Building fully customisable middleware using an aspect-oriented approach

R.M. Pratap, F. Hunleth and R.K. Cytron

Abstract: Applications accrue features in response to the needs of all users, yet the associated code bloating and performance loss often render an application unsuitable for some other users whose needs differ. As a result, developers are often faced with either reinventing pieces of an application, custom tailored to their needs, or they are faced with the daunting task of refactoring an existing application to obtain an appropriate subset of that application's functionality. In either case, subsequent development, maintenance and testing of the application becomes more complex, due to the effects of future revisions on all of the derived subsets. In the paper, the authors report their experience of using aspect-oriented programming in building a CORBA event channel using a compositional approach to building software. Instead of refactoring a large, feature-rich application, a feature can be added by including that feature's aspect in the set provided to an aspect compiler. In addition to generic feature management, a study of the effect of the transport layer (CORBA) on overall performance of the event channel is conducted. Finally, experimental results on how such an approach gives fine control over footprint and throughput performance is presented.

1 Introduction

Traditional distributed object computing (DOC) middleware, such as the Common Object Request Broker Architecture (CORBA) [1], COM+ [2] and Java RMI [3], is designed and built to provide a wide feature set to suit the needs of multiple problem domains. This technique increases the appeal of using the middleware, but perhaps at the expense of supplying extra features that may contribute to unnecessary code bloat and configuration complexity for some applications.

An increasingly important area for DOC middleware is embedded and realtime systems. In general, these systems have stricter requirements for predictability and often impose harsher limitations on the available resources for computation and storage. To support these environments, middleware such as the ADAPTIVE Communication Environment (ACE) [4] and the ACE Object Request Broker (ORB) (TAO) [5] have both been designed with customisability in mind and have been subsetted extensively. However, current techniques for subsetting middleware such as ACE and TAO have numerous shortcomings:

1. Standard subsetting techniques such as the use of macros to include code selectively, or the use of design patterns [6] such as the Strategy or Template Method, require *a priori* knowledge of customisation points.
2. As a result of adding macros and strategising customisation points, the basic functionality of core code can be obfuscated by the customisation infrastructure. This complicates program maintenance and evolution.
3. The Strategy and Template Method patterns introduce code that is present at runtime, even for those features *excluded* from a particular runtime configuration. At best, this introduces an insignificant amount of runtime overhead, especially if this customisation point is in a performance-critical loop, or if the call is made to another software context (such as across shared libraries); then the additional method-call can impact system performance and predictability.

To address these shortcomings, in developing the 'framework for aspect composition for an event' channel (FACET), we take an alternative approach to developing middleware – we use *aspects* [7] to introduce features incrementally. In this process, we aim to make features as independent as possible; this allows middleware users to reconfigure the software to meet their needs without adding any extraneous interfaces, code bloat or null strategies. At a high level, each feature is implemented as one or more aspects. Through the use of a configuration file, the appropriate aspects can then be woven through the base middleware code (and possibly user code) to achieve the desired configuration [8].

A key challenge when subsetting middleware is to provide a mechanism to identify dependences between features and to signal errors when two mutually exclusive features are selected. The most common technique for solving this problem is to use conditional compilation macros to check for all possible violations. This method is error prone, due to the amount of manual work involved. Additionally, as shown in FACET, aspect-oriented programming (AOP) allows one to develop many fine-grain features, making the feature-management problem more severe. To solve this problem, we present a feature-management framework that automatically validates feature configurations and simplifies management of the features' dependences.

Reliability is always a concern when developing software, especially for embedded or realtime systems that may be located in remote locations or perform safety-critical

© IEE, 2004

IEE Proceedings online no. 20040923

doi: 10.1049/ip-sen:20040923

Paper first received 3rd June 2003 and in revised form 12th May 2004

The authors are with the Department of Computer Science and Engineering, Washington University, St. Louis, MO 63130, USA

tasks. An important tool to ensure software quality is the creation and automated execution of unit tests. Additionally, to ensure the quality of customisable middleware, not only should every feature be validated, but also all meaningful *combinations* of features should be checked to identify unintentional interference between features. A naive approach of exhaustive feature enumeration is intractable, since the number of (valid and invalid) feature combinations grows exponentially. However, by using the feature-management framework in FACET, it is possible to identify only those feature combinations that produce a *viable* configuration. By reducing thorough testing to an automatic, relatively efficient process, it is likely that software developers will perform testing routinely and frequently, thus shortening development time and increasing the reliability of the delivered middleware.

An additional issue that arises when enabling testing over all combinations of features is when one feature changes the expected behaviour of another feature's unit tests. This issue actually arises frequently, typically because an enbabled feature requires that some additional work be performed at initialisation, or before uses of some core functionality. Traditionally, testing techniques for middleware overlook testing in this area due to the added complexity for allowing for these cases. The complexity of feature interaction is often sufficiently daunting that developers tend to bundle sets of interactive features without testing their interactions. However, by taking advantage of AOP techniques, FACET can automatically update unit tests to handle modifications to core functionality by other unrelated features.

To further demonstrate the ease and efficacy with which AOP techniques developed here can be used to finely control crosscutting features like the transport layer (CORBA), we report our experience in abstracting the use of CORBA in FACET. In addition to functionality, aspects also govern systemic concerns like whether the event channel makes use of CORBA interfaces and an ORB, or not (in the case distribution and inter-language support are unnecessary). We use this to evaluate the effect of CORBA on the footprint and throughput of the event channel.

In this paper, we describe our use of AOP to identify the core functionality of a middleware framework and then to codify all additional functionality into separate aspects. The advantage of using AOP is that the hooks and callbacks required for subsetting (using standard, object-oriented techniques) disappear, obviating the need to preconceive where points of variation are needed in the code. The patterns described in Section 5 make achieving these advantages in AspectJ easier.

Desirable combinations of features are selected by middleware users, based on the minimum functionality needed to support a given application. Feature dependences add complexity for both the middleware user and developer. We address this issue in Section 4.2 by providing a framework to manage features, their relationships, and by integrating this knowledge into the build environment.

2 Background

In this Section, we provide a brief overview of DOC middleware such as event channels and introduce some terminology related to AOP and AspectJ.

2.1 DOC middleware

Developing large software projects is notoriously difficult. Programming platforms vary widely, outdated and unwieldy programming interfaces abound, and frameworks for addressing communication issues either may not be available or may not be interoperable. It is for these types of problems that middleware has proven to be very useful in practice [9].

DOC middleware is a specific category of middleware that addresses the many accidental and inherent complexities [9] of network and distributed programming. Accidental complexity refers to the programming issues with using tools, languages, interfaces and frameworks that are difficult to use and prone to errors. Network programming has historically been difficult due to the lack of availability of anything besides low level socket interfaces. On the other hand, inherent complexities arise out of inherent difficulties with developing any program in the domain regardless of language, tools or libraries. For networking, these include issues such as fault tolerance, security, concurrency and program distribution.

ACE [4] and TAO [5] are two of many examples of DOC middleware frameworks that address the difficulties of distributed network programming. Both of these frameworks have matured over many years of use for both research and industry applications [10, 11]. Issues identified during their development and evolution, though, have led to current research and this work.

FACET is an implementation of a CORBA [1] *event channel* that uses AOP to achieve a high level of customisability. Its functionality is based on features found in the Object Management Group (OMG) Event Service [12], the OMG Notification Service [13, 14], and the TAO Real-time Event Service [15, 16].

An event channel is a common middleware framework that decouples event suppliers and consumers. The event channel acts as a mediator through which all events are transported. Figure 1 shows the main participants in an event-channel framework. At its simplest:

- suppliers push events to the event channel
- the event channel applies any filtering, correlation or other specified features to the events
- the event channel pushes appropriate events to consumers.

Event channel implementations differ in the types of events that they handle and in the processing and forwarding that occurs within the channel.

2.2 AOP and AspectJ

Separation of concerns [17] is the general term given to the process of identifying and encapsulating related ideas and concepts together. Separation of concerns for object-oriented programming (OOP) involves identifying the structure of classes and interfaces that define an application. However, separating concerns based on structural elements is only one of many dimensions where separation can occur. The inability of OOP to separate other concerns has led to significant research in identifying new approaches [7, 18–21]. These approaches are collectively termed Advanced Separation of Concerns (ASoC), due to their ability to enable more flexible separations. As a result, separation of concerns is no longer an abstract concept; it has become linguistically real in aspect-oriented software development (AOSD) languages and tools, to the extent that software units can reflect separate concerns [22].

Before describing the languages and paradigms used to encapsulate nonstructural concerns, it is useful to describe other types (or dimensions) of concerns. These can be broadly categorised as *systemic* and *functional* concerns [23]:

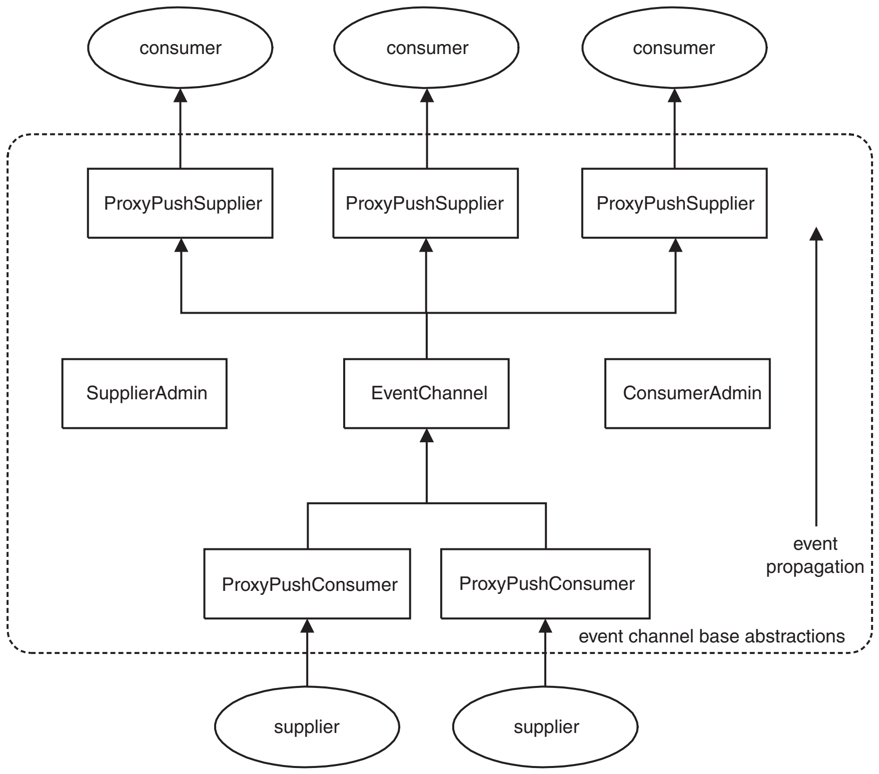

Fig. 1 *Main participants in an event channel*

• Systemic concerns include synchronisation, realtime, scheduling, transaction semantics, caching and prefetching strategies and memory management concerns.
• Functional concerns comprise application logic and features. These differ from systemic concerns in their scope and intention. For example, an application logic such as a new business rule may effect several computations and decisions in separate classes, but a systemic concern such as synchronisation affects many classes systemwide.

