In *Proceedings of the 22nd International Conference on Software Maintenance, pp. 14–23.*
IEEE Computer Society Press, September 2006.

Managing Concern Interfaces

Jean-Sébastien Boulanger and Martin P. Robillard
School of Computer Science
McGill University, Canada
{*jboula2, martin*}*@cs.mcgill.ca*

Abstract

Programming languages provide various mechanisms to support information hiding. One problem with information hiding, however, is that providing a stable interface behind which to hide implementation details involves fixing in advance the services offered through the interface. We introduce a flexible approach to define and manage interfaces to achieve separation of concerns in evolving software. Our approach involves explicitly specifying interface and implementation classes for individual concerns, and automatically classifying implementation classes based on their relation to the interface. Our approach is supported by JMantlet, a tool that provides advanced interface management within an integrated development environment. We report on a case study of a large system that provides evidence that flexible interface management is desirable and adequately supported by our approach.

1. Introduction

Information hiding [13] is an important principle of software engineering. In its general form, this principle dictates that design decisions likely to change should be hidden such that their implementation can evolve without affecting the rest of the system. Information hiding motivates the separation of a program into concerns that each represent a particular concept or purpose that can be thought of or changed separately. In practice, programming languages support separation of concerns and information hiding in a number of ways: abstract data types, object encapsulation, application programming interfaces (APIs), modules, Java packages, C# assemblies, and others. These mechanisms provide a *stable interface* that hides a concern's implementation from the rest of the system and allows the implementation of a concern to evolve without directly affecting the rest of the program that interacts with that concern. The use of language mechanisms to enforce separation of concerns has the advantage that access to a concern's implementation can be automatically prevented by the compiler (e.g., the Java compiler disallows access to `private` members outside of the declaring class).

One problem with information hiding is that providing a stable interface behind which to hide implementation details involves fixing in advance the services offered through the interface. In practice, however, the services offered by an interface in one context might not be sufficient or appropriate in a different context. In such cases, developers are faced with the decision of either complying with the interface (possibly giving up on a service), or accessing the implementation (which introduces coupling and may lead to maintenance problems). Design idioms, such as interface segregation, can help mitigate this problem, but have limitations of their own (e.g., proliferation of interfaces, versioning problems).

In this paper we present a tool-based approach to information hiding. Our approach provides support to define a concern model that can restrict or permit access to the implementation of a service based on the context (project) in which it is used. Our approach also provides a flexible mechanism to manage the evolution of concern interfaces and provides information that can help the programmer reason about the interface of concerns. To conduct the initial experimentation and validation of our approach, we implemented JMantlet, an Eclipse plug-in for the Java programming language. As an initial validation of our approach, we conducted a study of the implementation of a concern in the JBoss application server.

The rest of this paper is structured as follows. In Section 2 we provide a detailed scenario that motivates our approach in the context of a Java program. In Section 3, we describe the features of JMantlet, a short usage scenario, and an overview of its implementation. We presents our study in Section 4. We discuss related work in Section 5, and conclude in the Section 6.

2. Motivation

Let us consider a software maintenance scenario involving high-level concerns. In this scenario, a developer is working on a Java application that uses an email service abstracted as the standard JavaMail™ API [17]. As for most

of the APIs in the J2EE architecture, [1] the actual implementation of the JavaMail API is left to third parties. For example, the GNU Classpath project[2] provides a free implementation of multiple transport protocols (e.g., IMAP, SMTP, POP3) that are organized in multiple packages. Figure 1 shows the seven packages of GNU's JavaMail implementation followed by the four packages of the standard API.

```
gnu.mail.providers.imap
gnu.mail.providers.maildir
gnu.mail.providers.mbox
gnu.mail.providers.nntp
gnu.mail.providers.pop3
gnu.mail.providers.smtp
gnu.mail.util
javax.mail
javax.mail.event
javax.mail.internet
javax.mail.search
```

Figure 1. Packages of GNU's JavaMail 1.0

Let us assume that the developer chooses the GNU implementation for the application and adds the GNU class library to the project's classpath. At this point, the developer will be able to refer to the GNU implementation classes of the JavaMail API from within the project. However, if the application code directly refers to classes in the `gnu.mail.*` packages, different parts of the system will become coupled to the implementation of JavaMail. If GNU classes change (i.e., are updated to a later release), the developer may have to modify and re-test multiple code locations. The obvious solution to this problem is to refer only to the JavaMail API (i.e., classes in the `javax.mail.*` packages). This strategy allows the application to evolve independently from the JavaMail implementation. The core of the J2EE architecture is based on this idea. Unfortunately, although it sounds simple in theory, avoiding references to implementation classes can be problematic in practice for at least two reasons: there is no mechanism to enforce module boundaries, and standard interfaces are inflexible.

No mechanism to enforce module boundaries. Even if there exists an explicit guideline not to refer to GNU classes except to instantiate objects, this design decision cannot be enforced by the compiler since many of the GNU classes have a `public` access qualification. This visibility is necessary to instantiate the classes, but also to allow code reuse across implementation packages. As a result, every present and future developer involved in the project must ensure that the implementation classes are not referred to.

