Using AOP to Ease Evolution

David Shepherd
Quantum Leap Innovations, Inc.
3 Innovation Way
Newark, DE 19716
dcs@quantumleap.us

Thomas Roper
Quantum Leap Innovations, Inc.
3 Innovation Way
Newark, DE 19716
taro@quantumleap.us

Lori Pollock
Computer and Information Sciences
University of Delaware
Newark, DE 19716
pollock@cis.udel.edu

ABSTRACT

This paper describes our experience in using Aspect Oriented Programming to greatly ease maintenance and evolution tasks in industrial software evolution. We have discovered that AOP can be effectively used to add unpluggable features to an existing code base. AOP has allowed us to place code related to certain features into separate modules, which eases evolution when requirements change. Furthermore, AOP allows us to compose these features with the base system in a flexible manner. We relate our positive experience using AOP in production code, describing its use in implementing our project, and detailing the advantages and disadvantages of its use. We abstract our experiences, and report a constrained version of a known AOP design pattern for use in constructing arbitrary, highly composable, independent features.

1. INTRODUCTION

AOP is motivated by the presence of cross-cutting concerns [17, 30, 3]. Cross-cutting concerns are high-level concepts, such as logging, persistence, caching, or synchronization, that are implemented in many different classes. An example of a display updating concern is presented in Figure 1. In the method figureChanged, there are calls to willChange and changed at the beginning and end of the method, respectively. When a figure object is about to change, the display object needs to know, so that it can wait until after the change to refresh. This is accomplished by the willChange call. Once the figure has changed, the display is then notified via the changed call, so it can refresh. Since every figure is required to behave like this, every figure's source code will contain similar calls to willChange and changed.

If one were to update this concern, for instance, to eliminate the call to willChange, one would have to change many different source files. In an Aspect Oriented programming language, such as AspectJ [18], this concern can be implemented in its own module, as shown in Figure 2(b). Within the aspect, a developer can note which methods this concern affects by a *pointcut*. The pointcut specifies that the aspect will affect all calls to the method figureChanged in type `Figure` and in `Figure`'s subtypes. The developer can also specify to call the willChange and changed methods before and after the methods that this concern affects; this is shown at the bottom of Figure 2(b). Thus, an implementation of the concern is contained in one module. In the OOP implementation, the calls to willChange and changed are scattered throughout code, whereas in the AspectJ solution, all calls are in one module. In Figure 2(a)), one can see how using AspectJ simplifies the method figureChanged's code, because it eliminates two statements that are part of a cross-cutting concern.

CH.ifa.draw.contrib.html.HTMLTextAreaFigure

```
public void figureChanged(FigureChangeEvent e) {
    willChange();
    super.basicDisplayBox
        ( e.getFigure().displayBox().getLocation(),
          Geom.corner(e.getFigure().displayBox()) );
    changed();
}
```

Figure 1: Part of a Cross-Cutting Concern

This paper describes the use of AOP to evolve an open source software project to support new features that meet our requirements. In our company, primarily an industrial research and development enterprise, we are often required to implement medium to large size prototypes quickly. To do this, we often leverage open source projects, as well as our own internal projects, to provide functionality to our prototypes that we do not wish to implement ourselves. However, a key barrier to adopting legacy software to fit new requirements is the cost of evolution. Often developers will reimplement functionality to avoid evolving legacy software, because modifying the source code of a stable project to support a new feature can quickly become complicated. If they were to modify the source code of a project to evolve it, they would then effectively have to diverge from the main development branch of that project. Then they

```
CH.ifa.draw.contrib.html.HTMLTextAreaFigure
public void figureChanged(FigureChangeEvent e) {
    super.basicDisplayBox
        ( e.getFigure().displayBox().getLocation(),
        Geom.corner(e.getFigure().displayBox()) );
}
```

(a) Method After Refactoring

```
Aspect MonitorChanges
pointcut changes(Figure f):
   call(* Figure+.figureChanged());

before(Figure f): changes(f){
   f.willChange();
}

after(Figure f): changes(f){
   f.changed();
}
```

(b) The Created Aspect

Figure 2: The AOP Solution

would not benefit from any bug fixes or improvements made to the main development branch, causing this new branch to become stagnant. We call this the *stagnant branch* problem.

During the implementation of a prototype, requirements change often, generally more often than for a typical implementation phase. This creates a situation in which our current project itself, with most of its core functionality in place, acts as legacy software, which we must evolve to meet new requirements. In this paper, we show how we used AOP to ease the evolution of both an open source project as well as our own project, avoiding the *stagnant branch* problem. From this experience, we develop a pattern for adding new *independent features* to existing source code bases. An *independent feature* is a feature that is not part of the core functionality of an application. It does not interact with other, orthogonal features. Information flows from the core application to the independent feature. Our experience suggests that many independent features can be easily added to a program, and that the inclusion or exclusion of a particular independent feature is unlikely to affect the core functionality of the program. Separating features in this way leads to a system that is highly composable, where features can be easily added or excluded to support different configurations.

In Section 2, we discuss the motivating problem that our system was designed to address, as well as some issues we encountered as we began to implement it. In Section 3, we explain why we chose to use AOP for our evolution tasks and we provide a description of the evolution process. In Section 4, we describe how AOP allowed us to add a feature late in the life-cycle of our own software. Section 5 generalizes our results and describes our proposed design pattern. Section 6 discusses related work. Section 7 provides a brief discussion and a conclusion.

