-aspects. XP-Elements are designed according to the "XP way": with the simplest design which meets the current needs and extending or refactoring it when needed.

A partial list of XP-Elements is presented in Listing 1. The interface *XPElement* represents any existing entity in an XP project, and provides the most basic distinction between humans and artifacts. *Artifact* represents an element made by humans and is classified according to the phase to which it belongs. *CodingElement* and *TestingElement* represent elements that belong to the coding and testing phase respectively. *UnitTest* represents any internal (non-customer) test, and finally the interface *PairProgrammers* represents a pair of programmers who develop together.

Besides their functional role, XP-Elements also serve as an important vehicle for the understanding of XP. For example, the methods *getPilotName()* and *getNavigatorName()* in *PairProgrammers* imply that two different roles exist in each pair: pilot and navigator. The pilot "owns" the keyboard and does the actual coding, while the navigator thinks more strategically hence providing a global point of view.

2.3 Test-first example

Test-first (also known as "test-driven development") is one of the core practices of XP. At the beginning of each development iteration, each developer is assigned several tasks derived from user stories. Then, a partner is found and together they start to implement the tasks, where each task is implemented in a series of steps. Working test-first means that in each step a test (unit-test) is first written and only then the pair writes the minimal portion of code that makes it pass that test. Test-first 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. Test-first is not considered to be an easy practice since it requires new and unfamiliar thinking. Steinberg and Palmer state [26]: "testing first is one of the more difficult practices

Listing 1: XP-Elements

```
public interface XPElement {

    public boolean isArtifact();
    public String getName();
    ...
}

public interface Artifact
            extends XPElement {

    public static final int PLANNING = 1;
    public static final int DESIGN = 2;
    public static final int CODING = 3;
    public static final int TESTING = 4;
    ...

    public int getPhase();
    ...
}

public interface CodingElement
            extends Artifact {

    public static final int CLASS = 1;
    public static final int METHOD = 2;
    ...

    public int getKind();
    ...
}

public interface TestingElement
            extends Artifact {

    public static final int UNIT = 1;
    public static final int ACCEPTANCE = 2;
    ...

    public int getKind();
    public boolean isPassed();
    ...
}

public interface UnitTest
        extends TestingElement {
         ...
}

public interface PairProgrammers
        extends XPElement {

    public String getPilotName();
    public String getNavigatorName();
    ...
}
```

Listing 2: Test-first example

```
public abstract aspect TestFirst {

    protected abstract pointcut creationOfUnitTests(UnitTest test);

    protected abstract pointcut creationOfCodingElements(CodingElement element);

    after(UnitTest test): creationOfUnitTests(test){
        existingUnitTests.add(test);
    }

    before(CodingElement element): creationOfCodingElements(element){
        if(!hasUnitTest(element, existingUnitTests)){
            disapproval(element);
        }
    }

    protected abstract boolean hasUnitTest(...);
    protected abstract void disapproval(...);
    ...

    private Collection existingUnitTests;
}
```

to embrace. However, because of its wide-ranging impact, it is also the most important".

The *TestFirst* aspect in Listing 2 provides partial support for test-first by expressing a more restricted policy:

- *Upon creation of a coding-element there should already be a corresponding unit-test.*

TestFirst monitors the creation of *UnitTests* and saves them in *existingUnitTests*. Then, when a *CodingElement* is created, it checks the existence of a corresponding unit-test, and *disapproval(..)* is called if the unit-test does not exist. The definition of *TestFirst* is very general and does not provide an answer to several immediate questions such as:

- What kind of coding-elements are affected by the policy?
- What kind of unit-test should a coding-element have?
- What does *disapproval(..)* do? Does it only monitor or does it enforce?