[1] Java 2 Platform, Enterprise Edition. java.sun.com/javaee

[2] www.gnu.org/software/classpath/

Inflexible standard interfaces. A second problem is that although the JavaMail API provides a very broad interface to a general mail system, the implementation of a specific protocol can provide additional features that are not accessible through the JavaMail API. For example, in the GNU implementation of the IMAP store there is a method to retrieve the disk storage quota of a mailbox. To make use of that functionality in the application, the developer would have to refer to the implementation class. In practice, this may be a necessary and cost-effective decision.

Concern Interface Management. As this scenario illustrates, high-level concerns are not always perfectly abstracted by an API. In our case we have a concern, MAIL, that corresponds to the use of an email service. Ideally, the interface to this concern is the JavaMail API and its implementation is the corresponding set of GNU packages. However, our development scenario required the *enforcement* of this concern's interface in most cases, and its *extension* in a few cases. To address these interacting requirements, we propose a technique and tool to support the management of concern interfaces in a structured way.

3. Tool Support

JMantlet [3] is a tool to manage and enforce concern interfaces in a program. JMantlet is implemented as a plug-in for the Eclipse Platform, [4] an integrated software development environment supporting the addition of functionalities through a plug-in architecture. In this section, we describe the features of JMantlet, illustrate the benefits of the tool with a usage scenario, and present an overview of the implementation.

3.1. Features

Concern Configuration File. JMantlet provides a mechanism to define a concern model that is independent from the source code and valid in the context of an Eclipse workspace. The *concern configuration file* is used to declare the interface and the implementation of the concerns of a system in XML format (see Figure 4). A concern configuration file can declare multiple concerns (with the `<concern>` XML tag) . Each concern has a *name* attribute that is used in the display of the model. A concern's interface (specified with the `<interface>` XML tag) consists of a set of types that is defined by matching patterns on type names using the same semantics as Eclipse's Java Search. [5] For example, in Figure 4, the interface of

[3] www.cs.mcgill.ca/∼jboula2/jmantlet

[4] www.eclipse.org

[5] help.eclipse.org/help31/index.jsp

the defined concern is the union of the types in the packages `javax.mail`, `javax.mail.event`, `javax.mail.internet`, and `javax.mail.search`. A concern's implementation (specified with the `<implementation>` XML tag) is defined using the same approach. If the intersection of the interface and the implementation is not empty, the interface has precedence over the implementation.

Concern Model Viewer. The *concern model viewer* of JMantlet displays the concern model generated from a concern configuration file (see Figure 2). The viewer displays each concern in the model and lists the classes that are part of the interface and implementation, respectively. For implementation classes, the viewer shows additional information such as the nature of the relationships with the interface (i.e., *implements*, *extends*) and the *category* of the implementation class.

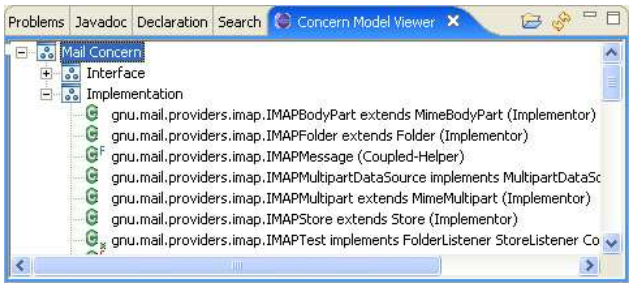

Figure 2. JMantlet's Concern Model Viewer

Our approach makes a classification of implementation classes that can help the programmer reason about the dependencies between a program and a concern. It can also help the programmer decide whether a given class should be part of a concern's interface. Our classification defines three categories of concern implementation classes:

1. **Implementors.** *Implementors* are classes that implement an interface or directly extend a class of a concern's interface. Our assumption is that the implementors realize key design decisions for a concern that should generally be hidden from clients [13].

2. **Coupled-Helpers.** *Coupled-helpers* are other classes that contribute to a concern's implementation and that reference the implementors or other coupled-helpers. Our assumption is that coupled-helpers know about the key design decisions realized by implementors so they should also generally be hidden from clients.

3. **Decoupled-Helpers.** *Decoupled-helpers* are classes that only rely on the concern interface. Technically, decoupled-helpers could be reused across multiple implementation of a concern without any modification. If needed, the programmer could even consider adding a decoupled-helper to the concern's interface so that it can be used by the rest of the program.

References Alerts. JMantlet alerts the programmer of illegal references to a concern's implementation classes similarly to a compilation error. JMantlet creates an *error marker* that is displayed in the Eclipse editor, next to the reference (see Figure 3). The error is also added to the standard list of problems (i.e., *Problems view*). This mechanism ensures that references to implementation classes, from classes other than classes in the concern's implementation, are prohibited within the Eclipse workspace.