2. TARGETED SOFTWARE EVOLUTION PROBLEM

2.1 Motivating Application

Imagine a nationwide system of databases, maintained by the government, that holds records of all legal cars, with each record storing information such as color, model, make, and year, as well as a flag that notes if the car is in police custody. Each state, having their own database, updates their database every time a car's record changes. Many different applications update this database. Clerks at the Department of Motor Vehicles have applications to alter cars' records when a driver's registration changes, police stations have applications to alter records when cars are impounded or reported stolen, etc. A detective also has an application (referred to as `DetectApp`), which scans for cars that pertain to a certain case, and operates at a site remote from the database. `DetectApp` can scan for a set of liscense plates, or even for all cars of a certain color. `DetectApp` operates by beginning a database session with a remote database, downloading the records of cars its seeks to analyze into its local cache, and then ending the database session. It then performs analyses on this data. However, if this data changes at the server, `DetectApp` needs to know, because these changes might affect its analyses' results. `DetectApp` can assure it does not have "stale data" by following one of the following protocols.

1. The interested `DetectApp`s can poll the state's database, selecting cars of interest every short time interval, and checking these new records against its old records to see if any record has changed. In Figure 3, we show that this solution will perform extra queries on the database. Even though the database is only queried twice, `DetectApp` polls the database four times.

2. The interested `DetectApp` can register with the state's database, and the database can send empty messages to registered `DetectApp`s when it is updated. The `DetectApp`s can then perform a select on the database, downloading for the records of the cars in question, checking these new records against its old records to see if any record has changed. Note that this solution performs less queries on the database than the first solution (if $\textit{short time interval} < \textit{average time between messages}$), because it only queries the database when there has been an update. In Figure 4, we show how this solution still performs unnecessary queries on the database. After the "un-interesting" query (one that does not affect data used by the `DetectApp`) `DetectApp` still queries the database to update its information.

3. The interested `DetectApp` can register with the state's database, and the database can send messages that include the primary keys of newly deleted or modified records to registered `DetectApp`s. The `DetectApp` decides whether it is interested in the affected keys. If it is, then the `DetectApp` can perform a select using the provided primary keys. This should perform the least amount of queries on the database (if the chances of an update or deletion being relevant are low) of all three so-

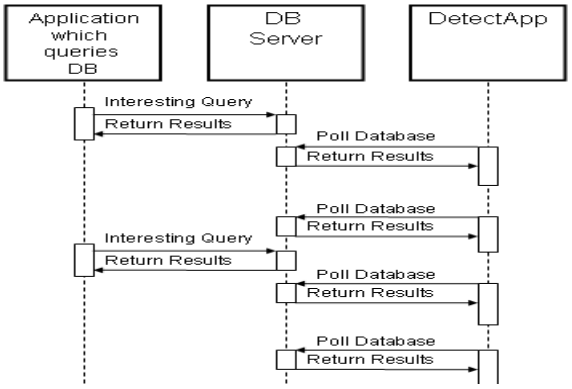

Figure 3: Polling Solution

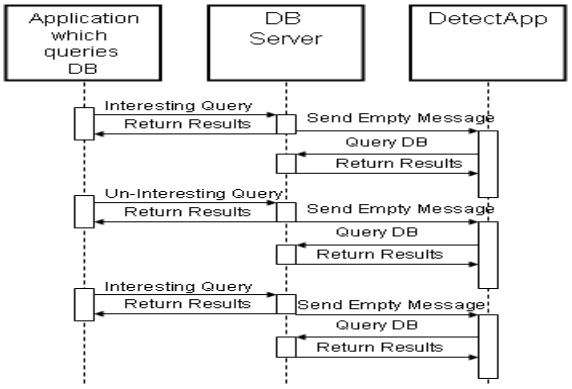

Figure 4: Empty Message Solution

lutions. It also does not require the client to determine which records were changed, thus eliminating some computation. In Figure 5, notice how the `DetectApp` does not update when an "un-interesting" query is executed on the database.

In a situation where the database has many entries (i.e., all of the cars for a state), and the chances that an update or deletion is relevant to a remote application (i.e., `DetectApp`) are low, we argue that solution 3 is the most effective solution because:

- It eliminates unnecessary queries on the database server.
- It requires only a small overhead on the server (i.e., tracking the interested parties).
- It updates remote clients quickly.

2.2 Distributed Data Sharing Framework

In a situation where databases have remote clients that work with a subset of the database's information, these clients would like to know when the database has changed, added, or deleted entries. This is because the remote client might be working with the database's information. Therefore, if the information changes, this might affect the client's computation. In the `DetectApp` example, if the `DetectApp` is performing a data mining operation on a set of cars, in order to detect the similarities between these cars, then `DetectApp` needs to know

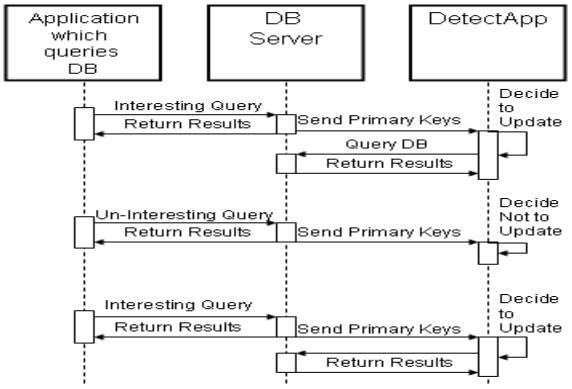

Figure 5: Primary Keys Solution

if any of these cars' information has changed in the database. `DetectApp` must restart its computation if any of these cars' records change, because the results would be invalid. Quick notification of updates allows `DetectApp` to restart sooner, which means the computation will finish sooner than if another notification scheme was used.

