A Comparison of Jiazzi and AspectJ for Feature-wise Decomposition

Bin Xin, Sean McDirmid, Eric Eide, and Wilson C. Hsieh

UUCS-04-001

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

March 23, 2004

Abstract

Feature-wise decomposition is an important approach to building configurable software systems. Although there has been research on the usefulness of particular tools for feature-wise decomposition, there are not many informative comparisons on the relative effectiveness of different tools. In this paper, we compare AspectJ and Jiazzi, which are two different systems for decomposing Java programs. AspectJ is an aspect-oriented extension to Java, whereas Jiazzi is a component system for Java. To compare these systems, we reimplemented an AspectJ implementation of a highly configurable CORBA Event Service using Jiazzi. Our experience is that Jiazzi provides better support for structuring the system and manipulating features, while AspectJ is more suitable for manipulating existing Java code in non-invasive and unanticipated ways.

A Comparison of Jiazzi and AspectJ for Feature-wise Decomposition

Bin Xin xinb@cs.utah.edu
Sean McDirmid mcdirmid@cs.utah.edu
Eric Eide eeide@cs.utah.edu
Wilson C. Hsieh wilson@cs.utah.edu

University of Utah, School of Computing
50 South Central Campus Drive, Room 3190
Salt Lake City, Utah 84112–9205

ABSTRACT

Feature-wise decomposition is an important approach to building configurable software systems. Although there has been research on the usefulness of particular tools for feature-wise decomposition, there are not many informative comparisons on the relative effectiveness of different tools. In this paper, we compare AspectJ and Jiazzi, which are two different systems for decomposing Java programs. AspectJ is an aspect-oriented extension to Java, whereas Jiazzi is a component system for Java. To compare these systems, we reimplemented an AspectJ implementation of a highly configurable CORBA Event Service using Jiazzi. Our experience is that Jiazzi provides better support for structuring the system and manipulating features, while AspectJ is more suitable for manipulating existing Java code in non-invasive and unanticipated ways.

1. INTRODUCTION

Feature-wise decomposition is an important technique for modularizing the implementation of a software system and for making that system configurable. Informally, a *feature* is a unit of software functionality: for example, the ability to use multiple fonts in a word processor is a feature, and the ability to invoke a spelling checker is a second feature. A great deal of research has been devoted to creating more formal models of software features (e.g., [12, 16, 17]), particularly motivated by the goal of software reuse. The general idea is to identify common and variable elements in the requirements of similar software systems, and then to use that information in defining and implementing a set of software "parts" that can be used in combination to construct all of the examined systems.

Feature-wise decomposition leads to modularity within the software implementation, because the implementations of different features are encapsulated within separate software modules, components, or other types of parts. This modularity is important for developers, who need to understand the features in isolation from each other. If all of the code for a feature is contained in one element of the system, then the feature can be manipulated and maintained as a whole. Furthermore, modularity helps programmers to understand the relationships between features, because the feature interconnections are explicitly represented in the code.

Feature-wise decomposition also leads to configurability of the system as a whole, because it allows designers to establish a *software product line* in which any particular system can be configured with all or just a subset of the possible features. Such configurability can be important, for instance, when a software developer needs to create a product that runs on a variety of computing platforms. The modern diversity of computing environments — including PCs, PDAs, cell phones, embedded chips in automobiles, and embedded computers in consumer appliances — makes it infeasible (both in cost and time) to develop the product separately for each platform. By using feature-wise decomposition, a developer can implement a suite of software parts that allow a shared software base to be customized for each platform to address concerns such as resource availability (code footprint, CPU power, memory size) and other environment-specific constraints (different user interfaces across devices).

There are many technologies that facilitate feature-wise decomposition [8, 21, 30], and there has been a significant amount of research on how to use these technologies to develop applications with configurable feature sets [5, 15, 19]. These technologies commonly correspond to programming language concepts such as modules, components, objects, and aspects. Because the underlying concepts are quite different from each other, the technologies can lead to implementations that have significant differences in terms of their modularity and configurability.

However, despite these differences, there has been relatively little research that compares the various technologies in a direct way, i.e., for performing feature-wise decomposition of a common system according to common set of identified features. Our work is a step toward filling this gap.

In this paper, we compare the effectiveness of two of technologies, AspectJ and Jiazzi, for performing the feature-wise decomposition of an event channel based on the CORBA Event Service [25]. AspectJ [8] is a Java language extension designed to separate and localize crosscutting concerns: i.e., to modularize the implementation of a concern that would otherwise be scattered throughout the system. A feature is such a concern that can be implemented as an aspect (or collection of aspects) in AspectJ. The feature can then selectively be composed into an application via *aspect weaving*. In contrast, Jiazzi [21] is a Java component system designed to man-

This research was largely supported by the Defense Advanced Research Projects Agency, monitored by the Air Force Research Laboratory, under agreement F33615–00–C–1696. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation hereon.

age imports and exports between modules that contain Java classes. In Jiazzi, a feature can simply be implemented as a Jiazzi component, called a *unit*. The feature can then selectively be composed into an application via linking.

On the surface, AspectJ and Jiazzi do not have much in common. As we have previously shown, however, Jiazzi components can be used to express some of the same crosscutting features that AspectJ can [21, 22]. As a result, we decided that it would be useful to compare the decompositions that AspectJ and Jiazzi support. We took FACET [15], an AspectJ implementation of the CORBA Event Service, and reimplemented it using Jiazzi. The CORBA Event Service has a rich set of identifiable features that can be separated and selectively composed. As a result, it is a good target system for comparing these two systems. It is important to note that we are not comparing the ideas of aspects and components in general: rather, we are comparing two implementations of these concepts in terms of how well they solve a particular feature-wise decomposition task. This paper describes our experience. This paper also presents the conclusions that we have drawn, based on our experience, that we think are applicable to the use of AspectJ and Jiazzi for feature-wise decomposition tasks in general.

The main contributions of this paper are twofold:

1. We identify several important issues that arise when doing feature-wise decomposition and composition. These concerns include separate compilation, incremental composition, compositional reasoning, feature dependency, composition order, refactoring, and expressiveness and readability.

2. We evaluate AspectJ and Jiazzi in terms of these issues and conclude that Jiazzi provides better support for structuring a system and manipulating features, while AspectJ is more suitable for modifying and extending existing Java code in non-invasive and unanticipated ways, such as adding new features without modifying existing code.

The rest of the paper is organized as follows. Section 2 presents background on AspectJ, the CORBA Event Service, FACET, and Jiazzi. Section 3 describes the reimplementation of the Event Service in Jiazzi. Section 4 discusses the differences between AspectJ and Jiazzi with respect to the feature-wise decomposition of the event channel. Section 5 examines additional Jiazzi usability issues. Section 6 discusses related work, and Section 7 concludes.

2. BACKGROUND

In this section, we provide some background information on AspectJ, Jiazzi, the CORBA Event Service, and FACET.

2.1 AspectJ

AspectJ [8, 18] is a Java language extension that enables programmers to modularize the implementation of crosscutting concerns. Its design is based on the observation that no single decomposition can modularize all the aspects or concerns in the system. There will always be certain design concepts or elements that crosscut many modules.

AspectJ provides an *aspect* construct to separate and localize the implementation of a crosscutting concern. An aspect consists of the application of *advice* to *pointcuts*. *Advice* consists of code that is introduced into a program. The base code is called the *advised* code, and the aspects constitute the *advising* code. The process of integrating advising code with advised code is called *aspect weaving*.

Pointcuts are descriptions of the points where the code is to be introduced: each point is called a *join point*. Join points are dynamic: they are events in the advised program's execution. Pointcuts, on the other hand, are static: they are syntactic descriptions of join points. For example, a pointcut might consist of all of the method calls where the method is named *foo()*.

AspectJ supports several kinds of advice. *Before* advice introduces code to be executed at a join point before continuing the normal execution from that point. *After* advice executes the introduced code after the normal execution finishes at a particular join point. *Around* advice shortcuts the normal execution at a join point with the execution of the introduced code. The keyword *proceed* can be used in the body of around advice to resume the execution of the normal code; after the normal code finishes, the remaining part of the around advice code is executed. AspectJ also offers the ability to introduce new fields or methods into existing classes and declare new parent interfaces and classes for existing types.