By being general, *TestFirst* allows many possible implementation options and it is the responsibility of the concrete sub-aspect to choose one. The abstract method *disapproval(..)*, for example, only suggests that it is "not good" to create coding-elements without having corresponding unit-tests. Hence it leaves open options for several different management strategies. As noted earlier, one strategy could be simply to monitor compliance with the practice. Here, *disapproval(..)* would be refined to a method that logs and reports the violation and the pair involved to management. In a more activist strategy, compliance could be enforced, disallowing conduct not consistent with the practice. Then *disapproval(..)* would announce to the pair that no test exists and not allow proceeding with the creation of the coding element. Moreover, in some cases, the aspect itself could have code that applies the practice, removing this obligation from the developer (or at least helping in the task). Here, *disapproval(..)* could itself create an empty unit-test, prompting the user to fill in specific details.

TestFirst does not cover the whole test-first practice but only provides a starting point. For broader coverage it should also treat the changes of coding-elements rather than only their initial creation. By doing that, the entire behavior of the developers may be analyzed to recognize deviations from test-first practice and to draw other important conclusions.

2.4 Collective ownership example

In describing the collective ownership practice, Beck explains [3]: "In XP, everybody takes responsibility for the whole of the system. Not everyone knows every part equally well, although everyone knows something about every part". In practice this means that the involvement of a pair in the project should not be restricted to a specific segment of it. As a result, "expert-only" segments are avoided and thus the project has greater potential to evolve smoothly. The *CollectiveOwnership* aspect in Listing 3 expresses the following policy derived from the collective ownership practice:

- *A pair not highly involved with an artifact should be encouraged to make changes to it.*

This simple aspect monitors change events of different project artifacts, logs them together with the pair that made the change, and then encourages the pair if its involvement in that artifact is limited. This is in contrast to a pair already involved in it. The log which is maintained by the aspect can provide the XP coach (one who manages the team) with valuable knowledge related to the practice. For example, it may be inferred from the log, possibly as additional methods local to this aspect, whether there are any "expert-only" project segments and their location, developers who have knowledge about a specific segment or those who restrict their work to only a few segments. Again, *CollectiveOwnership* does not specify which are the directly monitored project artifacts or what is the meaning of a change, nor does it address the method of encouragement or the definition of "highly involved pairs". As before, these open issues are left for the concrete aspect.

Listing 3: Collective ownership example

```
public abstract aspect CollectiveOwnership {

    protected abstract pointcut changeOfArtifacts(PairProgrammers pair, Artifact artifact);

    after(PairProgrammers pair, Artifact artifact):
        changeOfArtifacts(pair, artifact){
                log.add(pair, artifact);
                if(!highlyInvolved(pair, artifact, log))
                    encourage(pair, artifact);
    }

    protected abstract boolean highlyInvolved(...);
    protected abstract void encourage(...);
    ...

    private Log log;
}
```

Listing 4: Rapid feedback example

```
public abstract aspect RapidFeedback {

    protected abstract pointcut executionOfTestingElements(TestingElement test);

    after(TestingElement test): executionOfTestingElements(test){
        communicate(test);
    }
    ...
    protected abstract void communicate(...);
}
```

2.5 Rapid feedback example

Rapid feedback is one of the basic principles of XP. In general, feedback means to have a response to a certain action. Rapid feedback means to get this response as soon as possible and thus improve learning time. In XP, this principle is expressed in several dimensions. For example, having small releases and short iterations provides the customer with rapid feedback about the team performance and the state of the product, and also feeds the team with the customer reaction.

The *RapidFeedback* aspect in Listing 4 deals with the automation of feedback, hence it makes feedback particularly rapid. *RapidFeedback* reports to the team on significant events that occur during development. By that, team members are provided with crucial knowledge about the current state of the project which may result in “online” decision making and contributes to the involvement of team members in the project. *RapidFeedback* uses an activist management strategy to achieve the XP principle, and the burden of remembering to report events is lifted from the developers.

