Aspect-Oriented Programming and Separation of Concerns

Proceedings of the International Workshop
Lancaster University, UK
24 August 2001

Awais Rashid, Lynne Blair (eds.)

Computing Department, Lancaster University
Technical Report No. CSEG/03/01

http://www.comp.lancs.ac.uk/computing/users/marash/aopws2001/

Preface

These proceedings contain the position papers and extended abstracts accepted for presentation at the Workshop on Aspect-Oriented Programming (AOP) and Separation of Concerns held on 24 August 2001 at Lancaster University, UK. The goal of the workshop was to increase awareness about AOP and other separation of concerns mechanisms in the UK computer science community by bringing together both UK-based and international researchers working in these areas. The workshop was aimed at providing a forum to present on-going or completed research work and exchange ideas about outstanding research issues. A tutorial was held before the workshop on 23 August 2001 to provide an introduction to AOP concepts and hands-on experience with AspectJ™: an AOP language from Xerox PARC which enables writing aspect-oriented programs in Java™. The tutorial was aimed at people relatively new to the area. Since the tutorial was held before the workshop, tutorial participants attending the workshop had an opportunity to expand their knowledge to state of the art research in the area.

The accepted papers for the workshop covered a wide but coherent set of topics including adaptive systems, persistence, mapping and automation, middleware and future directions for AOP. We wish to extend our thanks to the authors of the papers for their contribution to the workshop. We also wish to express our gratitude to Cooperative Systems Engineering Group at Computing Department, Lancaster University for providing funding for the workshop and the British Computer Society Advanced Programming Group for their help in advertising the workshop. In addition, we wish to thank Aimee Doggett, Chris Needham and Michelle Spence for local arrangements. Last but not least we thank all the participants without whom the workshop would not have been possible.

Awais Rashid
Lynne Blair

August 2001

Table of Contents

Miscellaneous

Grouping Objects using Aspect-Oriented Adapters

Stefan Hanenberg, Rainer Unland

Institute for Computer Science
University of Essen, D - 45117 Essen
{shanenbe, unlandR}@cs.uni-essen.de

Abstract. Aspect-Oriented Programming (AOP) is an approach for realizing separation of concerns and allows different concerns to be weaved into existing applications. Concerns usually cross-cut the object-oriented structure. Whenever a concern needs to invoke some operations on objects of the given structure the problem arises, that those objects have different types, but the concern expects them to be handled in the same way. Therefore a mechanism for grouping objects of different types is needed.This paper discusses different mechanisms and proposes aspect-oriented adapters for grouping types and shows how this approach permits a higher level of flexibility and reduces the limitations of known approaches. Aspect-oriented adapters are not limited to a specific general purpose aspect language (GPAL). Nevertheless the examples in this paper are realized in AspectJ, which is by far the most popular and well-established general purpose aspect language.

1 Motivation and Problem Description

Let us assume we want to make objects persistent, which are created by an existing simulation-application. As pointed out in [3] persistency is a concern and so this is a typical application of Aspect-Oriented Programming [4]. Every newly created object should be added to a persistent storage and whenever the state of a certain object changes, its representation on the store must be updated. There is no need to offer an interface for retrieving objects, because the simulation itself does not use former objects. Instead the information is used by another application which directly accesses the storage for retrieving information about the simulation. The objects to be stored are all instances of class `Point`.

A suitable (straight-forward) solution for this problem in AspectJ [5] would be an aspect, which writes the state to the store every time an object is created and whenever its state changes (fig. 1). An instance of the aspect `PersistentPoint` is created for every `Point` instance. The aspect generates an object id (realized as an instance counter) and stores it in its attribute `id`. After creating a new `Point` the object is written to the persistent storage realized in the constructor of `PersistentPoint`.

The state of a point changes, whenever the methods `setX()` or `setY()` are invoked. Therefore a *pointcut* `setPC()` is defined for any instance of `Point` receiving a set-message. Whenever this happens the corresponding *pointcut method* (or *advice* in the AspectJ terminology) is executed which reads a point's state (`getX()`, `getY()`) and updates the persistent storage.

```
class Point {
  private float x=0;
  private float y=0;
  public float getX() {
    return x;}
  public float getY() {
    return y;}
  public void setX(float x) {
    this.x = x;}
  public void setY(float y) {
    this.y = y;}
}
```

```
aspect PersistentPoint
      of eachobject(instanceof(Point)){
  private static int idnum = 0;
  private int id = ++idnum;
  public PersistentPoint() {
    .. write new (unitialized) object to storage}
  pointcut setPC(Point p): instanceof(p) &&
            (receptions(void setX(float)) ||
             receptions(void setY(float)));
  after(Point p):  setPC(p) {
    float x = p.getX());
    float y = p.getY());
    …update x,y of object id}
}
```

Figure 1: a) Class `Point`, b) Aspect `PersistentPoint`

Let us assume there is another (similar) application having its own implementation of a point `AnotherPoint` identical to `Point`. The proposed solution directly depends on the class `Point` and cannot be used for other classes. Therefore it would be more desirable to define a persistency aspect without being limited to class `Point`.

AspectJ supports inheritance relationships between aspects and allows to declare abstract aspects, so it seems to be a good choice to define an abstract aspect `PersistentObject`, which is responsible for creating the object id and reading the object's state (fig. 2, see [2] for a detailed discussion on inheritance and AOP). Its subaspects only have to define the class this aspect should be weaved to. Therefore `PersistentObject` contains an abstract pointcut `weavedClassPC()`, which has to be defined by the subaspects. We want the aspects to be instantiated for every instance of `Point` and `AnotherPoint`, so the definitions of `weavedClassPC()` in our concrete aspects corresponds to that.

But now a new problem arises: how can the state of the object be read in the pointcut method? The intention of the aspect is to be woven to classes, having the methods `getX()`, `getY()`, `setX(float)` and `setY(float)`. The set-methods are used for the pointcut definition, and the get-methods are needed by the aspect instance to read an object's state. But although knowing those method signatures the concrete type of those classes is unknown and left to those aspects, which make the abstract pointcut concrete. Because aspects crosscut the inheritance structure of classes usually those classes do not have any common type but `java.lang.Object`. So it is not possible to send getter-messages to the related object, because the type is unknown and therefore a typecast is not possible.[1] A pos-

[1] We assume here general purpose aspect languages with static type checking like AspectJ or Sally [8] which are both based on the programming language Java.

sibility would be to use reflection for those method calls, but that requires an enormous effort.

```
abstract aspect PersistentObject of
      eachobject(weavedClassPC) {
  private static int idnum = 0;
  private int id = ++idnum;
  public PersistentObject() {
    .. write new (unitialized) object to storage}
  abstract pointcut weavedInstances(Object o);
  pointcut setPC(Object o): weavedInstances(o) &&
           (receptions(void setX(float)) ||
            receptions(void setY(float)));
  after(Object p):  setPC(p) {
            ..write state to data storage}
}
```

```
aspect PersistentPoint
    extends PersistentObject {
  pointcut
   weavedInstances(Point p):
   instanceof (p);
}

aspect PersistentAnotherPoint
    extends PersistentObject {
  pointcut weavedInstances
    (AnotherPoint p):
    instanceof (p);
}
```

Figure 2: abstract persistency aspect (trial)

The concrete problem is, that aspect-oriented programming groups objects in another way than the predefined object-oriented structures do. So a mechanism is needed how to group objects of different types and allow to sent messages to them.

In the next section we discuss approaches related to this problem and demonstrate that they do not solve this problem appropriately. Afterwards we introduce and discuss *aspect-oriented adapters* for grouping types and show how this approach allows a higher level of flexibility and reduce the limitations of other approaches. We will also apply the adapter to the introducing example. In the forth section we map the introducing example to aspect-oriented adapters. Finally we summarize and conclude the paper.

2 Related Work

AspectJ offers a mechanism called *introductions* which can be applied to the given problem. The mechanism allows aspects to change the structure of the object-oriented classes. In this way additional methods and attributes can be inserted into existing classes. For the purpose of grouping objects introductions allow to insert new types to the target classes. So the interface needed by an aspect has to be defined and afterwards integrated into those classes.