In the AspectJ programming model, there is an asymmetry between the advising code and the advised code: the references from the advising code to the advised code are unidirectional. In other words, the advised code does not have knowledge about how advising code will affect its behavior. This property makes AspectJ a useful tool for retrofitting new features into existing systems. However, this unidirectional property can sometimes make the resulting system hard to understand, because the advised code contains no apparent references to the aspects. IDE support can alleviate this problem by providing navigation between the advised code and the advising code, but the dynamic nature of some AspectJ pointcut designators (such as run-time type tests and control-flow tests) can be difficult to visualize in an IDE.

By implementing features with aspects and by selectively incorporating aspects into a program during aspect weaving, AspectJ can be used to do feature-wise decomposition. When multiple features are selected for a configuration, aspects that implement these features might introduce code at the same join points in the program's execution. AspectJ employs a set of nontrivial rules to prioritize the running order of different pieces of advice when they are advising the same join point [8, 18]. These rules prioritize advice based on advice type and specificity, and can be overridden by explicit precedence declarations.

2.2 Jiazzi

Jiazzi [21] is a component system that we have developed for Java. In Jiazzi, components are called *units*. A Jiazzi unit is a "container" of Java classes that can be instantiated multiple times. A unit imports and exports *packages*, which are groups of classes. An important feature of Jiazzi is that classes in a unit can inherit from imported classes. This feature allows a Jiazzi unit to support mixin-like behavior. Jiazzi also allows a unit to specify that an imported class must inherit from another imported class, or even from an exported class. This latter feature is important, as we shall see.

Jiazzi provides its own language for defining and linking components. In this language, *signatures* are used to describe groups of classes and interfaces in a unit's imports and exports. Signatures allow units to hide classes and class members between components. Jiazzi's signatures are designed to support type-safe separate compilation, even in the presence of cross-unit inheritance.

Jiazzi provides support for feature-wise decomposition with the open-class pattern [21, 22]. The open-class pattern is an organization of units that makes use of cross-unit inheritance and inheritance of imports from exports, as illustrated in Figure 1. The feedback loops in the figure introduce the most derived versions of classes into all intermediate classes in the inheritance hierarchies. This ensures that only a single class type for a given class hierarchy is used to instantiate objects in a final composed system. However,

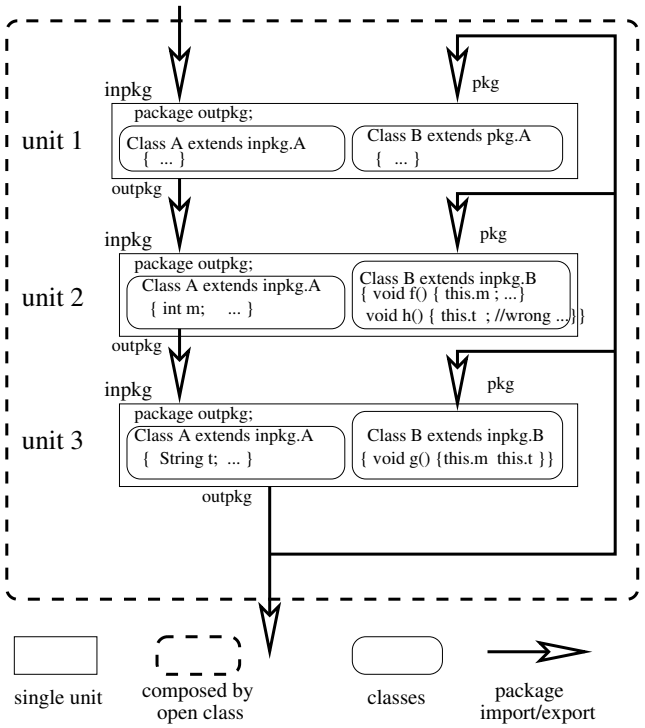

Figure 1: Inheritance across units and inheritance of imports from exports. Import and export packages are named and are mapped correctly when units are composed.

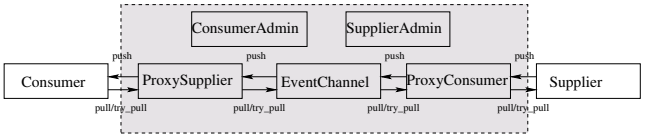

Figure 2: This graph shows the components and interactions in an event channel. Components in the shadowed box are implemented by the Event Service. Arrows indicate the direction of method invocations.

in each unit, class members introduced by the downstream units are not visible (In Figure 1, `B` cannot access `A.t` in unit 2). More importantly, the open-class pattern supports the construction of class frameworks, where a group of related classes can all be subclassed together.

This pattern implements open classes, as supported in MultiJava [7]. Open classes allow a programmer to add new members to existing classes. The open-class pattern allows members to be added through inheritance between units in Jiazzi. This support for open classes enables Jiazzi to be used for feature-wise decomposition. Classes and class members that implement a feature are packaged into a Jiazzi unit. For each configuration, feature units are selectively chosen as the parts of an open-class pattern instance. These feature units are then linked together in the open-class pattern to form the target configuration.

2.3 CORBA Event Service

CORBA [25, 31] is a distributed object system designed to enable communication between objects that may be written using different programming languages, possibly executing in different processes, and possibly running on different hardware platforms. The

```
interface PushConsumer {
  void disconnect_push_consumer();
  void push();
}
interface PushSupplier {
  void disconnect_push_supplier();
}

class EventCarrier {
  EventCarrier();
  ...
}
class EventChannelImpl {
  public void pushEvent(EventCarrier eventCarrier);
  public void destroy();
  public ConsumerAdmin for_consumers();
  public SupplierAdmin for_suppliers();
  ...
}
class ConsumerAdminImpl {
  protected EventChannelImpl eventChannel_;
  public ProxyPushSupplier obtain_push_supplier();
  ...
}
class SupplierAdminImpl {
  protected EventChannelImpl eventChannel_;
  public ProxyPushConsumer obtain_push_consumer();
  ...
}
class ProxyPushConsumerImpl {
  ProxyPushConsumerImpl(EventChannelImpl ec);
  public void connect_push_supplier(PushSupplier
                                    push_supplier);
  public void disconnect_push_consumer();
  public void push();
  ...
}
class ProxyPushSupplierImpl {
  ProxyPushSupplierImpl(EventChannelImpl ec);
  public void push(EventCarrier eventCarrier);
  public void connect_push_consumer(PushConsumer
                                    push_consumer);
  public void disconnect_push_supplier();
  ...
}
```

Figure 3: A skeleton of an implementation of the CORBA Event Service. All the `XXXImpl` classes implement the corresponding `XXX` interfaces and the `ProxyPushConsumer` and `ProxyPushSupplier` interfaces extend the `PushConsumer` and `PushSupplier` interfaces, respectively. An event consumer proceeds by first obtaining a reference to an `EventChannel` object, through which a reference to a `ConsumerAdmin` object is obtained. The consumer can then obtain a reference to a `ProxyPushSupplier` object, to which the consumer can connect itself. Then, when events are available, they are pushed to the consumer. An event producer works similarly.

Event Service is one of the Common Object Services defined by CORBA; the Event Service implements event channels. An event channel decouples event suppliers from event consumers, and provides proxies for consumers and suppliers to register themselves. Figure 2 shows the architecture of an event channel.

Push and pull modes are used to communicate events between suppliers and consumers. In push mode, the supplier controls the flow of events. When an event is generated by the supplier, it pushes the event to the event channel without knowledge of which consumers will receive the event. The event channel pushes the event to registered consumers, and can also perform filtering and aggregation. In pull mode, the consumer initiates communication and pulls events from the event channel that in turn pulls events from registered suppliers. If no event is available, the consumer will block. Alternatively, the consumer can poll to determine if

events are available. Push and pull modes can also be used in combination: for example, a supplier may push events to the event channel, which can later be pulled by a consumer.

Event data can be generic or typed. In the generic case, all event data are packaged into a single object, for example, a CORBA `Any` object. In the typed case, the event data is defined in the Object Management Group's (OMG) Interface Definition Language (IDL) and can have an arbitrary structure. For example, the event data structure can be defined to have a custom header and payload. The event header field can be visible to the event channel so that filtering can be done based on the header.

A general Event Service implementation can incorporate all of the features that an end-user might need. However, such a monolithic implementation is not satisfactory for situations in which some features cannot be supported or are not needed. For example, the memory footprint of a certain feature might be too large for the feature to be used in a resource-constrained embedded system. Therefore, an Event Service implementation with a configurable feature set is desirable.

