Proceedings of the Fifth AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software

March 21, 2006

Held in conjunction with the Fifth International Conference on Aspect-Oriented Software Development (AOSD 2006)

Bonn, Germany

Yvonne Coady, David H. Lorenz, Olaf Spinczyk, and Eric Wohlstadter, editors

Department of Computer Science
University of Virginia
Charlottesville, VA 22904
Technical Report CS-2006-01

Table of Contents

Creating Pluggable and Reusable Non-functional Aspects in AspectC++

Michael Mortensen
Hewlett-Packard
3404 E. Harmony Road, MS 88
Fort Collins, CO 80528
and
Computer Science Department
Colorado State University
Fort Collins, CO 80523
mmo@fc.hp.com

Sudipto Ghosh
Computer Science Department
Colorado State University
Fort Collins, CO 80523
ghosh@cs.colostate.edu

ABSTRACT

An object-oriented framework enables both black box reuse and white box reuse in client applications, serving as an important infrastructural building block. We are refactoring framework-based applications to modularize cross-cutting concerns with aspects. In this paper, we explore implementation issues we encountered while creating non-functional aspects in AspectC++ that are pluggable and reusable.

1. INTRODUCTION

Aspect-oriented programming can be used to modularize cross-cutting concerns in application software. Non-functional aspects modularize cross-cutting concerns that do not directly affect functionality, but instead improve characteristics such as performance, dependability, and configurability. Though important, they are orthogonal to the primary functionality. Modularizing non-functional cross-cutting concerns with aspects reduces duplicated code. In addition to reducing duplicated code, aspects improve pluggability, since a single pointcut can enable or disable the advice code [6]. Designing aspects with virtual pointcuts and virtual methods allows an aspect to be reused in different applications, with application-specific behavior or pointcuts specified in a concrete aspect.

Object-oriented frameworks serve as infrastructure building blocks for the development of large-scale industrial applications. A framework's application programming interface (API) enables blackbox reuse [14] while framework classes enable white box reuse through object-oriented mechanisms such as composition, inheritance, and polymorphism [13].

Domain-specific frameworks are typically designed as large, layered systems that include a kernel layer of foundational services such as access to the operating system, container classes, and external data stores and domain-specific layers [2]. VLSI chip design involves a complex, multi-step flow that requires many individual tools to share data. VLSI CAD frameworks provide common translators for different file formats and enable applications to focus on a specific task while leveraging framework components for data translation, persistence, and memory models [1].

Frameworks provide extension points for client applications that enable object-oriented reuse. However, such extension points must be anticipated by framework developers during the design of the framework. Aspects provide an additional way of providing extension points for client applications. Advice can be layered on top of existing code whether or not the developer anticipated the extension.

Although frameworks can be designed or refactored to use aspects [5], changing existing frameworks can be problematic due to license restrictions, the large number of applications already using the existing framework implementation, or the effort required to modify a large framework. Rather than modifying these frameworks, we can identify cross-cutting concerns common to many framework-based applications, particularly cross-cutting concerns that relate to framework use and extension. In like manner, Ghosh et al. [4] differentiate between business logic and code that interacts with middleware services, and propose that middleware-related code be modularized as an aspect.

We use aspects to enhance the use of existing, unmodified OO frameworks. We are creating a library of framework-based aspects that represent cross-cutting concerns in framework-based applications. We refactor applications (not the framework) and weave in aspects from the aspect library; the refactored applications use both the aspect-library and the framework. For our aspect library to be beneficial to many framework-based applications, aspects must be designed with pluggability and reuse in mind. While developing an aspect library for an VLSI CAD framework used at Hewlett-Packard, we found that tradeoffs between alternative implementations have implications for reusing aspects. We also found that aspects themselves can be developed in a modular way so that a caching aspect can monitor how much benefit (in terms of cache hit rates) it provides to the system.

The remainder of the paper is structured as follows. Section 2 describes the design and implementation of a caching aspect for framework-based applications. In section 3 an

AOSD:ACP4IS 2006 Bonn, Germany

aspect for enabling user-configuration of a large CAD application is considered. Section 4 discusses related work, followed by conclusions and future work in Section 5.

2. CACHING

2.1 Motivation

Caching is used to improve performance while maintaining the same functional behavior. Caching is a common example of a cross-cutting concern that can be modularized as an aspect. The implementation is largely the same across all functions that are cached [7, 8].

Framework-based applications contain functions or methods that implement caching in an ad-hoc way. In VLSI CAD applications, these functions calculate some property of a circuit element (an electrical net or transistor). Calculating the property can involve complex, time-consuming steps, such as reduction of RC networks for resistance calculation or graph traversals to calculate reachability. In a framework-based application, the objects being cached are often instances of framework classes, while the function doing the calculation is application-specific. One of the framework-based applications that we are refactoring is an electrical rules checking (ERC) tool developed at Hewlett-Packard, the `ErcChecker`. The `ErcChecker` consists of approximately 80,000 lines of C++ code.

2.2 Aspect Identification

We can identify cached functions idiomatically since they use a manually inserted static set or static map. The `ErcChecker` contains 38 functions that separately implement similar caching code. These functions were in various classes and did not have a common naming convention, so a list of pointcut expressions grouped by the logical `or` operation can be used to capture them, and the original tangled code can simply be deleted.