Both of these types of concerns crosscut many classes, and by encapsulating them into separately compilable units, one can selectively enable or disable their behaviour. A key observation is that software requirement tracability is much more apparent for languages that support separation of nonstructural concerns [20].

AOP [7] is a software development paradigm that enables one to separate concerns that crosscut sets of classes and encapsulate those concerns in self-contained modules called *aspects*. Since AOP's introduction, several projects and tools have evolved that embrace AOP's goal of separation-of-concerns [23–26].

The AspectJ [27] programming language adds AOP constructs to Java [28] and uses the following terminology. Within an aspect, the locations at which *advice* should be applied are defined using *pointcuts*. Each pointcut is made up of one or more *joinpoints*, which are well defined locations in the execution of a program. The code applied at a pointcut is called *advice*. In addition to applying advice, languages supporting AOP often allow new methods or other language features to be *introduced* into existing classes. Of all the separation of concerns languages suitable for developing middleware, AspectJ is currently the most mature and was thus selected for the experiments documented in this paper.

3 Related work

The need to subset (or extend) software systems selectively has existed for some time [29], and many areas of AOSD provide successful approaches. In this Section, we examine some of the current techniques for subsetting functionality and consider the case study of subsetting Real-Time CORBA 1.0 (RTCORBA) from TAO. We also summarise and compare our efforts with feature-oriented programming (FOP) – a compositional approach to product-line generation that offers similar advantages.

3.1 Subsetting techniques

Developing middleware to be flexible in diverse environments often involves the following process [30]:

1. building flexibility and extensibility around known variation points at the beginning.
2. refactoring functionality out of the core middleware to extensions and adding flexibility as new features are added and as the footprint becomes too large.

Unfortunately, the first relies on a designer's ability to preconceive feature extension points. As this is impractical with all but the smallest frameworks, functionality inevitably gets added that will need to be refactored to extensions later. Quite a few object-oriented design patterns have been identified that document successful strategies to subsetting middleware. These include patterns such as Strategy [6], Interceptors, Extension Interface, Service Configurator and others [31]. Although such patterns have been used extensively in middleware such as ACE [4] and TAO [5], these come with some limitations:

1. They require additional infrastructure within the framework to support their presence. For example, Strategy and Interceptors require method invocation hooks to be placed at key locations throughout the code. From a programmer standpoint, these hooks and the additional infrastructure lessen the readability and maintainability of the code.
2. If the locations where subsetting should have occurred are not preconceived, time-consuming refactoring may be needed to extract functionality into separate libraries.

3. The hooks and infrastructure themselves can lead to degradations in performance and increase in footprint size if some or all of them are not used.

Case study: subsetting TAO: TAO is a full-featured CORBA ORB developed by the DOC Group at Washington University and University of California at Irvine. Although much of it was designed to be configurable from the beginning, it has nonetheless grown to the point that its footprint size has become too large for many embedded systems (as well as some desktop systems). As a result, functionality has been subsetted from its core many times. In fact, this is an ongoing effort, as TAO is increasingly considered for use in environments with tighter memory constraints.

One feature recently subsetted from TAO is support for the RTCORBA specification [1]. RTCORBA defines standard mechanisms that allow applications to control the priorities at which CORBA requests are processed and how threads are allocated internally in an ORB. Many applications do not need the features of RTCORBA, and developers of such applications find that the overhead in footprint and processing of those features is burdensome.

The time and effort required to subset RTCORBA features from TAO was significant: it took two expert ORB developers nearly five months' time to refactor code throughout the ORB and its associated libraries, and to verify its operation on all supported platforms.

After RTCORBA was removed, the size of the core library was reduced by $\sim 10\%$, and many method calls were removed from the critical path of the ORB, resulting in a small but noticeable performance improvement.

A benefit of this process is that the core TAO code became more extendable for future features. However, in addition to having to go through the tedious subsetting process, the resulting TAO code is now:

1. more complicated due to the additional strategy classes and RTCORBA interception points
2. not as fast as possible due to the overhead of maintaining the hooks for RTCORBA even when RTCORBA is not being used
3. suffering from additional overhead from new calls between the core code and the RTCORBA library when RTCORBA is in use.

3.2 Other additive approaches

While subsetting (subtractive) techniques can be tedious and error-prone, this paper documents our experience with an additive approach based on AOP. In a sense, we use AOP for reasons advocated by the more general notion of FOP, in which the layering and relationship of a program's features can be more directly expressed. In this Section, we consider other additive approaches, such as FOP, mixins and generative component modelling. While those approaches offer certain advantages over (our use of) AOP, we explain why AOP currently provides greater expediency in achieving our results.

Feature-oriented programming: FOP [32] is a programming methodology that advocates program construction by feature composition. Dijkstra's notion of step-wise refinement, through which a program is developed in careful stages or evolved to include new features, is expressed in FOP using convenient syntax and formalised using algebraic equations so that it can be applied on a much larger scale [33]. A feature is truly crosscutting, so that feature composition has the effect of refining multiple classes by introducing fields and methods that support the feature. FOP has also considered the effect of feature composition on noncode artefacts such as documentation, testing and a program's interface definition language (IDL).

Batory *et al.* have proposed an algebraic model for FOP that resembles the *mixin* specifications described below. However, their composition operator applies to an entire program – not just a single class, as is the case with mixins. As with AOP, the FOP composition results in a form of weaving applied to an entire program. In comparing FOP with our use of AOP to achieve essentially the same results, the following issues arise:

- As discussed in Section 4.2, FACET's features are not independent. Some features depend on the inclusion of other features; some features exclude other features. Of the approximately 8 million potential feature compositions, only 10 000 of them make sense when feature dependences are taken into account. While FOP offers the potential for feature-oriented language extensions in which feature dependences could be specified, and the algebraic model of Batory *et al.* could be extended to include such information, such mechanisms are not currently available at the level of maturity of the AspectJ tool.
- Because FACET can be composed for deployment with or without CORBA, there are non-Java artefacts whose behaviour must be compositionally altered based on feature inclusion. Specifically, CORBA requires IDL that exposes the services offered by FACET at the ORB level. FOP and the model-generative techniques described below can service noncode artefacts. Currently, our need to modify Java and IDL is best served by using AspectJ to weave IDL-creating code into FACET and staging the configuration so that IDL is generated before FACET is fully customised. Again, this a tool-maturity issue and there is no barrier to creating FOP tools that can accomplish this particular task.
- The premise of refinement is that construction of a program is accomplished by a sequence of steps. In FOP, the steps are feature-related, so that the resulting program is composed through addition of a specific set of features. If the features are relatively independent, then the order in which features are added may not matter.

  However, with extant FOP systems, the set of features is introduced in a particular order – the *steps* of the step-wise refinement process. Consider features *A* and *B* that are introduced in that particular order. While *B* can affect the behaviour of *A* in this composition, *A* cannot actively affect *B* unless *B* chooses to customise its behaviour based on *A*'s presence – a violation of the 'separation of concerns' doctrine.

  For our purposes, AOP (in particular, AspectJ) allows a *set* (instead of a sequence) of aspects to be specified up-front; the aspects are then applied in an order commensurate with the effects of the aspects' application to the base code and to each other.

  In summary, FOP has been applied to generating product-line architectures [34, 35], and is thus certainly the methodology of choice for FACET. While our approach is consonant with the goals of FOP, we achieve similar results using AOP. Our choice of AspectJ is based on the maturity of that tool and its ability to weave a set of aspects as described above.

Other approaches: Mixins [36] offer a limited implementation of FOP by way of class *functors*: a mixin specification transforms a class (to an extended class) by introducing or overriding the class's fields and methods. A class is initially specified in its minimal (feature-free) form. Class transformations are then applied to add features. If a given class could offer a large set of relatively orthogonal features, then mixins nicely anticipate any combination of those features and promote reuse at that level. Mixins do not offer the level of joinpoint specification seen in AspectJ and other AOP languages. Moreover, mixins do not directly express the kinds of crosscutting (multiple class) issues that are advocated by FOP and seen in recent FOP implementations [33].

At a much higher level, aspect-oriented domain modelling (AODM) is emerging as a mechanism for reaping the benefits of FOP for component modelling. With this approach, noncode artefacts (such as real-time and footprint concerns) are expressed at the modelling level. Program generators and weavers then synthesise an application by making appropriate choices and transformations of classes. The relationship of joinpoints and advice in AOP (using AspectJ) and AODM has been considered [37]. AODM essentially takes AOP a step further by automating the specification of joinpoints and advice in the context of the larger, noncode concerns.

4 FACET

We next present the design and architecture of FACET and show how the use of AOP alleviates many of the problems experienced with *ad hoc* subsetting techniques.

4.1 Feature decomposition

FACET is separated into a base and a set of selectable *features*. The base represents a fundamental and indivisible level of functionality. Each feature adds a structural and/or functional enhancement to the base or to other features. AOP language constructs are then used to integrate or weave feature code into the appropriate places in the base or into other features [38].

The base consists of a simple implementation of interfaces similar to those found in the CORBA Event Service with a few caveats. The *pull* interfaces and their implementation are not included in the base, since they are much less frequently used. Also, since the event payload-type varies with each application, it too has been relegated to a feature.

To support functionality not found in the base, FACET provides a set of features that can be enabled and combined subject to some dependence constraints. These features include:

1. interfaces and implementation to support *pulling* events through the event channel
2. various event payload types such as CORBA Any's, CORBA octet sequences and strings
3. event structures such as headers that are made visible to the event channel and used by other features. These include event-type labels for dispatch and filtering, a time to live (TTL) field to support federated event channels, timestamp fields for profiling, and others
4. dispatch strategies that trade off channel performance and memory usage
5. event correlation support that allows consumers to specify sequences of events that should be received by the channel before notification
6. event channel profiling and statistics generation
7. tracing hooks to aid application debugging.

The construction of FACET follows a bottom-up approach in which features are implemented as needed. As new requirements are presented, they are decomposed into one or more features. In the case of FACET, the requirements for several existing event services were selected one-by-one for incorporation. By using AOP techniques, the code for each of these features can then be weaved into the base at the appropriate points.

Although FACET offers 23 user-specifiable features, not all combinations of those features make sense, as discussed in Section 4.2. Because FACET is an event channel, and the features of an event channel are well specified [39], we formulated aspects intially to mimic the exposed features of an event channel. As we discovered commonality and opportunities for reuse, we factored features into the abstract features described above. Because AspectJ generates relatively efficient code as an output of its weaving process, we were not concerned with any bound on the minimal size of an aspect nor the size of a joinpoint set in our code.

Some features, such as TTL do not heavily crosscut FACET's classes; however, TTL's inclusion does crosscut heavily into the test cases, as discussed in Section 4.4. Other, more classical applications of aspects, such as logging, profiling and statistics-generating, crosscut most of the code in FACET. In between are features such as dispatching strategies and pull (instead of push), which affect many but not all of the FACET classes.

In addition to the base and features, Fig. 2 illustrates three other major components in FACET. The feature registry maintains all of the relationships and metadata concerning every feature. It has the responsibility for validating event-channel configurations and providing dependence relation information to the other components. The build system is then responsible for selecting and compiling the appropriate source files that correspond to the desired feature configuration.

Finally, the test framework has the responsibility of verifying that each feature and its compositions perform actually as intended. It is used to gain a high level of confidence that changes to the base or to other features that do not have unintended consequences in any configuration.