2.4 FACET

FACET [15] is a feature-wise decomposition of a CORBA Event Service using AspectJ.[1] FACET incorporates features found in the CORBA Event Service, the CORBA Notification Service, and the TAO Real-time Event Service [13, 25]. The FACET event channel implementation has 25 features. Although not all the features are concrete or directly usable from an end-user's point of view, most of them are what an end-user might need in an Event Service configuration. These features are listed in Table 1.

In FACET, features are implemented using aspects and auxiliary classes. Feature composition is performed through aspect weaving using AspectJ. Features desired for a configuration of an Event Service are selected using a configuration file. At build time, a set of aspects specified by the configuration file are passed to the AspectJ compiler and woven into the base implementation of the Event Service. The base implementation of the Event Service offers only the push feature with no event payload. In other words, the base implementation of the Event Service allows the supplier to notify the consumer that an event has occurred, but not to include any data about the event. The Event Service implementation skeleton is shown in Figure 3. We follow this implementation structure of the Event Service in the Jiazzi approach. All the `XXXImpl` classes implement the corresponding `XXX` interfaces and the `ProxyPushConsumer` and `ProxyPushSupplier` interfaces extend the `PushConsumer` and `PushSupplier` interfaces, respectively. An event consumer proceeds by first obtaining a reference to an `EventChannel` object, through which a reference to a `ConsumerAdmin` object is obtained. The consumer can then obtain a reference to a `ProxyPushSupplier` object, to which the consumer can connect itself. Then, when events are available, they are pushed to the consumer. An event producer works similarly

3. EXPERIMENT

To compare AspectJ and Jiazzi, we reimplemented the feature-wise decomposition of the event channel in FACET using Jiazzi. We started by examining the features identified by FACET, shown in Table 1. In our Jiazzi implementation, we included only those features that are identifiable from an end-user's point of view and excluded those features that are merely artifacts in FACET: i.e., those that are not intrinsic features of the Event Service. The excluded features are the tracing feature and the features marked as abstract in Table 1. The tracing feature has been referred to under many situations as an example that demonstrates the expressive power of AspectJ and how AspectJ improves modularity. Although these arguments about the tracing aspect are true, they do not generally apply to other aspects: the tracing aspect only introduces printing code at uniform locations in the program. In FACET, the features marked as abstract are used to do dependency checking among features selected for a configuration. Feature dependencies are managed differently in Jiazzi, so these abstract features can be excluded.

In this section, we first show how a feature unit is constructed in AspectJ and Jiazzi in Section 3.1. Then we demonstrate how a feature is composed with other features in Section 3.2.

3.1 Construct a Single Feature

We use the *Event Pull* feature to demonstrate the difference between how features are implemented in AspectJ and Jiazzi. The *Event Pull* feature adds pull-mode event communication to the Event Service and is optional since some users might only need push mode. In both AspectJ and Jiazzi, the code for the pull feature should be separated into a separate module that can be included in or excluded from an Event Service configuration. The files used to implement the pull feature in AspectJ and Jiazzi are summarized in Table 2. Files in the same row serve similar functions in the two approaches.

In AspectJ, new members (fields or methods) can be introduced into existing types from aspects. These members are scoped relative to the aspects, not to the target types. This idiom is called "inter-type member declaration". In FACET, the implementation of the pull feature relies on this idiom to extend interfaces and classes. A code snippet is shown in Figure 4. It shows how various methods are introduced to various types using an aspect (`pullIntroAspect.aj`). The implementations of introduced methods are added at different places for existing classes and new classes. For existing classes, they are placed in an aspect (`pullAspect.aj`). For new classes, normal Java class files are created (`ProxyPullSupplierImpl.java` and `ProxyPullConsumerImpl.java`). It is also clear that there is no direct language support for describing the feature or the connections with other features. The mechanism to separate the pull feature from other features is by organizing all the implementation files into a directory. The pull feature is selected into a configuration by instructing the build tool to include the source files in the directory.

In Jiazzi, the pull feature is implemented using a unit, as illustrated in Figure 5. The dotted boxes represent other Jiazzi units. The rounded boxes represent normal Java classes. For the nested rectangle boxes, the inner one represents the Java code implementation of the pull feature, and the outer box represents a Jiazzi unit that wraps the Java code with a unit definition described by `pull.unit`. The unit definition describes the import and export packages of the unit. Parts of the Jiazzi code are shown in Figure 6. The unit definition file describes the structure of the pull feature and the location of the Java code that implements the feature. The signature files describe the structure of the packages that the pull features import and export. These mechanisms force the designer to describe the structure of the system explicitly, which helps others to understand the system. The pull feature can be developed separately from other features: the unit can be compiled and saved in a binary format, which can then be linked with other features.

The open class pattern is the main vehicle for doing feature-wise decomposition in Jiazzi. In Figure 6, lines commented with "ocp"

[1]FACET contains other subsystems, including an automatic testing framework. For our comparison we focus on the parts of the system that contribute to a deployed event delivery service.

NO	Feature	Dep.	Function
1	Base	(none)	The base implementation of the Event Service
2	Consumer Dispatch	12	Filter events when received by consumers
3	Consumer QoS	1	Support for consumers to register quality of service requirements
4	Correlation Filter	10, 12	Deliver events to consumers only when a combination of events are received
5	Dependency	3	Support for consumers to specify dependencies upon events
6	Dispatch Mutex	1	(*abstract*) Prevent two dispatcher features from being used simultaneously
7	Event Any	13	Provide `Any` as a container for event data
8	Event Header	10	Add a header field to the `Event` struct
9	Event Pull	10	Add pull mode interfaces for consumers and suppliers
10	Event Struct	13	Provide `Event` struct to replace `Any` as event data
11	Event Type	8	Add a type field to the event header
12	Event Type Filter	5, 11	(*abstract*) Provide common code for all event type filtering features
13	Event Type Mutex	1	(*abstract*) Prevent Event Struct feature and Event Any feature from being enabled simultaneously
14	Event Vector	10	Support for sending multiple events to consumers at once
15	Eventbody Any	10	Support CORBA `Any` payloads in the event structure
16	Eventbody Object	10	Support `Object` payloads in the event structure
17	Eventbody Octetseq	10	Support octet sequence payloads in the event structure
18	Eventbody String	10	Support `String` payloads in the event structure
19	Realtime Dispatch	2, 6, 20	Support for event dispatch based on consumer priorities
20	Source Filter	5	Filter events based on the source field in the event header
21	Supplier Dispatch	6, 12	Filter events when received in an event channel
22	Throughput Test	8	Calculate throughput of the Event Service
23	Timestamp	8	Add timestamps to events
24	Time-to-Live	8	Provide TTL support for connected event channels
25	Tracing	1	Support for tracing and debugging calls to the Event Service

Table 1: This table shows features in FACET, features on which they depends, and their functions. *Abstract* features have no meaning to end-users, and are only used to support dependency checking in FACET.

AspectJ Approach	Jiazzi Approach
File Name (line counts): What the code does	File Name (line counts): What the code does
pullAspect.aj (28): New methods `obtain_pull_supplier()` and `obtain_pull_consumer()` are introduced into classes `ConsumerAdminImpl` and `SupplierAdminImpl`, respectively. With these new methods, consumers and suppliers can set up the link for pull-mode event communication.	*ConsumerAdminImpl.java (20)* and *SupplierAdminImpl.java (19)*: `ConsumerAdminImpl.java` extends the imported class `ConsumerAdminImpl` as described in `base.sig` with a new method `obtain_pull_supplier()`. `SupplierAdminImpl.java` performs a similar task, adding a new method `obtain_pull_consumer()`.
pullIntroAspect.aj (15): New abstract methods, `pull()`, `try_pull()`, `connect_pull_consumer()`, `disconnect_pull_supplier()` etc., are introduced into interfaces, `PullSupplier`, `PullConsumer`, `ConsumerAdmin`, `SupplierAdmin`, `ProxyPullSupplier`, and `ProxyPullConsumer`. These introductions will enable end-users to use the pull feature.	*EventComm.sig (17)* and *EventChannelAdmin.sig (21)*: Describes the interfaces to be implemented by this unit. The end-users use the pull feature through these interfaces.
ProxyPullSupplierImpl.java (81) and *ProxyPullConsumerImpl.java (51)*: Two auxiliary classes that implement the pull feature.	*ProxyPullConsumerImpl.java (51)* and *ProxyPullSupplierImpl.java (80)*: Two new classes introduced to implement the pull feature.
(What is required and what is produced are implicit in AspectJ's approach. They are scattered out in the files `pullAspect.aj` and `pullIntroAspect.aj`.)	*base.sig (10)* and *pull.sig (15)*: `base.sig` is the signature file for the imported package which corresponds to an Event Service implementation without pull feature. `pull.sig` is the signature file for the exported package which describes an Event Service implementation with pull feature.
(There is no way to describe features directly in AspectJ. Features are identified implicitly by grouping the files implementing the same feature into one directory.)	*pull.unit (16)*: `pull.unit` is the unit definition file that describes the signature of this unit.