2.3 Design challenges

Caching consists of storing the result value for some input. In C++, the input can be one of several things, since we can have procedural functions where the input is a value, method calls where the input is the object invoking the method, or functions or methods where an object is passed as a parameter.

Non-functional aspect pluggability is important since these aspects may not always be needed. For example, user comments in one manually cached `ErcChecker` function indicate that the cache was not improving performance. Caching is only effective if cached values are used: a function that does not recompute the same value during program execution will not benefit from caching. In prototyping and developing caching as an aspect that could easily be enabled and disabled and also monitor its benefits, we found a variety of interesting trade offs and challenges.

2.4 Cache implementations

2.4.1 Two simple caches

We implemented two simple caching aspects: one for caching functions and static methods, and one for caching object-based method calls. These caches are similar to a generic caching strategy described by Lohmann, Blaschke, and Spinczyk [8], although they focus on using C++ template metaprogramming with AspectC++, and demonstrate capabilities such as handling an arbitrary number of function or method parameters.

Example code for caching a function or static method based on its first argument is shown in Appendix A, as the `AbstractFunctionCache`. The second cache is structured like the `AbstractFunctionCache`, but the cache key is the object on which the method is called rather than the first parameter. This changes the static map definition:

```
static std::map <
      typename JoinPoint::Target*,
      typename JoinPoint::Result >
   theResults;
```

Retrieving the target is done dynamically with `tjp->target()`.

Both aspects declare a virtual pointcut so that use only requires creating a concrete aspect that defines the pointcut, as shown below. Because the cache creates a static map inside the around advice, a concrete aspect like `MethodObjCaching` (see below) whose pointcut matches multiple join points will have separate caches for each one.

```
1  aspect MethodObjCaching :
2     public AbstractMethodCache {
3        pointcut ExecAroundMethod() =
4              execution("% Circle::getArea(...)")
5           || execution("% Donut::getArea(...)");
6  };
7  aspect SquareCache :
8     public AbstractFunctionCache {
9        pointcut ExecAroundArgToResult() =
10             execution("% SquareArea(...)");
11 };
```

2.4.2 TrackingCache

In order to measure the effectiveness of the caching concern, we add hit and miss counts directly to the implementations above. However, we immediately recognize two constraints: we must initialize the Statistics before the first access, and after all calls to the caching functions are done the Statistics need to be displayed.

Because a concrete aspect (such as `MethodObjCaching`) may match multiple join points, hits and misses should also be tracked for each join point. We can create a simple C++ struct for representing hits and misses whose constructor does initialization, and can store the struct for each join point with `JoinPoint::signature()`. We print out the map `after` the `main` function has executed to get the final results. We show the statistical tracking implementation in Appendix B. This caching aspect defines the `Stats` struct and overrides the output operator (`<<`) so that `Stats` variables are easily printed.

The around advice (`ExecAroundMethod`) has two lines of cache Statistics code (lines 35 and 40) mixed in with the caching code, with the Statistics functionality is spread across two advice bodies. Any changes tracking cache results will require manually editing these lines, which look like the kind of tangled code that aspects strive to reduce or eliminate.

2.4.3 Caching with a utility class

Since cache hit and miss statistics are linked to cache behavior, next we modify the aspect to use a utility class,

which we can extend via inheritance to add tracking information. This provides an aspect implementation that enables easily plugging or unplugging the cache tracking (hit/miss) functionality.

First, we define the helper class, which is a templatize cache class that can handle arbitrary key and value types. Source code is shown in Appendix C. Next, we modify the aspects (parameter based and object-based) to use `AspectCache`. The caching aspect that uses the class can simply instantiate it and call `get` and `set` rather than directly manipulating the cache data. A caching aspect for storing parameter/return value pairs is shown in Appendix D.

Note that the `AspectCache` class cannot be accessed in the `after` advice associated with `main`. Because of C++ scoping rules, the `AspectCache` variable (`theResults`) is only accessible in the `around` advice for the cached function. It is needed there for two reasons. First, we need it to be a static object inside the around advice so that it is created once for each join point, allowing this aspect to have a pointcut that matches many join points and yet have each join point be separately cached. Second, the cache is instantiated with static type information available *only* inside the join point (`JoinPoint::Arg<0>::Type` and `JoinPoint::Result`).

We rely on the destructor for the static class when extending the `AspectCache` via inheritance to create the `CountingAspectCache`, which we show in Appendix E.

The `CountingAspectCache` can be directly substituted in the aspect, or we can have an aspect associated with each class (`AspectCache` and `CountingAspectCache`) and then create a concrete aspect based on whether or not we want tracking. Since the `CountingAspectCache` may be associated with one of many join points, we pass in the join point signature to the constructor so that it will be accessible (at the end of execution) in the destructor (lines 11-17), which prints out join point names and cache statistics.

The `CountingAspectCache` inherits concrete methods from `AspectCache`. While reuses code, it could lead to problems if the `AspectCache` is modified without regard to how that might impact the `CountingAspectCache`. In fact, the relationship between the `CountingAspectCache` and `AspectCache` is analogous to the relationship between `CountingBag` and `Bag` in the example code for the fragile base class problem [11]. An alternative that avoids this would be an abstract caching class that only defines pure virtual methods (which forms an interface class in C++), from which both `AspectCache` and `CountingAspectCache` could be derived and which could be implemented separately.