In XP, both unit-tests and acceptance-tests provide continuous feedback on the current state of the system and *RapidFeedback* provides the team members with the results of those tests. Note that the pointcut *executionOfTestingElements(..)* does not mention which kind of executed tests are monitored. Possible refinements may monitor only those tests answering some criteria, e.g., tests belong to a specific module or following a check-in operation. *RapidFeedback* may be extended to provide the team with more information such as the last integration time, new added tasks or customer feedback.

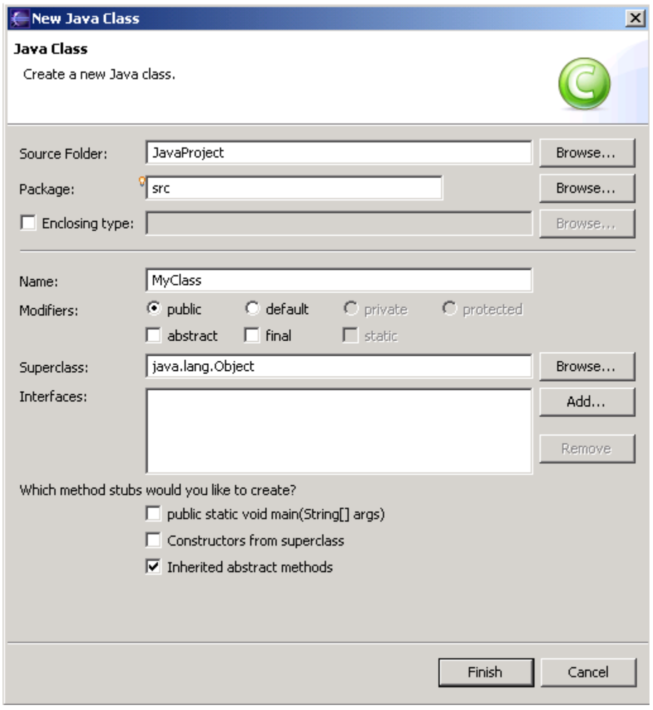
3. XP-ASPECTS OVER ECLIPSE

3.1 The Eclipse platform

The Eclipse platform is an open-source extensible development environment [7]. 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 where usually one of them is 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 (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.

3.2 AJEER

In this paper, XP-aspects are intended to be woven into Eclipse. Since Eclipse is a collection of plug-ins, the weaving is actually into the Eclipse plug-ins. One way to do the weaving is by using AspectJ Development Tools (AJDT) [5], which is an extension that provides Eclipse with support for development with AspectJ. Although usually used with applications, there is no obstacle to using it for plug-ins of the development environment itself or even to AJDT. However,



Java Class

Create a new Java class.

Source Folder: JavaProject

Package: src

☐ Enclosing type:

Name: MyClass

Modifiers: [✓ public] default private protected

☐ abstract ☐ final ☐ static

Superclass: java.lang.Object

Interfaces:

Which method stubs would you like to create?

☐ public static void main(String[] args)

☐ Constructors from superclass

☑ Inherited abstract methods

Figure 1: Class creation wizard

at least up to AJDT 1.3, such an approach means that whenever we want to weave an aspect into a group of plug-ins, or we want to remove or update it, we need to recompile all of them.

Thanks to Martin Lippert, the contributor of the AspectJ-Enabled Eclipse Runtime (AJEER)[20], we have another way to do the weaving which is much smoother. AJEER is an extension that adds support for *load-time weaving* of Eclipse plug-ins. With AJEER, AspectJ aspects are written (using AJDT) in a separate plug-in which extends a provided extension-point. Then, upon Eclipse initialization the aspects are activated and woven into the relevant plug-ins when these plug-ins are loaded. Thus no recompilation is needed. The use of this load-time weaving mechanism makes the removal or update of an aspect very easy: we just need to remove the aspect's plug-in from the system or to change its code. The next time Eclipse is activated, the aspect will, respectively, not be used, or be updated.