4.2 Feature management

The feature registry maintains all of the relationships between features and provides interfaces to query those relationships. All of the functionality provided by the registry can be adapted easily to be reused in other contexts.

Nearly every part of FACET takes advantage of the feature registry, including the build environment, test environment and statistics-collection framework. Additionally, every feature must interact with the feature registry to

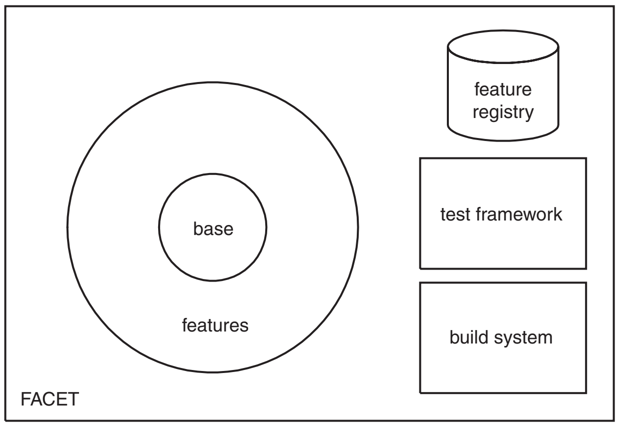

Fig. 2 *Main components in FACET*

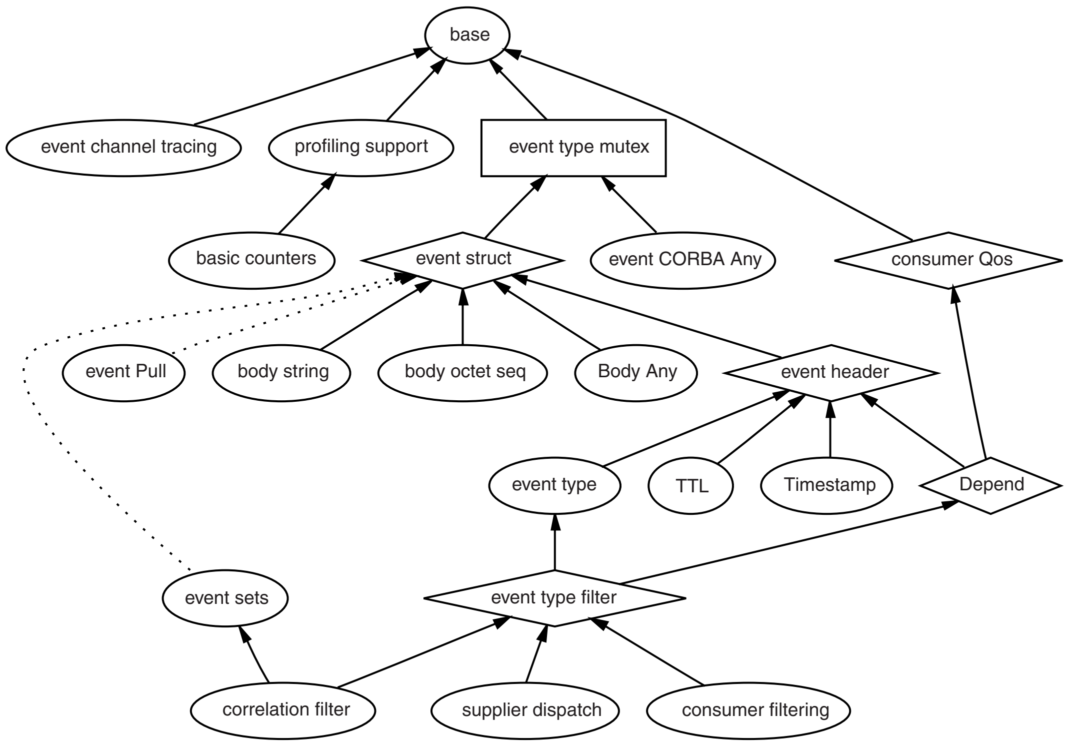

Fig. 3 *Feature dependence graph*

inform it of its requirements. In this sense, FACET provides a higher level AOP meta-programming framework for middleware [8].

Features in FACET can relate to each other and to the base in several different ways. These relationships are important, since they essentially determine valid configurations of the middleware. Fundamentally, each feature can be assigned one of the following properties based on the usage requirements of the feature:

1. *Concrete features:* These features can be included in any configuration given the stipulation that any feature that they depend on is also included.
2. *Abstract features:* These features provide a structural or functional enhancement that cannot exist on its own. A concrete feature must augment this feature for the configuration to be valid.
3. *Mutual exclusion features:* These features do not necessarily provide any structural or functional enhancements but are used to allow only one of a set of features to be enabled at one time. For example, the type of event passed through the FACET may either be a structure or a CORBA Any, but not both at the same time.
4. *Inferred features:* These features exist only within FACET and are created when one feature refers to a nonexistent feature. That nonexistent feature is *inferred*, and if it is never loaded, it signals a configuration error.

The base is modelled as a concrete feature with no dependences. All other features depend on each other or on the base. It is always possible to reach the base from any feature following the dependence relationship. There are two types of dependence relationships [30]:

1. *Depends:* Most relationships between features are of this type. A feature that depends on another cannot exist in a valid configuration unless all of its dependences are also part of the configuration.
2. *Contains:* Some features create or use data structures that *contain* data structures introduced by some other feature. These features still depend on the presence of the other feature but cannot be used to fulfil dependence requirements of that other feature. An example of this is the *pull* feature that allows users to be able to pull events through the event channel. This feature does not care what kind of event is used, but it does care that an event type feature has been enabled [Note 1]. Since enabling the pull feature does not specify an event type, the depends relationship cannot be used, and it is said that the pull feature contains the event feature.

By using the depends and contains relationships to connect features together, a directed acyclic graph (DAG) can be created and analysed to validate a given feature configuration. It should be noted that, although cycles can be created when features depend on each other in a loop, this representation assumes that it is always possible to combine the features in the cycle together or to factor out functionality to break the cycle. Our experience with FACET indicates that this is not an issue.

Figure 3 shows the relationships between the base and the current features implemented in FACET. In the feature dependence graph, oval nodes are concrete features, diamond nodes are abstract features, and rectangular nodes are mutual exclusion features. Nodes that are related by the depends relationship are shown with a solid arrow, and those related by the contains relationship are shown with a dotted arrow.

Figure 4 shows the overall system architecture of FACET. Each feature is represented as an AspectJ file in our system. The feature selections are analysed early in the build process to determine their validity, according to the feature dependence graph described above. If the specified features are valid, then feature closure is computed

Note 1: Theoretically, the pull feature could be implemented to handle the case where no event type has been selected. However, this case has very limited usefulness compared to the amount of complexity added to the feature code.

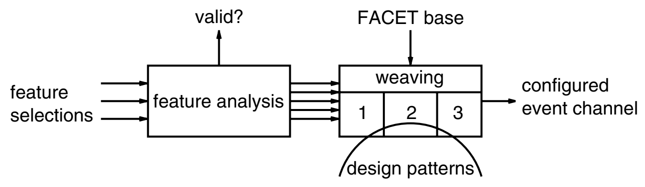

Fig. 4 *FACET system architecture*

The three-stage weaving process is described in Section 4.6, and the patterns applied to resolve implementation difficulties are described in Section 5

according to the dependence graph, and the resulting set of aspects is submitted for weaving, as shown in Fig. 4 and explained in Section 4.6.

For the feature registry to manage feature dependences, every feature must register its dependences at initialisation. This is accomplished using the template advice pattern (see Section 5.2), so that the registration is performed in the feature as opposed to a centralised location. This has the advantage that the feature metadata (a feature concern) is kept with the feature implementation.

4.3 Build environment

When developing customisable software, the build environment is usually charged with determining what features are included in the delivered library or executable. For FACET, the environment has an even greater importance, due to the large number of features that are available. If the build environment does not provide a logical, simple configuration mechanism, it will likely frustrate potential users [8].

The FACET build environment was designed with several goals in mind:

1. Feature identification should be as automated as possible. For example, adding new features to FACET should involve a minimal number of changes.
2. The environment should be portable.
3. The resulting code should be validated against unit tests for all selected and dependent features.
4. The user should be able to specify only those features that are directly needed. Dependent features should be added automatically.

Of these, portability is achieved by using the ant build tool [40], which is essentially a *make* utility designed specifically for building Java applications.

Features are enabled and disabled by using a configuration file. Figure 5 shows one such configuration file. Each listed directive corresponds to a desired feature that is identified by 'use_' and the identifying part of the package for that feature.

The build system automatically includes all dependent features for those that are listed. This greatly reduces the chance of specifying invalid FACET configurations. It does not completely eliminate it, though, due to the *contains* relationship [8].

Feature-set testing and verification uses information found in the feature registry and is described in the following Section.

4.4 Testing framework

Software verification is necessary and important for any application. Proper testing is even more important for

```
use_feature_event_pull=yes
use_feature_tracing=yes
use_feature_eventtype=yes
```

Fig. 5 *Example configuration file*

FACET than for many other software projects for two main reasons:

1. FACET supports a large number of different configurations of features that interact with each other in numerous ways. Validating a subset of legitimate configurations does not guarantee that every configuration will work or even compile.
2. It is difficult to verify that a change made to the base or a feature does not remove or change the semantics of a joinpoint used in another feature.

jUnit [41] is a commonly used framework to automate the regression-testing process for Java applications. Its application programming interface (API) provides various methods to validate code and report errors. Additionally, it comes with GUI and text based tools that can run one or more tests, create testing reports and quickly summarise test-run results. FACET uses jUnit as the basis for its test framework, and by default, the build system invokes a jUnit test runner to execute all relevant tests for a configuration after every build.

Although jUnit is very useful, it does not address several issues that arise when developing highly reconfigurable middleware. These issues include:

1. support for automatically running tests that correspond to the set of features that were enabled
2. support for upgrading tests written using one configuration to work under another configuration that includes other features.

Both of these issues are addressed in FACET by using AOP techniques. The latter issue, in particular, would have been very difficult to support with standard object-oriented techniques, but with AOP it is relatively simple [8].

Running the appropriate subset of tests: The main requirement for this issue is that when validating an event channel configuration, every test associated with every feature within that configuration must be run. Several options are possible for achieving this:

1. Create a jUnit test suite to call each appropriate test. This is the standard jUnit method for running more than one test. It has the major disadvantage that FACET has thousands of configurations that can be selected. Writing test suites would have to be automated to be practical.
2. Modify the build system to search the directory hierarchy for Java source files that begin with the word *Test* and invoke a jUnit test runner on each of them. Unfortunately, this option adds complexity to the build system, is slow since it has to launch a new JVM for each test, and does not allow jUnit to summarise the results.
3. Use an aspect to automatically register each test case with a well known test suite.

By using standard object-oriented techniques, it is possible to come up with another option that sounds promising but actually does not work in Java. This is to create a static registration method to add test cases to the main test suite and then to add a *static* block to each of the test cases that has a call to this method. This does not work, since Java does not run *static* blocks until class load time, and if no other code references the class, the *static* blocks will never be run. Thus, the third option above is by far the most desirable one.