Applied to the example from the first section that means, that an interface needed by the aspect `PersistentObject` has to be specified. This interface has to contain the getter-methods the aspect needs to invoke for reading a point's state. The concrete aspects `PersistentPoint` and `PersistentAnotherPoint` have to introduce this interface to the classes `Point` and `AnotherPoint`. The type of the parameter in the pointcut and pointcut method must be of that introduced interface, so the advice can invoke the getter-methods.

But this way to handle the problem leads to additional problems:

- *Tangled introduction statements*: The introduction-statements in every sub-aspect logically belong to the abstract aspect. They have to be implemented redundantly and are in that way tangled.
- *Confusing class structure*: After weaving the classes of the original application implement a new interface (fig. 3). If a lot of aspects are woven this approach leads to numerous interfaces spread all over the class structure. In this way the original code becomes confusing and makes it difficult for developers to understand the original code for reasons of reuse.
- *Lack of structure after unweaving*: After weaving developers extending the application can use the common interfaces introduced by the aspects, because they cannot distinguish the original interface from the introduced ones. So after unweaving the aspects the original classes do not implement those interfaces any longer, and so the extended application is incorrect.

Because of this we regard introductions to be inappropriate for the given problem. The problem of grouping objects has already been discussed widely in the context of object-orientation. Classes are templates from which objects are created (cf. [10]) and in that way group objects. [9] pointed out the importance of classification as a mechanism for conceptual modeling in object-oriented programming. The difference to the problem handled here is that the needed classification is not an inherent property of the objects, but depends on an aspect's subjective perspective on the system.

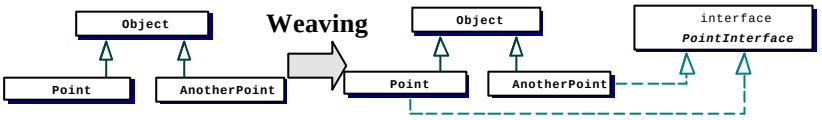


Figure 3: Using introductions for `Point` and `AnotherPoint`

In that way the mechanism of generalization introduced in [7] seems to be appropriate for the problem stated. Generalization permits to define a super-type based on an existing class. In [7] one of the main purposes of generalization is to achieve a late classification. That is exactly what aspects are doing: while they cross-cut an existing structure they accomplish a late classification for their special purposes.

Neglecting the fact, that generalization is not available in popular object-oriented programming languages, the criticism of that mechanism is corresponds to the of criticism introductions in AspectJ: developers extending the original application can not distinguish between the original classes and those created for the purpose of late classification. Also the problem of the confusing class structure stays the same.

3 Aspect-Oriented Adapters

Adapters (cf. [1]) are special classes, which adapt the interface of a class in the way expected from its clients. In that way the functionality of adapters match the problem

stated above. The traditional use of adapters for mapping interfaces assumes, that clients expect a certain interface of a class which differs from that class which is able to fulfill the requests. The problem depicted here is different: advices expect their parameters to have some method signatures to send messages to them. Although the signatures are known, the type of those objects is unknown, respectively those objects do not have any common type. So an adapter is needed which has the interface expected by the aspect and which forwards messages to a certain object.

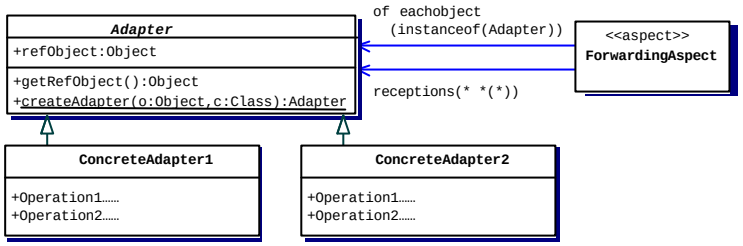


Figure 4: Aspect-Oriented Adapters[2]

An object-oriented solution for this problem is quite complex: the type of the object to which the messages have to be forwarded is unknown, so the developer has to use reflection to realized it. This code has to be used in every method which means an enormous effort.

```
public aspect ForwardingAspect
      of eachobject(instanceof(Adapter)) {
    ...
    around() returns Object: receptions(* *()) {
      ...
          return invokeMethodFromReceptionsJoinPoint((ReceptionJoinPoint) thisJoinPoint);
      ...}
    private Object invokeMethodFromReceptionsJoinPoint(ReceptionJoinPoint jp) throws Exception {
      Method m = getMethodFromReceptionJoinPoint(jp);
      return m.invoke(((Adapter) jp.getExecutingObject()).refObject, jp.getParameters());}

    private Method getMethodFromReceptionJoinPoint(ReceptionJoinPoint jp) throws Exception {
          Adapter wrap = (Adapter) jp.getExecutingObject();
          MethodSignature sig = (MethodSignature) jp.getSignature();
          String methodName = sig.getName();
          Class[] paramTypes = sig.getParameterTypes();
          return wrap.refObject.getClass().getMethod(methodName, paramTypes);}
}
```

Figure 5: Example-implementation for forward-adapter

The aspect-oriented solution for such adapters is much easier and allows a higher degree of reusability (figure 4). The abstract class `Adapter` contains the reference to the object to which every message is to be forwarded. The aspect `ForwardingAspect` is responsible for forwarding every message received by an instance of `Adapter`. Therefore an instance of `ForwardingAspect` is created for every instance of `Adapter` and the aspect contains pointcut methods, which forward every message received by the adapter to the corresponding object.

[2] The UML-like notation used here serves the understanding of the ingredients of the aspect-oriented wrapper, but does not match the UML standard.

The aspect-oriented adapter is used by creating a `ConcreteAdapter`, subclass of `Adapter`, which contains all methods needed by the aspect. Those methods have to contain a dummy implementation needed for compiling the class. The implementation will never be executed, because the `ForwardingAspect` replaces it by forward implementations.

Whenever a client wants an object to be adapted, he has to create an adapter instance by invoking the static `createAdapter(..)`-method of `Adapter`. The parameters of this method are an instance of the object which is about to be adapted, and a reference to the concrete adapter class. The adapter uses reflection to create a new instance of the concrete adapter and initializes `refObject` with the adapted object. The developer doesn't have to write glue code for forwarding messages, because this is already done by the `ForwardingAspect`.

Figure 5 shows an extract from the implementation of a forward aspect in AspectJ. The advice overrides every method of the adapter having an arbitrary return type. This is realized by a receptions pointcut consisting only of wildcards. The implementation uses the Reflection API part of AspectJ for finding out, what the target method is and the Java Reflection API for getting a reference to and invoking the target method .

For applying the aspect-oriented adapter to the introducing example a concrete adapter (`PointAdapter`) has to be created containing both getter-methods used by the advice in `PersistentObject`. The advice has to create an adapter object for the incoming object using the create method of the abstract adapters:

```
PointAdapter a = (PointAdapter) Adapter.createAdapter(p, PointAdapter.class);
```

Afterwards object a can be used as if it is an instance of `PointAdapter`.

4 Conclusion and further work

We introduced aspect-oriented adapters as a mechanism for grouping objects and compared it to existing approaches. The main advantage of using adapters is, that objects can be grouped without touching the existing inheritance structure. The effort of using aspect-oriented wrappers is compareable to introductions known from the GPAL AspectJ.

Nevertheless aspect-oriented adapters need to be used very carefully and in a disciplined maner. Because forwarding messages is realized on object-level using reflection there is no static type-checking available. So the developer has to be sure, that the interface of the adapted object really fulfills the signatures specified in the concrete adapter.

This presented approach can be used for composing *aspectual components* [6], which represent aspects whose interfaces have to be adapted to let them interact with their environment. In the future we will examine, how such components can be realized in existing general purpose aspect languages.