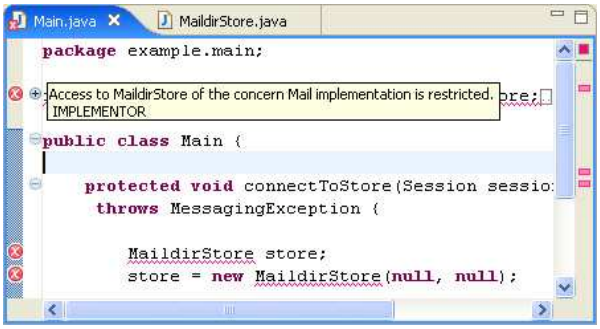

Figure 3. Error Alert of an Access to the Implementation

3.2. Usage Scenario

We reuse the JavaMail example of Section 2 to describe a typical use of JMantlet.

Defining the Concern Interface and Implementation. The developer first defines the interface and implementation of MAIL by editing the configuration file (see Figure 4). In our example the concern's interface is the set of classes defined in the four packages of the JavaMail API and the concern's implementation is the set of classes defined in the seven packages of the GNU implementation of JavaMail. To enable the instantiation of the GNU implementation classes, the developer creates a class containing factory methods [4] that knows how to instantiate the types of the GNU implementation (i.e., `example.mail.Factory`) and adds it to the interface of the mail concern (Figure 5).

Hiding the Concern's Implementation. The developer loads the configuration file into the concern model viewer of JMantlet, which then displays the types matching the concern's interface and implementation definitions, respectively. It also displays the category (i.e., implementor,

```
<concerns>
  <concern name="Mail">
    <interface>
      <type pattern="javax.mail.*"/>
      <type pattern="javax.mail.event.*"/>
      <type pattern="javax.mail.internet.*"/>
      <type pattern="javax.mail.search.*"/>
    </interface>
    <implementation>
      <type pattern="gnu.mail.providers.imap.*"/>
      <type pattern="gnu.mail.providers.maildir.*"/>
      <type pattern="gnu.mail.providers.mbox.*"/>
      <type pattern="gnu.mail.providers.nntp.*"/>
      <type pattern="gnu.mail.providers.pop3.*"/>
      <type pattern="gnu.mail.providers.smtp.*"/>
      <type pattern="gnu.mail.util.*"/>
    </implementation>
  </concern>
</concerns>
```

Figure 4. MAIL **Concern Configuration File**

coupled-helper, or decoupled-helper) of each implementation class. Once the tool is enabled in Eclipse, the developer will be alerted of any reference to a class of MAIL's implementation (Figure 3).

Extending the Concern's Interface. Later, a second developer tries to extend the application to display the remaining quota of users' mailboxes. The developer notices that there is a method to retrieve this value in the GNU implementation of an IMAP store (i.e., IMAP protocol message store) but that there is no such method in the concern's interface (i.e., JavaMail API). To solve this limitation, the team decides to define their own interface for the `IMAPStore`, which they then add to the mail concern's interface (Figure 5). The second developer then implements the interface with a class that wraps the GNU implementation, and adds this class to the concern's implementation (Figure 5).

```
<concerns>
  <concern name="Mail">
    <interface>
      <type pattern="javax.mail.*"/>
      <type pattern="javax.mail.event.*"/>
      <type pattern="javax.mail.internet.*"/>
      <type pattern="javax.mail.search.*"/>
      <type pattern="example.mail.Factory"/>
      <type pattern="example.mail.IMAPStore"/>
    </interface>
    <implementation>
      <type pattern="gnu.mail.providers.imap.*"/>
      <type pattern="gnu.mail.providers.maildir.*"/>
      <type pattern="gnu.mail.providers.mbox.*"/>
      <type pattern="gnu.mail.providers.nntp.*"/>
      <type pattern="gnu.mail.providers.pop3.*"/>
      <type pattern="gnu.mail.providers.smtp.*"/>
      <type pattern="gnu.mail.util.*"/>
      <type pattern="example.mail.gnu.*"/>
    </implementation>
  </concern>
</concerns>
```

Figure 5. MAIL **Concern Configuration File with Extensions**

Replacing the Concern's Implementation. Before the release date, the project manager informs the development team that they cannot use the GNU library because of a licensing issue. The developers will have to replace the use of the GNU library by Sun's implementation. Because the dependencies to the MAIL concern are managed by JMantlet, the task will be localized to the MAIL concern as defined in JMantlet. A developer performs this task by modifying the `example.mail.Factory` class to instantiate Sun's classes instead of GNU's classes, implements a wrapper class for the IMAP store that uses Sun's classes, and updates MAIL in the concern configuration file.

3.3. Implementation Overview

The JMantlet Eclipse plug-in takes as input the *concern configuration file*. The *Concern Model Builder* builds a model that includes all the classes that define the concerns' interface and implementation, as well as the dependencies between the implementation classes. The *Reference Checker* analyzes each class in the workspace and determines the references to the implementation of the model's concerns. The *Reference Checker* generates an error (in Eclipse an error marker) for each reference found. The *Incremental Project Builder* is responsible for calling the *Reference Checker* on the compilation units that are built in the workspace. The *Concern Model Viewer* displays the concern model in the workbench. We describe the components that have not been discussed in Section 3.1.