Since we are providing caching as part of an aspect library for framework-based applications, the best approach seems to be to let users directly access both cache aspects. During development and testing, cache Statistics could be tracked to determine the value of caching a particular function. Before production release, the application developer would switch to the regular (non-counting) cache to reduce cache overhead. For aspect library stability, implementing the cache classes based on a C++ interface base class would be preferred.

2.4.4 Object Identity

Many of the functions cached are based on calculating a value for a parameter that is an object (e.g., pointer to an electrical net, pointer to a transistor). Object identity and aliasing introduce additional complexity to caching. Using pointers is efficient with a map. However, multiple references to the same object result in each one being added to the cache rather than reusing the calculated value, as shown by lines 32, 38 and 39 of example in Appendix F.

Hewlett-Packard's framework, like other VLSI CAD frameworks, reads in a persistent store such as a netlist and builds an in-memory model in which pointers are usually unique and are associated with circuit components. However, in general we need to be careful for cases where we can have different memory references to the same object.

There may also be distinct objects that are equivalent in some respect. In an electrical design, there may be many instances of the same circuit component. In the example in Appendix F, square structs `s1` and `s3` (on lines 24 and 28) have the same coordinates. Depending on the cost to compare objects versus the cost of computing some function, we may want to cache properties of objects and avoid recomputation when we have seen an alias to an object or an equivalent object.

2.5 Impacts on the ErcChecker

Originally, developers manually implemented caching for some functions of a large application to improve performance. The implementation did not use aspects, and resulted in similar code being scattered in many functions. Implementing caching as an aspect improves modularity and results in the removal of duplicated cache code from 27 locations. By using a helper class and inheritance, application developers can easily switch between a fast cache and a slightly slower one that measures the benefit of the cache. We did not find a noticeable performance penalty when running the refactored `ErcChecker`. Benchmark testing of the aspects[1] found that the aspect implementation was only 15 seconds slower than the direct implementation (even with gathering hit and miss data) for a loop of 9 million method calls.

Aspects, like application code, must be maintained. Using inheritance and abstract interface classes when implementing aspects improves cache pluggability, allows the aspect to be reused in many contexts, and that limits the change impact if the aspect or utility classes change. When designing caching as an aspect, subtle language features such as aliasing and domain knowledge about object equivalence are important to consider since they affect cache and application performance.

3. RUN-TIME USER CONFIGURATION

3.1 Motivation

A second type of extra–functional aspect we consider is run-time user configuration of the `ErcChecker`. ERC systems automate a number of electrical circuit checks. Example checks include checking for proper transistor ratios between the pull-up and pull-down transistors of an inverter, checking for fan-out limits, or checking for drive strength problems [12]. An ERC checking tool typically performs many types of checks on a circuit, reporting violations to the user.

The `ErcChecker` implements 58 different electrical checks as a class hierarchy. Because of the number of queries and

[1]Performance data are from a 2.4GHz Linux Xeon desktop system with 3 GB RAM.

the run-time of the tool (often several hours for a circuit), the `ErcChecker` allows users to turn any of the checks off through a user configuration file. Before time-intensive circuit traversal or electrical query evaluation, each of the 58 types of electrical calls a configuration method to see if the user has disabled that electrical check. Although this does affect the functionality of the final run, this functionality is orthogonal to the primary functionality: the actual electrical rules checking.

3.2 Aspect Identification

In the `ErcChecker`, each class has a static method, `createQueries()`, that calls `ErcQuery::getQueryConfig` before creating and evaluating query objects to see if the query is disabled:

```
void InformAlwaysOnPassGate::createQueries
(bcmInstance *fet)
{
   // See if this query is turned off
   if (ErcFet::getQueryConfig
      ("InformAlwaysOnPassGate")== eOff)
      return;
   ...
```

Because the `ErcFet::getQueryConfig` call to check for run-time user configuration crosscuts all electrical queries, and its behavior is always the same (disable running the check if a user configuration command is found), it is a good candidate for an aspect. A query that fails to call `ErcFet::getQueryConfig` in this manner will function, but cannot be disabled by configuration commands, which is a system-wide policy for the `ErcChecker`.

The original C++ code embedded the literal string for the electrical check's name (such as "InformAlwaysOnPassGate"). This was done because C++ lacks reflection, which otherwise could query the name of the class and method name. AspectC++ provides this level of reflection through `thisJoinPoint`. In this application, the string literals match the class name for the method. Without such regularity, it would be difficult to refactor to an aspect and automatically calculate the electrical check context.

3.3 Design challenges

The aspect must first extract the electrical query name. We could simplify the aspect code by modifying `ErcFet::getQueryConfig()` to accept the join point signature and parse the string. Doing so would change the functionality of an existing behavior to depend on the aspect, which would limit aspect pluggability. In addition, `ErcFet::getQueryConfig()` is an overloaded function that can take additional arguments (such as an electrical net or transistor) to provide more fine-tuned user configuration. The fine-grained calls to `ErcFet::getQueryConfig()` with extra parameters are interspersed throughout the electrical checks and do not have a regular structure.