With AJEER, new plug-ins which are added dynamically to Eclipse are also affected by the installed aspects, without the need to restart Eclipse. The implementation of AJEER, based on AspectJ, keeps an intensive type-information cache in memory in order to realize the weaving itself. Thus currently using AJEER requires workstations with at least 512MB of RAM. (This memory consumption is supposed to be improved in upcoming versions of AspectJ.) The user experiments described in Section 5 confirm that, given such workstations, the overhead of using SP-aspects during the ongoing use of Eclipse is negligible, and may only be noticed during the initialization of Eclipse or when plug-ins are loaded.

3.3 Implementing TestFirst

The following concrete regulation is derived from the vague test-first policy (Section 2.3) of the *TestFirst* aspect:

- *Upon creation of a Java class there should already be a corresponding JUnit test-case named Test{ClassName}.*

JUnit [14] is a popular framework for writing and executing unit-tests for Java applications, and a JUnit test-case is a container of unit-tests. The idea behind the regulation is that each class in the project should have a JUnit test-case that serves as a "frame" for testing it, and contains the unit-tests for that class. The test-case is also related to the class with a naming convention.

As stated earlier (Subsection 1.2), a concrete implementation of SP-aspect is generated by refining its abstract key events (pointcuts), activities (advices), and predicates (boolean methods). Hence, to implement the above regulation, a user first needs to tell the repository to implement the abstract pointcuts *creationOfCodingElements(..)* and *creationOfUnitTests(..)*, defined in *TestFirst*, with concrete pointcuts corresponding to a creation of a Java class and a JUnit test-case respectively. The second step is to specify a concrete management strategy instead of the abstract *disapproval()*. In our example, we choose a strategy of type enforcement, i.e., the developer will not be able to create the class unless the relevant test-case exists. Finally, the abstract boolean method *hasUnitTest(..)* should be refined to handle the desired naming convention. However, it is important to recall that this is only one particular form to realize the *TestFirst* aspect, and many other forms exist as well.

In Listings 5 and 6 we see two fragments of the definition of the aspect *EclipseTestFirst*, extending *TestFirst*.

Listing 5: EclipseTestFirst (Implementation of pointcuts and methods)

```
protected pointcut creationOfCodingElements(CodingElement element):
    execution(* org.eclipse.jface.wizard.IWizard.performFinish(..))
        && this(element) && within(NewClassCreationWizard);

protected pointcut creationOfUnitTests(UnitTest test):
    execution(* org.eclipse.jface.wizard.IWizard.performFinish(..))
        && this(test) && within(NewTestCaseCreationWizard);

protected void disapproval(CodingElement element){
        enforce("JUnit_TestCase_Named_Test" + element.getName() + "_Does_Not_Exist");
}

protected boolean hasUnitTest(CodingElement element, Collection existingUnitTests){
    return isTestNameExists("Test" + element.getName(), existingUnitTests);
}
...
```

Listing 6: EclipseTestFirst (Inter-type declarations)

```
declare parents: NewClassCreationWizard implements CodingElement;

declare parents: NewTestCaseCreationWizard implements UnitTest;

public int NewClassCreationWizard.getKind(){
    return CodingElement.CLASS;
}

public int NewClassCreationWizard.getPhase(){
    return Artifact.CODING;
}

public String NewClassCreationWizard.getName(){
    return getNameFromWizardPage();
}
...
```

EclipseTestFirst is realized by packing it as an Eclipse plug-in which extends AJEER. When Eclipse is launched, AJEER takes care of weaving it into the relevant plug-ins: JDT and JUnit. In order to implement the abstract pointcuts *creationOfCodingElements(..)* and *creationOfUnitTests(..)* we need to search in Eclipse code for join-points that correspond to creation of a Java class and a JUnit test-case. As a starting point for the search we consider the developer's perspective.