Concern Model Builder. The *Concern Model Builder* receives the sets of types that make up the interface and the implementation of a concern as an input. It finds the set of classes that are specific to the implementation by filtering out the classes that intersect both sets. Then, the algorithm finds the *implementors*, the classes that implements or extends a class of the concern's interface. Recursively, it finds all the classes of the implementation that refer to an implementor or to a class that refers an implementor, and classifies these classes as *coupled-helpers*. Remaining implementation classes are classified as *decoupled-helpers*.

Reference Checker. The *Reference Checker* uses the concern model to determine if a given compilation unit contains any reference to an implementation-specific type of a concern. If it finds such a reference, it creates a marker at its location. The message of the marker indicates the category of the class that is referenced in the code (implementor, coupled-helper, or decoupled-helper).

Incremental Project Builder. The *Incremental Project Builder* is added to Eclipse's build process such that when

a resource or a project is built, it can pass the built compilation units to the Reference Checker, which will find the references and re-generate the markers.

4. JBoss Case Study

In this section we present a study of the evolution of the implementation of a Java API across the history of the JBoss source code. The goal of the study was to investigate actual changes to a concern's interface and implementation as part of the evolution of a well-designed system. The study provides evidence that concern interfaces need to be explicitly managed in practice. The study also provides evidence in support of the intuitions underlying our classification scheme.

4.1. Target System and Concern of Interest

JBoss is an open-source J2EE-compliant application server that provides services to simplify the development of distributed applications. The JBoss project began in 1999 as a simple Enterprise Java Bean container and has since grown into a full-blown enterprise development platform. Today JBoss comprises multiple projects and has over six million lines of code in its repository. The JBoss project has a number of desirable characteristics that made it particularly suitable for our study: it is open-source, it implements multiple Java APIs that are associated with high-level concerns, and it is a mature project (over six years) with a relatively clean object-oriented design.

The concern we chose to investigate is the support for transactions, or the implementation of the Java Transaction API (JTA) [16]. Transactions and distributed transactions come in many flavors and are not trivial to implement. Moreover, implementation decisions (e.g., optimistic vs. pessimistic concurrency control [18]) involves many trade-offs that can influence quality attributes of the system such as integrity, robustness, and performance. Transaction support is a concern that has evolved and is likely to continue to evolve, thus, of particular interest for our study. For the remainder of this section we will refer to this concern as TRANSACTION. In the J2EE 1.4 API, [6] TRANSACTION's API is defined by seven interfaces and ten exceptions within two packages (Figure 6). JBossJTA [7] is the JBoss implementation of the standard API and its source code is located within the `org.jboss.tm` package. The package contains a number a classes implementing the core interfaces of JTA such as `TransactionManager` and `Transaction`. The package also contains extra interfaces and helper classes that contribute to the concern's implementation.

[6] java.sun.com/j2ee/1.4/docs/api

[7] wiki.jboss.org/wiki/Wiki.jsp?page=JBossJTA

javax.transaction	
Status	HeuristicCommitException
Synchronization	HeuristicMixedException
Transaction	HeuristicRollbackException
TransactionManager	InvalidTransactionException
UserTransaction	NotSupportedException
	RollbackException
	SystemException
	TransactionRequiredException
	TransactionRolledbackException
javax.transaction.xa	
XAResource	XAException
Xid	

Figure 6. Java Transaction API

4.2. Methodology

We analyzed the evolution of TRANSACTION's implementation classes from their creation to their last revision on the MAIN branch up to the 1 February 2006. To browse the CVS repository we used the FishEye web-based tool provided on JBoss website. [8]

To study the evolution of TRANSACTION's implicit interface, we examined 26 versions of JBoss exhibiting non-trivial differences in the implementation of TRANSACTION (not tagged versions, but snapshots of the repository at midnight on chosen dates). We selected the different versions for our study as follow:

1. Using the FishEye tool, we browsed through each file in the `/jboss-transaction/src/main/-org/jboss/tm` folder of the JBoss repository. We compared each pair of consecutive revisions with the difference tool, and found all cases for which a public member of a TRANSACTION class had changed (e.g., method signature, deleted members). For example, we inspected the difference between revision 1.1 and 1.2 of `TransactionImpl`. In that case we found that the constructor `TransactionImpl(TxCapsule, XidImpl)` had been modified to `TransactionImpl(TxCapsule, Xid)`, we recorded the date of that change. Then, for the same file, we inspected the difference between revision 1.2 and 1.3; then, between 1.3 and 1.4, and so on.