References

1. Gamma, E., Helm R., Johnson, R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995
2. Hanenberg, S., Unland, R.: *Concerning AOP and Inheritance*. In: Mehner, K., Mezini, M., Pulvermüller, E., Speck, A. (Eds.): Aspect-Orientation - Workshop. Paderborn, Mai 2001, University of Paderborn, Technical Report, tr-ri-01-223, 2001
3. Hürsch, W., Lopes C.: *Separation of Concerns*. Northeastern University, technical report, no. NU-CCS-95-03, 1995.
4. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., Irwing, J.: *Aspect-Oriented Programming*. Proceedings of ECOOP '97, LNCS 1241, Springer-Verlag, pp. 220-242, 1997
5. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: *An Overview of AspectJ*. Appears in ECOOP 2001
6. Lieberherr, K., Lorenz, D., Mezini, M.: *Programming with Aspectual Components*, Technical Report, NU-CCS-99-01, Northeastern University, Boston, 1999
7. Pedersen, C.: *Extending Ordinary Inheritance Schemes to Include Generalization*. In: Meyrowitz, N. (Ed.): Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA'89), October 1-6, 1989, New Orleans, Louisiana, Proceedings. SIGPLAN Notices 24(10), October 1989
8. Sally: A General-Purpose Aspect Language, `http://www.cs.uni-essen.de/dawis/research/ aop/sally/`, January 2001
9. Taivalsaari, A.: *On the Notion of Inheritance*. ACM Computing Surveys, Vol. 28, No. 3, pp. 439-479, 1996
10. Wegner, P.: *Dimensions of object-based language design*. In: Meyrowitz, N. (Ed.), Proceedings of OOPSLA '87, SIGPLAN Notices 22 (12), pp. 168-182, 1987

From Software Parameterization to Software Profiling

Philippe Bouaziz[1,2] and Lionel Seinturier[1]

[1] Univ. Paris 6, Lab. LIP6, 4 place Jussieu, F-75252 Paris cedex 05, France
{Philippe.Bouaziz, Lionel.Seinturier}@lip6.fr
[2] Prodware Group, 45 quai de la Seine, F-75019 Paris, France

Abstract. Over the last years, software engineering research applied to separation of concerns has focused on new software paradigms such as Aspect Oriented Programming. Aspects are abstractions which capture and localize crosscutting concerns. Many works have been conducted with regard to non functional concerns such as performance or semantics of component. This paper wants to demonstrate their interest for functional concerns. It introduces a new software engineering process that we call Software profiling which represents a step further in software parameterization using functional/non-functional aspects to provide highly adaptable and evolving software.

1 Introduction

Over the last years, software engineering research applied to separation of concerns has focused on new software paradigms such as Aspect Oriented Programming [8][7]. It aims at optimizing software development by providing tools to improve modularization, so that it corresponds to the natural view of concerns such as defined by developers [10]. Aspects are abstractions which capture and localize crosscutting concerns e.g. code which cannot be encapsulated within one class but that is tangled over many classes. Synchronization, failure handling, load balancing, real time constraints, memory management, optimization are classical examples of aspects. Many works have been conducted with regard to non functional concerns such as performance or semantics of component whereas crosscutting concerns are widely presents in codes dealing with functional aspect.

Research in AOP is in its early stage with programming tools available, but its contribution to software development process is still not clearly defined, especially in term of penetration degree in the functional part. We think that capabilities of aspect are broader than system and environmental concerns, and that they can be widely used in the software development process, and especially concerning software parameterization that has been the last ten years goal in software industry.

We present here the foundation of a software engineering process that we call Software profiling which represent a step further in software parameterization by

using functional/non-functional aspects to provide highly adaptable and evolving software. It permits to obtain clear, modular and strongly evolving software that can be profiled statically or dynamically depending on the problematic of the user with no alteration of the standard source code which is only concerned by standard evolutions, keeping this way ascendant compatibility in software versions.

First of all, we describe quickly Aspect Oriented Programming. In a second part we present the goals and fundamentals of Software profiling and its contribution to software engineering industry. Finally we consider the benefits of a Case tools for software profiling as a base for further works.

2 AOP

Aspect Oriented Programming aims to achieved separation of concerns by improving code modularization using Aspects. Most of the time, aspects represent non-functional requirements. In AOP, components and aspects are separate codes and the weaving process is done by a static (compile time) or dynamic (run time) compiler that is called an aspect weaver. The aspect weaver, weaves aspects and components at specific points named join points which can be implicit such as language keywords or explicit. Specific code is then added at this points. [6] classify join points among open, class-directional, aspect-directional, and closed depending on the fact that the aspect or the class knows about each other or not:

- Open: Both classes and aspects know about each over,
- Class-directional: the aspect knows about the class,
- Aspect-directional: the class knows about the aspect,
- Closed: neither the aspect nor the class knows about the other.

AspectJ [1] and the Composition Filter Object Model (CFOM) [2] are some of the leader tools in AOP. AspectJ is an aspect-oriented extension to Java that is being developed at the Xerox Palo Alto Research center. It offers a language to define a new kind of module, called an aspect. Aspects are defined separately from the standard code. AspectJ provides a static aspect weaver in Java, and other development tools. It is widely used in AOP research community, and version 1.0 is due next fall. A version of AspectJ for C [4] is under development. CFOM is a project developed by the Trese group. The composition filter approach [2] extends objects with filters that deal with inter-objects messages. Input and output filters are used to localize aspect code. Other AOP approach exists, among them Subject Oriented Programming [5], Adaptive Programming [9], and other language extension like AOP/St for Smalltalk [3].

3 Software profiling

3.1 Software parameterization

Customer requirements and needs tend to evolve as technology, business and company processes advance. Software developed in the last decade with initial

customer needs in mind tend to be unsuitable to follow these changes as no proper design technique is available for this. Development teams that try nevertheless to achieve this goal, end up with source code more and more difficult to maintain, upgrade, and reuse. Along a long embryo period, software is install with many difficulties and is in permanent beta stages to meet new requirements or requirements that emerge from the analysis.

The increasing speed of technological evolution over the last twenty years and the fact that computer software became more and more common, created a clear need for rationalizing software processes to deliver cheaper products, with short integration times, quality-oriented maintainability and evolution capabilities to guarantee a longer software life.

To achieve this goal, software industry evolved, just as the textile industry did with ready-made clothes, and committed itself in developing standard business software, based on common needs of a significant community of customers. Base on this, customizations are being made possible with parameters that reflect the various management practices of customer organizations. These software are developed by editors that provide maintenance, that are permanently auditing their market, and that are arbitrating the functional and technological changes of software.

These days, the existing level of parameterization existing in major products, such as ERPs[1] (e.g. SAP), provides many important features to meet needs of various industry fields. Nevertheless, the level of adaptability of these products to non-mutualizable features is weak due to the complexity introduced by the huge number of available parameters. Most of the time, this adaptation is done in an *ad-hoc* manner. Faced with this problem, partial solutions have been proposed based on object technologies, n-tiers architectures and in the industry, relational databases. They allow delocalizing treatments such as reporting, that can (re) become specific, and to slightly amend software based on entry points, triggers or stored procedures, to better integrate it with the information system.

Nevertheless, for simple needs such as ascending compatibility, the behavior of the application or of the data model can never be altered. This prevents a straight and optimized answer to above mentioned issues. Furthermore, programs end up being tangled due to the many possibilities introduced by parameterization and cannot be reuse.

3.2 The benefits of functional aspects

Software design aims at: (1) factorizing functionalities, and (2) allowing that these functionalities be parameterized in order to meet customer specific needs. From an industrial point of view, the modification of these parameters can induce too deep modifications of the software internal structure preventing the addition of new or customer specific features. Many technical solutions exist, but they bring either an overload of scattered code, or a parameterization of existing parameters, and in all cases are to difficult to maintain (each case needs to be

[1] Entreprise Resource Planing

individually treated). Aspect oriented development should allow to standardize and centralize these specificities in order to avoid overloading and tangling.

3.3 Software profiling

The notion of aspect allows to profile software depending on the needs and notably:

- to encapsulate in a clear and centralized way, parameters leading to massive cross-cutting all along the code,
- to perform without altering the main code, modifications or extensions in functional part, for example altering the behavior of business objects to obtain a different action on the information flows, and even its replacement by another object,
- to encapsulate in a clear and centralized way system features such as synchronization, load balancing, real time, ...,
- to ease code reuse,
- to bring capabilities of dynamic reactivity to the software depending on the evolution of profiles such as dynamic design of GUI and contents (for instance to profile Internet applications furnished by ASPs - application service providers).

Usage of aspects in parameterized software therefore allows to profile software depending on data and strategy defined in static or dynamic ways. The pending difficulty is to be able to define what in a profile is relevant to aspects.