In FACET, aspects are used to add unit tests automatically by using the template advice pattern (described in Section 5.2.) The basic idea behind this is to provide an abstract aspect that encapsulates the knowledge of where

```
import junit.framework.TestSuite;

public abstract aspect TestSuiteAdder {
    abstract protected void addTestSuites(TestSuite suite);

    private pointcut addTestSuitesCut(TestSuite suite) :
        call(void AllTests.addTestSuites(TestSuite)) && args(suite);

    before (TestSuite suite) : addTestSuitesCut(suite) {
        this.addTestSuites(suite);
    }
}
```

Fig. 6 *TestSuiteAdder aspect*

and when unit-test registration should occur. Feature-specific unit tests merely subclass this aspect and provide a suitable implementation of the registration method.

Figure 6 shows the code for the TestSuiteAdder abstract aspect and Fig. 7 shows a common use of the TestSuiteAdder.

Automatically upgrading tests: To validate the operation of FACET, we would like to run every unit test for every feature successfully. However, in certain cases, combinations of features can easily break unit tests. An example of this problem is to consider the interaction between the event type feature and the TTL feature.

Enabling the event type feature simply adds an event type field to the EventHeader structure. A unit test for the event type feature may send events between suppliers and consumers and test whether the values stored in the field arrive unchanged at the consumers. Such a unit test would have no knowledge of the TTL feature (nor should it). This is illustrated in Fig. 8.

The TTL feature adds a TTL field to the EventHeader structure and adds code to decrement and check it as events pass through the event channel (show in Fig. 9). A unit test for the TTL feature may check that events with a TTL of zero get dropped and events with other TTL values arrive at the consumer with their TTL decremented. Note that the TTL field must be set by the supplier or it will receive the default value of zero.

In configurations that have only one of these two features, their unit tests will work. The problem arises when both features are combined in one event channel. Since the `update_ttl` method introduced by the TTL feature disallows events with a TTL value of zero, event type feature unit tests break since they do not set the TTL field in the EventHeader to a positive value.

One solution is to write an aspect to intercept executions of the EventHeader constructor and set the TTL field to a nonzero default. This solution has two main problems. The first is that not all feature conflicts can be resolved by modifying a default value. The second is that this change in processing can be seen by user applications. Since the Java IDL mapping specifications specify that the default field value is zero, this change makes the FACET API appear inconsistant with expectations associated with CORBA programming. Also, when the tests are not compiled with the FACET library, the code to automatically set the TTL field may not be included, which may then break user code that relies on this behaviour.

The solution to this problem is to use the interface tag pattern (described in Section 5.3) to selectively mark the features that a test case supports, and use *upgrader* aspects to update unit test code to support new features.

The actual upgrader aspect specifies that any code that it modifies must be inside classes that implement Upgradable and do not implement its associated feature interface. Figure 10 shows the code for the TTL feature's upgrader. Other feature upgraders are similar. The upgradeLocations pointcut limits the applicability of the aspect to appropriately tagged classes. It is separated from the advice for clarity, since more than one advice block may need to use it. Following the pointcut, the advice in the TtlUpgrader intercepts all calls to the EventHeader constructor and initialises the TTL field to a sufficiently large number for any test.

4.5 Transport abstraction

FACET was originally designed to use CORBA so that events can be sent to and received from remote consumers and suppliers. The advantages of CORBA include the ability to distribute the consumers and suppliers as well as to fashion their implementation for any language that maps to CORBA (e.g. Java and C++). The language independence is obtained by specifying interface definitions via CORBA's IDL.

However, in certain usage scenarios where distribution and multi-language support is unnecessary, the use of CORBA becomes unnecessary. As described in previous work [42], there are important examples of this case. In this situation, the underlying transport mechanism can be a simple method call, doing away with the need to make use of an ORB. Indeed, one form of this optimisation is routinely used in the Bold Stroke event service, in the form of the Subscription and Filtering configuration [16].

We sought to provide a standard interface for the event channel while making the use of CORBA optional as well. In other words, merely by selecting an EnableCorba or DisableCorba feature (which are mutually exclusive since it would make no sense to enable both) at build-time, it

```
public class TestEventTtl extends EventChannelTestCase {

    /* Test case implementation */

    static aspect AddTests extends TestSuiteAdder {
        protected void addTestSuites(TestSuite suite) {
            suite.addTestSuite(TestEventTtl.class);
        }
    }
}
```

Fig. 7 *Typical use of TestSuiteAdder*

```
//
// Called by the event channel whenever an event is received.
//
public synchronized void push (Event data)
{
   int eventNum = data.getHeader().getType();

   Assert.assertEquals (eventNum, EVENT_TYPE);
}
```

Fig. 8 *Event type unit test*

```
aspect TtlAspect {

        //
        // Update the TTL for the event.
        // If the TTL is still ok, return true.
        //
        boolean update_ttl (Event event)
        {
                event.getHeader().setTtl (event.getHeader ().getTtl () - 1);
                return event.getHeader().getTtl () >= 0;
        }

        //
        // Update the TTL, and possibly drop the event if it gets too low
        //
        void around (EventCarrier ec) :
                call (void EventChannelImpl.pushEvent (EventCarrier))
                && args (ec)
        {
                if (update_ttl (ec.getEvent ()))
                        proceed (ec);
        }
}
```

Fig. 9 *Code inserted by TTL feature*

should be possible to obtain an event channel with the desired configuration. Again, the use of AOP techniques greatly simplified this task. Through the use of placeholder class and method patterns [42], it was possible to write the implementation of the event channel in a manner which was independent of whether the interfaces were plain Java interfaces or interfaces generated from IDL.

4.6 Weaving and IDL generation

When different features are enabled in FACET, the IDL of the event channel needs to change accordingly to reflect the presence of those new features, in the case CORBA is enabled (there is no need to generate IDL when CORBA is disabled). In previous work, the changes to the IDL of the event channel were conducted by the use of scripts, which make the changes using primitive text processing and search-and-replace style techniques [38]. However, for our purposes, using such a script would entail having to use different aspects for the cases when CORBA is enabled and when it is not. From a software engineering standpoint, this would be very inefficient and greatly reduce the maintainability of the code.

To address this, we chose to investigate a new technique which involves the generation of the IDL for a given configuration by *reflection* on the classes comprising the event channel's interface. With this, it is only necessary to specify the aspect introductions in the Java code – the corresponding changes to the IDL happen automatically since the mapping from CORBA to Java is well known [42].

The generator is run as part of the three-stage weaving process depicted in Fig. 4:

```
aspect TtlUpgrader {

    pointcut upgradeLocations() :
        this (Upgradable) &&
        !this (TtlFeature);

    after () returning (EventHeader header) :
        call (EventHeader.new()) &&
        upgradeLocations () {
        header.ttl = 255;
    }
}
```

Fig. 10 *Upgrader aspect for TTL feature*

1. Aspects that perform introductions are applied to the classes which form the public interface of the event channel.
2. The IDL generator reflects on these classes and generates the IDL. The IDL compiler is then run to generate the stub and skeleton classes. In the case CORBA is disabled, this step is automatically skipped.
3. All classes comprising the event channel along with the relevant aspects for the particular feature set are compiled and the relevant jUnit [41] tests are run [38].

The IDL Generator we have developed is generic in its implementation and can be used to generate the IDL interfaces corresponding to any set of Java classes. Conceivably, this technique can be extended to any language which has a strong runtime type system and allows reflection.

5 Aspect oriented design patterns in FACET

Our aspect-oriented approach to feature accrual in FACET is conceptually accomplished as shown in Fig. 4. However, due to the nature of AspectJ and to the interaction of various features, the following problems arose repeatedly in the implementation:

- Feature introduction can cause the API of a given method to widen, so as to accommodate information related to that feature. The base of FACET is intentionally agnostic about all features, and the API must remain consistent as features are introduced. This recurring difficulty is resolved using the *encapsulated parameter* pattern described in Section 5.1.
- It is expected that code and feature development and modification would continue after FACET's initial design and implementation. In AspectJ, advice is applied at specified pointcuts, yet the nature of a pointcut can become unstable as code is modified: types may change, method names may change, etc. To obtain a stable set of interception points where advice can be applied, we relied on the *template advice* pattern described in Section 5.2.

We next elaborate on these patterns and their application in FACET.

Design patterns are solutions to recurring problems [6]. Patterns are usually identified by reflecting on experiences from previous programming projects where a common problem has repeatedly arisen. Since AOP has only found its way into programming projects recently, there is a lack of significant experience from which to gather patterns that are useful for AOP systems. On the other hand, AOP techniques have been considered for instilling design patterns on extant code [43].

Over the course of developing FACET, many of the core interfaces and aspects had to be significantly refactored to surmount difficulties when implementing new features. The ability to add new features to FACET using AOP is critical to its ability to be precisely customisable for its users. As a result, much attention was focused on ways of using AOP mechanisms to enhance feature flexibility, scalability, efficiency and maintainability. The patterns identified in this Section represent the most common and useful of those found in developing FACET [8].

5.1 Encapsulated parameter

5.1.1 FACET context:

In addition to introducing code, features in FACET have the effect of passing around extra parameters that the base has no knowledge of. For instance, the event type parameter requires that the an extra parameter named 'type' be passed around the event channel and eventually be delivered to the consumer. Accomodating this extra parameter would ordinarily involve updating all the methods that are responsible for delivering event data with this new parameter – a task that can be accomplished using aspects but which is tedious, nevertheless.

The encapsulated parameter pattern solves this problem by making the additional data, members of a structure (or Java class) that is used consistently by all the relevant methods in the base. This way, the programmer's interface is unaffected by the addition of new features.

5.1.2 Intent:

Allow new features to add parameters to API calls while keeping the programmer's interface simple.

5.1.3 Motivation:

It should be possible to add parameters to API method calls to support new functionality added by using aspects without having to bloat the API with numerous overloaded versions of the same method. The following are possible approaches to solving this problem:

1. Introduce a new method to the API to set the parameter.
2. Introduce a new method with the extra parameter and add advice to the old method to call the new one with a default value.
3. Use a cflow aspect [27] to determine parameters from the context of the caller.
4. Hard code all possible method parameters to the base methods.

The encapsulated parameter pattern avoids the liabilities of the above approaches [8] by merely passing a structure (or Java class with public member variables) to API methods. Additional parameters can then be added to those methods by introducing new member variables to the passed structure. If a parameter has a default value, it can be initialised in the constructor for the parameter class, so that the user does not need to set it.

5.1.4 Applicability:

Use the encapsulated parameter pattern when:

1. a method call will need to be passed by additional parameters to support functionality added using aspects.
2. the new parameters to the method are most logically set when that method is called.
3. the parameters cannot be determined from the calling context or the calling context is unknown.

5.1.5 Structure:

Figure 11 shows the structure for the encapsulated parameter pattern. Since unified modelling language (UML) [44] does not currently support the depiction of AOP interactions, these interactions are shown using stereotypes for advising and introducing functionality to existing classes.

5.1.6 Participants:

- *Encapsulated parameter:* holds parameters introduced by the enhancement aspect.
- *Invoker:* initialises an encapsulated parameter instance and passes it to the appropriate API method.
- *Enhancement aspect:* introduces parameters to the encapsulated parameter and adds advice to use the parameters in the internal class.
- *Internal class:* one or more classes that are not directly accessable through the API facade.
- *API facade:* defines an interface to a part of the framework to the user.