2. We collected all the dates recorded as part of step 1. This resulted in a set of 26 dates.

3. We decremented each date by two days. For example, because we observed changes to TRANSACTION

[8] fisheye.jboss.com

being committed on 5 July 2002, we recorded 3 July 2002. This way, we know that the 3 July 2002 version will be different from any version that comes after 5 July 2002. We chose to backtrack two days based on the assumption that a given change task would be committed to the repository within the same 24 hour period, since programmers tend to avoid committing partial changes that may break a build. [9]

After selecting the versions for study, we wrote a configuration file that defined the interface and the implementation of TRANSACTION. We coded the interface as the set of types within the `javax.transaction` and `javax.transaction.xa` packages and the implementation as the set of types within the `org.jboss.tm` package. Then, for each of the 26 dated versions, we checked out a copy of the JBoss source code from the repository, which we then imported into Eclipse. Using JMantlet we identified the references to the concern's implementation classes from the remainder of the program. For each referred class, we recorded each referring class as well as the category of the referred class (i.e., implementor, coupled-helper, or decoupled-helper). For example on 3 July 2002, two recorded references were from the class `org.jboss.jms.asf.StdServerSession` to the implementor `XidImpl`. We recorded the other references in the same fashion.

4.3 Concern Interface Extension

As part of our case study we observed a number of cases of concern interface extension for TRANSACTION that illustrate the practical need for our approach.

Table 1 contains the number of references to TRANSACTION's *implementation* classes from the rest of the program. The first column (Date) contains the date of the version analyzed. The second major column (TXN Referenced Classes) contains the number of classes in TRANSACTION's implementation that are referenced in the rest of the program. The third major column (Classes Refering To TXN) contains the number of classes in the rest of the program that reference a class of TRANSACTION's implementation. For both variables, we made an additional subdivision to distinguish between the number of implementors (I), coupled-helpers (CH), decoupled-helpers (DH), and the aggregation of all three (All). (The last two major columns, Ref. to Changed and Changed Ref., will be explained later in this section.) The line break in the table (Rollback from 4.0 to 3.2) indicates a major change to most files of the system that was caused by a rollback on the MAIN branch from version 4.0 to version 3.2. The rollback explains the abrupt change in the number of references at that point in time but does not threaten our results since the system evolved differently after the rollback.

If the concern interface had been strictly enforced to JTA standard interfaces, Table 1 would contains only zeros. However, as one can see, many implementation classes (TXN Referenced Classes) are referenced by many external classes (Classes Refering to TXN), and the number generally increases as the system evolves.

A detailed analysis of the references revealed that they origin from classes implementing multiple other concerns, some of which are closely related to TRANSACTION (i.e., CORBA and aspect-oriented support for transactions), and some of which are less related concerns (i.e., the MESSAGING SERVICE, SECURITY, and CLUSTERING concerns) but also make references to the classes of the implementation (mostly decoupled helpers). There are multiple situations where access to information or functionalities that are not provided through the JTA interfaces is required, but for space constraints we only describe three of the most representative.

References in MESSAGING. In the project that implements a messaging service for JBoss (`messaging`), the class `org.jboss.mq.pm.jdbc2.PersistenceManager` refers to `TransactionTimeoutConfiguration` (Decoupled-Helper), an interface implemented by the transaction manager. `TransactionTimeoutConfiguration` allows to retrieve the timeout of the current transaction, information that is not available through JTA's `TransactionManager` interface.

References in SECURITY. In the project that implements the JBoss security framework (`security`), the classes `org.jboss.security.auth.spi.DatabaseServerLoginModule` and `org.jboss.security.auth.spi.Util` both refer to `TransactionDemarcationSupport` (Coupled-Helper). This class provides utilities to suspend and resume a transaction associated with the current thread, functionalities that are not provided through the JTA API.

References in CLUSTER. In the project that implements JBoss clustering capabilities (`cluster`), the class `org.jboss.invocation.jrmp.interfaces.JRMPInvokerProxyHA` refers to the class `TransactionPropagationContextFactory` (Decoupled-Helper), which allows to retrieve the transaction propagation context.

In general we observed that JBoss programmers created interfaces decoupled from the implementation (i.e.,

[9] In some cases where changes were observed on two consecutive days the two versions were almost identical and we took one common date for the two changes