In Eclipse, the most common way for a developer to create a class or a test-case is by using a *wizard*, which is a graphical user interface that guides the developer through a sequenced set of tasks. An example of a wizard for creation of a Java class is presented in Figure 1, and a similar wizard is also used to create a test-case. After filling in the relevant fields, the developer "requests" the creation of the class by pressing the finish button. Then, after some processing, the developer gets feedback that the request is accomplished. This analysis infers that the concrete pointcuts may be based on the event which corresponds to the pressing of the finish button. A related investigation of Eclipse code reveals that both wizards are represented by classes that implement the interface *IWizard*. Furthermore, *IWizard* declares a method called *performFinish(..)* which is invoked right after the developer presses the finish button.

As seen in Listing 5, the method *performFinish(..)* is used in both concrete pointcuts. However, the pointcuts are based on different join-points. The first is based on the creation of a Java class while the second on the creation of a JUnit test-case. The distinction is made by the primitives *this()* and *within()*. In the first pointcut the primitive *this()* captures events where the 'this' object implements *CodingElement*, while in the second it relates to events where 'this' implements *UnitTest*. The same primitive also exposes the context to be used by the advice of the abstract aspect. That is, *element* of type *CodingElement* and *test* of type *UnitTest*, respectively. The primitive *within()* also guarantees that similar but irrelevant methods of super-types are not captured.

Recall that both *CodingElement* and *UnitTest* are not Eclipse built-in types but are XP-Elements (Section 2.2). Hence, we need to explicitly add these interfaces to the relevant classes "behind" the generic *IWizard* interface. As seen in Listing 6, we use AspectJ inter-type declarations to add the interface *CodingElement* to class *NewClassCreationWizard* and the interface *UnitTest* to class *NewTestCaseCreationWizard*. The rest of the code adds to the classes the methods which are defined by the interfaces. In particular, the method *getNameFromWizardPage()* extracts the name of the newly created type from the relevant wizard page which is retrieved from *IWizard*.

The code fragment in Listing 5 also implements the desired management strategy and the predicate. The abstract method *disapproval(..)* is refined by the method *enforce(..)*

which handles the enforcement by displaying an error message and then throwing an exception. The abstract method *hasUnitTest(..)* is implemented to indicate whether a test with the name TestClassName exists.

3.4 Implementation notes

3.4.1 Relating key events to Eclipse

As shown in the previous example, in order to define key events during the development process in the form of concrete pointcuts, we first need to understand how these key events are related to Eclipse. In the example, the event is initiated by pressing a button and then waiting for feedback that the task is accomplished. This simple sort of “press and wait” event is quite common during development and is relatively well-defined. That is to say, it is quite clear when the event starts (right after the pressing and before the task is started) and when it ends (right after the related task is accomplished). Hence, it is reasonable to surround these events with *before* or *after* advice.

But other less defined key events exist as well. For example, consider the event *changeOfArtifacts(..)* in Listing 3, and suppose we deal with change of classes. Although in Eclipse some changes of classes do belong to the “press and wait” family (such as automatic refactoring), most of them are handled by hand using the editor. These “editorial” events are much less trivial for definition and handling. We need to define what a change of a class is, whether it is the addition or change of a single character, an entire string, or perhaps the addition or change of a method. Moreover, for interactive changes to text, there are obvious difficulties in analyzing additions or changes that have not yet been made by the user, so *before* advice may be restricted.

3.4.2 The need for expert knowledge

The most difficult task in implementing XP-aspects (or SP-aspects) is to find the appropriate concrete join-points in Eclipse (or any other development environment) which correspond to the abstract pointcuts. For a join-point to be suitable, it should first satisfy the desired pre and post conditions derived from the abstract pointcut. For example, consider a pointcut that represents creation of certain types (e.g. *creationOfUnitTests(..)*, *creationOfCodingElements(..)*). In this case, an appropriate join-point is one that just *before* its occurrence the type has not been created yet, and right *after* that the type is already created. Indeed, the method *performFinish(..)* in Listing 5 satisfies both of these conditions. Another requirement from an appropriate concrete join-point is that the context we wish to expose will be accessible. For example, a suitable join-point for *executionOfTestingElements(..)* (Listing 4) should provide access to the test results.