5.1.7 Collaborations:

The invoker creates an encapsulated parameter object and passes it to a method in the API facade. The API facade then may pass the

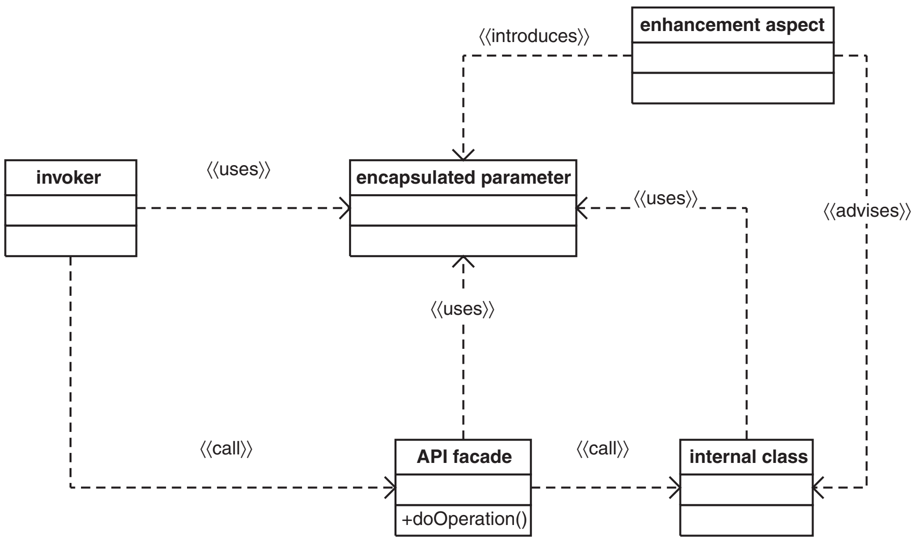

Fig. 11 *Encapsulated parameter pattern structure*

encapsulated parameter to other internal classes. One or more enhancement aspect instances can add new parameters to the encapsulated parameter class and then use those parameters internally.

5.1.8 Consequences:
The encapsulated parameter pattern has the following consequences:

1. It makes it possible to extend the parameters of API calls.
2. It simplifies the procedure of writing enhancement aspects, since parameters are easy to access.
3. It allows default parameters to be specified in the encapsulated parameter class to simplify the introduction of new enhancements.
4. It complicates the API somewhat, since parameters are not stored in the definition of the encapsulated parameter class.

5.1.9 Negative consequences:
The encapsulated parameter pattern has the following negative consequences:

1. It complicates the API somewhat, since parameters are not stored in the definition of the encapsulated parameter class and are introduced by aspect code elsewhere.
2. It affects code readability as it is not possible to tell from the signature of the method what exact data a method manipulates.

5.1.10 Known uses:
The encapsulated parameter pattern is used in several places in FACET:

1. *Event passing:* The Event class in FACET is itself an instance of the encapsulated parameter pattern. It is passed into a ProxyPushConsumer instance by the user to send the event, then passed through the event channel and finally through a ProxyPushSupplier instance to another user. The base code of FACET sends empty events, and features introduce fields into the Event class to hold a payload, headers, source and destination fields, etc.
2. *Consumer registration parameters:* The connectPushConsumer method to register consumers with the event channel takes a ConsumerQOS class to pass in quality of service (QoS), filtering and correlation parameters. Initially, this class is empty, and, for example, when filtering is enabled, parameters are added to this structure to specify what events are desired.

5.1.11 Related patterns:
The encapsulated parameter pattern has some similarity to the command pattern [6], since both patterns encapsulate data in a class that is passed like a parameter. The patterns differ in context, and also since the command pattern passes code in the class, and it is not meant to be extended through the use of aspects.

At the API interface, this pattern is similar to using named parameters in languages that support this. Named parameters can be specified in any order and can take on default values in the same way as the encapsulated parameter. If Java had support for named parameters, they could be used as an alternative to the encapsulated parameter pattern. However, passing parameters in structures as done in this pattern is convenient and is easy and efficient to pass around internally.

5.2 Template advice

5.2.1 FACET context:
Much of the aspect code in FACET's features relies on weaving code into key interception points in the base. To weave code in, one needs to specify precise joinpoints where aspect weaving needs to be applied. For instance, features need to subclass the TestSuiteAdder aspect and register their testcases by inserting registration code at a specific joinpoint. The specification of these joinpoints can, however, get quite tricky, especially when the code is unfamiliar and there is no guarantee about whether the structure that is assumed by the writer of the joinpoint shall be preserved in subsequent releases.

The template advice pattern solves this problem in FACET by making use of abstract aspects that supply predetermined pointcuts which can be used by advice without exposing private implementation details.

5.2.2 Intent:
Export key interception points to API users and extension developers and decouple advice from hard coded pointcuts.

5.2.3 Motivation:
One of the benefits of using AOP is that it provides a mechanism for extending existing code without explicit modification by using an aspect compiler to weave new code at the desirable joinpoints. This ability to break through layers of encapsulation to add crosscutting functionality is what makes AOP useful. However, specification of joinpoints can be very tricky, especially when the joinpoint applies to unfamiliar code. Additionally, if the base code is undergoing active development, that joinpoint may not exist in the next release. Even worse, the joinpoint may be reached in a completely different way in a subsequent release, causing the advice to behave unexpectedly.

AspectJ provides a potential solution to this problem by allowing pointcuts to be specified in abstract aspects and then *concretised* by sub-aspects. By using this mechanism, a core-code developer can specify an interception point by creating an abstract aspect with the appropriate pointcut. A user of that interception point can then create an aspect and inherit the pointcut. This approach still requires that the user know whether before, after or around advice should be used and if any preprocessing needs to be done to convert parameters from the pointcut to an appropriate *external* form.

5.2.4 Applicability:
Use the template advice pattern when

1. An interception point may be used by many aspects.
2. It is desirable to decouple the join point and advice location from the actual advice implementation. This may be the case if the base and feature code are developed independently.
3. Common processing is needed to adapt internal variables, parameters and the join point to an exportable form.
4. It is desirable to expose interception points as part of an aspect oriented API.

5.2.5 Structure:
Figure 12 shows the structure of the template advice pattern.

5.2.6 Participants:

- *Abstract aspect:* defines a pointcut and a skeleton advice implementation that calls abstract methods (doAction) to be filled in by the concrete aspect.
- *Concrete aspect:* implements the logic that should be applied at the interception locations defined by the abstract aspect.
- *Internal class:* contains the pointcuts defined by the abstract aspect and receives the advice from the concrete aspect.

5.2.7 Collaborations:
The concrete aspect relies on the abstract aspect to execute its methods at the appropriate interception points.

5.2.8 Consequences:
The template advice pattern has the following consequences:

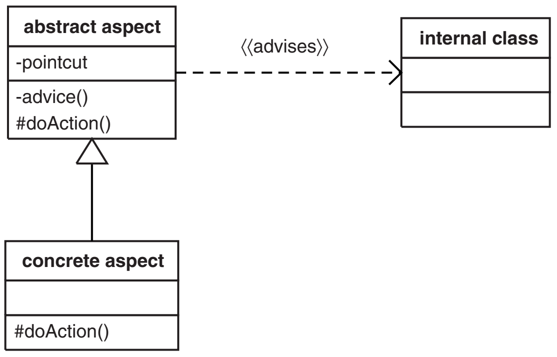

Fig. 12 *Template advice pattern structure*

1. Decouples the pointcut and advice location from the actual implementation of the advice. This adds the flexibility to change internal code without worrying about breaking important pointcuts and retains the advantages of being able to use aspects.
2. Simplifies the extension of a framework by exposing common interception pointcuts.

5.2.9 Negative consequences:

The template advice pattern has the following negative consequences:

1. Can limit the parameters accessable to the concrete aspect's implementation. This can be advantageous since it reduces the number of variables that need to be considered when extending a framework.
2. Requires that the abstract aspect's designer be able to preconceive all possible pointcuts of interest since it is not possible to extend it without breaking encapsulation.

5.2.10 Implementation:

Consider the following issues when implementing the template advice pattern:

1. *Use access control:* Like the template method pattern [6], access control can prevent unintended uses of pointcuts and advice. For example, the pointcut should be declared as private, and the abstract methods in the abstract aspect should be protected.
2. *One aspect per pointcut:* To reduce the complexity of using the template advice pattern, define one abstract aspect per interesting pointcut. Most likely, only one abstract method will be needed for the implementation of the advice.
3. *Provide access to enough parameters to advice implementation:* In order for the concrete aspect to implement its advice, it will need some parameters from the interception point. An implementation must decide how many internal details it should reveal to the concrete aspect.

5.2.11 Known uses:

The template advice pattern is used in FACET to register features with the FeatureRegistry. The FeatureRegistry has an empty method that it calls whenever it needs to build a list of the features in the system. As shown in Fig. 13, the AutoRegisterAspect abstract aspect contains the pointcut for this empty method. Individual features derive concrete aspects from AutoRegisterAspect and implement the appropriate registration code. An example of this is shown in Fig. 14. Note that the actual feature registration encompasses many more details that have been left out here for simplicity.

```
aspect RegisterTtlFeature extends AutoRegisterAspect {
    protected void register(FeatureRegistry fr) {
        fr.registerFeature(CorbaTtlFeature.class);
    }
}
```

Fig. 14 *Register TtlFeature registration implementation*

5.2.12 Related patterns:

The template advice pattern has many similarities to the template method pattern. Both patterns define a general skeleton that defers an operation definition to derived types. They differ in their mechanism (use of aspects) and in intent. The intent of the template advice pattern is to decouple the knowledge of a pointcut and where advice should be placed from the actual implementation of that advice.

The template advice pattern is also related to the interceptor pattern [31] in its use. Both patterns provide mechanisms to add logic at predefined interception points. The template advice pattern, though, takes advantage of AOP techniques to avoid requiring a registry to manage interceptors or the need to add interceptor callbacks throughout the code. Consequently, since template advice is applied at compile-time, it has a higher performance than an equivalent implementation that uses interceptors. Finally, many of the high level design techniques for the interceptor pattern are also useful for the template advice pattern.

5.3 Interface tag

5.3.1 FACET context:

Quite often, it is necessary to apply advice to a certain group of classes (see Fig. 15). For instance, in writing upgrader aspects, one needs to apply certain advice to all classes that require upgrading. Specifying these classes by name is highly impractical since there is little scope for extensibility and it is not an approach that promotes type-safety.

The problem of tagging all classes that certain advice should be applied to is solved by the interface tag pattern by making use of empty interfaces that are implemented by all classes that are interested in being the target of certain advice.

5.3.2 Intent:

Tag a set of arbitrary classes and aspects as possible recipients of advice.