Table 1. References to TRANSACTION

	TXN Referenced Classes				Classes Refering To TXN				Ref. to Changed				Changed Ref.			
Date	**I**	**CH**	**DH**	**All**	**I**	**CH**	**DH**	**All**	**I**	**CH**	**DH**	**All**	**I**	**CH**	**DH**	**All**
3 July 2002	1	1	2	4	1	2	6	8	1	2	0	2	1	0	0	1
19 July 2002	0	1	3	4	0	2	11	12	0	0	0	0	0	0	0	0
15 Sep. 2002	0	1	4	5	0	2	14	15	0	2	0	2	0	0	0	0
24 Nov. 2002	0	1	4	5	0	2	16	17	0	2	0	2	0	0	0	0
27 Nov. 2002	0	1	5	6	0	2	17	18	0	0	0	0	0	0	0	0
28 Nov. 2002	0	1	5	6	0	2	17	18	0	0	5	5	0	0	0	0
14 Feb. 2003	0	1	5	6	0	1	15	15	0	0	0	0	0	0	0	0
23 Apr. 2003	5	4	9	18	5	6	22	26	0	0	0	0	0	0	0	0
3 May 2003	4	5	9	18	5	18	22	38	0	1	0	1	0	0	0	0
14 May 2003	4	5	10	19	5	13	28	38	2	0	0	2	1	0	0	1
30 June 2003	4	5	9	18	5	12	24	34	0	0	10	10	0	0	1	1
ROLLBACK FROM 4.0 TO 3.2																
28 Aug. 2003	1	2	6	9	2	9	17	25	0	6	0	6	0	0	0	0
9 Sep. 2003	1	2	5	8	2	15	16	30	0	0	0	0	0	0	0	0
17 Jan. 2004	1	2	6	9	2	16	17	31	0	0	0	0	0	0	0	0
3 Apr. 2004	4	5	6	15	3	23	13	33	0	0	0	0	0	0	0	0
12 Apr. 2004	4	5	7	16	4	23	14	35	0	0	0	0	0	0	0	0
17 Apr. 2004	4	5	8	17	4	23	14	35	1	0	0	1	0	0	0	0
3 Jan. 2005	4	5	9	18	8	30	22	54	1	0	0	1	0	0	0	0
4 Apr. 2005	4	6	9	19	11	35	24	62	0	2	0	2	0	0	0	0
9 Apr. 2005	4	6	9	19	12	35	24	62	4	0	0	4	2	0	0	2
16 Apr. 2005	4	7	10	21	14	37	24	64	4	0	2	4	0	0	2	2
4 May 2005	4	7	12	23	16	37	23	64	6	10	0	13	2	0	0	2
15 June 2005	4	7	12	23	14	36	23	62	5	0	0	5	0	0	0	0
25 July 2005	4	7	12	23	18	36	24	65	16	3	0	17	3	1	0	4
3 Aug. 2005	4	7	14	25	18	40	25	69	6	0	0	6	1	0	0	1
12 Oct. 2005	4	7	15	26	20	40	33	79	6	0	1	7	1	0	1	2

decoupled-helpers) to access this additional information (see Figure 7). A good example is the *transaction context propagation* (discussed above) which is necessary in order to share the context of a transaction between its participants (i.e., threads). JBoss programmers defined two interfaces and one class to handle context propagation: `TransactionPropagationContextFactory`, which is used to retrieve a transaction propagation context at the client side; `TransactionPropagationContextImporter`, used for importing a propagation context within the transaction manager; and `TransactionPropagationContextUtil`, which provides methods to statically access an instance that implements the two interfaces. Table 2 shows the total number of references in other concerns (# of References) and the number of classes with at least one reference (# of Referencing Classes) summed up over all 26 studied versions, as well as the number of times the class changed (# of Changes) over its history. We can see that the three decoupled-helpers have a considerable number of references in other concerns. They are also very stable as they only changed one or two times (for each file one change was actually attributed to change in the header comment). This example illustrates a case where JBoss programmers created an *extension* to the TRANSACTION concern interface to provide additional functionality that was not provided by the standard API.

We observed many other similar situations which supports our hypothesis that programmers can positively take advantage of the freedom to extend interfaces using design idioms. Our approach supports verifying and reasoning about such idioms.

4.4. Classification of Implementation Classes

Our case study was also an opportunity to assess our classification scheme. We observed that although some implementation classes were heavily referenced, other classes

Table 2. Transaction Propagation Context Implementation Types

Class	# of References	# of Referencing Classes	# of Changes
`TransactionPropagationContextFactory`	526	15	2
`TransactionPropagationContextImporter`	332	10	1
`TransactionPropagationContextUtil`	171	17	2

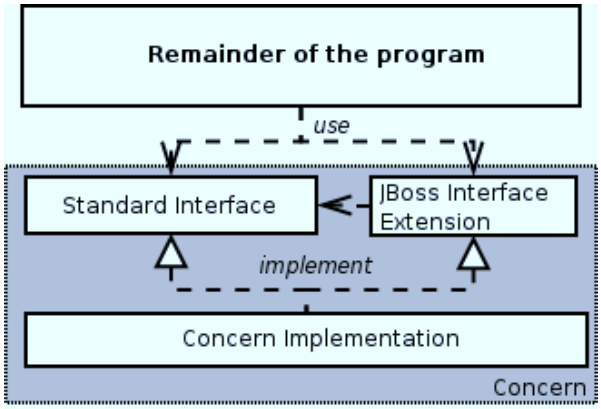

Figure 7. Concern Interface Extension Idiom

were infrequently or never referenced. The data in Table 1 supports these observations: only a small number of implementors are referenced in comparison to decoupled-helpers. For instance, we recorded that on 5 July 2002 (version 3 July 2002 in Table 1) a reference from the class `org.jboss.jsm.asf.StdServerSession` to the constructor of the implementor `org.jboss.tm.XidImpl` was replaced by the use of a factory interface decoupled from the implementation.

The last two major columns of Table 1 provide additional information on the impact of changed implementation classes. The fourth major column (Ref. to Changed) contains the number of classes in the rest of JBoss that referenced a TRANSACTION class for which we had recorded a change to a public member (see Section 4.2). The fifth major column (Changed Ref.) contains the number of classes in the rest of the system that also changed as part of the same change set because of the change to a public member of the TRANSACTION class.