To achieve quick notifications, we use a strategy where client applications can declare interest in a database's activities. When that database is updated, the registered clients are notified, and given a reference to the modified data's primary key. Then, each client can either update itself, or, if the information does not affect the current computation, ignore it. This mechanism helps the client applications avoid stale or missing data, without placing a large burden on the server or the network.

2.2.1 *Requirements*

When designing a Distributed Data Sharing Framework in the way we have described, there are a few key requirements to be met.

Requirements

1. The database must have the ability to inform a monitoring process that data has been committed.
2. The database must track which records have been committed, and it must have access to the primary keys of these records.
3. The monitoring process must allow clients to register with it.
4. The monitoring process must inform the registered clients when it receives pertinent information.

2.2.2 *Implementation Issues*

When we consider implementation, we are forced to decide whether to use a proprietary database, without source code, or an open source database. Unfortunately, most proprietary databases do not provide us with enough functionality to develop our framework. These databases do provide the user with a "trigger", which is a segment of stored code that is executed upon an event on a database. Initially, triggers appear to be a possible solution. However, triggers are most often used for enforcing constraints and checking permissions,

thus they do not easily provide state-saving functionality [6, 22, 27]. We decided that a trigger was not the appropriate technology to use for two major reasons:

- The trigger cannot provide all of the needed information, such as the primary keys of the modified items, without re-querying the database, which is too much overhead.
- Triggers cannot save state (without an additional database statement per execution), and thus, could not track which clients are interested in the database.

We also investigated the use of stored procedures [6, 27]. However, stored procedures suffer from the same problems that triggers do.

Therefore, we decided to use an open source database, the Mckoi Database Server [21], with the hopes that we could add functionality to it using AOP to meet our requirements.

3. AOP SOLUTION

We use AOP to evolve the Mckoi Database Server, because it allows us to:

- change the functionality of the database without altering the source code of the database
- separate the original source code from the added source code
- modularize the feature of monitoring the database
- substitute the monitoring feature for a different monitoring feature (try different strategies)

Modifications made to the database monitoring feature late in the life cycle should be eased because the feature is well modularized (i.e., located in one place in source code).

3.1 Adding Monitoring Code

In order to provide the functionality that we need, we must evolve the Mckoi Database Server to notify a monitoring process when it is updated. Furthermore, the database must report which primary keys were affected. In order to do this, we could have actually augmented the Mckoi Database Server source code itself. However, by doing so, we could easily introduce errors into the server. These errors would be particularly hard to correct, because once we make changes to the code base, our code becomes entangled with the original code. After several changes, it becomes hard to tell which code is new (i.e., related to the database monitoring feature) and which code is old (i.e., not related to the database monitoring feature). The database monitoring feature's code can become *tangled* among unrelated code.

Tangling is a known problem in the AOSD domain, and it has several ill effects on the maintainability, evolvability, and readability of the source code [30, 17, 3]. Even though CVS provides the ability to browse the history of a file, and thus, to discover which code is newer, this is a time consuming task that can be avoided. It is better to have our new code completely separate from the original source code. This way, when the functionality of the database monitoring feature is behaving incorrectly, it is much easier to find and fix problems, because the feature is well-modularized.

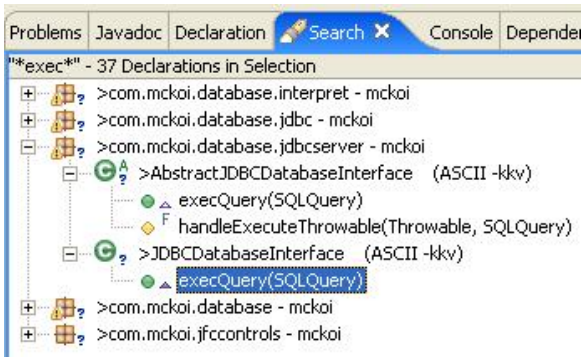

Figure 6: Search for "execute" in Mckoi Server Database

```
1|  try{
2|    // Evaluate the sql query.
3|    Table result =
4|      sql_executor.execute(database_connection, query);
5|*   query.notifyObservers(result);
6|    error = false;
7|    return t;
8|  }
```

Figure 7: OOP: Excerpt from Method `execQuery` In AbstractJDBCInterface

3.1.1 The Search Begins. . .

To evolve Mckoi to meet our requirements, we first begin looking for the places in source code where SQL statements are being executed. We suspect that, at or near the sites where SQL statements are executed, we will be able to access the information that we need (i.e., primary keys, cardinality, table name, etc.). Figure 6 shows the results of searching for a method declaration using the string "exec*". In Figure 6, the most promising package is the `jdbcserver` package, because we are trying to augment the behavior of the server. Within this package, the AbstractJDBCInterface type appears promising, because it has a method named `execQuery`. Upon further inspection of `execQuery`, we find that it is indeed a good candidate, since a query's result is stored in a local variable, as shown in Figure 10. Initially, we thought that we had found the appropriate place in code to inject our database monitoring feature. This was actually not the appropriate place, and we tried several other points in code before we found the correct point.