3.4 Aspect Implementation

The aspect for this concern uses the join point signature to get the calling context (`void InformAlwaysOnPassGate::createQueries (bcmInstance *fet)` and calls the same method as the original code (`ErcFet::getQueryConfig()`).

```
aspect QueryConfigAspect {
   pointcut createQuery() =
      execution("% %%::createQueries(...)");
   advice createQuery() : around() {
     std::string jpName = JoinPoint::signature();
     int first_space = jpName.find(' ');
     int scope_operator = jpName.find("::");
     std::string className=jpName.substr(
          first_space+1,
          scope_operator-first_space-1);
     if(ErcFet::getQueryConfig(className)==eOff)
        return;  //user config exists, SKIP
     tjp->proceed();
   }
};
```

The aspect above handles the call to `ErcFet::getQueryConfig()` that occurs at the beginning of the static method `createQueries` for each `ErcQuery` class, but does not handle the additional fine-grained user configuration calls mentioned in the design challenges section. The fine-grained calls do not occur in a regular way that can be described by an AspectC++ pointcut. Instead, they are associated with loops and conditional statements in the body of the `createQueries` method for each class.

3.5 Impacts on the ERC application

Using an aspect for the call to `ErcFet::getQueryConfig()` inside the `createQueries()` method for each electrical query class improves the modularity of the code. Since the scattered code is only a single method call, there is not significant code reduction, but using an aspect provides an automated way to enforce the policy that each class check the code. A query class that does not call `ErcFet::getQueryConfig()` will not provide the desired user configuration.

The policy itself can easily be enabled, modified, or disabled for the `ErcChecker`. Other VLSI CAD applications developed at Hewlett-Packard provide similar configuration options, although the implementation varies by application. We are investigating how easily this aspect, developed for the `ErcChecker`, can be reused by other applications.

4. RELATED WORK

Lohmann, Gal, and Spinczyk [9] demonstrate that C++ template metaprogramming can be used to develop code with an aspect-oriented style, but without the obliviousness of aspects: everything must be explicitly instantiated through templates, which need to have the extension points designed in. Lohmann, Blaschke, and Spinczyk [8] have shown that aspects in AspectC++ can use templates and template metaprogramming. Their work on caching influenced our work, but their focus is on using template metaprogramming to create a single aspect that can cache a function with an arbitrary number of parameters, while our focus is on the implementation trade-offs that occur in making any cache pluggable and reusable.

Lohmann et al. [10] use AspectC++ to modularize extra-functional (also called non-functional) aspects. They focus on emergent properties that result from architectural properties, while we focused on explicit non-functional aspects. They also observe that while non-functional aspects are feasible, the implementation can contain challenges and subtle details that require in-depth analysis of their viability.

Duclos, Estublier, and Morat [3] consider mixing component-based design with aspect-oriented programming, which is

similar to our goal of using aspects with existing frameworks. Their approach, rather than extending frameworks with a single aspect language, proposes providing two languages so that component development and component use with aspects are done separately.

5. CONCLUSIONS AND FUTURE WORK

Aspects are effective for modularizing non-functional crosscutting concerns. We explored two types of non-functional concerns: caching and user run-time configuration. Although aspects were well suited for both, we identified specific issues that should be considered when using aspects:

- Non-functional aspects should be designed (where appropriate) to monitor their own usefulness.
- Non-functional aspects should make use of stand-alone classes, aspect inheritance, interfaces, and other features that promote ease of maintenance of aspects.
- Subtle implementation issues such as object identity and object equivalence can complicate seemingly simple aspects.
- When refactoring to use an aspect, obtaining the context (such as the query name in the user configuration aspect) may be different in the aspect code than the equivalent functionality in the (original) base code. For example, the original base code might not have a regular naming structure that facilitates using pointcuts.
- A non-functional aspect may only be able to be *partially* factored to an aspect, as our `QueryConfigAspect` demonstrated.

These issues should not discourage the use of aspects, but can provide guidance and insight as aspect-based refactoring and design is performed.

The aspects improve pluggability of the non-functional features. The caching aspect is generic enough to be reused in multiple applications. The run-time configuration aspect modularizes a feature that is used in other VLSI CAD applications, although its current implementation may be too closely tied to how run-time configuration is done in the `ErcChecker`.

We continue to identify aspects for refactoring the ErcChecker. This work may yield additional insights into design considerations, and may identify more aspects. We are also interested in seeing if the fine-grained calls to `ErcFet-::getQueryConfig()` described in Section 3.3 can be refactored to one or more additional aspects.

6. ACKNOWLEDGMENTS

The authors are grateful for the use of AspectC++ (www-.aspectc.org) and for valuable feedback from Andreas Gal and Olaf Spinczyck through the AspectC++ mailing list.

7. REFERENCES

[1] Si2: Silicon integration initiative. http://www.si2.org/?page=72.

[2] D. Baeumer, G. Gryczan, R. Knoll, C. Lilienthal, D. Riehle, and H. Zuellighoven. Domain-driven framework layering in large systems. *ACM Comput. Surv.*, 32(1es):5, 2000.

[3] F. Duclos, J. Estublier, and P. Morat. Describing and using non functional aspects in component based applications. In G. Kiczales, editor, *Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002)*, pages 65–75. ACM Press, Apr. 2002.