Table 2: Summary and comparison of the files used to implement the pull feature in AspectJ and Jiazzi. Only non-comment, non-blank lines are counted towards the line numbers of each file.

```
// file ./pullAspect.aj: introduce method definitions.
aspect pullAspect {
  ProxyPullSupplier ConsumerAdminImpl.obtain_pull_supplier()
  { ... }
  ProxyPullConsumer SupplierAdminImpl.obtain_pull_consumer()
  { ... }
}
```

```
// file ./pullIntroAspect.aj: introduce method signatures.
aspect pullIntroAspect {
  ProxyPullSupplier ConsumerAdmin.obtain_pull_supplier();
  ProxyPullConsumer SupplierAdmin.obtain_pull_consumer();
  Event PullSupplier.pull();
  Event PullSupplier.try_pull (BooleanHolder has_event);
  void PullSupplier.disconnect_pull_supplier();
  void PullConsumer.disconnect_pull_consumer();
  void ProxyPullConsumer.connect_pull_supplier(
                        PullSupplier pull_supplier);
  void ProxyPullSupplier.connect_pull_consumer(
                        PullConsumer pull_consumer);
}
```

```
// file ./ProxyPullConsumerImpl.java: method implementations.
public class ProxyPullConsumerImpl {
  private EventChannelImpl eventChannel_;

  protected ProxyPullConsumerImpl(EventChannelImpl ec)
  { ... }
  synchronized void connect_pull_supplier(
                        PullSupplier pull_supplier)
  { ... }
  synchronized void disconnect_pull_consumer()
  { ... }
}
```

```
// file ./ProxyPullSupplierImpl.java: method implementations.
public class ProxyPullSupplierImpl {
  private EventChannelImpl eventChannel_;

  protected ProxyPullSupplierImpl(EventChannelImpl ec)
  synchronized void connect_pull_consumer(
                        PullConsumer pull_consumer)
  { ... }
  synchronized void disconnect_pull_supplier()
  { ... }
  Event pull()  {...}
  Event try_pull(BooleanHolder has_event)
  { ... }
}
```

Figure 4: FACET's implementation of the event pull feature using AspectJ. It shows how various methods are introduced to various types using an aspect (`pullIntroAspect.aj`). The implementations of introduced methods are added at different places for existing classes and new classes. For existing classes, they are placed in an aspect (`pullAspect.aj`). For new classes, normal Java class files are created (`ProxyPullSupplierImpl.java` and `ProxyPullConsumerImpl.java`). It is also clear that there is no direct language support for describing the feature or the connections with other features.

show how to prepare the pull feature unit to be used in an open class pattern instance. Also shown in the figure is the inheritance from export declaration which prepares the pull unit for a feedback loop link when the pull unit is composed with other units, as discussed in Section 2.2.

Compared to AspectJ, extra code in Jiazzi is needed to express unit and package signatures. However, this gives us the power to develop each unit separately, as we discuss in the next section.

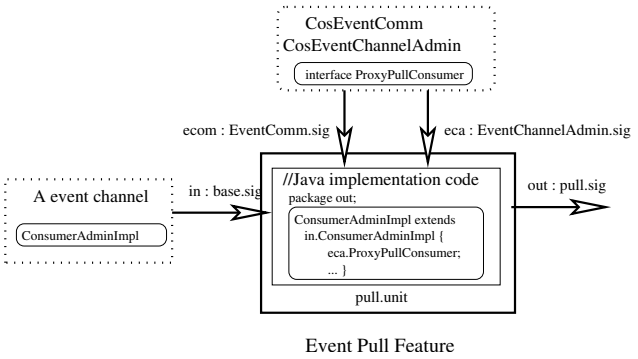

Figure 5: The Event Pull feature as a unit in Jiazzi. The dotted boxes represent other Jiazzi units. The rounded boxes represent normal Java classes. For the nested rectangle boxes, the inner one represents the Java code implementation of the pull feature, and the outer box represents a Jiazzi unit that wraps the Java code.

This example does not show how features implemented by `before`, `after` and `around` advice in AspectJ are implemented in Jiazzi. Simulating these types of advice in Jiazzi presented no great difficulty. However, in some cases a function that implements one aspect in FACET had to be spread over multiple Java files when the same feature was implemented in Jiazzi.

3.2 Feature Composition

The Jiazzi decomposition and the FACET decomposition also differ in how feature units are selected and composed. In our decomposition, one can follow a high-level "boxes and arrows" approach by treating each feature unit as a box and the connection from one unit's export to another unit's import as an arrow. The composition task involves a couple of declarative statements as shown in the following Event Service configuration with pull mode support:

```
compound pushPullAny_config {
  export EventComm : EComm_pullAny;
  export EventChannelAdmin : ECA_pullAny;
  export EventConfig : pullAny;
}{
  link unit bt : base, pat : pushAny, pt : pull,
            cc : pull_cc;
  link package
       cc@EventComm to *@EComm,
       cc@EventChannelAdmin to *@ECA,

       bt@outbase to pat@inbase,
       pat@outbase to pt@inbase,
       pt@outbase to *@base, // open-class

       cc@EventComm to EventComm,
       cc@EventChannelAdmin to EventChannelAdmin,
       pt@outbase to EventConfig;
}
```

The `export` clauses define exported packages from this composition in the form of `<package name> : <package signature>`. The `link unit` clause declares unit instances that are composed in this configuration in the form of `<unit instance name> : <unit definition>`. The `link package` clauses connect the imports and exports of the unit instances in the form of either `<unit name> @ <export package> to <unit name> @ <import package>` or `<unit name> @ <export package> to <export package of this configuration>`. The `*` can be used to denote all the units that have an import package named `<import package>`. The code also illustrates how exported package `outbase` from `pt` is fed back

```
// file ./pull.unit: unit definition
atom pull{
        import EComm : EventComm.sig;
        import ECA : EventChannelAdmin.sig;

        import inbase : base.sig;  //ocp
        export outbase extends inbase : pull.sig; //ocp
        import base extends outbase; //ocp, inherit from export
bind package
        base to *@base,  //ocp
        ...
}
```

```
// file ./EventComm.sig: interfaces implemented
// by the pull feature.
// (similarly for EventChannelAdmin.sig)
signature EventComm = {
interface PullConsumerOperations {
        abstract void disconnect_pull_consumer();
}
interface PullSupplierOperations {
        abstract corba.Any pull();
        abstract corba.Any try_pull(
                        corba.BooleanHolder a0);
        abstract void disconnect_pull_supplier();
}
...
}
```

```
// file ./base.sig: what the pull feature imports
signature base = {
  public class ConsumerAdminImpl { ... }
  public class SupplierAdminImpl { ... }
...
}
```

```
// file ./pull.sig: what the pull feature exports
signature pull= {
  public class ConsumerAdminImpl {
        ...
        EventChannelAdmin.ProxyPullSupplier
                        obtain_pull_supplier();
  }
  public class SupplierAdminImpl {
        ...
        EventChannelAdmin.ProxyPullConsumer
                        obtain_pull_consumer();
  }
  public class ProxyPullConsumerImpl {
        ProxyPullConsumerImpl();
        void disconnect_pull_consumer();
  }
  public class ProxyPullSupplierImpl {
        ProxyPullSupplierImpl();
        Event pull();
        Event try_pull(BooleanHolder hasEvent_);
        void disconnect_pull_supplier();
  }
...
}
```

Figure 6: Jiazzi's implementation of the event pull feature. It shows the direct support for describing features in Jiazzi, using the unit definition file, and the explicitness of describing the connections of this feature with regard to other possible features, using the import/export package signatures. These are the key mechanisms available in Jiazzi that ease the structuring and the understanding of a system. The lines commented with "ocp" are related to the usage of open-class pattern.

to all three units (`bt`, `pat` and `pt`) in an open-class pattern (cf. Figure 6.).