For example, for 3 August 2005, six classes referred to an implementor that changed. Out of these references, only one reference was modified at the same time. In that case it was the method `getTimeLeftBeforeTimeout()` of `org.jboss.tm.TransactionImpl` that was modified to throw a `RollbackException`. The class `org.jboss.tm.iiop.OTSSerant` had to be modified to handle the `RollbackException`.

The numbers of changed external classes (major column Changed Ref.) show that there are more cases in which a change in an implementor is associated with a change in the rest of the system. This observation is interesting because there are fewer references to implementors than to coupled-helpers and decoupled-helpers. It could suggest that implementors change more often than helpers.

We thus decided to investigate whether there was a correlation between the change frequency of a class and its category. We made the assumption that a class changing more frequently has a higher probability to create evolution problems with classes that refers to it. If we can determine that decoupled-helpers are more stable than implementors we could infer that a reference to a decoupled-helpers is *safer* from an evolution perspective. Thus, our classification could be useful to determine if a particular class should be part of the concern's interface or implementation.

We compiled the number of revisions on the main branch of the JBoss repository for each class of TRANSACTION excluding the first revision (i.e., when the class was added) and the last revision (i.e., if the class was removed). Files that existed less than one day were not considered. For each class we normalized the number of revisions over the number of days it existed in the repository. The category of each class (i.e., implementor, coupled-helper, or decoupled-helper) was identified from the last revision of that file in the repository. Table 3 shows the average of revisions per day for each category. A statistical analysis of variance (ANOVA) revealed that there was a significant difference between the change rate of files in different categories ($p=0.0018$). [10] This means that if our hypothesis holds for other systems and concerns as it does for JBoss TRANSACTION, programmers can benefit from knowing that they can more safely refer to *decoupled-helpers* because they tend to be more stable.

Table 3. Revision Rate Per Category

Category	Revisions/Day
Implementors	0.02110
Coupled-Helpers	0.01521
Decoupled-Helpers	0.00517

[10] In other words the probability of this phenomenon being observed by chance is estimated to be 0.0018. Differences are usually considered to be significant with $p < 0.01$.

4.5. Discussion

Our study of the JBoss system documents a concrete case where flexibility for a concern's interface is needed. We observed that in the case of the TRANSACTION concern, JBoss programmers had to refer to a number of implementation classes from the rest of the system. This observation provides evidence in support of our hypothesis that it can be necessary and desirable to extend a concern's interface, and motivates the need for a tool to check and enforce extended concern interfaces.

Since our investigation focused on a single concern, we cannot draw any conclusions about the frequency with which it might be necessary to extend the interface of concerns. However, our approach provides value even in the case where developers respect an interface, as it can be used to restrict all accesses to implementation classes.

As can be expected, our quantitative study of the classification scheme is subject to a number of nuisance factors. First, although we only classified each type once using the last revision of the corresponding file, it is possible that the category changed during the system's history. Second, the assumptions required for an ANOVA test (independence of observations, normal populations, and homogeneity of variance) may not perfectly hold as our sample is a relatively small set of files taken from the same package. Although such factors can have a minor impact on the values reported and the strength of the statistical test, the observed phenomenon is clear enough that we can reasonably expect that the overall observation will hold. Additional investigation should help determine if the observed difference between the rate of change of different categories of implementation classes generalizes over multiple concerns and systems.

5. Related Work

Classpath Access Rules. Eclipse 3.1 supports the definition of *access rules* on a project's classpath entries [8]. Access rules restrict the access to specified packages or types of a particular project's dependencies. Access rules could be used in some cases to have the same restrictive effects as JMantlet. However, access rules do not support the specification of a high-level model of concerns that can help the programmer reason about the concerns of a system. Furthermore, access rules cannot be used to restrict access to classes of a same project. Thus, each concern would have to be implemented as a separate project, an impractical limitation.

Concern Modeling. Many tools such as FEAT [14], ConcernMapper [15], CME [6], and the IntensiVE environment [10], allow users to create views of structurally-related elements that define a mapping between a high-level concern and source code. They are different from JMantlet in that they typically allow modeling of concerns at a finer level of granularity (e.g., methods, fields), but do not enforce or verify interactions between a concern and the rest of the program. The exception is the IntensiVE environment which supports the definition of relations between views that can be checked for consistency. However, the goal and mechanism of the two approaches differ as IntensiVE can verify more generic and fine-grained structural properties between views while JMantlet specifically verifies information hiding between an entire source base and a concern. JMantlet also automatically detects predefined relations between implementation and interface classes from which it generates a classification.