4 Further works

As mentioned before, parameterized software design, is a global process. To be a part of it, the notion of aspect should impact all levels, analysis, design and implementation. To do so, CASE tools should integrate this notion, furnishing an environment taking all this considerations in account. We can imagine to find there :

- a set of rules to help decide whether to use aspects or not,
- an extension to existing methods to take aspects into account during the analysis phase,
- a graphic design tool to describe aspects and their relationships with other aspects and components,
- an environment for managing and testing projects integrating aspects characteristics.

Some works have been realised in this sense such as UML/UXF [11] for the design phase or AJDE [1] as a development environment. Nevertheless, as far as we know, no break throws have been made in terms of designing aspects and their relationships or about methods, or decisional purpose. Our goal is to provide a CASE tools for software profiling design taking all this needs in account. One of the first steps will be to define, method specifics, components specifications and rules.

5 Conclusion

We show in this paper the interest of introducing aspects in software parameterization, and by the way demonstrate the capabilities of aspects in the functional field of software development. Aspects in the development process will bring clear, modular, strongly evolving and adaptable parameterized software. In this article we enlarged the scope of aspect to functional domains. We are now working, on defining rules to decide where and when aspects should apply. The next step will be for us to define fundamentals for Software profiling, especially in terms of method specifics, components specifications and rules.

References

1. AspectJ home page. `http://www.aspectj.org`.
2. Aksit, M., Wakita, K., Bosch, J., and Bergmans, L. Abstracting object interactions using composition filters. vol. 791 of LNCS, pp. 152–184.
3. Bollert, K. On weaving aspects. In Workshop Aspect-Oriented Programming at ECOOP'99 (June 1999). `http://trese.cs.utwente.nl/aop-ecoop99/`.
4. Coady, Y., Kiczales, G., Feeley, M., Hutchinson, N., and Ong, J. Structuring system aspects. In Proceedings of the workshop on Aspect-Oriented Programming at ICSE'01 (2001).
5. Harrison, W., and Ossher, H. Subject-oriented programming (A critique of pure objects). In OOPSLA 1993 Conference Proceedings, A. Paepcke, Ed., vol. 28 of ACM SIGPLAN Notices. ACM Press, Oct. 1993, pp. 411–428.
6. Kersten, M., and Murphy, G. Atlas: A case study in building a web-based learning environment using AOP. In Workshop Aspect-Oriented Programming at ECOOP'99 (June 1999). `http://trese.cs.utwente.nl/aop-ecoop99/`.
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. An overview of AspectJ. In Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01) (2001), Lecture Notes in Computer Science.
8. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. Aspect-oriented programming. In Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP'97) (June 1997), vol. 1241 of Lecture Notes in Computer Science, Springer, pp. 220–242.
9. Lieberherr, K. Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston, 1996. `http://www.ccs.neu.edu/research/demeter/biblio/dem-book.html`.
10. Parnas, D. On the criteria to be used in decomposing systems into modules. Communications of the ACM 15, 12 (1972), 1053–1058.
11. Suzuki, J., and Yamamoto, Y. Extending UML with aspects: Aspect support the design phase. In Workshop Aspect-Oriented Programming at ECOOP'99 (June 1999). `http://trese.cs.utwente.nl/aop-ecoop99/`.

Aspect-Based Workflow Evolution

Boris Bachmendo and Rainer Unland

Department of Mathematics and Computer Science
University of Essen, D - 45117 Essen
{bachmendo, unlandR}@cs.uni-essen.de

Abstract. In this position paper we propose an approach for the flexible evolution of object oriented workflow implementations using AOP. We show how reusable aspects can apply changes (e.g. insertion of activities or control flow constructs) to OMG compliant implemented processes. Besides aspects providing different workflow auditing methods can trigger necessary alternations. In that way a cyclic workflow improvement can be realized.

1. Introduction

Aspect-Oriented Programming is a new software engineering paradigm which supports a separation of concerns. Concern composition is realized by extending programming language with special constructs: joinpoints (which define relevant points for the insertion of concern-related code in the application class structure), pointcuts (which describe interactions between joinpoints) and pointcut-methods (also known as advises, define action to be performed before, after or instead the invocation a certain pointcut is activated by) [6].

Different perspectives of workflow modelling and implementation, e.g. control flow (execution order), data flow (data interchange) or resources are described in [7]. The applicability of AOP for supporting flexible workflow execution was first identified in [12], that propose implementing these perspective separately using AOP and weaving them together in a workflow application.

Although workflow management arose from automating well structured repetitive production processes, the need for supporting dynamic altering workflows, e.g. in office and scientific areas, is obvious nowadays. [3] distinguish between *static workflow evolution*, i.e. modifying workflow models, and *dynamic evolutio*n, i.e. adapting running process instances. Unlike [12] we propose dynamic evolution of existing object oriented workflow implementations by weaving appropriate aspects. This approach allows the reuse of both adaptation cases realized as aspects (e.g. insertion of control flow constructs) and workflow implementations. At the same time aspects implementing arbitrary auditing methods can be used to control process execution and trigger workflow evolution when necessary.

In the next section we briefly present an object-oriented workflow implementation approach. Possible evolution scenarios of control flow as well as resource and auditing perspective are described in the third section. Section four summarizes the paper and discusses some open issues.

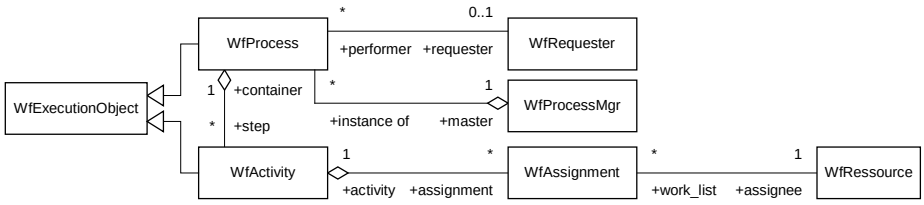


Fig. 1. Simplified Workflow Management Facility Model

2. Object-Oriented Workflow Implementation

First workflow management systems (WfMS) had a highly centralized architecture with a single workflow engine (e.g. FlowMark) or multiple replicated execution units (e.g. MOBILE [7]). But an optimal level of flexibility and scalability can only be provided by a truly distributed object-oriented implementation, where workflows are realized as independent distributed objects. In order to standardize object-oriented WfMS and to make them compliant to Object Management Architecture the OMG proposed the Workflow Management Facility Specification. Since we will explain our approach using this model as a reference, we will briefly outline it in the following.

A simplified version of Workflow Management Facility Model [10] is depicted in Figure 1. `WfRequester` interface represents a request for some job to be done. `WfProcess` has to perform the work, inform its requester upon completion and deliver him the execution result. The requester uses `WfProcessMgr` to create a process instance, i.e. it is a factory and locator for the instances of certain process type. `WfExecutionObject` is a basic interface which contains common members of `WfProcess` and `WfActivity`. While a `WfProcess` implements an instance of a particular process model, the single process steps are represented by `WfActivity`. The process creates corresponding activities and defines its context (i.e. input data). The result that is produced by activity can be used to determine the following one. `WfActivity` can also act as `WfRequester`, thereby the process it creates becomes a subprocess of its owner. `WfResorce` represents a resource necessary for activity execution.

3. Workflow Adaptation Using Aspect-Oriented Programming

3.1 Control Flow Perspective

First of all we consider the control perspective and describe how the control flow can flexibly be changed using AOP. Figure 2 depicts the insertion of activities and control flow constructs, defined by the Workflow Management Coalition [14], such as sequence, split, join and iteration. Activity diagrams on the left side show a process fragment, while sequence diagrams to the right represent interactions between objects. Replaced control flows are depicted by dashed arrows in activity diagrams, while interactions contain some special constructs showing interceptions by the aspects.

In Figure 2a an activity (A1) which is an instance of `WfActivityA` is replaced by the instance of `WfActivityB` (A2). We assume `WfActivityA` and

`WfActivityB` to be subtypes of the OMG-interface `WfActivity`. In this case we use an aspect that has a pointcut activated by the invocation of an `WfActivityA`-constructor performed by `WfProcess` instance P. The corresponding pointcut method is executed *instead* of the original invocation. In the sequence diagram the original call is depicted as a dashed connector with a transparent dot at the beginning and transparent arrow at the end. Although this call is not executed it should especially be depicted for instead-invocations to clarify what pointcut was activated. The aspectual invocation is depicted by the connector between the caller object and the aspect instance with the black dot on the aspect side. It is labelled with the original method call.

