Expert Systems with Applications 37 (2010) 6675–6681

Contents lists available at ScienceDirect

Expert Systems with Applications

journal homepage: www.elsevier.com/locate/eswa

TMiner aspects: Crosscutting concerns in the TMiner component-based data mining framework

Fernando Berzal *, Francisco J. Cortijo, Aída Jiménez

Department of Computer Science and Artificial Intelligence, ETSIIT, University of Granada, Granada 18071, Spain

ARTICLE INFO

Keywords:
Data mining
Component-based systems
Aspect-oriented programming
Crosscutting concerns
Design patterns
Performance monitoring

ABSTRACT

TMiner (Berzal, Cubero, & Jiménez, 2009) is a component-based data mining framework that has been designed to support the whole KDD process and facilitate the implementation of complex data mining scenarios. This paper shows how aspect-oriented programming techniques support some tasks whose implementation using conventional object-oriented programming would be extremely time-consuming and error-prone. In particular, we have successfully employed aspects in TMiner to evaluate and monitor the I/O performance of alternative data mining techniques. Without having to modify the source code of the system under analysis, aspects provide an unintrusive mechanism to perform this kind of performance analysis. In fact, aspects let us probe a system implementation so that we can identify potential bottlenecks, detect redundant computations, and characterize system behavior + lessons learned during the development of TMiner.

1. Introduction

All programming methodologies provide some kind of support for separation of concerns, which entails breaking down a program into distinct parts that overlap in functionality as little as possible. The structured and object-oriented programming paradigms resort to procedures and classes, respectively, to encapsulate concerns into single entities and thus achieve some separation of concerns. However, some concerns defy these forms of encapsulation and lead to tangled, difficult-to-maintain code, since they cut across multiple modules in a program. Aspect-oriented programming overcomes this problem by enabling developers to express these crosscutting concerns separately (Kiczales et al., 1997).

In this paper, we employ aspect-oriented software development techniques for solving a common problem programmers must face in the development of complex systems. In particular, we describe how aspects can be woven within a component-based data mining framework in order to support the fine-grained performance evaluation of data mining techniques.

Aspects let developers dig into their system at leisure. Since aspects provide an unintrusive way to tuck probes into their system, developers do not have to tweak their underlying system implementation for enabling system monitoring. As keen observers, they can study system performance without inadvertently introducing subtle errors nor degrading actual system performance in a production environment (aspects can easily be removed once the performance evaluation has taken place).

Our paper is organized as follows. Section 2 introduces some of the fundamental concepts and terms behind aspect-oriented software development, including a subsection that describes how crosscutting concerns, or aspects, can be specified using the AspectJ extension to the Java programming language. Section 3 describes the component model and the architectural design of the data mining framework we have fitted with aspects to study the implementation of crosscutting concerns in complex systems. Section 4 presents a case study on the evaluation of the I/O performance of some well-known data mining techniques. Finally, Section 5 concludes our paper by summarizing the results of our study.

2. Background

Aspect-oriented programming (AOP), in particular, and aspect-oriented software development (AOSD), in general, has been identified as a promising area of research in programming languages and software engineering (Kiczales, 1996). AOSD techniques attempt to improve system modularization by the explicit identification of crosscutting concerns.

Crosscutting concerns are aspects of a program which affect, or crosscut, other concerns. These concerns have a clear purpose, yet they often cannot be cleanly decomposed from the rest of the system, and usually result in tangled code that is notoriously hard to maintain.

* Corresponding author.
E-mail addresses: fberzal@decsai.ugr.es (F. Berzal), cb@decsai.ugr.es (F.J. Cortijo), aidajm@decsai.ugr.es (A. Jiménez).

0957-4174/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.eswa.2010.03.052

Logging offers the quintessential example of a crosscutting concern, since a logging strategy necessarily affects every single logged part of the system. Logging thereby crosscuts all logged system modules (classes and methods in an object-oriented implementation).

In general terms, a software system is the realization of a set of concerns. One of the central premises of object-oriented programming is that each concern should be implemented as a separate module. However, there are certain system-wide concerns, such as logging, security, performance, and so forth, that often need to be addressed in many, if not all of the modules. Consequently, the code to handle these system-wide concerns may be mixed in with the core logic of a huge number of modules, resulting in lower productivity, poor quality, and systems that are hard to evolve.

Hence, the implementation of such concerns using conventional programming techniques introduces some problems, since it introduces redundant code (i.e. the same fragment of code typically appears in many different places). This redundant code makes it difficult to reason about the (non-explicit) structure of software systems. It also makes software more difficult to change because developers have to find all the scattered code that might be involved and they must ensure that changes are consistently applied.