Writing abstract XP-aspects in AspectJ is quite easy. But the implementation of these aspects certainly requires expert knowledge of the underlying system and is time consuming. In the general case where the implementors of the aspects do not have the required expert knowledge, this resembles searching for a needle in a haystack. In our case, the fact that Eclipse is open-source, has a supportive development community, and is relatively well-documented, definitely made the search easier but not trivial.

3.4.3 Maintenance of XP-aspects

During the development process, changes to XP-aspects are likely to occur. The first class of changes to consider is due to updates in the software process itself. Here, changes to both abstract XP-aspects and their concrete implementations are possible. However, due to their general definitions, changes in abstract XP-aspects are considered to be less common. Changes in concrete XP-aspects, on the other hand, are much more common and reflect changes in the way that abstract policies are realized. These frequent changes are to be supported by the repository.

Another class of changes is related to the inevitable modifications in the implementation of the underlying environment. XP-aspects depend on Eclipse in two ways. First, they use its services, e.g., *EclipseTestFirst* in the previous example uses Eclipse services to open an error dialog. Such dependencies are stable since the aspects use programmatic interfaces (APIs) supplied by Eclipse plugins and thus a 'contract' is made. However, a second class of dependencies, namely augmenting the environment code, is not (yet) contracted. Most Eclipse plugins do not have an API and even for those that do, their API definitely does not provide all the desired events or expose the relevant context. Hence, XP-aspects advise Eclipse code that is considered 'internal'. That is, this code is not supposed to be used by clients and is unstable, i.e., might be changed in subsequent releases. As a result, in each major release of Eclipse plugins, some sort of review and validation is required for the links to the relevant unstable parts of the aspects.

A similar problem is discussed in [11] which suggests creating a middle layer of aspects called XPIs (no related to XP), that hold the unstable details along with constraints on the advised code. Concrete XP-aspects already serve as a layer that separates the details from the abstract aspects. XPIs do provide some useful design techniques that may ease the maintenance of the aspects. However, there still would be an unstable layer that needs to be reviewed and validated when underlying changes occur. This problem is also addressed by Aldrich who suggests the idea of Open Modules [1], which are modules that have more awareness to AOP. Open Modules extend the familiar API with pointcuts that represent internal events that are semantically important. This solution definitely makes the dependencies between aspects and the underlying system more stable but restricts the power of AOP.

4. RELATED WORK

In this section, SP-aspects are compared with existing approaches. In subsection 4.1 we discuss approaches related to the field of process-centered software engineering environments. Subsection 4.2 evaluates our approach compared to Eclipse-based solutions, and in the final subsection, other works that relate AOP with the software process are considered.

4.1 PCEs/PMLs

In the landmark paper “software processes are software too” [22], Osterweil argues that software processes should be treated like software: they should be specified, designed, implemented, etc. Among others, the paper initiated the development of process centered software engineering environments (PCEs). 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). The process model 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 as stated in Section 1, despite their promising potential, commercially available PCEs are rare.

The 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 (as their name suggests) 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 [8], 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 [18] defines an architecture for process support that preserves the original working environment of the developers. As in SP-aspects, Provence is based on monitoring significant events during development and acting upon them. In practice, a smart file system reports low-level events such as file changes and tool invocations to an event manager. Then, they are filtered and events of interest are passed to a process server which acts according to a defined process model. Provence introduces an interesting architecture but it has several drawbacks. First, it is quite difficult and even impossible to identify significant high-level process events based on low-level monitoring. Second, Provence is mainly an observer and its process support is limited to monitoring and very partial automation. Finally, using Provence depends on several components such as a smart file system, an event manager, and a process server.