[4] S. Ghosh, R. B. France, D. M. Simmonds, A. Bare, B. Kamalakar, R. P. Shankar, G. Tandon, P. Vile, and S. Yin. A middleware transparent approach to developing distributed applications. *Software Practice and Experience*, 35(12):1131–1154, October 2005.

[5] S. Hanenberg, R. Hirschfeld, R. Unland, and K. Kawamura. Applying aspect-oriented composition to framework development - a case study. In *International Workshop on Foundations of Unanticipated Software Evolution, FUSE 2004*, Barcelona, Spain, 2004 (to be published).

[6] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In C. Norris and J. J. B. Fenwick, editors, *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA-02)*, volume 37, 11 of *ACM SIGPLAN Notices*, pages 161–173, New York, Nov. 4–8 2002. ACM Press.

[7] R. Laddad. *AspectJ in Action.* Manning Publications Co., Grennwich, Conn., 2003.

[8] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In *Third International Conference on Generative Programming and Component Engineering (GPCE'04).* ACM, 2004.

[9] D. Lohmann, A. Gal, and O. Spinczyk. Aspect-Oriented Programming with C++ and AspectC++ (Tutorial). In K. Lieberherr, editor, *Proc. 3rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2004).* ACM Press, Mar. 2004.

[10] D. Lohmann, O. Spinczyk, and W. Schrder-Preikschat. On the configuration of non-functional properties in operating system product lines,. In D. H. Lorenz and Y. Coady, editors, *ACP4IS: Aspects, Components, and Patterns for Infrastructure Software*, Mar. 2005.

[11] L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In E. Jul, editor, *ECOOP '98 — Object-Oriented Programming, 12th European Conference , Brussels, Proceedings*, volume 1445 of *Lecture Notes in Computer Science*, pages 355–382. Springer-Verlag, July 1998.

[12] S. M. Rubin. *Computer Aids for VLSI Design.* Addison-Wesley VLSI Systems Series. Addison-Wesley, 1987.

[13] D. Schmidt and M. Fayad. Object-oriented application frameworks. *Communications of the ACM*, 10(40):32–38, 1997.

[14] J. van Gurp and J. Bosch. Design, implementation and evolution of object oriented frameworks: concepts and guidelines. *Software Practice & Experience*, 10(31):277–300, October 2001.

APPENDIX

A. ABSTRACT FUNCTION CACHE

```
%#include <map>
aspect AbstractFunctionCache {
   pointcut virtual ExecAroundArgToResult() = 0;
   advice ExecAroundArgToResult() : around()
   {
      JoinPoint::Result *result_ptr = NULL;
      static std::map <
        typename JoinPoint::Arg<0>::Type,
        typename JoinPoint::Result > theResults;
      JoinPoint::Arg<0>::Type *arg_ptr =
          (JoinPoint::Arg<0>::Type*) tjp->arg(0);
      JoinPoint::Arg<0>::Type arg = *arg_ptr;
      if( theResults.count( arg ) ) {
         //already have the answer, return it
                result_ptr = tjp->result();
         *result_ptr = theResults [ arg ];
      } else {
         //proceed and store the answer
         tjp->proceed();
             result_ptr = tjp->result();
         theResults [ arg ] = *result_ptr;
      }
   }
};
```

B. TRACKING CACHE

```
struct Stats {
   int hits, misses;
   Stats() : hits(0),misses(0) {}
};
std::ostream& operator <<(std::ostream& os,
                          const Stats& stats) {
   os << "(Stats hits: " << stats.hits
      << " misses: " << stats.misses << ")";
   return os;
}
aspect AbstractTrackingMethodCache {
  pointcut virtual ExecAroundMethod() = 0;
  std::map<std::string, Stats> theStats;
  advice execution("% main(...)") : after()
  {
     std::map<std::string,Stats>::iterator iter;
     for(iter = theStats.begin();
       iter != theStats.end(); ++iter) {
       std::string jpName = (*iter).first;
       Stats  jpStats = (*iter).second;
       std::cerr << " JP: " << jpName
                 << " "<< jpStats << std::endl;
     }
  }
  advice ExecAroundMethod() : around()
  {
     JoinPoint::Result *result_ptr = NULL;
     static std::map<typename JoinPoint::Target*,
       typename JoinPoint::Result > theResults;
     JoinPoint::Target *target = tjp->target();
     if( theResults.count( target ) ) {
       //cache already had answer, return it
       result_ptr = tjp->result();
       *result_ptr = theResults [ target ];
       theStats[JoinPoint::signature()].hits++;
     } else { //not in cache, proceed and store
       tjp->proceed();
       result_ptr = tjp->result();
       theResults [ target ] = *result_ptr;
       theStats[JoinPoint::signature()].misses++;
     }
  }
};
```

C. CACHE CLASS FOR CACHING ASPECT

```
template < typename Key, typename Value>
class AspectCache {
protected:
   std::map< Key, Value> theData;
public:
   AspectCache() {}
   virtual ~AspectCache() {} //important for OO
   virtual bool has(Key& k)
   {  return (theData.count(k)>0); }
   virtual void add(Key& k, Value& val)
   {  theData[ k ] = val; }
   virtual Value& get(Key& k) {
      assert( has(k) ); //ensure we have it
      return theData[k];
   }
};
```