5.3.3 Motivation:

AspectJ enables one to specify the places at which advice is applied in aspects by defining joinpoints. When joinpoints are not precisely known by an

```
public abstract aspect AutoRegisterAspect {

    abstract protected void register(FeatureRegistry fr);

    private pointcut registry(FeatureRegistry fr) :
        execution(void FeatureRegistry.buildGraph()) && target(fr);

    after(FeatureRegistry fr) : registry(fr) {
        register(fr);
    }
}
```

Fig. 13 *AutoRegisterAspect abstract aspect*

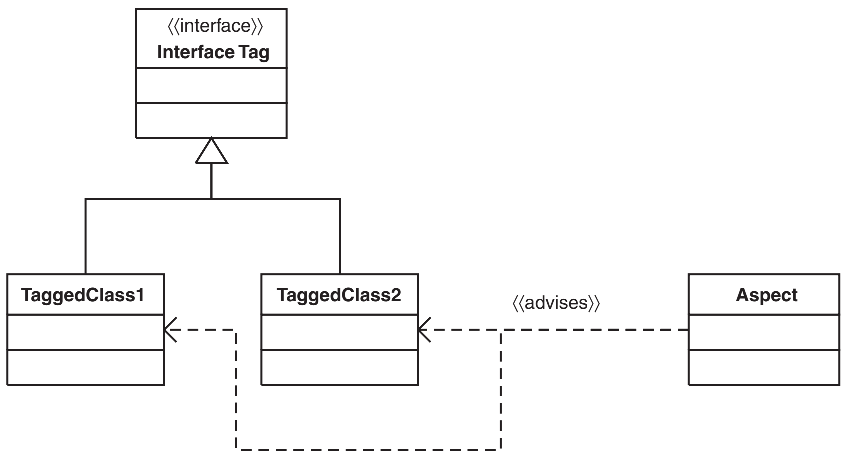

Fig. 15 *Interface tag pattern structure*

aspect, a standard technique is to create an abstract pointcut and then to specialise the pointcut in derived aspects. In many cases, though, the specialisation serves simply to identify those classes into which an aspect should be woven. For example, a trace aspect that logs a message whenever a method is entered or exited may specify before and after advice around method calls, but the classes to which it is applied may vary. If the classes to which the advice is applied are arbitrary, each particular class's name needs to be hardcoded in the aspect's pointcut. As indicated above, aspect inheritance could be used to help decouple the pointcut locations by creating a derived aspect each time a new location is discovered. This solution can become cumbersome when the number of aspects that need to be created becomes large.

An alternative solution used in the interface tag pattern is to create an empty interface class or a *tag* and use the tag to mark every class that should be affected by the aspect's advice. For example, in the trace aspect example, a Traceable interface could be created and any class that would like to be traced need only implement the Traceable interface. The trace aspect itself would only need to specify that its advice apply to all classes that implement Traceable to work.

In this example, the tracing concern is still encapsulated in the tracing aspect. However, the pointcut to which the tracing concern applies has been decoupled from the aspect. As classes are added to the system, the decision whether or not the tracing aspect should be applied can be made. Note that it is also possible that the tracing aspect is not enabled or has a stricter pointcut that may not apply to all classes that request it. In cases where the aspect is not applied, the presence of the interface tag results does not add any runtime overhead.

5.3.4 Applicability:

Use the interface tag pattern when:

1. An aspect's advice applies to arbitrary classes that are difficult or impossible to categorise in a central location.
2. The class intends to have advice applied to it that is consistent with the tag.
3. Knowledge of classes to which a pointcut applies breaks the encapsulation boundaries of what an aspect should know.

5.3.5 Structure:

Figure 15 shows the structure of the interface tag pattern structure.

5.3.6 Participants:

• *Interface tag:* defines an empty interface that is implemented by a class to *tag* it.
• *TaggedClass1 and TaggedClass2:* implement the interface tag interface.
• *Aspect:* implements advice that affects classes that implement the interface tag interface.

5.3.7 Collaborations:

The aspect advises joinpoints within the tagged classes.

5.3.8 Consequences:

The interface tag pattern has the following consequences:

1. *The pointcut is more general:* By using the interface tag pattern, the pointcut in the aspect can be specified without specifically referencing target classes. This allows new classes to be added without revising the pointcut.
2. *The aspect is more reusable:* Since the aspect does not contain references to specific classes, it is no longer coupled tightly with the code that it advises.
3. *Standard aspect semantics are inverted:* Normally, aspects apply their advice to target pointcuts that they specify. By using the interface tag pattern, classes request that aspects be applied to them.

5.3.9 Negative consequences:

None.

5.3.10 Implementation:

Consider the following issues when implementing the interface tag pattern:

1. *Tag all application classes:* It is easy to miss tagging some classes such as inner classes in Java. To be sure that classes are not missed, it may be possible to write a *tagging* aspect that tags all affected classes in a particular set of source files. The motivation for this pattern precludes tagging all classes in one aspect, but it may be possible to localise the tagging.
2. *Take advantage of classes that act like tags:* In some cases, classes may already exist that *tag* other classes. This obviates the need to define a specific interface tag.

5.3.11 Known uses:

The interface tag pattern is used to identify classes and aspects in FACET that should be upgraded when unrelated features are included in the system. For this purpose, it is actually used twice. The first usage is to mark a class as upgradable by implementing the Upgradeable interface, and the second is to mark which features that class knows about. This latter marking is done by implementing an interface defined by each feature. Additionally, if a feature depends on another feature, its interface will, in turn, extend that feature. Section 4.4 describes the Upgradeable interface in detail and provides more information on the automated upgrading of unit tests.

5.3.12 Related patterns: The intent behind the interface tag pattern is similar to that used in Java to mark classes that can have their state *serialised* to or from a stream. Such classes are identified to the JVM by implementing the java.io.Serializable empty interface. Another such interface in Java is java.lang.Cloneable.

6 Experimental results

By enabling the user to select only those features that are necessary, FACET enables both code footprint and performance advantages over traditional middleware implementations. In this Section, we present experimental data obtained on the footprint and throughput performance of a number of different configurations of FACET, both in the presence and absence of CORBA [42].

To collect such data, a set of popular configurations was identified based on feedback from several developers of the TAO users' community who are using event channels in their application development [30]. In addition, to gauge the effect of individual features on the overall size and performance of the FACET event channel, each feature was studied by measuring its effect across all configurations that included or omitted the given feature.

We found that the numbers obtained were indeed quite encouraging and a solid proof of the applicability of AOP to the development of middleware in the manner we have described.

One method to measure the footprint of a Java application is to sum the size of all the .class files that are loaded. Embedded systems that use Java interpreters or just-in-time compilers could use this metric to estimate the amount of RAM needed. Another method consists of generating native code using a compiler (such as GCJ [45]) and then measuring the size of the resulting executable. The compiled code is more suitable for comparisons with C and C++ code. Moreover, embedded real-time applications are likely to precompile to native code for execution predictability. An overall observation has been that the size of the GCJ produced object files are generally larger than the corresponding .class files [30]. This is commensurate with the design of .class files to be small so as to reduce transmission time over networks.

Here, we report results based on .class files that are interpreted and executed using the Java Virtual Machine (JVM) 1.4.0 with Just-In-Time compilation enabled. The experiments were performed on a dual-Xeon processor machine running at 2.40 GHz, with 512 MB of RAM.

With Java and .class files, the footprint of the running program increases as classes are loaded. We report the maximum footprint, achieved when all code has been loaded; such measurements are most appropriate for an embedded system. For a native-code compiled version, both the footprint and the resulting throughput are expected to increase.

6.1 Common configurations

The following are the 10 event channel configurations used in collecting experimental data:

Configuration 1 (base): Although the applications requested by developers all required more functionality than the base, it is useful in that it is a lower bound on the footprint. Note that all subsequent tests use the full functionality provided by the base.
Configuration 2: Several developers only needed configurations similar to the standard CORBA COS event service specification. This configuration has CORBA Any payloads and does not support filtering. The pull interfaces were not included in this configuration since they were not used.
Configuration 3: This configuration is the same as the previous except that the tracing feature is enabled.
Configuration 4: Structured events and event sets are enabled. This configuration also adds the TTL field processing to eliminate loops created by federating event channels. This configuration is still minimal, however, and does not support any kind of event filtering.
Configuration 5: This configuration has support for dispatching events based on event type. It uses a CORBA octet sequence as the payload type and is a common optimisation over using a CORBA Any. This configuration is similar to that used in the TAO real-time event channel (RTEC).
Configuration 6: This configuration adds support for the event pull interfaces to Configuration 5 and uses a CORBA Any as the payload.
Configuration 7: This configuration enhances Configuration 5 by replacing the simple event type dispatch feature with the event correlation feature. In the corresponding application, event timestamp information was also needed, but the event pull feature was not.
Configuration 8: This configuration represents one of the largest realistic configurations of FACET. It supports the pull interfaces, uses event correlation, and adds support for statistics collection and reporting. It uses structured events carrying CORBA Any payloads and headers with all possible fields enabled.
Configuration 9: This configuration adds the tracing feature to Configuration 8.
Configuration 10: This configuration is representative of that used in the Boeing Bold Stroke architecture. It includes a number of features like type filtering, event correlation, event timestamps and the realtime dispatcher feature – a feature that allows consumers and suppliers to set realtime priorities on event delivery.

6.2 Footprint measurements

As shown in Fig. 16, the base FACET configuration (Configuration 1) is 3 times larger when CORBA is present: 166 921 bytes with CORBA and 55 250 without.

At the other extreme, one of the fuller-featured FACET configurations (Configuration 9) has a size of 475 100 bytes with CORBA and a size of 342 226 bytes without – approximately 1.4 times larger with CORBA enabled. This is expected since there are a significant number of Stub and Skeleton classes that are generated by the IDL compiler, which are absent in the non-CORBA case.

It must be noted that the size of the ORB has not been included in this study. It follows that if it were indeed included in these measurements, there would be an even bigger difference in the footprint observed. Generally, in full-featured ORBs such as JacORB [46] that are not subsettable, the most casual reference to the ORB causes the entire ORB to be included in the resulting executable code. While ORBs vary in size [47], and some ORBs do offer reduced-feature versions, the choice of which features to include or omit is not made on an application-specific basis. Conceivably, our AOP approach for including features in an event channel could be extended to include only those ORB features needed to support a given event-channel configuration.

Figure 16 shows that the disabling of CORBA for the configurations we considered mostly reduced footprint by

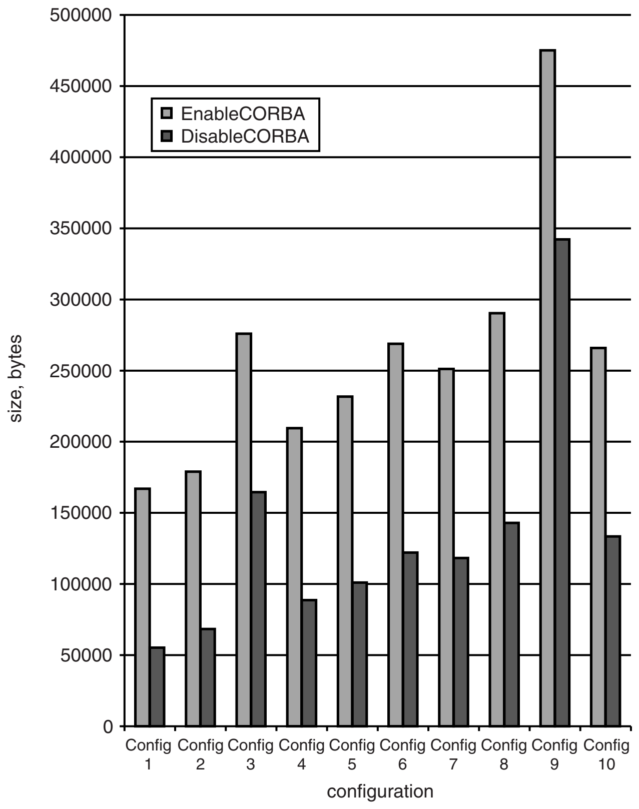

Fig. 16 *FACET footprint under different configurations*

about half – appreciable savings for small embedded systems.