In this case (Fig. 2a) the pointcut-method creates an instance of `WfActivityB` and returns it to P instead of a `WfActivityA`-object (we assume the process is handling its activities through the `WfActivity` interface). The context used for the creation of A2 can differ from the original data. A2 considers P as its owner process and reports it the execution results. Deletion of activities can be realized analogously by replacement by dummy activities.

In Figure 2b a new activity (A2) is inserted between two existing ones and all of them are executed in a sequence. The activity constructor invocation is once again intercepted by the aspect. But in contrast to the first example the pointcut method is executed *before* the constructor. It creates a new instance of `WfActivityB`. This activity cannot report its results to the process, because P is not aware of its existence and it would interpret the call as the result of A1. So the result is reported to the aspect and thereafter the intercepted constructer call is executed. The context passed over to A1 can be derived from the A2 result which in that way can influence the rest of the process.

Figure 2c shows an inserted AND-Split between the activities A0 and A1. A single thread of control now splits into two concurrently executing threads [14]: the old (starting with A1) and the new one (A2). In contrast to the previous case the aspect does not wait until A2 is finished before it continues the instantiation of A1. Therefore the context of A1 cannot be affected by A2, whose returning result is omitted since it is not relevant. An OR-Split (i.e. branching into several alternative threads) can be inserted analogously to the activity replacement with the help of an instead pointcut method. The method evaluates given conditions and decides on creating either A1 or A2 (XOR-Split) or both of them.

Multiple threads converge into a single one by using the join construct [14]. The insertion of an AND-join that merges parallel threads is depicted in the Figure 2d. Since the coordination of A0 and A1 is already handled by the process, the aspect has to ensure that A1 can only start after A2 is finished. Therefore one pointcut observes the final call of A2 that return the result and sets an internal flag as soon as it was executed. Another pointcut method intercepts the instantiation of A1 and lets it proceed only after the flag was set. If the converging branches are alternatives (OR-Join) the aspect has to detect the termination of the both, A0 and A2. It has to trigger the creation of A1 at the moment the first of these events occurs and has to prevent the instantiation when the second one takes place.

An iteration (i.e. repetitive execution of a process segment) is added in Figure 2e. Activity A1 is performed repeatedly as long as a certain condition is fulfilled. A before pointcut detecting the result delivery of A1 and starting it again and again, is

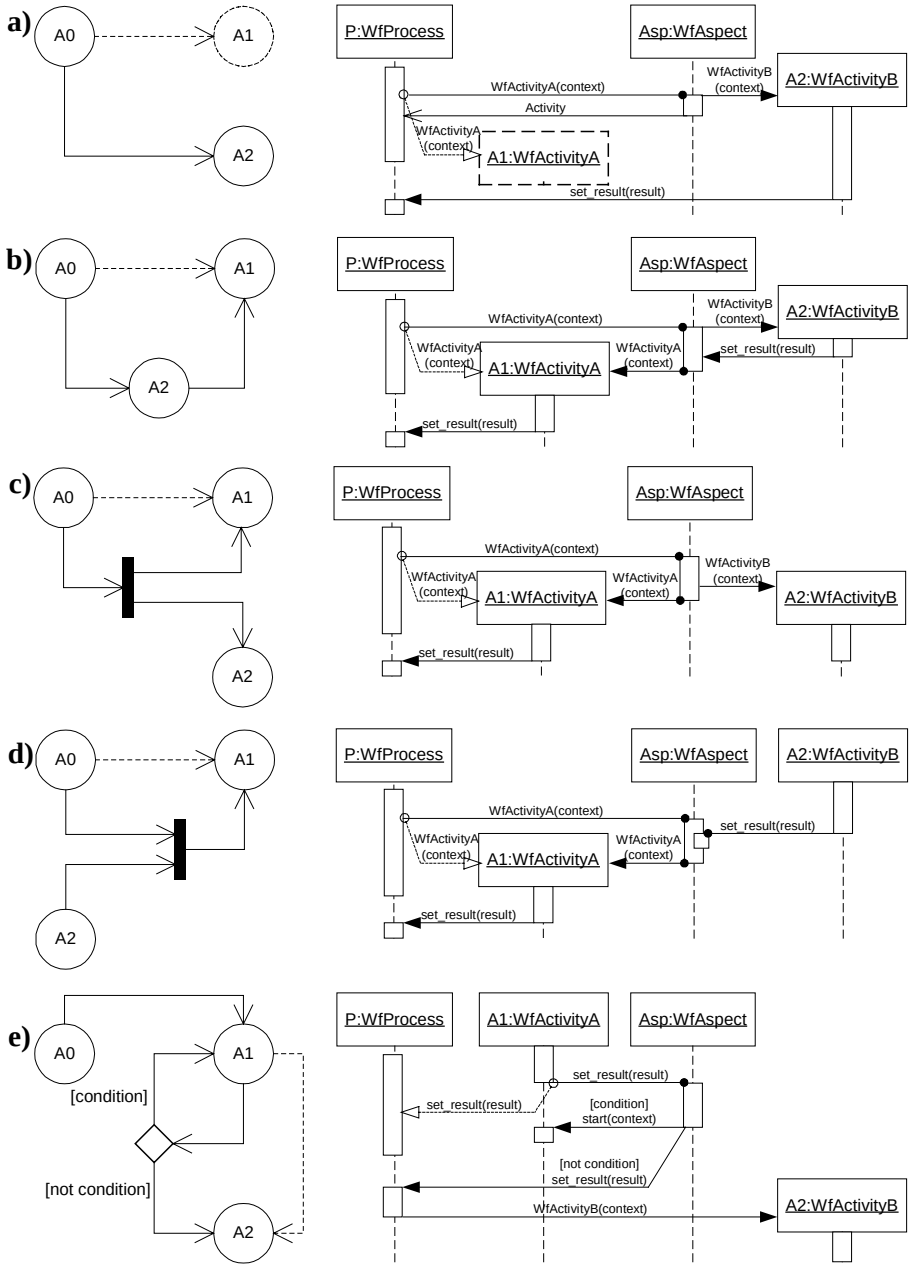


Fig. 2. Control flow adaptation using aspects.

not appropriate in this case, since it would mean that results are reported repeatedly to the process object, that cannot handle them, since it is not aware of the repetition. So the instead pointcut method is used. It checks the condition (which can be based on the returned result or an independent of A1) and either starts the activity once more or forwards the result of the last execution to the process. After the process object gets this result it instantiates the next activity A2.

3.2 Resources Perspective

Another workflow execution concern where flexibility is an essential requirement is the dynamic assignment of resources to activities. The WfMC differentiates four kinds of resources: human (person), organizational unit, role (e.g. the function of a person within an organization) and system (i.e. automated machine resource) [13]. As proposed in [15] the workflow resource model can be separated into the static meta model, the dynamic assignment rules and access synchronisation mechanisms. Although the frequent changes in the resource meta model are extremely seldom and, therefore, can be realized by redesigning the application, they can also be implemented by using aspectual *introductions*. It is a mechanism provided by the general purpose aspect language AspectJ [8] that allows extending given types or certain objects with additional member fields and methods.

Much more undecided and, therefore, changeable units are the dynamic assignment rules also referred to as policy resolution. They handle the resource assignment to process activities at runtime. In the Workflow Management Facility Specification of the OMG an object implementing the `WfAssignment` interface is responsible for linking `WfActivity` with `WfRessource` objects. It selects appropriate resources according to the given activity context and other process independent information. Although eligible resources are selected dynamically, the used resolution policy depends on the `WfAssignment` object the activity is related to. But often the assignment strategy itself has to be changed or extended dynamically. In this case reusable aspects can be used to either replace the assignment objects or extend the activity selection procedure by inserting additional code before or after it. Possible extensions can consider the actual workload (e.g. appropriate aspects can be dynamically added in overload situations) or history dependent assignment either in order to take advantage of personal experience or to ensure equal work partitioning [2]. On the other hand it can be necessary to replace a resource either for all assignments resp. activities (e.g. if an employee is absent and his work has to be delegated) or only for selected ones (e.g. for security reasons). This changes can also easily be realized by adding an aspect intercepting the resource invocations.