D. CACHING ASPECT THAT USES CACHE CLASS – PARAMETER VERSION

```
aspect AbstractFunctionCache {
   pointcut virtual ExecAroundArgToResult() = 0;
   advice ExecAroundArgToResult() : around()
   {
      JoinPoint::Result *result_ptr = NULL;
      static AC::AspectCache <
         typename JoinPoint::Arg<0>::Type,
         typename JoinPoint::Result > theResults;
      JoinPoint::Arg<0>::Type *arg_ptr =
         (JoinPoint::Arg<0>::Type*) tjp->arg(0);
      JoinPoint::Arg<0>::Type arg = *arg_ptr;
      if( theResults.has( arg ) ) {
         result_ptr = tjp->result();
         *result_ptr = theResults.get( arg );
      } else {
         tjp->proceed();
         result_ptr = tjp->result();
               theResults.add( arg, *result_ptr );
      }
   }
};
```

E. COUNTING ASPECT CACHE

```
template < typename Key, typename Value>
class CountingAspectCache :
public AspectCache <Key,Value> {
```

```
protected:
   int hits, misses;
   std::string joinPointName;
public:
   CountingAspectCache(std::string jpName) :
      AspectCache<Key,Value>()
      {hits=misses=0; joinPointName = jpName;}
   ~CountingAspectCache() {
      std::cerr << "Deleting CountingAspectCache,"
                << " jp=" << joinPointName
                << " hits="  << hits
                << " misses =" << misses
                << std::endl;
   }
   void add(Key& k, Value& val) {
      misses ++;
      AspectCache<Key,Value>::add(k,val);  }
   Value& get(Key& k) {
      hits++;
      return AspectCache<Key,Value>::get(k); }
};
```

F. OBJECT IDENTITY EXAMPLE

```
/* Simple C-style struct, and a
   function that calculates its area */
struct Square {
    int x1,y1,x2,y2;
};
void InitSquare(Square* s, int x1,int y1,
                int x2, int y2) {
   s->x1 = x1; s->y1 = y1;
   s->x2 = x2; s->y2 = y2;
}
int AreaOfSquare(Square *s) {
   int w = s->x1 - s->x2;
   int l = s->y1 - s->y2;
   int area = w*l;
   if(area<0)
      area = -area;
   return area;
}
int main() {
   Square *s1,*s2,*s3,*s4;
   Square s5;
   s1 = (Square*) malloc(sizeof(Square));
   InitSquare( s1, 0,0, 2,2);
   s2 = (Square*) malloc(sizeof(Square));
   InitSquare( s2, 3,3, 4,4);
   s3 = (Square*) malloc(sizeof(Square));
   InitSquare( s3, 0,0, 2,2);
   /*s3 and s1 have the SAME contents but
     are different memory objects */
   InitSquare( &s5, 4,4, 7,7);
   s4 = &s5;  /* s4 is a pointer to s5 */
   /* consider a sequence of calls that call
      AreaOfSquare while cached by an aspect*/
   int y = AreaOfSquare(s1);
   y = AreaOfSquare(s2);
   y = AreaOfSquare(s3);
   y = AreaOfSquare(s4);
   y = AreaOfSquare(&s5);
}
```

Coordination as an Aspect in Middleware Infrastructures

Mercedes Amor, Lidia Fuentes, Mónica Pinto
Departamento de Lenguajes y Ciencias de la Computación,
University of Málaga, 29071 SPAIN
{pinilla,lff,pinto}@lcc.uma.es

ABSTRACT
In this paper we discuss the shortcomings derived from having coordination and computation tangled in the same software entities and from having coordination protocols scattered through the several components participating in an interaction. We show a possible solution to this problem by using aspect-oriented techniques to separate coordination as an independent entity in middleware infrastructures.

Keywords
Coordination, AOSD, MultiTEL, CAM/DAOP, MALACA

INTRODUCTION
Developing distributed applications can be seen as the combination of two distinct activities: a computing part that comprises programming a number of entities (objects, components, agents, web services) involved in manipulating data, and a coordination part responsible for the communication and cooperation between these entities.

Given a set of possibly heterogeneous computational entities, the purpose of the coordination paradigm is to provide mechanisms and primitives to specify the synchronized interaction for putting all these components together, and make them interact in such a way that form a single application. This paradigm provides a clean separation between individual software components and their interactions within their overall software organization. This separation, together with the high-level abstractions offered by coordination models, allows viewing computational entities as black boxes, promoting their reuse in different applications.

Most standard middleware infrastructures directly support some kind of coordination mechanisms based on traditional coordination models such as the publish-and-subscribe, the tuple-based and the blackboard models [1]. However, such mechanisms are not sufficient for managing complex interaction protocols, typically comprising several lines of code. For instance, in the publish and subscribe coordination model, the support for expressing patterns about distributed events and algorithms for detecting correlations among these events are still largely unexplored [2]. By failing to support separate abstractions for representing such complex protocols, most standard middleware infrastructures force programmers to distribute and embed them inside the interacting components..