3.1.2 AOP Solution

In Figure 12, we present an abbreviated version of the aspect oriented solution for monitoring the database[1].

[1]For those very familiar with AspectJ, you will notice that the code in Figure 12 will not work as presented, because of a precedence issue. Since it is not possi-

```
1|public class SQLQuery implements Subject{
2|  ...
3|  public void notifyObservers(Table result){
4|    //Get the primary keys, table, etc. from results
5|    ResultInfo resultInfo =...
6|    //Get query info from the query
7|    QueryInfo queryInfo =...
8|    for(Iterator e = observers.iterator();
9|      e.hasNext(); ){
10|    ((DBObserver)e.next()).
11|       update(resultInfo,queryInfo);
12|   }
13| }
14| public void addObserver(Observer o){...
15| public void removeObserver(Observer o){...
16| public SQLQuery(){
17|   addObserver(DatabaseDistributedNotifier.
18|     getInstance());
19|   ...
20| }
```

Figure 8: OOP: Additions to SQLQuery

```
1|public class DatabaseDistributedNotifier
2|   implements Observer{
3| ...
4| public void update(ResultInfo r, QueryInfo q){
5| ...
6| }
7| ...
8| public DatabaseDistributedNotifier(){
9|   onlyOne = this;
10|  ...
11| }
12| ...
13| public DatabaseDistributedNotifier getInstance(){
14|   return onlyOne;
15| }
16| ...
17|}
```

Figure 9: OOP: Additions to DBDistributedNotifier

We decided to implement this solution as an instance of the Observer Design Pattern, [8] since the new feature will need to observe the database's activities. However, we decided not to use the OOP version of the design pattern, but the AspectJ adaptation of the pattern, because of its improvements in modularity [13].

On lines 3-5 we assign roles to the types in our system (i.e., the `SQLQuery` type functions as the `Subject`). The pointcut on lines 10-12 specifies exactly which changes to the database the feature wants to observe. It specifies a pointcut that captures when method `execute` is called on a `SQLQueryExecutor` object within the class `AbstractJDBCDatabaseInterface` with an argument that is playing the role of a `Subject`. The advice on lines 14-22 is not a standard part of this pattern's implementation, but in this specific case we must use it. The advice has access to the query and to the results table, and so it gathers the information from these two sources and packages it into the `Subject`. This information is forwarded to the `DatabaseDistributedNotifier` in the method `updateObserver` (lines 24-29), which is automatically triggered by `ObserverProtocol`. The details of the observer protocol are implemented in `ObserverProtocol`; intuitively, after the subject changes (i.e., the pointcut on line 10 matches) the information is first packaged up (i.e., the advice on lines 14-22 is executed), and then the observer is notified (i.e., the `updateObserver` method is executed). Note that, conceptually, information flows from the original source code into the Aspect, but not vice versa.

Another important part of the `DBObserver` aspect is the registration code, which serves to attach observers with subjects. The pointcut on line 33 captures the creation of any new `SQLQuery`s. The advice on lines 36-39 attaches these queries with the _ `dbDistribNotifier` from line 31. Having this code in the aspect, instead of an object, makes this pattern easier to unplug.

3.1.3 OOP Solution

In Figures 7, 8, and 9, we present the possible object-oriented solution for monitoring the database. It involves implementing the standard observer design pattern [8]. This causes us to add code to three different classes, `AbstractJDBCInterface, SQLQuery,` and `DBDistributedNotifier`. The observing feature becomes a crosscutting concern, because its implementation resides in three different classes. The feature also gets tangled, at a method level, with unrelated methods, as represented by the "..." separating the relevant methods in Figure 8. It also gets tangled at the statement level, as shown in Figure 7.

3.1.4 Discussion

We prefer the AOP solution, in this case, because it is a non-invasive, well-modularized solution, having several positive effects.

Exploring Possible Insertion Points

When adding an observer to existing code, it is often unclear exactly where to trigger a subject change event. There are often several points in code which appear appropriate, although there is usually only one point which is most appropriate. AOP allows us to easily change this trigger point, without a cut and paste routine, which allows us to investigate the solution space quickly.

In order to change the trigger point in the AOP solution, we only need to change parts of lines 10-12 in Figure 12. For instance, in order to change the pointcut to affect a method call in a different source file, we would replace the text `AbstractJDBCDatabaseInterface` with the new source file's type.