6.3 Throughput measurements

Figure 17 shows the difference in throughput performance for various configurations with and without CORBA enabled. When configured as the standard CORBA COS Event Service [12], the throughput with CORBA enabled was 1651 events/s as compared with 131 758 events/s without – a difference of 2 orders of magnitude! This can be explained by the fact that the Java ORB, JacORB, does not include optimisations for collocated objects, which means that the Stubs and Skeletons perform marshalling and communication over network sockets, assuming a truly distributed system. With an ORB such as TAO that does include such optimisations, the performance difference is likely to be less dramatic but still substantial.

This level of improvement without CORBA held for all configurations of the event channel that we studied, with the exception of configurations which included the tracing feature (Configurations 3 and 9). The reason for this can be attributed to the enormous amount of code weaved in by the AspectJ compiler onto all the methods of every class in the event channel, when the tracing feature is enabled. The overhead of these extraneous method calls to the log4j [48] logging library contribute significantly to performance degradation and to the size of the footprint. This observation is consistent with findings in a previous study [30].

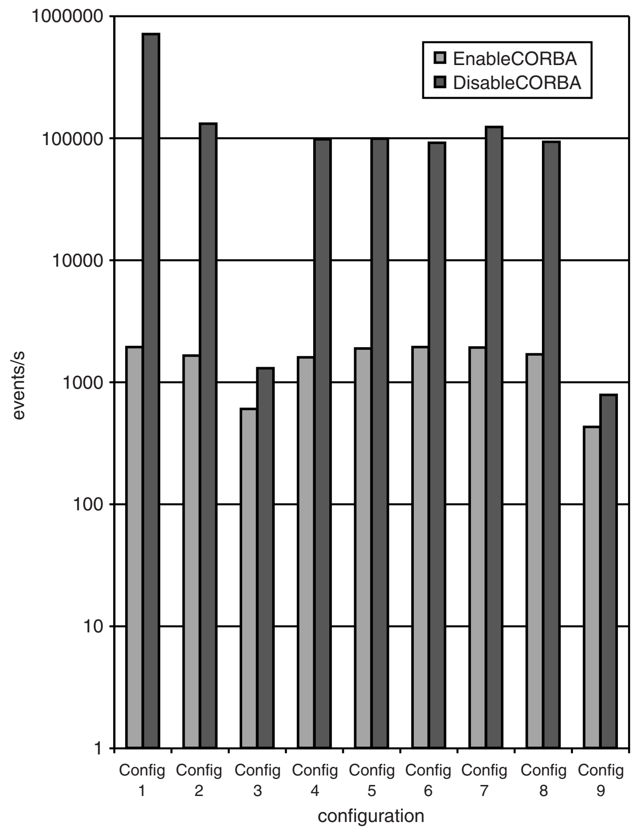

Fig. 17 *Throughput under different configurations*

6.4 By-feature study

We next measured footprint and throughput for various configurations in which only a single feature (and features on which it depends) was enabled at a time. This experiment quantifies the size contribution and throughput degradation of a given feature.

Figure 18 shows footprint reduction by feature, with and without CORBA. For an embedded system, even modest savings can be crucial to a component's cost.

A much greater impact can be seen as we study performance. As shown in Fig. 19, the difference for each feature with and without CORBA is dramatic. The interesting observation is that no difference is observed *among* the features when CORBA is disabled. An explanation for this is that the aggregation of features on the base and the overhead associated with the code weaved in by the AspectJ compiler is negligible, so that the throughput at this point is limited by the operating system and/or hardware. This indicates that the throughput performance of the event channel with CORBA disabled is at a maximum and is quite unaffected by the feature set (again with the exception of the tracing feature).

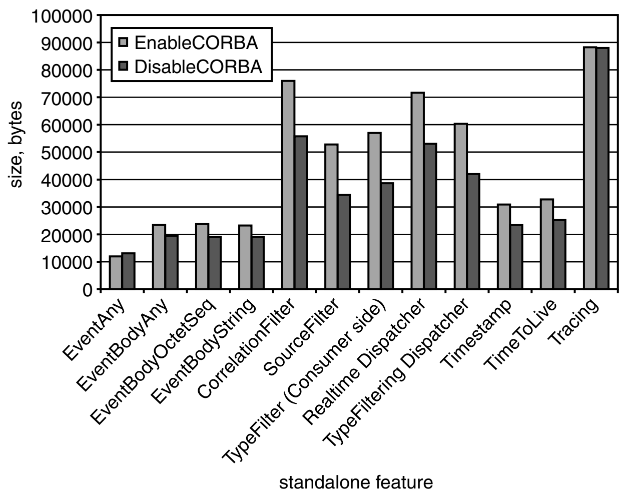

Fig. 18 *Impact of different features on footprint*

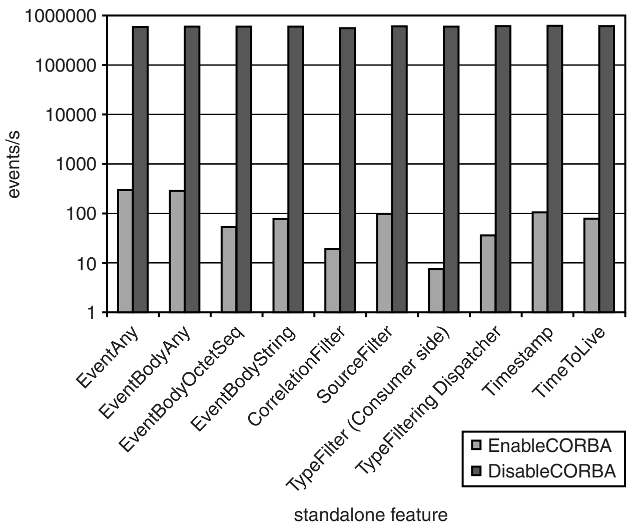

Fig. 19 *Impact of different features on throughput*

6.5 Impact of external libraries

Features can have an even more substantial impact on the overall footprint when they depend on auxiliary libraries to provide some functionality. An example of this is the tracing feature, since it pulls in the *log4j* logging libraries [48], which require an additional 290 kilobytes of .class files [30]. In nonembedded Java applications, the Java class loader can limit the amount of code and data in memory by dynamically loading only what is needed. On the other hand, embedded applications often require that all possibly executed code be linked or packaged together prior to runtime, so that such code can be deployed in ROM or some other local memory device. Therefore, a middleware user also needs to consider external libraries that are referenced as byproducts of enabling features.

In addition to referencing other libraries, features may make more or less use of libraries required in the base. This becomes apparent in FACET's use of the JacORB [49] CORBA ORB. For example, CORBA Anys require support from the ORB and additional code to be generated from IDL files to marshal and demarshal Any variables. Since FACET can be configured to avoid using CORBA Anys, it would be desirable to remove all Any support from the ORB to reduce code footprint. As the ORB libraries contribute a substantial amount of code to the end application, it is desirable to trim other functionality as well. This is not possible in the JacORB implementation, however, since the degree to which these concerns can be separated from the ORB is not as significant as what can be accomplished using AOP techniques – as in FACET.

6.6 Savings from using aspects

By using aspects to weave features together, FACET does not require the programming infrastructure to support varying functionality that traditional middleware needs. This includes if statements to choose alternate paths, virtual function calls to strategised methods, and abstract factories to select functionality at runtime. This entire infrastructure impacts the performance and code size of the middleware both when extended features are enabled *and* when they are not included.

Determining the overhead saved by using AOP instead of traditional techniques in FACET is not straightforward. At a minimum, several features introduce fields to existing data structures such as the event structure. Any Java-only implementation of FACET could not allow this flexibility [Note 2]. Other features have mutually exclusive relationships that cannot be easily rewritten for the same functionality just using object-oriented techniques. Additionally, the flexibility of aspects to augment code directly at the appropriate interception points serves to minimise the commonality of interception points between aspects. If converted directly to an object-oriented program with similar flexibility, this would result in a high number of hooks to call extensions. An object-oriented designer would probably try to reduce the number of these hooks and find more commonalities between features to reduce the complexity of the resulting code.

7 Concluding remarks and future work

As embedded software becomes more complex, it becomes increasingly desirable to use middleware and, in particular, DOC middleware in distributed embedded applications. Two major impediments to using middleware frameworks such as ACE and TAO are their footprint size and the inability to subset them enough to fit on platforms with limited program storage. Unfortunately, existing object-oriented techniques to subset middleware are time-consuming and can make existing code more complex.

In this paper, we have developed a novel approach to constructing middleware by using AOP techniques. By designing an essential base implementation and using *aspects* to encapsulate optional features, the middleware user now has the ability to select only those features that are truly needed. This has distinct advantages in that the resulting middleware contains very little code bloat for unused features – they are simply not compiled. Additionally, by using aspects, the hooks, strategies, registries and other infrastructure needed to support subsetting object-oriented middleware are no longer needed. This simplifies the readability of both the feature and the base code.

Murphy *et al.* [50] and others have questioned whether a collection of code and aspects simplifies the task of writing, understanding or evolving a system. Concerns are separated, but their effects on code are not always obvious. It is certainly possible to misuse aspects to the point that the system's concerns are not reflected in the aspectual decomposition. As discussed in Section 3, we have used AOP to achieve a form of FOP, whose principles mitigate the complexity that could arise from misuse of an AOP system.

To research the feasibility of developing middleware using AOP, we built FACET, a CORBA event channel modelled after the OMG Event and Notification Services and the TAO Real-time Event Channel. The base FACET implementation which forms the structure on top of functionality is added by means of features – to send payloads, provide correllation and filtering, support statistics collection, provide *pull* interfaces, and more. In addition to managing functional concerns, we demonstrated how systemic concerns such as the transport layer (CORBA) can be abstracted into a feature.

Managing the many features in FACET is itself an issue, since dependences between features make some event channel combinations invalid. A framework was developed to describe the characteristics of features within the system and their relationships to other features. In FACET, every

Note 2: Languages such as C and C++ that support preprocessor macros could allow for this flexibility at a major cost to code readability.

feature registers its characteristics and dependences with the FeatureRegistry. With this information, the FeatureRegistry can be used to validate configurations and automatically select missing dependent features.

Managing features alone, however, is not enough for developing highly customisable middleware. Especially for high reliability environments, verification that selected configurations actually do work is also needed. In FACET, this is provided by including a test framework that is used by every feature. For any particular combination of features, the build system can run all relevant unit tests. Additionally, FACET can itself verify that every possible combination of features and associated unit tests compiles and runs by using information from the FeatureRegistry.

To aid the development of future middleware that uses AOP, several design patterns were identified and documented that proved very useful in the development of FACET. These include lessons learned when extending API calls to contain new parameters, encapsulating and exposing joinpoints and advice, and using aspects to augment arbitrary code.

Lastly, footprint and performance measurements were taken that quantify the advantages of using AOP to selectively enable and disable middleware functionality. These measurements show that disabling complex unneeded middleware features can significantly improve middleware's applicability to more constrained environments. In this paper, we have presented these measurements for event service configurations presently in use by members of the TAO user community.