Architectural Description. A lot of work has been done in recent years on the description and verification of software architectures [1, 9, 11, 12]. A large portion of that work focuses on *architecture description languages* (ADLs), which allow the description and verification of software architectures [9]. ArchJava [1] is a recent instance of an ADL for the Java programming language (i.e., an extension to Java) which supports the definition of the architecture of a software system, and guarantees the communication integrity between the architecture and its implementation at run-time. ArchJava and other ADLs explore a set of tradeoffs that is different from JMantlet. For instance, supporting ArchJava's finer-grained model requires additional syntax, a different compiler, and possible runtime performance penalties. In contrast, JMantlet's form of checking is simpler, but can be seen as less intrusive since it is only an extension to the development environment which does not interfere with the Java source code or binaries.

Other tools such as software reflexion models [12] and the virtual software classifications [11] are closer to our approach, wherein a separate high-level model is mapped to source code and used to verify properties of the model in the source code. However, the tools above describe and enforce the concerns of a system's architecture and the communication between those concerns, while our approach provides a way to control the interface to a service (concern) that can differ based on the context in which the service is used.

Reverse Engineering Tools. There is a wide range of reverse engineering tools such as SA4J [7] and JDepends [2] which are useful to analyze the dependencies between classes and packages of a program. These tools can help the programmer create a high-level model of a system that can be used to inspect the dependencies between the concerns of the model. JMantlet differs from reverse engineering tools as it actively enforces the properties of a concern model on a source base.

Classification of Classes. Micro patterns [5] and class blueprint visualization patterns [3] are recent attempts to use mechanically recognizable patterns to create a classification of classes that could be used to assess the evolvability of a system. Our classification scheme is similar in that it is also mechanically recognizable and it could be used to assess the evolvability of a concern. However, our categories are broader and they differ in the level of abstraction. Micro patterns and visual patterns are mainly recognized based on the properties of a class, while our categories are recognized from the relation between classes (i.e., inheritance, dependencies).

6. Conclusion

We presented an approach to allow developers to explicitly specify, manage, and check the implicit interface to a concern's implementation. Our approach is supported by JMantlet, an Eclipse plug-in that uses a simple developer-specified concern model to automatically classify implementation classes, detect external references to classes, and reports disallowed references to a concern's implementation.

We reported on a case study providing evidence of the existence of the practices supported by our approach. Our case study also documents a case where there is a correlation between the stability of a class and its category. Based on this evidence, we form the hypothesis that there might be a generalized relation between the category of an implementation class and its stability in the context of a concern's implementation. If this hypothesis holds, our categorization should provide valuable insights to developers about the potential risk of introducing references to implementation classes. We conclude that tool-based management of concern interfaces is a feasible and practical way to mitigate problems caused by the evolution of code accessing or implementing standard interfaces.

Acknowledgments

The authors thank the anonymous reviewers for their valuable comments. This work was supported by an IBM Eclipse Innovation Grant.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implementation. In *Proceedings of the 24th International Conference on Software Engineering*, pages 187–197, 2002.

[2] M. Clark. *JDepend*. Clarkware Consulting, Inc., 2006. http://www.clarkware.com/software/JDepend.html.

[3] S. Ducasse and M. Lanza. The Class Blueprint: Visually supporting the understanding of classes. *IEEE Transactions on Software Engineering*, 31(1):75–90, 2005.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns—Elements of Reusable Object-Oriented Software*. Professional Computing Series. Addison-Wesley Longman, Inc., 1995.

[5] J. Y. Gil and I. Maman. Micro patterns in Java code. In *Proceedings of the 20th Conference on Object oriented Programming, Systems, Languages, and Applications*, pages 97–116, 2005.

[6] W. Harrison, H. Ossher, S. Sutton Jr., and P. Tarr. Concern modeling in the concern manipulation environment. Technical Report RC23344, IBM Research, 2004.

[7] IBM. *Structural Analysis for Java*, 2004. http://-www.alphaworks.ibm.com/tech/sa4j.

[8] R. J. Lorimer. *Use Access Rules to Enforce API Engagement Rules*. EclipseZone, November 2005. http://www.eclipsezone.com/forums/-thread.jspa?messageID=91952212.

[9] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[10] K. Mens and A. Kellens. Towards a framework for testing structural source-code regularities. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 679–682, 2005.

[11] K. Mens, R. Wuyts, and T. D'Hondt. Declaratively codifying software architectures using virtual software classifications. In *Proceedings of TOOLS-Europe 99*, pages 33–45, 1999.

[12] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: Bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4), April 2001.

[13] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[14] M. P. Robillard and G. C. Murphy. FEAT: a tool for locating, describing, and analyzing concerns in source code. In *Proceedings of the 25th International Conference on Software Engineering*, pages 822–823, 2003.

[15] M. P. Robillard and F. Weigand-Warr. ConcernMapper: Simple view-based separation of scattered concerns. In *Proceedings of the Eclipse Technology Exchange at OOPSLA*, 2005.

[16] Sun Microsystems, Inc. *Java Transaction API (JTA), Version 1.0.1*, April 1999.

[17] Sun Microsystems, Inc. *JavaMail™ API Design Specification, Version 1.2*, September 2000.

[18] A. S. Tanenbaum and M. van Steen. *Distributed Systems Principles and Paradigms*. Prentice-Hall, Inc., 2002.