If we had used OOP (as in Figures 7, 8, and 9), and simply inserted code into existing methods and classes, we would have had to cut and paste the call that was added in Figure 7 elsewhere, because this point in code does not provide us with the best trigger point. This particular point in code does provide us with all of the result set information that we need, but the results are not actually stored in the database yet. If we tell clients

ble in AspectJ to declare precedence between abstract and concrete advices, we altered `ObserverProtocol` and `DBObserver` slightly to, in effect, declare the precedence of `DBObserver` to be higher than `ObserverProtocol`.

```
// Evaluate the sql query.
Table result =
    sql_executor.execute(database_connection, query);

// Put the result in the result cache... This will
// lock this object until it is removed from the
// result set cache.  Returns an id that uniquely
// identifies this result set in future
// communication. NOTE: This locks the roots of
// the table so that its contents may not be altered.
result_set_info =
    new ResultSetInfo(query, result);
result_id = addResultSet(result_set_info);
```

Figure 10: Result Table Stored in Local Variable

about this most recent update to the database, and then they query the database, they might query the database before the results are written. We had to try many different places in code before we found the appropriate pointcut. However, it was easy to try many different pointcuts, by simply changing the pointcut definition, without changing the advice code. Through trial and error, we were able to determine that MConnection's method `execQuery` was the appropriate pointcut.

Making Changes Without Accidental Errors

When attempting to explore the solution space with OOP, the programmer often finds himself cutting and pasting his code (i.e., line 5 in Figure 7) into and out of several OOP methods in order to find the correct final resting place of his code. This practice is dangerous, because it is easy to accidentally modify the OOP code. For instance, when cutting from Figure 7, the programmer could accidentally cut lines 5-6, when only intending to cut line 5. When the programmer pastes that code into a new position, it is likely that the code will cause no compilation problems, because `error` is a common field among these related classes. Therefore, the programmer would probably not notice his mistake. However, this would cause corrupted code in the original class.

Another disadvantage of the OOP solution is that, when the original source code base is modified (i.e., for a bug fix), the new version of the source code base and our altered version of the source code base will have to be merged. In the AOP solution, unless the new version involves a major refactoring that negates the validity of our pointcut, no changes to our code need to be made. The AOP solution seems to automatically evolve with the source code base.

3.2 Adding Functionality

Once we had the observer infrastructure in place, implementing the rest of the feature was accomplished in a straightforward, object-oriented fashion. The method in Figure 12 on lines 24-29 calls the object-oriented code's (`update` method), and from there the feature of notifying distributed clients is implemented in a typical OOP style. AOP provided the ability to inject functionality into an existing code base in a modularized way, allowing for unpluggability of this feature.

3.3 Subsequent Evolution

Once the basic code was in place and our framework was working on smaller examples, we realized that our requirement that the framework report the primary keys to the clients could heavily degrade network performance, because large data sets could be inserted at one time, causing a large amount of primary keys to be sent. Therefore, we decided to change the requirement to only report primary keys if the table has less than 32 records that are modified.

This evolution task was easy because we had modularized the concern of monitoring the database. Instead of searching for exactly which classes implement the crosscutting concern of observing the database, we are able to immediately visit the `DBObserver` aspect, which houses the feature we wish to evolve as a first class entity. From here, we can easily add a conditional to the body of the advice on lines 14-22 to affect whether we collect and send primary keys.

4. FLEXIBLE CUSTOMIZATION PROBLEM

Quantum Leap is involved in bringing new technology to market. Therefore, the product we are selling is very often not the small application that we have written as a demo, but the underlying high-technology framework that is driving the demo. We often need to be able to show and explain how our technology works, so the clients can understand how our technology can help them solve their problems. To do this, we often build a GUI that represents objects and actions in our system. In addition to the software evolution tasks just described, we used Aspect Oriented Programming to build a visualization tool with the following properties:

- Can easily be disconnected from code, when we need speed (in production use)
- Can easily be added to the code, when we need the demo
- Does not slow down production code
- Connecting and disconnecting the GUI should not cause changes in the original source code

4.1 AOP for Monitoring

We used AspectJ to serve as an unpluggable observer, to observe the original application and inform the GUI demo of updates. The aspect in charge of monitoring the normal program observes method calls to functions that represent interactions between units in our system. The observing aspect also tracks the creation of important objects in our system, informing our GUI of the system changes that affect its state. Our GUI provides a view that shows the created objects of the system, as well as the messages sent between them. Note that information flows again only from the source code base into the feature.

4.2 AOP for Enhancing Performance

We also used AspectJ to augment the performance of our GUI. For instance, we include an Aspect that augments the amount of time that messages are left on

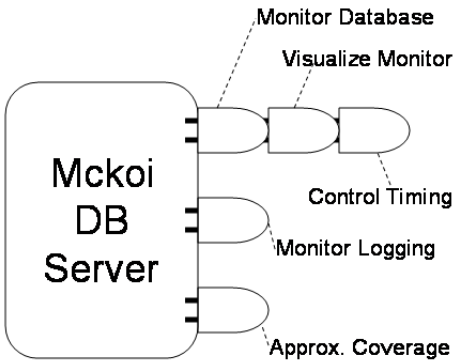

Figure 11: Complete System

the screen before being cleared by observing the clearing calls and making the update function sleep for the appropriate amount of time. In this way, we can either slow down or speed up the visualization. In this case, we consider our snap-on GUI the core application, with the timing feature composed onto it. Note, the information flows from the GUI into the timing feature.

4.3 Overview of Complete System and Feature Interaction

An overview of our completed system is presented in Figure 11. We show how we augmented the core system (the Mckoi Database Server) by using AspectJ to monitor the database, monitor logging, and monitor approximate statement coverage for a given program execution. We also show how we used AspectJ to augment the database monitor by adding a visualization monitor onto it, and how we augmented the visualization monitor by adding a timing controller onto it. This diagram highlights the use of AspectJ to make several additions to a program's core, as well the use of AspectJ to make additions to these additions.