One of the issues found when using FACET is that currently a general shared library cannot be created that supports all possible configurations. For example, it would be ideal if an application could specify its required features, and an aspect-aware linker (or dynamic library loader) could weave in the features to create the desired event channel. Currently, a separate library needs to be created for every desired configuration, and an application needs to link against the library that supplies the right features. In FACET, since every configuration is in the same package, only one configuration can be in use at a time. Simply creating a different package for every configuration is impractical due to the large number of configurations and the space required. We plan to investigate this in future work.

FACET is configured *statically* – well ahead of executing the event channel. As such, its set of features must be known prior to installing FACET. A more dynamic approach would allow features to be added on-the-fly, performing AspectJ's weaving at runtime. With meta-object protocols (MOP) [51], the behaviour of the event channel could be customised at runtime, and this is the subject of future work. Other dynamic approaches include Java aspect components (JAC) [52], which provides an AOP server that weaves at runtime. However, note that FACET's static configuration is very well suited to its deployment in embedded systems – an important target for our work.

Additionally, we intend to take the transport layer abstraction further and to support other transport mechanisms such as Java RMI [53] through encapsulation in a feature. Finally, we are also investigating extending FACET to make real-time guarantees about event delivery.

8 Acknowledgments

We thank David Sharp and Edward Pla of The Boeing Company for testing FACET extensively and providing us continuous feedback on its deployment; Morgan Deters for providing interesting ideas on the use of aspects and for answering many questions in that regard; Krishnakumar Balasubramanian for general ideas on transport layer abstraction and help with CORBA; and Anand Krishnan for help with various things, including proofreading draft versions of this paper. We thank the reviewers for their comments and suggestions, which we believe have resulted in a substantially improved paper. This work and its authors were sponsored by DARPA under contract F33615-00-C-1697.

9 References

1 Object Management Group. The common object request broker: architecture and specification, 2.4 edition, October 2000
2 Morgenthal, J.P.: Microsoft COM+ will challenge application server market. www.microsoft.com/com/wpaper/complus-appserv.asp, 1999
3 Wollrath, A., Riggs, R., and Waldo, J.: 'A distributed object model for the Java system', *USENIX Comput. Syst.*, 1996, **9**, (4), pp. 265–290
4 Schmidt, D.C.: The ADAPTIVE communication environment (ACE). www.cs.wustl.edu/ ~ schmidt/ACE.html, 1997
5 Institute for Software Integrated Systems. The ACE ORB (TAO). www.dre.vanderbilt.edu/TAO/, Vanderbilt University
6 Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: 'Design patterns: elements of reusable object-oriented software' (Addison-Wesley, Reading, MA, 1995)
7 Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.-M., and Irwin J.: 'Aspect-oriented programming'. Proc. 11th European Conf. on Object-oriented Programming, Jyvaskla, Finland, June 1997, pp. 220–242
8 Hunleth, F.: 'Building customizable middleware using aspect-oriented programming'. Master's thesis, Washington University in Saint Louis, 2002
9 Schmidt, D.C., and Huston, S.D.: 'C++ Network programming, Volume 1: Mastering Complexity with ACE and Patterns' (Addison-Wesley, Boston, 2002)
10 DOC Group: ACE Success Stories. www.cs.wustl.edu/~ schmidt/ACE-users.html
11 Schmidt, D.C.: Successful project deployments of ACE and TAO. www.cs.wustl.edu/~ schmidt/TAO-users.html, Washington University
12 OMG. CORBAServices: Common Object Services Specification, Revised Edition. Object Management Group, 97-12-02 edition, December 1997
13 Object Management Group. Notification service specification. Object Management Group, OMG Document telecom/99-07-01 edition, July 1999
14 Gore, P., Cytron, R.K., Schmidt, D.C., and O'Ryan, C.: 'Designing and optimizing a scalable CORBA notification service'. Proc. Workshop on Optimization of Middleware and Distributed Systems, Snowbird, Utah, June 2001, pp. 196–204
15 O'Ryan, C., Schmidt, D.C., and Noseworthy, J.R.: 'Patterns and performance of a CORBA event service for large-scale distributed interactive simulations', *Int. J. Comput. Syst. Sci. Eng.*, 2002, **17**, (2), pp. 115–132
16 Harrison, T.H., Levine, D.L., and Schmidt, D.C.: 'The design and performance of a real-time CORBA event service'. Proc. ACM Conf. on Object-oriented Programming, Systems, Languages and Applications (OOPSLA), Atlanta, GA, October 1997, pp. 184–199
17 Dijkstra, E.W.: 'A discipline of programming' (Prentice-Hall, Englewood Cliffs, NJ, 1976)
18 Elrad, T., Filman, R.E., and Bader, A.: 'Aspect-oriented programming: Introduction', *Commun. ACM*, 2001, **44**, (10), pp. 29–32
19 Czarnecki, K., and Eisenecker, U.: 'Generative programming: methods, tools, and applications' (Addison-Wesley, Boston, 2000)
20 Tarr, P., Ossher, H., Harrison, W., and Sutton, S.M.: 'N degrees of separation: multi-dimensional separation of concerns'. Proc. Int. Conf. on Software Engineering, Los Angeles, CA, May 1999, pp. 107–119
21 Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., and Ong, J.S.: 'Structuring operating system aspects: using aop to improve os structure modularity', *Commun. ACM*, 2001, **44**, (10), pp. 79–82
22 Constantinides, C., and Skotiniotis, T.: 'Providing multidimensional decomposition in object-oriented analysis and design'. Proc. Int. Assoc. of Science and Technology for Development (IASTED), Software Engineering Conf., Innsbruck, Austria, 17–19 February 2004, pp. 101–110
23 Ossher, H., and Tarr, P.: 'Using multidimensional separation of concerns to (re)shape evolving software', *Commun. ACM*, 2001, **44**, (10), pp. 43–50
24 Ossher, H., and Tarr, P.: 'Multi-dimensional separation of concerns in hyperspace'. Technical Report RC 21452(96717)16APR99, IBM, 1999
25 Herrmann, S., and Mezini, M.: 'Pirol: a case study for multidimensional separation of concerns in software engineering environments'. Proc. 15th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, Minneapolis, MN, 15–19 October 2000, pp. 188–207
26 Ossher, H., and Tarr, P.: 'Hyper/j: multi-dimensional separation of concerns for java'. Proc. 22nd Int. Conf. on Software Engineering, Limerick, Ireland, 4–11 June 2000, pp. 734–737

27 The AspectJ Organization. Aspect-oriented programming for Java. www.aspectj.org, 2001
28 Arnold, K., Gosling, J., and Holmes, D.: 'The Java programming language' (Addison-Wesley, Boston, 2000)
29 Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D.: 'Refactoring - improving the design of existing code' (Addison-Wesley, Reading, MA, 1999)
30 Hunleth, F., and Cytron, R.K.: 'Footprint and feature management using aspect-oriented programming techniques'. Proc. Joint Conf. on Languages, Compilers and Tools for Embedded Systems, Berlin, Germany, 19–21 June 2002, pp. 38–45
31 Schmidt, D.C., Stal, M., Rohnert, H., and Buschmann, F.: 'Pattern-oriented software architecture: patterns for concurrent and networked objects' (Wiley, New York, 2000), vol. 2
32 Batory, D.: 'A tutorial on feature oriented programming and product-lines'. Proc. 25th Int. Conf. on Software Engineering, Portland, OR, 3–10 May 2003, pp. 753–754
33 Batory, D., Sarvela, J.N., and Rauschmayer, A.: 'Scaling step-wise refinement'. Int. Conf. on Software Engineering (ICSE), Baltimore, MD., 17–21 May 1993
34 Clements, P., and Northrop, L.: 'Software product lines: practices and patterns' (Addison-Wesley, Boston, 2002)
35 Batory, D., Lopez-Herrejon, R., and Martin, J.-P.: 'Generating product-lines of product-families'. Presented at Automated Software Engineering Conf., Edinburgh, Scotland, 23–27 September 2002
36 Flatt, M., Krishnamurthi, S., and Felleisen, M.: 'Classes and mixins'. Proc. 25th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, San Diego, CA, 19–21 January 1998, pp. 171–183
37 Gray, J., Bapty, T., Neema, S., Schmidt, D.C., Gokhale, A., and Natarajan, B.: 'An approach for supporting aspect-oriented domain modeling'. Proc. 2nd Int. Conf. on Generative Programming and Component Engineering (GPCE), Erfurt, Germany, September 2003, pp. 151–168
38 Hunleth, F., Cytron, R., and Gill, C.: 'Building customizable middleware using aspect oriented programming'. Presented at OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-oriented Systems, Tampa Bay, FL, October 2001. ACM. www.cs.ubc.ca/~kdvolder/Workshops/OOPSLA2001/ASoC.html
39 Object Management Group. The common object request broker: architecture and specification, 3.0.2 edition, December 2002
40 Apache Software Foundation. Apache Ant. jakarta.apache.org/ant/
41 Gamma, E., and Beck, K.: JUnit. www.xProgramming.com/software.htm, 1999
42 Pratap, M.R., and Cytron, R.K.: 'Transport layer abstraction in event channels for embedded systems'. Proc. Joint Conf. on Languages, Compilers and Tools for Embedded Systems, San Diego, CA, June 2003, pp. 144–152
43 Hannemann, J., and Kiczales, G.: 'Design pattern implementation in java and aspectj'. Proc. 17th ACM SIGPLAN Conf. on Object-oriented Programming, Systems, Languages, and Applications, Seattle, WA, 4–8 November 2002, pp. 161–173
44 Object Management Group. Unified Modeling Language (UML) v1.4, OMG Document formal/2001-09-67 edition, September 2001
45 GNU is Not Unix. GCJ: The GNU Compiler for Java. gcc.gnu.org/java, 2002
46 Brose, G.: 'JacORB: implementation and design of a Java ORB'. Proc. IFIP WG 6.1 Int. Working Conf. on Distributed Applications and Interoperable Systems (DAIS), September 1997, pp. 143–154
47 Gill, C., Subramonian, V., Parsons, J., Huang, H.-M., Torri, S., Niehaus, D., and Stuart, D.: 'ORB middleware evolution for networked embedded systems'. Proc. 8th Int. Workshop on Object Oriented Real-time Dependable Systems (WORDS), Guadalajara, Mexico, 15–17 January 2003, pp. 169–176
48 Apache Software Foundation. log4j. jakarta.apache.org/log4j/
49 Brose, G., Noffke, N., and Müller, S.: JacORB 1.4 Programming Guide. jacorb.inf.fu-berlin.de/ftp/doc/ProgrammingGuide_1.4.pdf, 2001
50 Murphy, G.C., Walker, R.J., Baniassad, E.L.A., Robillard, M.P., Lai, A., and Kersten, M.A.: 'Does aspect-oriented programming work?', *Commun. ACM*, 2001, **44**, (10), pp. 75–77
51 Zimmermann, C.: 'Metalevels, MOPs and what the fuzz is all about', in Zimmermann, C. (Ed.): 'Advances in object-oriented metalevel architectures and reflection' (CRC Press, Boca Raton, FL, 1996), pp. 3–24
52 Pawlak, R., Seinturier, L., Duchien, L., and Florin, G.: 'JAC: A flexible solution for aspect-oriented programming in java', *Lect. Notes Comput. Sci.*, 2001, **2192**, pp. 1–24
53 SUN. Java Remote Method Invocation (RMI) Specification. java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html, 2002