In this paper we propose a possible solution to this problem encapsulating the coordination among a set of software entities in an aspect. The coordination aspect is defined as an entity that encapsulates the interaction pattern or coordination protocol that governs the communication and interchange of information among two or more software entities. A coordination protocol can be defined as the list of messages and/or events that a software entity is able to send and receive, along with a set of coordination rules that state the order in which messages and events must be interchanged by the participant entities of the interaction. A coordination protocol can be specified by a STD (State Transition Diagram), or any other similar formalism. Other works for enhancing traditional coordination model by embedding the description of coordination information in a third-party entity are starting to appear [2,3,4,5].

After this introduction the paper is organized as follows. Next section briefly describes the state of the art of coordination in component-based and multi-agent domains. We continue with our motivation to separate coordination as an aspect and with the solutions we have adopted to separate coordination in MultiTEL [6], CAM/DAOP [7] and MALACA [8]. Some initial discussion about how coordination might need to interact with other aspects in the middleware infrastructure is presented in the following section. Finally, we present the conclusions of the paper.

STATE OF THE ART
Coordination plays a central role in technologies used to develop distributed applications such as component technologies, web services and agent technologies.

With respect to component-based technologies, they usually provide coordination mechanisms based on the publish-and-subscribe coordination model. Examples of these are CORBA, CCM/CORBA and J2EE. The main advantage of this mechanism is that communication is anonymous. Therefore, suppliers and consumers of events are decoupled among them due to the use of a third-party object that acts as intermediary between them. However, by using this mechanism software developers do not achieve a complete separation between computation and coordination for two main reasons: (1) the coordination model is spread throughout the objects acting as suppliers, consumers and event channels. That is, suppliers do not simply throw events that are intercepted by the middleware infrastructure and managed by the coordination entity. Instead, suppliers and consumers need to create and/or

localize the event channels, including code related to the particular coordination model they follow as part of the code of the objects that implement consumers and suppliers; (2) the event channel does not usually provide support to encapsulate complex interaction protocols. In consequence, this service is mainly suitable for applications that just need to be aware of "change notifications". Basically, a supplier produces an event and all the consumers registered in the event channel will receive the notification that such event was produced. The third-party objects in the Jini Distributed Event Service are an exception to the latter shortcoming. The reason is that Jini does not impose anything about the implementation of third party objects. They just need to implement the interfaces to register or to notify events but do not need to extend a particular object of a specified type. Therefore, Jini provides an event based programming model plus the possibility of encapsulating a coordination protocol in the third party objects. As commented before, other proposals that extends the publish and subscribe model to address the coordination of activities between decoupled components are [2,3,4]. Finally, ObjectPlace [5] is another proposal extended the traditional tuple space coordination model with support for role-based coordination between individual components.

Coordination also plays an important role in web services that use a loosely coupled integration model to allow flexible integration of heterogeneous systems in a variety of domains including business-to-consumer, business-to-business and enterprise application integration. However, the use of standards such as SOAP, WSDL and UDDI and interaction following a loosely coupled is not enough for such integration. Systems integration requires an additional layer able to describe and perform web services composition and orchestration. The latter term is the description of interactions and message flow between services in the context of a business process. This concept is not new; in the past it has been called workflow. Until now, different XML-based languages have been introduced to cover web services orchestration where two of the more well-known are the Web Services Choreography Description Language (WS-CDL) [9] and the Web services Business Process Execution Language (WS-BPEL) [10]. WS-CDL describes the set of rules that explains how different web services may act together, and in what sequence, giving a flexible and integral view of the process. WS-BPEL enables the composition of existing web services into a more complete web service. The composition is defined in terms of a workflow process consisting of a set of activities and the composition itself is exposed as a web service.

Nowadays, Multi-Agent Systems (MAS) are an effective paradigm for the design and implementation of complex software applications. Agent-based applications are understood as a system consisting of autonomous agents whose interactions are coordinated through an organizational structure [11]. The necessity of coordinating agents in MAS relies on the common idea that an agent should be able to engage in, possibly, complex communications with other agents in order to exchange information or to ask their collaboration in pursuing a goal. Currently two coordination models are the most representative and effectively used to realize agents coordination mechanisms: Interaction protocols and Shared Dataspaces. A large portion of the agent community, which includes the standardizing organism FIPA, considers coordinating agents using interaction protocols, i.e. predetermined patterns of interaction. This approach is based on considering coordination essentially as a problem of communication. Thus the Agent Communication Language (ACL) plays a fundamental role, providing a means to achieve a higher-level of interoperability between agents. We can find in the agent models supporting this coordination model that the functional part of the code of an agent is interleaved with code that is there only to support coordination. This dependency derived from the intermingled code makes difficult the reuse of the agent functionality in other applications and platforms.

MOTIVATION

The conclusion is that coordination models provide support for loosely coupled interaction, mainly focusing on decoupling the source and the target of the communication. However, they do not achieve a complete separation among computation and coordination, since they do not provide support for the description of complex interaction protocols. As stated before, a complex interaction protocol specifies not only the information interchanged by coordinated entities but also the coordination rules.

To illustrate this shortcoming we will focus on an auction system and, concretely, in the simplified interaction between the seller and the buyer. In this example, Figure 1 shows the entities participating in an auction, while Figure 2 describes a scenario of the interaction among them. However, when the system is finally implemented using, for instance, a component based approach, the coordination information shown in Figure 2 is usually lost. That is, the coordination protocol among buyers and sellers is directly hard coded as part of the implementations of the *Seller* and the *Buyer* components. As a consequence, there may be situations in which a change in the interaction protocol requires changing the component implementations.