5. INDEPENDENT FEATURE PATTERN

After our experiences in using AOP to evolve software, we have created a constrained version of the AOP Observer design pattern that will enable us and others to apply our experience to many situations. We call this constrained version of the Observer pattern the `Independent Feature Pattern`. This feature is designed to be independent of both other features, and the core application is not dependent upon this feature. Because of this, many *independent features* can be added to a core application simultaneously without the fear that they will affect each other or the core application's functionality.

5.1 Intent

Avoid complicating the source code that comprises the core functionality of an application, add new *independent features* by implementing an instance of the Observer design pattern which considers the original Application to be the subject and the Independent Feature to be the observer. Constrain this implementation by:

```
1| public aspect DBObserver extends ObserverProtocol{
2|
3|  declare parents SQLQuery implements Subject;
4|  declare parents DatabaseDistributedNotifier
5|    implements Observer;
6|
7|  protected pointcut independentFeaturePackage();
8|      within(dbObserverPackage...*);
9|                                                    A
10|  pointcut subjectChange(Subject query):
11|    call(* SQLQueryExecutor.execute(..)) &&
12|      within(AbstractJDBCDatabaseInterface) && args(*,query);
13|
14|  after(Subject query) returning (Table results)
15|    :subjectChange(query){
16|    //Get the primary keys, table, etc. from results
17|    ResultInfo resultInfo=...
18|    //Get query info from the query
19|    QueryInfo queryInfo=...
20|    //Package results
21|    query.addResults(resultInformation, queryInformation);
22|  }
23|
24|  protected void updateObserver(Subject s, Observer o)
25|  {
26|    ResultInfo r = s.getResultInfo();
27|    QueryInfo q = s.getQueryInfo();
28|    ((DatabaseDistributedNotifier)o).update(r,q);
29|  }
30|                                                    B
31|  DatabaseDistributedNotifier _dbDistribNotifer;
32|
33|  pointcut registration():
34|    (SQLQuery+.new(..));
35|
36|  SQLQuery around(): registration(){
37|    SQLQuery query = proceed();
38|    DBObserver.aspectOf(). addObserver(query,_dbDistribNotifier);
39|  }                                                 C
40| }
```

Figure 12: IFP example code

1. not allowing information to flow from the observer to the subject,
2. not allowing the Independent Feature to modify variables belonging to the original Application, and
3. requiring registration to occur in the Connecting Aspect or in the Independent Feature, not in the original Application.

Implementing features in this way makes them highly composable, and unlikely to destroy required behavior of the original Application.

5.2 Motivation

Consider our database monitoring feature, which we added as a new feature to the Mckoi Database Server. Adding feature code to the original Mckoi Database Server source code causes the implementation of this feature to be tangled and scattered throughout the system. Implementing this feature as an Independent Feature allows the feature to be well-modularized. The goal

of this pattern is to add composable, (un)pluggable features to an application without complicating the core functionality of that application.

5.3 Applicability

Use Independent Feature Pattern when all of the following hold:

- The Independent Feature does not flow information back to the core functionality of the original Application.
- The Independent Feature monitors or gathers information from the original Application.
- Adding the Independent Feature to the application in an object-oriented manner would cause tangling or scattering.
- The Independent Feature does not need to interact with other orthogonal Independent Features. For instance, in Figure 11, the Monitor Database feature does not interact with the Monitor Logging feature. The interaction between the Monitor Database and the Visualize Monitor feature is allowed because, in this case, the Monitor Database feature is behaving as a `Core System` with the Visualize Monitor feature augmenting it.

5.4 Structure

This constrained Observer Design Pattern is structured exactly like the original AspectJ implementation of the Observer Design Pattern, with a few additions. The concrete observer, for instance a database observer, is implemented by extending the `ObserverProtocol` aspect. This is not a change to the original pattern. However, the registration functionality must now be implemented inside of this extension. Having the registration performed here makes the pattern less connected to the original Application, and thus easier to unplug. This satisfies Constraint 3 from above. Additionally, the developer can constrain the use of the subject in the Independent Feature's code using an aspect that disallows any calls to the subject's methods except (with caution) getter methods, satisfying Constraint 1 and 2. If calls to getter methods are allowed in the Independent Feature, the developer must manually verify that this does not cause any of the subject's fields to be set, as this clearly would violate Constraint 2.

5.5 Participants

- Application
- Connecting Aspect
- Independent Feature

5.6 Collaborations

When the Application is executing, the Connecting Aspect intercepts certain points of execution, and passes information to the Independent Feature.

5.7 Consequences

Independent Feature Pattern has the following benefits and liabilities:

- Feature is well-modularized
- Feature is highly composable
- Feature is truly independent (cannot easily interact with other orthogonal features)
- Features themselves can be considered an Application, and thus can be augmented using the Independent Feature Pattern.

5.8 Known Uses

This pattern has been used to implement monitoring specific data, gathering application statistics, and controlling timing. We also suspect it could be used to implement many types of logging or tracing, and for any general purpose monitoring task. For instance, in order to implement a feature that warns sleepy data input workers when their key strokes and mouse activity drop below a certain threshold, we could use the `Independent Feature Pattern`.