The third component of the workflow resource model mentioned above is the synchronisation of the concurrent access of multiple activities to a single resource. Since synchronisation of concurrent threads was the first application of AOP and the main purpose for the specification of the domain specific aspect language COOL [9], the suitability of aspects in this area doesn’t needs no further elucidation.

3.3 Auditing Perspective

Monitoring and logging of workflow executions as well as a comprehensive evaluation of recorded audit trails is an essential part of workflow management, since it closes the workflow development cycle comprising workflow identification,

modelling, implementation, execution and controlling [11]. The main tasks of workflow auditing are acquisition of execution data, its analysis and the utilization of the results. Both acquisition and utilization can be differentiated in short and long term, as well as active and passive approaches.

In the OMG Workflow Management Facility the acquisition of execution data is realized by the `WfEventAudit` and its subtypes which record certain types of workflow events (e.g. process or activity start and termination, context and result changes etc.). But this scheme means a passive way of acquisition, even if using the OMG Notification Service as proposed in [10], because only events published by workflow execution objects can be received. An active acquiring component can obtain arbitrary information it is interested in. It can be achieved by implementing the acquisition with the help of aspects. In the case of object-oriented (and especially OMG compliant) implementations all the relevant execution events can be detected by appropriate pointcuts. An arbitrary replacement or combination of multiple aspects without any modification of execution objects provides the necessary flexibility. For example the short term acquisition aspect providing an order processing status for a customer can be combined with an aspect implementing a long term history logging.

Using aspects modules implementing different analysis methods for audit data can be dynamically added or replaced. While passive utilization implies simple recording and/or visualisation of the results an active approach intends an intervention in the workflow execution. Long term utilization means changes to the process models that influence all future executions. Short term intervention concerns the current running process instances and can be realized by aspects adding or replacing activities and modifying control flow as described in the section 3.1, or changing the context data.

4 Summary

In this position paper we proposed an approach for the dynamic evolution of workflow instances by using aspects. It allows flexible process adaption and reuse of both the object-oriented process implementation and the adopting aspects. The changes can either be caused externally or triggered by the auditing component that can be realized by aspects too. In that way a cyclic workflow improvement can be realized.

Unfortunately the most implementations of aspect languages only support static aspect weaving at the pre-compile time (a good overview is e.g. offered by [4]). Though if using this languages a workflow has to be restarted, in order to be changed, the aspect-based adaption still allows the reuse of both primary workflow implementation and adapting aspects. Dynamic run-time aspect assignment is desirable, in order to realize automatic improvement cycle. The approaches allowing dynamic weaving are e.g. *AOP/ST* [1], that makes use of reflective capabilities of Smalltalk, or *Aspect Moderator Framework* [4], which is implemented in Java and introduces a special design pattern for objects aspects are assigned to. General purpose aspect language *Sally* [5] is an extension of Java realized by a pre-compiler. It also supports dynamic aspect assignment at the run-time.

Other potential application areas like process error handling are to be examined in the future. An open implementation issue is the aspect realization in distributed environments, which is especially important for workflow management systems.

References

1. Boellert, K.: On Weaving Aspects. In: Proceedings of the Aspect-Oriented Programming Workshop at ECOOP'99.
2. Bussler, Ch.: Policy Resolution in Workflow Management Systems. In: Digital Technical Journal, Vol. 6, No. 4, Maynard, MA: Digital Equipment Corporation, 1995.
3. Casati F.; Ceri S.; Pernici B.; Pozzi G.: Workflow Evolution. In: Proceedings of the 15 the International Conference on Conceptual Modeling, ER'96, Cottbus, Germany. Springer Verlag, Lecture Notes in Computer Science, 1996.
4. Constantinides, C.; Bader, A.; Elrad, T.: A framework to address a two-dimensional composition of concerns. In: Proceedings of the First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems at OOSPLA'99.
5. Hanenberg, S.; Bachmendo, B.; Unland, R.: An Object Model for General-Purpose Aspect Languages. To appear in the Proceedings of the Third International Conference on Generative and Component-Based Software Engineering (GCSE) 2001.
6. Hanenberg, S.; Unland, R.: Concerning AOP and Inheritance. In: Mehner, K., Mezini, M., Pulvermüller, E., Speck, A.(Eds.): Aspect-Orientation - Workshop. Paderborn, Mai 2001, University of Paderborn, Technical Report, tr-ri-01-223,2001
7. Jablonski, S.; Bussler, Ch.: Workflow Management. Modeling Concepts, Architecture and Implementation. International Thomson Computer Press. London et. al. 1996.
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. To appear in ECOOP 2001.
9. Lopes, C.: D: A Language Framework for Distributed Programming. A Ph.D. Thesis. College of Computer Science. November 1997.
10. Object Management Group: Workflow Management Facility Specification, Version 1.2. April 2000.
11. Rosemann, M.: Workflow Monitoring and Controlling. In.: Jablonski, S.; Boehm, M.; Schulze, W. (Eds.): Workflow Management. Development of Applications and Systems. Heidelberg 1997, pp. 201-210. (in German).
12. Schmidt, R.; Assmann, U.: Extending Aspect-Oriented-Programming In Order To Flexibly Support Workflows. In: Lopes, C.; Murphy, G.; Kiczales, G.: Proceedings of the Aspect-Oriented Programming Workshop at ICSE'98.
13. Workflow Management Coalition: Interface 1: Process Definition Interchange Process Model. Document Number WfMC TC-1016-P. Version 1.1. October 29, 1999
14. Workflow Management Coalition: Terminology & Glossary. Document Number WFMC-TC- 1011. February 1999.
15. zur Muehlen, M.: Resource Modeling in Workflow Applications. In.: Becker, J.; R zur Muehlen, M.; Rosemann, M.(Eds.): Proceedings of the 1999 Workflow Management Conference "Workflow-based Applications" in Muenster. November 1999.

Some Insights on the Use of AspectJ and Hyper/J

Christina Chavez[1,2], Alessandro Garcia[1], and Carlos Lucena[1]
{flach, afgarcia, lucena}@inf.puc-rio.br

[1] Computer Science Department, Pontifical Catholic University of Rio de Janeiro, Rio de Janeiro, Brazil.
[2] Computer Science Department, Federal University of Bahia, Bahia, Brazil.

Abstract. In this paper, we report some insights collected during the implementation of the `Portalware` system, using both AspectJ and Hyper/J, having as a starting point an aspect-oriented design model. One of the motivations behind this development exercise is a practical assessment of each approach's similarities and differences, strengths and weaknesses to help in the validation of a generic aspect-based design model that can be mapped to the implementation models supported by AspectJ and Hyper/J.

1 Introduction

The identification and structuring of software components, crosscutting aspects, and their relationships are essential tasks in order to produce effectively reusable components and aspects. However, these tasks are not trivial and consequently require a disciplined design approach as well as suitable implementation tools. In this context, two research activities concerned to multi-dimensional separation of concerns (MDSoC) are currently being held within the Software Engineering Laboratory (LES) at PUC-Rio: (i) the assessment and evaluation of applying aspect-oriented design and programming techniques to the design and implementation of multi-agent object-oriented software, and (ii) the assessment and evaluation of the strengths and weaknesses of approaches and tools that support MDSoC to help in the validation of a generic aspect-based design model under development.

An unifying subject of study for these activities is the development of the `Portalware` system [4], a web-based environment that applies groupware concepts in order to support a disciplined approach for the construction and management of e-commerce portals. Software agents were introduced in `Portalware` to assist its users with time-consuming activities and to automate repetitive user tasks. An innovative aspect-based agent model proposed in [5] was applied in the early design of the `Portalware` system, to allow the separation and flexible integration of multiple aspects of agenthood, such as interaction, autonomy, adaptation and others. Moreover, an exploratory implementation for `Portalware` using AspectJ [8] was developed. Some interesting results have been found and discussed elsewhere [6].

The resulting aspect-based design model (highly influenced by the AspectJ programming model) and the AspectJ/standard Java code were reused in the implementation of `Portalware` with Hyper/J [11], a tool that supports multi-dimensional separation and integration of concerns in standard Java software. The adoption of an aspect-based design model supports our belief that the base-aspect dichotomy simplifies concern-based modularity [10].