SP-aspects are woven into existing development tools and thus, like Provence, they preserve the original working environment. Unlike Provence, SP-aspects are based on high-level events (join-points) which are closely related to the development activities. Furthermore, since SP-aspects act directly on the tools, they can include improved process support beyond monitoring, i.e., enforcement and enhanced automation. SP-aspects pose one critical demand on the underlying environment: it should 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 provides 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 [16, 4] and Merlin [15] 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 [2], 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 [24]), the level of integration of each tool is predefined by the tool. SP-aspects, on the other hand, capture events 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, in SP-aspects, manipulation of the tools is only possible if the tools have an API. Otherwise, monitoring and enforcement is still possible, but the level of automation support decreases.

A PML, beyond facilitating automated support, is an important vehicle for process understanding and communication between process participants. Curtis et al. [6] pointed out that PMLs cannot be used if they cannot be understood. They also mentioned that facilitating human understanding has received less attention from the research community than has machine automation. PMLs have various modes of expression such as programming languages based PMLs (APPL/A [27]), petri-net based (SPADE,Process Weaver), and rule-based (Marvel, Merlin). However, eight years after the above remark of Curtis et al., Fuggetta stated: "existing PMLs are complex, extremely sophisticated, and strongly oriented toward detailed modeling of processes" [10]. We believe that aspects are potentially more comprehensible than existing modeling paradigms, especially for developers familiar with the tools of the underlying environment, with Java, and with aspects.

Still, a modeling notation cannot by itself automatically provide improved comprehensibility. To do that, the process model should be defined considering also *cognitive* techniques. Some of these techniques have been used in our application of SP-aspects to the XP paradigm. First, the approach encourages partial (abstract) modeling of the process that can hide unnecessary detail, to be revealed only when needed. Second, the natural terminology of the paradigm can be easily incorporated, and third, the flexibility of the approach allows easy adjustments and reevaluation. A more detailed analysis of such techniques is out of the scope of this paper.

4.2 Eclipse-based solutions

Recall that plugins may introduce extension points to be used by other plugins. Potentially, a plugin may introduce extension points that correspond to important events during its execution, e.g., JUnit introduces an extension point that corresponds to an execution of a unit-test. Other plugins, by extending it, may monitor test executions. The Eclipse core and JDT also offer an expressive API to be used by other plugins. For example, they offer an API to register change-events on different resources. Basically, these APIs may be used to provide some sort of process support. However, in practice extension-points of the type described above are rare, and most of the plugins do not provide an API at all. Even if they did provide extension-points or APIs, it would be limited, since it is impossible to foresee all the places where process support is needed. Furthermore, such approaches do not provide a coherent and uniform process model.

The notion of software process is gaining more and more importance. The Eclipse Process Framework project [9], which is in its very early stages, aims to provide an architecture and web-based tools allowing to incorporate and exchange software process related knowledge, emphasizing agile software development. Obviously, SP-aspects related knowledge may also be incorporated, including the SP-aspects repository itself.

4.3 AOP and the software process

The first work to connect AOP to the area of software process modeling was [23]. Aspect-oriented concepts were proposed to complement and to assist the design of existing process modeling languages. The idea is investigated in the context of a specific PCE and PML called APSEE. The main difference from our work is that we use aspects as the process model *itself* acting directly on the development environment, whereas in the cited work AOP is used to complement an existing PML, which still requires the involvement of a special-purpose environment.

In [25], 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 seen in this paper, the problem they identify can be solved.

5. CONCLUSIONS

Aspects for XP over Eclipse are only one case study for applying aspects to aid in a development process. Another prime candidate for such a treatment is the Rational Unified Process (RUP) [19]. RUP is both a collection of development tools emphasizing UML design, and guidelines for using them. It seems natural to encapsulate the guidelines into a collection of RUP-aspects, along with a variety of enforcement strategies.

As noted, connecting the abstract pointcuts and entities to the specific join-points and types of the development environment is a non-trivial task. It should be conducted by a domain expert on the development environment implementation. However, it is a one-time activity for each (major version of a) development environment, with updates needed only as new abstract pointcuts and entities are defined. These concrete connections are hidden from the users, namely managers defining specific aspects to be applied and a strategy in terms of the abstract entities, and developers working in a team according to a given methodology.