5.9 Example Code

In Figure 13 we present the `ObserverProtocol` implementation. All unshaded areas are similar to the standard implementation. Shaded area `A` specifies that if any code within the Independent Feature or the Connecting Aspect violates the information flow constraints then it will declare a compiler error. Code would violate these constraints by calling any other method than a method that starts with the string "get". Area `B` defines an abstract pointcut that children of this aspect must implement to define the packages that comprise their Independent Feature. Area `C` defines an abstract pointcut that children must implement to specify when registration occurs.

In Figure 12, we present an example implementation of the `Independent Feature Pattern`. The code that is not shaded is similar to the normal AspectJ implementation of the Observer design pattern. The shaded code labeled `C` represents the registration code, which must be included in the aspect (as specified by the parent aspect). Lines 33-34 specify that every `SQLQuery` that is constructed must be registered with the `DBObserver`. The shaded code labeled `B` is specific to this solution, and provides the `DatabaseDistributedNotifier`, which is used to broadcast to remote clients. The shaded code labeled `A` specifies a pointcut which defines the packages and classes that make up the independent feature of monitoring the database. The definition of this pointcut is used by the `ObserverProtocol` aspect to enforce information flow constraints.

5.10 Related Patterns

We believe that this pattern is worth documenting separately even though it is only an augmentation of a known design pattern because:

- **Intent is different.** The intent of this design pattern is to add new features to an existing code base without complicating the original source code. The observer pattern is focused on maintaining consistent state between objects.
- **Observers have only limited access to subjects.** In this solution, the observer is required not to modify the subject. In this way, his interaction with the subject is more limited than in the original pattern, and they are more loosely coupled.

```
public abstract aspect ObserverProtocol {

    protected interface Subject  { }

    protected interface Observer { }

    private WeakHashMap perSubjectObservers;

    protected List getObservers(Subject subject) {
        if (perSubjectObservers == null) {
            perSubjectObservers = new WeakHashMap();
        }
        List observers = (List)perSubjectObservers.get(subject);
        if ( observers == null ) {
            observers = new LinkedList();
            perSubjectObservers.put(subject, observers);
        }
        return observers;
    }

    public void    addObserver(Subject subject, Observer observer) {
        getObservers(subject).add(observer);
    }

    public void removeObserver(Subject subject, Observer observer) {
        getObservers(subject).remove(observer);
    }

    protected abstract pointcut subjectChange(Subject s);

    after(Subject subject): subjectChange(subject) {
        Iterator iter = getObservers(subject).iterator();
        while ( iter.hasNext() ) {
            updateObserver(subject, ((Observer)iter.next()));
        }
    }

    protected abstract void
            updateObserver(Subject subject, Observer observer);

    pointcut stopInformationFlow():
        call(* ObserverProtocol.Subject+.*(..)) &&
        ( within(ObserverProtocol+)  ||
          within(independentFeaturePackage) )  &&
        !(call(* ObserverProtocol.Subject+.get*(..)));

    declare error: stopInformationFlow(): "Bad Information Flow";

    abstract protected pointcut independentFeaturePackage();

    abstract protected pointcut registration();
}
```

A

B

C

Figure 13: IFP example code

6. RELATED WORK

6.1 Design Patterns

Design Patterns describe patterns of good solutions to known problems in object-oriented programming [8]. Design Patterns have been refactored from OOP into AOP, with many patterns improving because of this change [13]. Several groups have studied how to implement OOP design patterns in AOP, and whether this is an improvement [9, 10, 24, 25]. However, there has been little work to date to develop AOP-specific design patterns, patterns whose structure uses a language feature not available in OOP that is available in AOP.

6.2 Feature Oriented Programming

Feature Oriented Programming deals with the addition of features to a core program, in order to produce different configurations of an application, or to simply increase the functionality of the core program [2]. Similar ideas have driven several case studies and frameworks [35, 34, 16, 1, 23, 7, 20]. Zhang et. al. focuses on customizing the addition of middleware features to an application after its implementation [35]. Three research groups are investigating the use of AspectJ to flexibly compose features of embedded applications [34, 33, 16, 1]. These works are similar to ours, because they use AspectJ to compose features. However, their features often flow information back into the core application, and therefore, are more complex than our *independent features*. Also, they do not present any design patterns for implementing such features.

6.3 Experience Reports with AspectJ

Many researchers have written experience reports after using AspectJ for various tasks [14, 29, 11, 19, 28, 15, 12]. Our paper differs from these reports because it does not intend to evaluate AspectJ generally, but only to evaluate AspectJ for the specific purpose of implementing and composing independent features.

6.4 Evolution and AOP

Rashid et. al. use AOP to implement the instance adaptation strategy of an object-oriented database in order to provide good modularization of this feature [26]. This paper demonstrates how AOP can provide good modularity and eased (limited) evolution. They are able to easily change which instance adaptation approach the database system uses. Researchers have discussed the difficulties of evolving AOP code [31]. Similarly, some have investigated the affects of introducing aspects on the evolvability of real code [4]. Researchers have also offered solutions to AOP evolutionary problems [32]. Researchers have combined AOP with other technology to ease evolution tasks [20]. Our paper is different from previous work in that it specifies a generic pattern for evolution of an OOP code base through the addition of independent features.

6.5 Triggers and Databases

An Active Database is a database that can respond, by executing code, to events that occur within itself [27]. Active Database research can be seen as applying AOP principles to databases. The AOP and the Active Database community have recently realized the similarities of their two research areas, and are working to combine their fields' efforts[5].