Composing features in FACET does not have any explicit support from the AspectJ language. To configure a system, all the source files for all the features in the system have to be fed to the AspectJ compiler. This can be done by a simple script or a more sophisticated, custom-built configuration system *a la* FACET. In FACET, each feature aspect must register itself and its feature requirements in a central registry. These aspects are compiled first and run to generate metadata about all the features in an XML file, including the source file names and required feature names for each feature. This information only needs to be updated each time new features are added to the feature repository. With this information, one can simply supply the build system with the feature names and the build process will automatically include other required features and build the configuration. However, the developer may still need to resolve ambiguities under certain circumstances, such as when multiple features can satisfy the requirements of a selected feature.

To summarize, Jiazzi's "boxes and arrows" approach to feature configuration is conceptually simple and helpful to the understanding of the system, which we consider important in constructing a system with many features. FACET, on the other hand, provides some implicitness with regard to specifying connections between features and can potentially help a system scale. However, since this implicitness relies on a global namespace, it will also make it harder to reuse aspects. Furthermore, we believe that understanding the structure of all the features in AspectJ is harder.

4. DISCUSSION

In this section, we discuss what we learned from applying feature-wise decomposition using Jiazzi to the CORBA Event Service and comparing it with an existing decomposition using AspectJ. An overall comparison of the decomposition results is given in Section 4.1. We then discuss incremental composition in Section 4.2, composition order and dependency in Section 4.3, refactoring in Section 4.4, and expressiveness and readability in Section 4.5.

From the discussion in this section, we conclude that Jiazzi is a better language tool to architect a system design, which specifies how one decomposes a system into modules, what each module requires and provides, and how modules can be connected. This capability is reflected in the typical process of using Jiazzi to develop a system, as discussed in Section 4.1.

In comparison, AspectJ is a better implementation tool when the scopes of aspects are limited to a local subsystem. It is a better tool to retrofit crosscutting concerns onto existing code due to its flexibility. However, if aspects crosscut an entire system, they prevent the system from being developed, debugged, and composed incrementally, and can also make it more difficult to understand the code.

4.1 Overall Decomposition Results

In our Jiazzi implementation, the decomposition of the Event Service corresponds closely to the decomposition in FACET. This is not surprising, since in doing feature-wise decomposition, the number of feature units is largely decided by the number of features in a system. However, there are certain abstract features in FACET that do not have meaning to an end user of the Event Service, and are only used to do feature dependency management. Since the dependency issue presents itself differently in Jiazzi (as discussed in Section 4.3), these abstract features are not needed in the Jiazzi decomposition.

An interesting comparison is how different language idioms or

design patterns are used in the AspectJ and Jiazzi approaches. Hunleth summarized three AOP design patterns used in FACET [14]: the encapsulated parameter pattern, the template advice pattern, and the interface tag pattern. The encapsulated parameter pattern groups parameters into a container, which can be extended to add parameters. The template advice pattern decouples the definition of pointcuts from the definition of advice, which reflects the principle of encapsulation and can be useful when the coding responsibilities of base and extensions fall onto different developers. Finally, the interface tag pattern tags a set of classes and interfaces that might undergo the same upgrade. Besides, the "inter-type member declaration" idiom has proven very useful in FACET.

In the Jiazzi implementation of the Event Service, we also used the encapsulated parameter pattern. For example, the *Event Struct* feature adds support for an `Event` structure to be communicated between suppliers and consumers. We used the encapsulated parameter pattern to implement features surrounding the *Event Struct* feature. This pattern solves the problem of maintaining a constant signature while enabling additional parameters to be added. For example, a client can always call `push(Event)` to push an event, but the actual members of `Event` can be different for clients using different Event Service configurations.

In Jiazzi, the open class pattern is the main vehicle to extend and compose a system. Writing Jiazzi code is a top-down process which makes the connections between components clear and improves the structure of the system. For example, when writing Jiazzi code, we designed the "interfaces" for each unit first, by specifying its package signatures for import and export. Skeletons are automatically generated by Jiazzi as an implementation of the unit. Finally, specific code is manually inserted into the skeletons. The top-down approach helps one to focus on the overall design of a system from the start.

In comparison, the AspectJ language supports a bottom-up approach to composition. A specific base implementation of a system is constructed first. Extensions are modeled as aspects that rely on pointcut definitions. However, pointcuts are low-level programming constructs that are not suitable for modeling connections between components, since specifying connections is a high-level system design task. The authors of FACET reported that many core interfaces and aspects had to be substantially refactored in the process of adding new features to FACET [14].

4.2 Incremental Composition

Incremental composition allows the composition of a system to be performed step by step. Feature components can be composed into units that can be further used in bigger units or in the final system. Incremental composition enables a certain degree of incremental development. It helps reduce the compositional complexity of the final system, helps find bugs earlier, and enables better reasoning about the code. With such support, debugging of the units can be performed incrementally, and the number of units used in the composition of the final system decreases. When reasoning about the resulting code, the decreased number of units also means fewer connections between units and a higher level of abstraction, which reduces cognitive complexity.

Jiazzi provides better support for doing incremental composition than AspectJ, because Jiazzi provides type checking of unit composition. Of course, type checking comes at a cost: as we showed in Section 3, a Jiazzi programmer must write signature files for every unit. In Jiazzi, after a feature is packaged into a unit, the unit can be compiled and debugged with respect to its import and export signatures. Also, several units representing a subsystem can be linked into a single unit, which can be used to build a bigger system. As long as its import requirements are met, a unit will provide whatever is in its exported package signatures. For example, the *Event Struct* and *Event Header* features in Table 1 frequently appear together in various configurations. By composing them first into an "*Event Struct* with a *Header*" feature, we reduced the number of components in composing a final configuration. More specifically, we have reduced the number of linking clauses in the final configuration. (Section 4.3 gives an example of linking and connecting units in Jiazzi.) This incremental composition ability helps manage the complexity in a large system with a great number of components.

The AspectJ language, on the other hand, does not provide support for type checking and debugging aspects incrementally.[2] As a consequence, errors in aspects may not be discovered until a complete system is in place. Because AspectJ lacks a signature mechanism to enable type checking, it is not as effective as Jiazzi in supporting incremental composition. In return, this lack of type checking enables AspectJ to express useful non-type-safe code transformations.

4.3 Feature Dependencies and Ordering

In this section, we compare Jiazzi and AspectJ with respect to the following issues: feature dependencies, feature execution order, feature composition order, and feature composition error detection. By "feature dependency," we mean that support for one feature requires the presence of other features. By "feature execution order," we mean that one feature's code must be executed before or after another feature's code. By "feature composition order," we mean an explicit specification of the order in which features are combined. Feature dependencies and feature execution order are tightly coupled in Jiazzi, while in AspectJ, aspect dependencies and advice execution order are relatively independent of each other. The differences have certain implications.

Feature dependencies require different treatment in Jiazzi and AspectJ. Consider the *Event Struct*, *Event Header*, and *Timestamp* features (as shown in Table 1): the *Event Header* feature introduces a header field into the event struct, which can be used in the event channel to do content-based filtering. The *Timestamp* feature introduces a field into the event header to record the system time when an event reaches an event channel. Hence, the *Timestamp* feature depends on the *Event Header* feature, and the *Event Header* feature depends on the *Event Struct* feature. These dependencies rule out some feature composition orders.

In Jiazzi, feature dependencies translate directly into a specific feature composition order that is specified by the linking clause in the definition of a compound unit. Here is how order is specified in Jiazzi to match the dependencies for the above example:

```
compound pushEvent_config {
...
link unit es:EventStruct, eh:EventHeader, tt:TimeStamp;
link package
...
        es@outparam to eh@inparam,
        eh@outparam to tt@inparam,
...

}
```

This compound links the exported `outparam` package from the *Event Struct* feature to the imported `inparam` package into the *Event Header* feature, and similarly for the *Timestamp* feature, which

[2] AspectJ release 1.1 takes advantage of the incremental compilation ability of the underlying Eclipse compiler, which is a different issue from separate compilation or incremental composition.

fixes the composition order. When reflected in the class hierarchy, classes in the `EventStruct` unit will be the super classes for those in the `EventHeader` unit.