Aspects of the type presented here for XP have been implemented as described over Eclipse and construction of a fuller repository for XP has begun. Preliminary experiments have been made with a number of users who were requested to perform several tasks using the Eclipse development environment and then to fill in a survey. Each task was performed twice, with and without XP-aspects. For example, the users were requested to create a specific Java project layout (i.e., packages and classes) including and excluding the *TestFirst* aspect. In the survey, they were asked about the differences between the scenarios regarding performance, about their understanding of the augmented policies, and on the advantages and disadvantages of the implementation.

The results verify that (due to the use of AJEER) there is noticeable delay in the initialization of Eclipse and when Eclipse plug-ins are loaded. However there is negligible overhead in the ongoing use of the Eclipse tools after they are loaded. The aspects indeed applied the desired strategy. However, the users tended to object to the enforcement policy of the *TestFirst* aspect that simply forbids continuing if a test class has not been created before a coding class and found it too restrictive. A more sophisticated version of enforcement could allow the user to override the policy occasionally, with a short justification reported to management. In general, either monitoring or activist policies (where the aspect takes over part of the task itself) were felt to be more user-friendly.

Special-purpose environments with Process Modeling Languages have not gained general acceptability and are not widely used. Yet there is a real need for greater support for the practices of any software development process desired by management. Bringing the flexibility, modularity, and reusability of aspects to bear on the software development process as applied in standard development environments has the potential of being one of the future major application areas for AOSD.

Acknowledgements

The authors thank Martin Lippert for his help with AJEER. This work is partially supported by the EU Network of Excellence AOSD-Europe.

6. REFERENCES

[1] 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.

[2] S. Bandinelli, M. Braga, A. Fugetta, and L. Lavazza. The architecture of SPADE-1 process-centered SEE. In B. Warboys, editor, *Software Process Technology - Proceedings of the 3^{rd} European Software Process Modeling Workshop*, pages 15–30, Villard de Lans, France, 1994. Springer.

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

[4] Israel Z. Ben-Shaul and Gail E. Kaiser. A paradigm for decentralized process modeling and its realization in the Oz environment. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 179–188. IEEE Computer Society Press, May 1994.

[5] Andy Clement, Adrian Colyer, and Mik Kersten. Aspect-oriented programming with AJDT. In Jan Hannemann, Ruzanna Chitchyan, and Awais Rashid, editors, *Analysis of Aspect-Oriented Software (ECOOP 2003)*, July 2003.

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

[7] Eclipse. http://www.eclipse.org/.

[8] 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.

[9] Eclipse Process Framework. http://www.eclipse.org/epf/.

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

[11] 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.

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

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

[14] JUnit. http://www.junit.org.

[15] 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.

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

[17] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersen, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Berlin, Heidelberg, and New York, 2001. Springer-Verlag.

[18] 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.

[19] Philippe Kruchten. *The Rational Unified Process.* Addison-Wesley, third edition, 2004.

[20] Martin Lippert. AJEER: an AspectJ-Enabled Eclipse Runtime. In *OOPSLA Companion*, pages 23–24, 2004.

[21] 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.

[22] 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.

[23] 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 Awais Rashid, Bedir Tekinerdoğan, Ana Moreira, Joao Araujo, Jeff Gray, Jan Gerben Wijnstra, and Paul Clements, editors, *Workshop on Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design (AOSD-2002)*, March 2002.

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

[25] 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.

[26] Daniel H. Steinberg and Daniel W. Palmer. *Extreme Software Engineering: A Hands-On Approach.* Pearson/Prentice Hall, 2004.

[27] S. Sutton, D. Heimbigner, and L. Osterweil. APPL/A: A language for software process programming. *ACM Transactions on Software Engineering and Methodology*, 4(3):221–286, July 1995.