AOSD builds on the object-oriented paradigm and streamlines complex systems development without sacrificing flexibility or scalability (Filman, Elrad, Clarke, & Aksit, 2004).

Some key AOSD terms include:

- *Crosscutting concerns*: Those design decisions whose implementation is scattered throughout the code, resulting in tangled code that is excessively difficult to develop and maintain (Kiczales et al., 1997).
- *Advice*: The additional code that you want to apply to your existing implementation at different places.
- *Point-cut*: The point of execution in the application at which a crosscutting concern needs to be applied.
- *Aspect*: The combination of the point-cut and the advice.

Aspect-oriented programming addresses those situation when neither procedural nor object-oriented programming techniques are sufficient to clearly capture some of the important design decisions the program must implement. AOP makes it possible to clearly express programs involving such aspects, including appropriate isolation, composition and reuse of the aspect code (Kiczales et al., 1997). AOP allows design and code to be structured to reflect the way developers want to think about the system (Elrad, Aksit, Kiczales, Lieberherr, & Ossher, 2001).

2.1. Specifying crosscutting concerns with AspectJ

The main idea behind AOP is therefore to capture the structure of crosscutting concerns explicitly, since these concerns are inherent to complex software systems but their implementation using conventional programming techniques leads to poorly-structured software. AOP languages provide a way to specify such concerns in a well-modularized way.

Gregor Kiczales started and led the Xerox PARC team that eventually developed AspectJ (Kiczales et al., 2001). AspectJ is an aspect-oriented extension for the Java programming language. AspectJ is available as an Eclipse Foundation open-source project, and it has become the de facto standard for AOP. By using Java as the base language and creating a final system that is compatible with Java byte code specification, AspectJ passes on all the benefits of Java. The use of Java as the base language also makes AspectJ a relatively easy language to learn.

All AOP languages include some constructs that encapsulate crosscutting concerns in one place. The difference between AOP languages lies in the constructs they provide for modularizing the system-wide concerns. AspectJ encapsulates them in a special class, an aspect, which is declared as a Java class using the `aspect` keyword:

```
public aspect DatasetScan {
// ...aspect implementation details...
}
```

Aspects in AspectJ define crosscutting types comprised of advices, user-defined pointcuts, and the usual field, constructor and method declarations of a Java class.

Pointcuts pick out sets of join points and exposes data from the execution context of those join points. Those join points are defined in terms of "points in the execution" of Java programs. For instance, the following pointcut.

```
pointcut move ():
  call (void Figure.setX (int))
|| call (void Figure.setY (int));
```

picks out each join point that is a call to `setX` or `setY`, the two methods that can be used to move a `Figure` in our system.

An advice defines the additional action to take at join points in a pointcut. It brings together pointcuts (join points) and code (to be run at each of those join points). AspectJ has several different kinds of advice:

```
before (): move () {
  System.out.println ("about to move");
}
after (): move () {
  System.out.println ("just moved");
}
```

In the first example, the `before` advice runs as a join point is reached, before the program proceeds with the join point. In the second one, the `after` advice on a particular join point runs after the program proceeds with that join point, just before control is returned to the caller.

AspectJ aspects can therefore alter the behavior of the base code (the non-aspect part of a program) by applying advice (additional behavior) at various join points specified by pointcuts. For those already familiar with current relational database management systems, we could say that AspectJ provides for object-oriented programs what triggers do for relational databases.

The complete details of the AspectJ language are covered in several programming handbooks (Laddad, 2003; Gradecki & Lesiecki, 2003). In those books, interested readers can find complete AspectJ solutions to implement crosscutting concerns such as logging, policy enforcement, resource pooling, business rules, thread-safety, authentication and authorization, as well as transaction management.

2.2. AOSD State of the Art

AOSD has drawn the attention of many researchers and it has also found early adopters in middleware products (Colyer, Greenfield, Jacobson, Kiczales, & Thomas, 2005), such as some commercial Java application servers.

Some researchers have turned their attention to earlier phases of the software lifecycle. For instance, Ivar Jacobson believes that use-case driven development and AOP complement each other very well: as *early aspects*, use cases are key to effectively separate concerns (Jacobson & Ng, 2004). This point of view has led to recent research on aspect-oriented requirements engineering, or AORE (Chitchyan, Rashid, Rayson, & Waters, 2007; Moreira, Rashid, & Araujo, 2005).