In AspectJ, dependencies between features imply that one cannot weave a feature without the code for the features on which it depends. Features can be implemented with many aspects, and each aspect can contain many pieces of advice. In AspectJ, the notions of feature dependency, aspect application order, and advice order are distinct: there are well-defined rules for ordering the execution of advice, but the containing aspects are conceptually applied to a base program in a simultaneous fashion, and inter-aspect dependencies are resolved in a simultaneous manner. For example, if two aspects are used to represent the *Event Header* and *Timestamp* features, and they use inter-type member declarations to implement these features, one can apply those two aspects simultaneously. If the *Event Header* feature is somehow missing, then compiling the *Timestamp* feature results in type-checking errors.

Composition order also needs to be considered for independent features in some cases. For example, application semantics can be affected by feature execution order, which might require a specific feature composition order. Given two independent features, different composition orders will produce different results. For example, the *Consumer Dispatch* feature in the Event Service blocks on events with types not registered by an event consumer, and the *Throughput Test* feature is used to count the number of events received by a consumer. Both of these features are implemented as refinements of the `ProxyPushSupplier` class. The execution order of these two features will determine the count number returned by the *Throughput Test* feature.

In Jiazzi, the placement of `super()` calls in subclasses determines the execution order of the code in the parent and child classes. In addition, when composing two independent units, programmers can choose which unit will be the parent unit. As a result, after several individual features have been coded and packaged, different application semantics can be obtained by just choosing different composition orders. For example, if a composition causes classes in the *Consumer Dispatch* feature to be superclasses of classes in the *Throughput Test* feature, the number of events being pushed to the proxy supplier will be counted, regardless of whether the events are filtered by the *Consumer Dispatch* feature. However, if the inheritance relationship is inverted between these two features, the actual number of events received by the consumer will be counted.

In AspectJ, the desired application semantics can be achieved by ensuring an appropriate execution order for advice. For example, the *Consumer Dispatch* and *Throughput Test* features are implemented using two aspects that advise the same pointcut: calls to `ProxyPushSupplier.push(EventCarrier)`. The *Consumer Dispatch* feature can be implemented as around advice, while the *Throughput Test* feature can be implemented using before advice. According to the precedence rules of AspectJ, the execution order of these two unrelated pieces of advice is undefined. If a particular AspectJ compiler runs the around advice first, only the number of events actually received by the consumer will be counted. If these semantics are not expected, i.e., the events filtered out by the *Consumer Dispatch* feature should be counted, then the following explicit precedence declaration should be used:

```
declare precedence:
        ThroughputTestAspect, ConsumerDispatchAspect
```

As demonstrated by this example, programmers must be extremely careful when two pieces of advice advise the same join point. If the implicit (or undefined) ordering of advice is not acceptable, explicit ordering must be specified.

Composition order and execution order are approached at different levels of abstraction in AspectJ and Jiazzi. In Jiazzi, both kinds of order are dealt with at the unit level, where order is largely decided by the order in which units are composed. In AspectJ, composition order is not meaningful, because aspects are applied simultaneously. Execution ordering is dealt with individually for each piece of advice. Execution order between advice is determined using AspectJ's implicit rules or using explicit precedence declarations on the enclosing aspects. Because a single aspect can provide multiple pieces of advice, managing execution order is nontrivial in AspectJ. By considering the composition order issue at a higher level, Jiazzi provides better support for managing unit dependencies and for wiring units together, especially in a system with a large number of components.

The tight coupling between dependencies and composition order in Jiazzi can make it easier to understand the structure of a feature configuration, but could conceivably cause difficulties in implementing some feature combinations. For example, in some situations the "sandwiching" technique [27] may be necessary to decompose features, so that all of the features' units can be assembled in a way that both satisfies unit dependencies and yields the desired run-time execution order. In other words, the designer may be forced to implement a feature as two separate units, due to the limitations of composing features-as-units in Jiazzi.[3] While decomposing the CORBA Event Service with Jiazzi, however, we did not encounter any such complex cases. Except for the *Tracing* feature, which we did not implement, the non-abstract features defined by FACET (listed in Table 1) were each mapped onto a single unit in our implementation. The units implementing these features compose naturally and easily. This suggests that "unnatural" decompositions of features into fine-grain units may be needed rarely, if at all, in real systems. Although Jiazzi's ability to compose units is perhaps more "coarse-grained" than AspectJ's ability to combine aspects, our experience is that the unit model is entirely adequate for feature-wise decompositions: the more complex composition rules of AspectJ are not generally required. Future experiments will be required to test this hypothesis for systems other than FACET and our implementation of the Event Service.

In Jiazzi, dependencies are directly resolved by composition order. Units cannot be wired together in Jiazzi if a required unit is missing. In contrast, AspectJ does not support dependency management very well. In fact, the implementors of FACET found it necessary to implement a special mechanism to validate dependencies between aspects that could not be expressed directly in AspectJ. Additionally, AspectJ silently ignores cases where advice pointcut designators are never matched, which results in functionality of an aspect not being used. This behavior might surprise developers. In contrast, in our implementation of the Event Service, Jiazzi prevents us from making invalid feature compositions and from accidentally omitting features.

4.4 Refactoring Effort

As pointed out by Murphy et al., features are usually identified during the development of a system [24]. Thus, it is often the case that one is forced to go back and refactor the base code when new features need to be added. For example, in the Jiazzi approach, we often need to change the visibility of class members in previously developed features, from private to protected and sometimes to public.

Since we did not perform the AspectJ decomposition of the Event

[3] Analogous refactorings are sometimes necessary in AspectJ. In AspectJ, though, the refactorings are usually used to expose join points.

Service, we have no direct knowledge about the refactoring effort that was required to add new features. However, the authors of FACET reported that many key interfaces and classes underwent significant refactoring during the course of adding new features. We can infer such effort from the interface tag pattern and from other programming idioms used. For example, the `TypeFilter(EventCarrier)` and the `SourceFilter(EventCarrier)` methods perform two types of filtering operations. They are called only once in the `ProxyPushSupplierImpl` class, where these filtering operations are implemented. Thus, their function could have been coded directly in the calling function. However, separating them out provides opportunities to easily add the *Consumer Dispatch* and the *Source Filter* features (shown in Table 1) in FACET.

It may appear that AspectJ provides better support for retrofitting new features onto poorly structured code than Jiazzi because AspectJ constructs for adding extensions to classes are more expressive than those in Jiazzi. Therefore, it should be the case that less refactoring of base code is needed when adding new features using AspectJ. However, as discussed above, refactoring in FACET involved significant invasive changes to the base code. When preparing a system for future extensions, the programming idiom used in AspectJ tends to refactor functionality into methods, or implement interfaces that are only used to tag classes. Because these idioms directly affect the base code, we believe that AspectJ's flexibility does not automatically benefit the process of extending systems.

4.5 Expressiveness and Readability

The programming model of AspectJ makes it difficult to understand AspectJ code in some situations. Although IDE support can make this task easier for programmers, traditional procedural and operational reasoning does not work well with AspectJ code. For example, finding the code affected by the following aspect is not easy for a big system:

```
aspect EventBodyAnyUpgrader {

        pointcut upgradeLocations() :
                this(Upgradeable) &&
                !this(CorbaEventBodyAnyFeature);

        after () returning (Event data) :
                call(Event.new())
                && upgradeLocations()
        { ... }
}
```

This code exemplifies one of the AOP design patterns used in FACET: the interface tag pattern. There are a dozen classes implementing the `Upgradeable` interface in FACET, and there are also many that do not. The choices of which class or interface to tag is not an obvious decision, because this sets up hooks for future extensions without knowing what those extensions might be. In Jiazzi, more code is needed to implement the same upgrade feature, but the upgraded classes are explicitly specified. To implement this upgrade feature in Jiazzi, the `Event` class is extended. In particular, an overriding no-argument constructor is added to the extending class. The rest of the system will use this extended `Event` class to create objects.

AspectJ can offer extremely succinct implementations for functionality that would otherwise be implemented with very complex and tangled code. The tracing concern is an often mentioned example in the literature. FACET provides the tracing feature, which would have required tedious and verbose implementation in Jiazzi. There is a tradeoff between the degree of separation of a feature and the readability of the resulting code. For example, AspectJ offers a higher degree of separation of concerns, while Jiazzi has a simpler model of how code might interact with other code and thus enables better readability. We can also think of the difference as the width of the interface that AspectJ and Jiazzi can manipulate. In AspectJ, this width is every method invocation, whereas in Jiazzi it is every method declaration. The tradeoff has been further explored by Bryant et al. [4]: their approach is to localize the implementation of a concern, but to make the crosscutting nature of this concern known to the rest of the system. Compared with AspectJ, their approach offers better readability, but always requires explicit hooks in the base code.