	AspectJ	Hyper/J
Crosscutting concern	aspect	hyperslice
Composition where how	 join point merge before, after, around	 corresponding unit merge, override
Composition Specification	inside aspect	outside hyperslice
Composition Precedence	"dominates" implicit rules	"order" hyperslice declaration
Composition Time	compile-time	compile-time

Table 1. Comparing AspectJ and Hyper/J

The primary purpose of this work is to report some insights we have got in implementing the same application, the `Portalware` system, using both AspectJ and Hyper/J, having as a starting point an aspect-oriented design model. In the following Section, we describe the mapping from the aspect-based design model to Hyper/J and discuss the insights we have gathered during the transformation process.

2 From Aspects to Hyperslices

The AspectJ programming model supports the base-aspect dichotomy. Crosscutting concerns are modularized by *aspects*. Composition between base and aspects is defined in terms of base-related *join points* [8]. Crosscutting behavior can be added *before*, *after* or *around* join points. Composition is defined inside aspects. Precedence among aspects is resolved implicitly (before, after, around rules) or explicitly (the *dominates* clause).

Hyper/J supports Hyperspaces [11], an evolution on the early work on subject-oriented programming (SOP) [7] that does not necessarily distinguish between base and crosscutting concerns. Concerns are modularized using *hyperslices*. Composition rules are defined in terms of *corresponding units* [11]. These units are related by *merge* or *override* relationships. Composition is defined independently from the hyperslices. Precedence among hyperslices is resolved explicitly (declaration of hyperslices in the hypermodule, the *order* clause). Table 1 summarizes these characteristics.

We have adopted a simple set of transformation rules, based on (i) rewriting each aspect to one or more classes encapsulated by a separate hyperslice, (ii) transforming advice code into ordinary method code and (iii) adopting the *mergeByName* general composition strategy. This decision was highly influenced by the fact that some constructs from Hyper/J are very limited (for example, the *bracket* directive[1]) or not available yet (the *merge* composition relationship to be used with the *equate* relationship) [11].

2.1 An AspectJ Solution

The AspectJ solution comprised 3 overlapping aspects (Interaction, Autonomy and Adaptation) and almost 50 classes. The following initial condition held, simplifying the transformation process:

[1] The *bracket* directive resembles the before/after advice constructs from AspectJ.

– The Java source files were available and compilable.
– Each aspect/class was defined in a separate file.
– Every pointcut used only the "executions" primitive pointcut.
– For each aspect definition, no before/after advices were defined over the same pointcut and there were no around advices.

2.2 The Transformation

A set of transformation rules were applied following the steps described below:

1. Separate files with standard Java code from files with AspectJ code[2].
2. Create a new file .hs (hyperspace file), declaring the class names that will live in the hyperspace.
3. Create a new file .hm (hypermodule file).
4. For each aspect $A_i, i = 1, n$ and $j \neq i$ with header
   `aspect` A_i `[dominates` A_j `] of eachobject(instanceof(CName))`
   (a) Create a package named A_i
   (b) Create a file named A_i/concerns.cm containing: `package` A_i : A_i`.Kernel`
5. Declare in the file .hm
   (a) The "base" hyperslice
   (b) The hyperslices A_i.Kernel, for each aspect $A_i, i = 1, n$ and $j \neq i$, containing
   `aspect` A_i `[dominates` A_j `] of eachobject(instanceof(CName))`
   where, if `Ai dominates Aj`, then A_i.Kernel is declared after A_j.Kernel[3].
   (c) The general composition strategy: `mergeByName`
6. For each aspect $A_i, i = 1, n$ and $j \neq i$ such that
   `aspect` A_i `[dominates` A_j `] of eachobject(instanceof(CName))`
   (a) Create a file named A_i/`Cname`.java and declare class `Cname` in it
   (b) Copy introductions from aspect A_i to class `Cname`
   (c) For each pointcut that contains `executions(T methodName)`,
      i. Define new method methodName with body B defined in the advice
      ii. If advice defined on pointcut is `before`
      Declare in the file .hm
      ```
      order  action Ai.Kernel.CName.methodName
             before action Agent.Kernel.CName.methodName;
      ```
      iii. If T $\neq$ void, create a summary function and declare `set summary function` in the file .hm.

2.3 An Hyper/J Solution

The resulting solution in Hyper/J was structured in an hyperspace with four dimensions (Agent, Interaction, Autonomy, Adaptation), four main hyperslices (Agent.Kernel, Interaction.Kernel, Autonomy.Kernel, Adaptation.Kernel), an hyperspace file, one concern map file for each dimension, an hypermodule file and a set of standard Java classes. Each source aspect file with an AspectJ aspect declaration has generated one or more Java files, depending on the requirements of declarative completeness imposed by Hyper/J.

[2] The original files with standard Java were not rewritten.
[3] For overlapping through after advices.

2.4 Discussion

Skipper [10] compares subject-oriented programming and aspect-oriented programming and drops interesting conclusions shared by this development experiment. For example, the base-aspect dichotomy, with explicit dependency on some "base" whose vocabulary is shared among the "aspects", may be regarded as a discipline that may simplify the development of MDSoC solutions, especially the rules of composition.

Lai, Murphy and Walker [9] describe an experiment with Hyper/J and discuss some code restructuring used to enable the capturing and composition of concerns. In this experiment, we have shared both the reported benefits and limitations presented ([9], section 4.2), although we did not restructure classes; instead, we have rewritten aspects into one or more classes. During the transformation process, we have noticed that AspectJ's aspects and named pointcuts are good mechanisms for improving the readability of the crosscutting code. By restructuring each aspect definition into multiple classes, rewriting advice code into ordinary methods, and fully using the *mergeByName* composition strategy in Hyper/J, we have simplified the transformation process, but we may have lost some of the reported benefits provided by aspects and named pointcuts [1].

Nevertheless, when the initial conditions reported in section 2.1 do not hold, the transformation may require a lot of additional work, and possibly, some kind of refactoring in some methods inside the aspect definition. One source of problem for the translation to Hyper/J is something that usually happens in AspectJ programs: the addition of something to the base via an introduction and a subsequent advice defined on it, all inside the same aspect. Since we can not have two methods with the same signature in a class, this would require a more complex rewrite rule, with different naming convention for the advice and the inclusion of an additonal *equate* clause in the hypermodule file. For a complete report on the problems found during this experiment, see [2].

The experiment also allowed us to evaluate how the composition mechanisms of each tool work in practice. In AspectJ, even with the expressive power of wildcards in pointcuts, it may be eventually necessary to invasively modify an aspect definition, because the composition between aspects and classes is hardwired in the aspect header. Moreover, the use of *dominates* establishes a strict relationship among aspects, and consequently, among *all* their pointcuts. It would be interesting if the designer could specify exceptions to the composition order for some of the pointcuts. Hyper/J provides a more elegant and flexible solution, by requiring the definition of the composition strategy explicitly outside the hyperslice definition and by allowing the specification of exceptions to the composition strategy and to the composition order among units (also at the level of operations). This approach increases the potential for non-invasive modification and reconfiguration. Finally, it is worth mentioning that an important requirement stated for `Portalware` – the attachment of different aspects to distinct instances of agents [5] – has not been fulfilled due to current limitations of the composition mechanisms of both tools.

3 Conclusions

We have reported some insights collected during the implementation of the same application, the `Portalware` system, using both AspectJ and Hyper/J, having as a starting point an aspect-oriented design model. These insights refer not to the maturity or performance of each tool, but to their ability to separate and compose concerns, and the correspondence among their programming elements.

We believe that, had we started this experiment with hyperslices instead of aspects, the transformation process would pose more difficulties, partially because the composition mechanisms provided by Hyper/J are more powerful and flexible than those provided by AspectJ, partially because we would have some additional work to consider the hyperslices under the base-aspect dichotomy perspective. In [3], Clarke argues that there is a potential for a relatively clean mapping from Composition Patterns – a design model inspired in the MDSoC model prescribed by SOP – to Hyper/J code, while the mapping to AspectJ code may be subject to scattering and tangling in aspects.

We expect that the generic aspect-based design model we have been working on allows the characterization and comparison of existing and new approaches, as well as an easy mapping from the design model to different programming models that support separation of concerns.