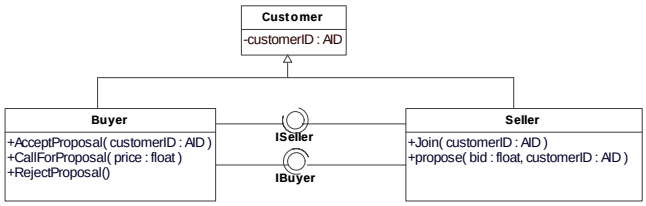

Figure 1. Part of the design of an Auction System.

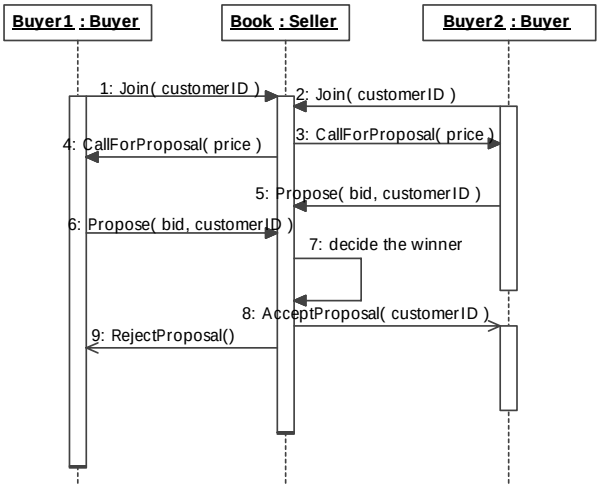

Figure 2. Buyer – Seller Interaction Protocol.

In Figure 2 when an auction finalizes, the *Seller* component decides the winner bid (step 7) and notifies to just one of the *Buyer* components that his/her bid won, using the *AcceptProposal(customerID)* operation (step 8). The other buyers are notified that they did not win invoking the *RejectProposal(customerID)* operation (step 9).

Let us suppose now that all the buyers have to be notified about which is the winner. This is a change in the coordination among the components and, with the current approach, implies to modify the implementation of the *Seller* component. Thus, the seller has now to invoke the *AcceptProposal(customerID)* operation over all the buyers.

This shortcoming may be overcame if the seller and the buyers use, for instance, the publish-and-subscribe interaction style. In this case, components register in a common communication channel. The use of a third-party object facilities the 1-to-many communication. This means that the seller just publishes an anonymous *AcceptProposal(customerID)* event and all the buyers subscribed to that event will receive the notification.

Another change may be to make the auction *public* (one in which all bids are shared among the participants). We desire to (re)use the *Buyer* and *Seller* components in Figure 1 and Figure 2. In this case the *Seller* component should have to notify the *Buyer* components that a new bid is proposed, and this requires modifying its implementation. Concretely, after receiving the *Propose(bid,customerID)* operation invoked by *customerID* buyer, the Seller has to notify the bid to the rest of buyers using the *CallForProposal(bid)* operation. Once again this is part of the interaction protocol codified crosscutting the functionality of the Seller component. In this case, even using the publish-subscribe interaction style it is needed to change the implementation of the seller component. The reason is that this change modifies the interaction protocol, which is tangled with computation and scattered throughout the participant components. The solution to this problem comes across not including the coordination rules as part of the components. This information should be included in the third-party object of coordination models. However, traditional coordination models do not provide support for this.

The extension of a coordination model with the necessary support to encapsulate complex interaction protocols may avoid to hard code this information as part of the coordinated entities. However, most of the realizations of a coordination model for particular technologies introduce some non desirable dependencies among the coordinated entities. For instance, the event service in CORBA follows a publish-and-subscribe coordination model. However, its implementation uses RPC (Remote Procedure Call). This implies that the channel is a CORBA object and it has to be located by the components before using it.

This problem can be alleviated if the responsibility of locating the adequate channels falls to the middleware infrastructure. This approach is followed in the CCM model of CORBA, where components just throw events and it is the container the responsible of managing channels.

The next section will show how AOSD mechanisms provide a natural solution to solve the limitations of the coordination models discussed in this section.

SEPARATING COORDINATION IN MIDDLEWARE INFRASTRUCTURES

In this section we describe our experience separating coordination as an aspect in three different middleware infrastructures we have developed: MultiTEL, CAM/DAOP and MALACA. Readers can obtain detailed information of these middleware infrastructures in [2,3,4].

A common feature of these infrastructures is that coordination is separated from computation and moved into a coordination aspect (Figure 3, label 1). Other important feature is that the weaving information is not part of the definition of components or aspects. Instead, it is explicitly described and allocated in the middleware infrastructure (Figure 3, label 2), which is the responsible of weaving components and the coordination aspect at runtime. This is very useful for implementing applications in open systems, in which components evolve over time. Thus, components do not need to choose or localize the proper coordination aspect, since this is the middleware infrastructure responsibility. This characteristic increases the reusability of components in different contexts. As shown in Figure 3, both load-time weaving and runtime weaving mechanisms may be provided, with divergent outcomes on flexibility and performance.

Other advantage of separating the coordination aspect is that it is easier to control the different states of a complex interaction (Figure 3, label 3).