5. JIAZZI USABILITY

Although our primary goal is to compare feature-wise decomposition in Jiazzi and AspectJ, we also found certain issues in using Jiazzi that are not comparable with AspectJ. These issues are important to consider when designing similar tools.

5.1 Signature Management

Although signatures are crucial for supporting separate compilation and incremental composition, signature management in Jiazzi is a non-trivial issue. For example, our project defines 20 feature units that depend on 39 package signatures. Even with this many package signatures, all legal feature compositions have yet to be explored. One reason for the all these different signatures is that each feature unit must specify its most general import requirements to maximize reusability. When two packages are composed, an additional export package signature is needed that exports a superset of what are exported in these two packages. An automatic tool for signature extraction might be useful to a limited extent.

5.2 Open Class Pattern Usage

Although the open class pattern is the main vehicle in Jiazzi for extending systems, there are subtle issues in using it correctly. For example, in using the open class pattern, each unit creates a new version of the extended classes and interfaces through subclassing. In the source code of each unit, the most extended version of these types should always be used. These types are imported into the unit by the back link in the open class pattern. (See Figure 1.) For example, in the pull unit of Figure 6, the `inbase`, `outbase`, and `base` packages describe three versions of the same set of types. It is important to use only the types in the `base` package.

When using the open class pattern, the constructor definitions in units might place surprising restrictions on composition order. In this project, we ran into the task of composing five units representing the *Base*, *Event Struct*, *Event Vector*, *Consumer QoS* and *Correlation Filter* features as shown in Table 1.

```
//order 1: incorrect
Base -> Event Struct -> Event Vector -> Consumer QoS
-> Correlation Filter
//order 2: correct
Base -> Consumer QoS -> Event Struct -> Event Vector
-> Correlation Filter
```

These five units extend the same package, `base`. As shown above, there are two composition orders for the five units: one is correct and the other is not. The reason for the incorrect order is that the `EventCarrier` class in the *Correlation Filter* unit has a constructor that is not present in the *Consumer QoS* unit, although it does appear in the *Event Vector* unit. The problem arises because constructors are not inherited in Java. Therefore, a `super` call in the `EventCarrier` constructor will cause a static error in the first composition.

This example illustrates that the non-inheritance of constructors interacts poorly with Jiazzi's module system. More generally, the

use of inheritance across modules in Jiazzi conflicts with the non-inheritance of constructors. In future languages with component systems, it might be useful to design some mechanisms that would allow constructors to be inherited.

5.3 Abstract Methods

In Java, a class declared as concrete is required to implement all abstract methods inherited from any superinterface. In the same spirit, when the implementation class is separated from the interface by being placed in different units or components, care must be taken to prevent introducing abstract methods into concrete classes when these components are composed together. However, there is one issue in this checking as illustrated by the following example. In the Jiazzi implementation of the Event Service, the `base`, `push`, and `pull` feature units all import the `EventComm` and `EventChannelAdmin` packages. However, the imported signatures for these two packages are different in the three units. The `push` unit imports an `obtain_push_consumer()` abstract method in the interface `SupplierAdminOperations` in `EventChannelAdmin`, the `pull` unit imports an `obtain_pull_consumer()` abstract method in the same interface, and the `base` unit imports neither of these methods. When composing these three units, only a single version of the `EventComm` package and a single version of the `EventChannelAdmin` packages are provided, which will have signatures that are the sum of the signatures for those three units. In this way, we have introduced an abstract method `obtain_pull_consumer()` into a concrete class in the `push` unit. (The same problem exists for the classes implementing `SupplierAdminOperations` interface in the `base` and `pull` units.) However, by using open class pattern, the three classes in these three units, together, implement all the methods in the `SupplierAdminOperations` interface.

The above problem is not generally associated with programming in Jiazzi, but is due to the conflict between programming the CORBA objects and programming an extensible system (in this case, the Event Service). The implementation of a CORBA object must implement the Java interface generated by a IDL compiler from its IDL definition. The IDL definition is determined by the features that are selected; the interface generated from the definition is then used by the implementation. However, in doing feature-wise composition, the implementation is divided into multiple units using the open-class pattern. As a result of this usage of Java interfaces, the Jiazzi signatures had to contain abstract methods.[4] As a solution, we relaxed the default checking of abstract methods at the unit level in Jiazzi. We still ensure composition safety, though: when a complete program is linked, Jiazzi ensures that all abstract methods are implemented in at least one constituent unit.

The binary compatibility rules in the Java Language Specification [11] address the same problem when the Java library is extended in unforeseen ways. The fundamental issue lies is in the use of Java interfaces. While Java interfaces are good for separating specification from implementation, they do not support evolution, which is common when reusing code and adding new functionalities, or designing a configurable and extensible system from the beginning. This problem is more generally referred to as the "subjectivity" problem [1] to provide different interfaces to a system depending on the system's configuration.

6. RELATED WORK

Jiazzi's component model was drawn from the research work on mixins and units. Mixins are class extensions with parameterized superclasses. Bracha studied mixins as a possible mechanism to bring some of the flexibilities of multiple inheritance OO languages back into single inheritance OO languages [3]. Flatt explored the benefits of integrating mixins with units, a component model with explicit module linking, in facilitating modular object-oriented programming [9, 10].

General issues regarding system decomposition and program families were explored in some of Parnas's classic papers [26, 28]. In constructing a complete program, decisions need to be made about data structures and algorithms. Designing program families is about representing programs at intermediate stages where certain decisions have already been made and some undesirable and uninteresting programs have been excluded, but where some more critical decisions has yet to be made to get a complete program. Step-wise refinement and module specification were considered two complementary techniques to develop program families. Using the former technique, decisions are made sequentially at each step and a (partial) program is obtained after each step. Thus, it does not add significant cost to get the first complete program running, but it is challenging at each step to decide which decisions will likely to change. Using the later technique, all the decisions are faced up-front. However, the decision making is hidden by some well-defined interfaces and placed in separate modules. The cost of designing modules and defining interfaces can be significant, but broader program families are produced. Jiazzi is a realization of the module specification technique with some enhanced features like the open-class pattern.

Subsetting and extending programs is a problem much related to program families [27]. Several techniques can ease the process, including identifying the minimal subset and all minimal increments in the system at requirement analysis stage, using modules, and designing proper "use" structure for subprograms. The "use" structure problem underlies the problem of managing feature dependency as discussed in this paper. Many concerns that we use to evaluate decomposition results in this paper can be found in the above work.

Feature-wise decomposition and feature management have been active research areas. Cardone et al. developed the Fidget [5] system to decompose a graphical user interface using the mixin layer approach. Lai et al. demonstrated techniques to decompose two Java packages, jFTPd and gnu.regexp, using Hyper/J [19]. Our work differs from both of these efforts in that we are not trying to show how feature-wise decomposition can be done with a particular system, but instead what we can learn from these different approaches to perform the same task, and how we can make such tools better.

Another approach to support feature composition is using software generators, including Batory et al's GenVoca model and Prehofer's feature-oriented programming model [1, 2, 29]. In both of these systems, features are composed by writing equations. In GenVoca, interfaces to components or features are called realms and implementations of realms are called components. GenVoca also generalizes the concept of parameterization to allow realm and data member name parameters, besides constant and type parameters. Component composition equations models a series of transformations, which are similar to function composition in mathematics. GenVoca also enables certain design rules to be specified for components that can be automatically checked for every compositions. Prehofer's feature-oriented programming model has a similar flavor with regard to feature compositions. A lifter is required for each pair of features that need method overwriting when composed together. Thus, lifters explicitly embody feature interactions.

Comparison studies try to bring into perspective the often incom-

[4]For Jiazzi, a better solution would be to have the IDL compiler generate Jiazzi signatures instead of Java interfaces.

patible evaluation found in the literature of each individual technology involved in the comparison. Lieberherr et al. [20] identified the tension between aspect-oriented programming (AOP) and modular programming (MP) [26] and the inadequacies of AspectJ and Hyper/J as flexible modular programming tools. They proposed a new language, called *Aspectual Collaboration* (AC), as a way to combine the expressiveness found in AOP and the modularity found in MP. They compared these three languages in decomposing an artificial problem, where the task is to verify that containers do not exceed their capacity and the task is augmented with a caching concern to improve performance.