7. CONCLUSION

In this paper, we have shown that AOP can be used to ease evolution tasks. We show how to use AOP to add functionality to existing source code bases, non-invasively, as well as how AOP makes it easy to adapt to changed requirements. We also demonstrate how to use AOP to snap-on GUI code to a prototype, in order to visualize the innerworkings of a program. This type of snap-on GUI does not affect the performance of the application when it is not in use.

More generally, we have presented an aspect oriented design pattern as a constrained version of the aspect oriented Observer design pattern that others can use to implement Independent Features and to compose them

with a legacy system. This design pattern has been demonstrated to be useful through our work, as well as through similar patterns in other known work.

8. ACKNOWLEDGEMENTS

This work was supported by a contract from the Office of Naval Research. I would like to thank Marius Marin and Trevor Young for their comments on an earlier draft of this paper.

9. REFERENCES

[1] V. Alves, P. Matos Jr., and P. Borba. An incremental aspect-oriented product line method for J2ME game development.

[2] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. In *ICSE '03: Proceedings of the 25th International Conference on Software Engineering*, pages 187–197. IEEE Computer Society, 2003.

[3] J. Brichau, M. Glandrup, S. Clarke, and L. Bergmans. Advanced separation of concerns. In *ECOOP 2001*.

[4] A. Brodsky, D. Brodsky, I. Chan, Y. Coady, S. Gudmundson, J. Pomkoski, and J. Suan Ong. Coping with evolution: Aspects vs asprin? In *OOPSLA-ASOC 2001*, 2001.

[5] M. Cilia, M. Haupt, M. Mezini, and A. Buchmann. The convergence of aop and active databases: towards reactive middleware. In *GPCE 03: Proceedings of the second international conference on Generative programming and component engineering*, pages 169–188. Springer-Verlag New York, Inc., 2003.

[6] O. Corporation. Oracle 9i Java Developer's Guide http://www.stanford.edu/dept/itss/docs/oracle/9i/index.htm. (April 1, 2005).

[7] C. Gacek and M. Anastasopoules. Implementing product line variabilities. In *SSR '01: Proceedings of the 2001 symposium on Software reusability*, pages 109–117, New York, NY, USA, 2001. ACM Press.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[9] A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena, E. Figueiredo, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 3–14. ACM Press, 2005.

[10] O. Hachani and D. Bardou. On aspect-oriented technology and object-oriented design patterns.

[11] S. Hanenberg and R. Unland. Specifying aspect-oriented design constraints in AspectJ.

[12] S. Hanenberg and R. Unland. Using and reusing aspects in AspectJ.

[13] J. Hannemann and G. Kiczales. Design pattern implementation in java and aspectj. In *Object Oriented Programming, Systems, Languages and Applications*, 2002.

[14] B. Harbulot and J. R. Gurd. Using AspectJ to separate concerns in parallel scientific Java code. pages 122–131.

[15] J. Huang. Experience using AspectJ to implementation cord.

[16] F. Hunleth and R. K. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *LCTES/SCOPES '02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 38–45, New York, NY, USA, 2002. ACM Press.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *ECOOP*, 2001.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect oriented programming. In *ECOOP*, 1997.

[19] K. Lieberherr, D. H. Lorenz, and P. Wu. A case for statically executable advice: Checking the law of Demeter with AspectJ. pages 40–49.

[20] N. Loughran, A. Rashid, W. Zhang, and S. Jarzabek. Supporting product line evolution with framed aspects, 2003.

[21] Mckoi SQL Server Homepage. http://mckoi.com/database/. 2005. (March 23, 2005).

[22] S. Mehrotra. Lecture on chapter 10. In *Introduciton to Data Management*. http://www.ics.uci.edu/ ics184/.

[23] M. Monga, F. Beltagui, and L. Blair. Investigating feature interactions by exploiting aspect oriented programming, 2002.

[24] N. Noda and T. Kishi. Implementing design patterns using advanced separation of concerns.

[25] M. E. Nordberg III. Aspect-oriented dependency management. pages 557–584.

[26] A. Rashid and P. Sawyer. Object database evolution using separation of concerns. *SIGMOD Rec.*, 29(4):26–33, 2000.

[27] K. V. Rhein. Lectures. In *Database System Design*. http://www-1g.cs.luc.edu/ van/cs468/.

[28] A. Schmidmeier. Transferring persistence concepts in Java ODBMSs to AspectJ based on ODMG standards.

[29] S. Soares, E. Laureano, and P. Borba. Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 174–190. ACM Press, 2002.

[30] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE*, 1999.

[31] T. Tourwe, J. Brichau, and K. Gybels. On the existence of the aosd-evolution paradox.

[32] E. Wohlstadter, A. Keen, S. Jackson, and P. Devanbu. Accommodating evolution in AspectJ.

[33] T. Young and G. Murphy. Trevor Young's Homepage http://www.cs.ubc.ca/ trevor/. (April 10, 2005).

[34] T. Young and G. Murphy. Using aspectj to build a product line for mobile devices. In *Demo Presentation at AOSD*.

[35] C. Zhang, D. Gao, and H.-A. Jacobsen. Towards just-in-time middleware architectures. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 63–74. ACM Press, 2005.