We share with them a common set of desired programming tool properties that are used to compare different tools. Although the examples used differ in scale, we arrived at similar conclusions in regard to AspectJ: AspectJ is designed to be a flexible programming tool rather than a powerful module system. Aspectual Collaboration requires radical changes to Java language syntax; Jiazzi requires no such changes, but instead superimposes the definition of units onto ordinary Java packages. AC and Jiazzi also differ in how composition is performed. AC adopts the Hyper/J style of composition, where classes defined in different "modules" are combined and class members renamed when necessary, while Jiazzi uses a explicit module linking language. AC also provides direct support for intercepting method invocations, which is more reusable than the method interception mechanism in AspectJ and more fine-grained than unit composition in Jiazzi.

Murphy et al. presented an exploratory comparison study on concern separation using Hyper/J, AspectJ and a lexically-based tool called LSOC [23]. They studied two cases of concern tanglement: tangled in a single method and tangled across a small number of classes. They compared the effects on base and separated code structures constructed by these three tools. They evaluated the effects using code cohesion and "knows-about" relation. The results were mixed. AspectJ was reported to produce a more clean and complete separation, but reasoning about aspects alone was potentially difficult, and the ordering constraints between aspects effectively coupled separated concerns. Hyper/J was reported to produce less cohesive base code, but it was easier to understand and maintain the system because separated concerns were colocated with the base structure. Compared with our study, their work focused on decomposition on a smaller granularity (within a method or a few classes), and as such, complements our work.

There have also been comparison studies conducted on different component frameworks. Casagni et al. developed a framework to structure the comparison process [6]. The framework identified three sets of features about a component framework. They include design properties, which describe the characteristics of resulting system using some component framework; framework impact issues, which describe framework features supporting component hosting and design ramifications due to framework considerations; and quality attributes, which describe the availability, usability, and modifiability of the resulting components. The authors applied their framework to a FIPA-compliant multi-agent system and the Web-centric J2EE platform, and they made suggestions as for which component framework is recommended to use when developing a new system. This framework is not generally suitable for our comparison on tools supporting feature-wise decomposition. For example, design size—number of components in the application—is one of the design properties compared in this framework, but it is not included in our comparison, since the number of components is largely decided by the features identified in the application. However, we do share a common subset of comparison elements, including interactions between components and component reusability.

7. CONCLUSION

We have described a comparison study of feature-wise decomposition using AspectJ and Jiazzi. As part of this study, we identified a set of concerns that are important for developers. Among these concerns, Jiazzi provides better support for doing separate compilation, incremental composition, compositional reasoning of the system, and writing loosely coupled features. AspectJ, on the other hand, provides more support for the ability to retrofit a new feature into a poorly structured code base and the ability to write concise, localized code for crosscutting features. Managing dependencies and execution order among features can be done at a higher level on units in Jiazzi, but requires explicit specification of the composition order of units. In AspectJ, dependency has to be managed outside the language and execution order has to be resolved at a lower level, namely advice and join points. However, AspectJ has implicit rules to resolve execution order of advice; it requires explicit specification in fewer cases than Jiazzi does. Finally, we have made some observations about language design that should aid in the future design of similar language tools.

8. REFERENCES

[1] D. Batory and B. J. Geraci. Composition validation and subjectivity in GenVoca generators. *IEEE Transactions on Software Engineering*, 23(2):67–82, Feb. 1997.

[2] D. Batory, V. Singhal, J. Thomas, S. Dasari, B. Geraci, and M. Sirkin. The GenVoca model of software-system generators. *IEEE Software*, 11(5):89–94, Sept. 1994.

[3] G. Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, University of Utah, 1992.

[4] A. Bryant, A. Catton, K. D. Volder, and G. C. Murphy. Explicit programming. In *Proceedings of the 1st International Conference Aspect-Oriented Software Development*, pages 10–18, University of Twente, Enschede, The Netherlands, Apr. 2002. ACM.

[5] R. Cardone, A. Brown, S. McDirmid, and C. Lin. Using mixins to build flexible widgets. In *Proceedings of the 1st International Conference Aspect-Oriented Software Development*, pages 76–85, University of Twente, Enschede, The Netherlands, Apr. 2002. ACM.

[6] M. Casagni and M. Lyell. Comparison of two component frameworks: The FIPA-compliant multi-agent system and the Web-centric J2EE platform. In *Proceedings of the 25th International Conference on Software Engineering*, pages 341–351, Portland, Oregon, May 2003. IEEE Computer Society.

[7] C. Clifton, G. T. Leavens, C. Chambers, and T. D. Millstein. MultiJava: modular open classes and symmetric multiple dispatch for Java. In *Proceedings of the 2000 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 130–145, Minneapolis, Minnesota, Oct. 2000. ACM.

[8] Eclipse Project. AspectJ Web site. http://eclipse.org/aspectj/, 2003.

[9] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *SIGPLAN Notices*, volume 34, pages 94–104. ACM Press, 1999.

[10] M. Flatt and M. Felleisen. Units: cool modules for HOT languages. In *Proceedings of the ACM SIGPLAN 1998*

conference on Programming language design and implementation, pages 236–248, Montreal, Quebec, Canada, 1998. ACM Press.

[11] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. The Java Series. Addison-Wesley, second edition, June 2000.

[12] M. L. Griss, J. Favaro, and M. d'Alessandro. Integrating feature modeling with the RSEB. In *Proceedings of the Fifth International COnference on Software Reuse*, pages 76–85, Victoria, BC, June 1998.

[13] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of the 1997 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 184–200, Atlanta, Georgia, Oct. 1997. ACM.

[14] F. Hunleth. Building customizable middleware using Aspect-Oriented programming. Master's thesis, Washington University, Saint Louis, Missouri, May 2002.

[15] F. Hunleth and R. Cytron. Footprint and feature management using aspect-oriented programming techniques. In *Proceedings of the 2002 Joint Conference on Languages, Compilers,and Tools for Embedded Systems & Software and Compilers for Embedded Systems (LCTES'02-SCOPES'02)*, pages 38–45, Berlin, Germany, June 2002. ACM.

[16] K. Kang, S. Cohen, J. Hess, W. Novak, and A. Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI–90–TR–021, Software Engineering Institute, 1990.

[17] K. C. Kang, S. Kim, J. Lee, K. Kim, E. Shin, and M. Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5:143–168, 1998.

[18] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In J. L. Knudsen, editor, *ECOOP 2001 — Proceedings of the 15th European Conference on Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, Budapest, Hungary, June 2001. Springer.

[19] A. Lai, G. C. Murphy, and R. J. Walker. Separating concerns with Hyper/J: An experience report. In *International Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE*, May 2000.

[20] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual Collaborations: Combining modules and aspects. Technical Report NU-CCS-02-03, Northeastern University, Nov. 2002.

[21] S. McDirmid, M. Flatt, and W. C. Hsieh. Jiazzi: New-age components for old-fashioned Java. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 211–222, Tampa, Florida, Nov. 2001. ACM.

[22] S. McDirmid and W. C. Hsieh. Aspect-oriented programming with Jiazzi. In *Proceedings of the 2nd International Conference Aspect-Oriented Software Development*, pages 70–79, Boston, Massachusetts, Mar. 2003. ACM.

[23] G. C. Murphy, A. Lai, R. J. Walker, and M. P. Robillard. Separating features in source code: an exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 275–284, Toronto, Ontario, Canada, 2001. IEEE Computer Society.

[24] G. C. Murphy, R. J. Walker, E. L. Baniassad, M. P. Robillard, A. Lai, and M. A. Kersten. Does aspect-oriented programming work? *Communications of the ACM*, 44(10):75–77, Oct. 2001.

[25] Object Management Group. Object Management Group Web site. http://www.omg.org/, 2003.

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

[27] D. L. Parnas. Designing software for ease of extension and contraction. In *Software fundamentals: collected papers by David L. Parnas*, pages 269–290. Addison-Wesley Longman Publishing Co., Inc., 2001.

[28] D. L. Parnas. On the design and development of program families. In *Software fundamentals: collected papers by David L. Parnas*, pages 193–213. Addison-Wesley Longman Publishing Co., Inc., 2001.

[29] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *ECOOP'97 - Preceedings of the 11th European Conference on Object-Oriented Programming*, volume 1241 of *Lecture Notes in Computer Science*, pages 419–443, Jyväskylä, Finland, June 1997. Springer.

[30] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, Los Angeles, CA, May 1999. ACM.

[31] S. Vinoski. CORBA: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 35(2), Feb. 1997.