References

1. B. Alwis et al. Coding Issues in AspectJ. In *Int'l Work. on Advanced Separation of Concerns at OOPSLA*, 2000.
2. C. Chavez, A. Garcia, and C. Lucena. An Experience Report on the Use of AspectJ and Hyper/J, 2001. Tech. Report, PUC-Rio (to appear).
3. S. Clarke and R. Walker. Mapping Composition Patterns to AspectJ and Hyper/J. In *Int'l Conference on Software Engineering (ICSE 2001)*, May 2001.
4. A. Garcia, M. Cortes, and C. Lucena. A Web Environment for the Development and Maintenance of E-Commerce Portals based on a Groupware Approach, 2001. Information Resources Management Association Int'l Conference (IRMA 2001).
5. A. Garcia and C. Lucena. An Aspect-Based Object-Oriented Model for Multi-Agent Systems. In *Advanced Separation of Concerns in Software Engineering at ICSE'2001, Toronto*, May 2001.
6. A. Garcia, C. Lucena, and D. Cowan. Engineering Multi-Agent Object-Oriented Software with Aspect-Oriented Programming, 2001. Submitted to Software: Practice & Experience, Elsevier, April 2001.
7. W. Harrison and H. Ossher. Subject-Oriented Programming (A Critique of Pure Objects). In *Proceedings of OOPSLA'93*, pages 411–428, 1993.
8. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ, 2001. (http://aspectj.org/documentation/overview/aspectj-overview.pdf).
9. A. Lai, G. C. Murphy, and R. J. Walker. Separating Concerns with Hyper/J: An Experience Report. In *Int'l Workshop on Multi-Dimensional Separation of Concerns in Software Engineering at ICSE*, 2000.
10. M. Skipper. The Watson Subject Compiler & AspectJ (A Critique of Practical Objects). In *Workshop on Multi-Dimensional Separation of Concerns at OOPSLA*, 1999.
11. P. Tarr and H. Ossher. Hyper/J User and Installation Manual, 2000.

Translation of Java* to Real-Time Java Using Aspects **

Extended Abstract

Morgan Deters, Nicholas Leidenfrost, and Ron K. Cytron***

Washington University Box 1045
Department of Computer Science
St. Louis, MO 63130 USA

Abstract. The Real-Time Specification for Java [1] (RTSJ) introduces the concept of nested-scope memory areas to Java. This design allows a programmer to allocate objects in areas that are ignored by the garbage collector. Unfortunately, the specification of scoped memory areas currently involves the introduction of unwieldy, application-specific constructs that can ruin the reusability of the affected software.
We propose the use of aspects [2], in particular the AspectJ [3] system, to transform a Java program into a scope-aware RTSJ program automatically. Moreover, we have developed analysis that automatically determines storage scopes, in response to information provided by an instrumented form of the application at hand. That instrumentation is also accomplished using aspects. Here we present our ongoing work in using aspects to detect and specify memory scopes automatically in Java programs.

1 Motivation

One major roadblock to the widespread acceptance of Java as a language for real-time and embedded systems is its reliance at runtime on an asynchronous garbage collector. Such a collector may preempt threads running in a real-time environment or take control of the system during memory allocation requests. The collector can then take an unbounded amount of time to complete its task, introducing unacceptable unpredictability into the system. To address these concerns and provide greater programmatic control over memory allocation and usage, the Real-Time Specification for Java [1] (RTSJ) introduces to Java the use of structural *memory areas* through the `MemoryArea` type, the subtyping of which can admit multiple strategies for allocating and deallocating storage.

One of the key memory strategies of RTSJ is provided with the `ScopeMemory` type, which allows for memory areas to be associated with a particular

* A registered trademark of Sun Microsystems

** Funded by the National Science Foundation under grant 0081214 and by DARPA under contract F33615-00-C-1697

*** Contact author: mdeters@cs.WUSTL.edu

scope of thread execution. When an execution scope is exited, objects in the memory area are released without the need for a garbage collector. However, the unit of a thread is not necessarily the most natural or useful level at which to create and manipulate memory areas. Consider the introduction of a `Cache` type into a system (Figure 1(a)). A singleton `Cache` object is constructed when the cache is first accessed. In a corresponding RTSJ design, we want this object to be allocated in `ImmortalMemory`, because the singleton `Cache` is around for the length of the program. Placing it in `ImmortalMemory` keeps the garbage collector from scanning or marking it. This would be a waste of time since we know that it will never become collectible: the scope of the `Cache` object is global. To realize this RTSJ design, the programmer replaces occurrences of `new Cache` with invocations of `newInstance` on the desired memory area, as shown in Figure 1(b).

If class `Cache` had a more complicated design that supported multiple instances, this recoding of `new` instructions to `newInstance` invocations would need to be pushed into the user's code at *every* site that a `Cache` object was constructed. This breaks the encapsulation of the `Cache` type. Memory instructions are sprinkled throughout user code rather than being factored out into a separate, decoupled memory strategy. The resulting code will not be easily reusable.

Figure 1(c) shows a better approach. With the `CacheMemory` aspect, we can write the `Cache` class just as we did in the Java implementation; the aspect assumes the responsibility of placing it into the correct type of memory area. Further, if we extended `Cache` to support multiple instantiations, then `CacheMemory` could weave into user code, allowing the user to construct a `Cache` instance naturally, without being responsible for its memory requirements.

We propose an automatic system for translating Java code into `MemoryArea`-aware RTSJ code by determining the necessary RTSJ memory scopes required to describe it. By employing *probing aspects*, we determine where scopes can be used and develop a graph from which we can compute provably legal scope hierarchies. A scope hierarchy is selected, runtime execution points are chosen for opening and closing scopes, and an aspect is generated to enforce the structure on the original program's execution. Objects in the modified program are located in scoped memory areas rather than in the garbage-collected heap.

2 Scope Determination

As specified by RTSJ, a thread becomes associated with a scope when that thread calls `enter` on the scope. The scope then resumes the thread by calling its `run` method. Any number of threads can enter a scope, and that scope can be deleted only when all threads that have entered it have exited their `run`

```
class Cache {
 protected static Cache singleton;
 public static Cache instance() {
  if(singleton == null)
   try {
    singleton = new Cache();
   } catch(Exception e) { ... }
  return singleton;
 }
 // etc.
}
```

(a)

```
class Cache {
 protected static Cache singleton;
 public static Cache instance() {
  if(singleton == null)
   try {
    singleton = (Cache)
     ImmortalMemory.instance().
      newInstance( Cache.class );
   } catch(Exception e) { ... }
  return singleton;
 }
 // etc.
}
```

(b)

```
aspect CacheMemory {
 around() returns Cache : calls(Cache.new(..)) {
  ConstructorCallJoinPoint ccjp =
   (ConstructorCallJoinPoint) thisJoinPoint;
  ConstructorSignature cs =
   (ConstructorSignature) ccjp.getSignature();

  return (Cache)( ImmortalMemory.instance().
                  newInstance( Cache.class )
 }
}
```

(c)

Fig. 1. (a) A partial Java implementation of a `Cache` type. **(b)** A partial RTSJ implementation of a `Cache` type in `ImmortalMemory`. **(c)** An AspectJ aspect used to rewrite `Cache` instantiations to be in `ImmortalMemory`. RTSJ `Cache` objects can now be constructed via `new`, just like objects in Java.

method; detection of this condition is accomplished by reference-counting the scope. Any memory allocated via a simple `new` instruction is allocated in the memory area currently associated with the allocating thread. Explicit `MemoryArea` allocation instructions are also permitted via the `newInstance` method, as in the `Cache` example of Figure 1.

`ScopeMemory` scopes can be nested, and it makes sense to design nesting relationships when small and perhaps iterative pieces of code produce a lot of garbage during their computation—this garbage can then be cleaned up all at once, on exit of the inner scope, without the need for a garbage collector. Programmers and program analysis tools tend to associate notions of storage scope with method scope. Thus, it may be desirable for a `ScopeMemory` to be associated with a particular scope of execution—a method, for example. The scope could then be deleted when the method exits. RTSJ avoids the improper collection of objects that are still reachable by mandating that object references may only point to objects within the same scope or outward from inner scopes

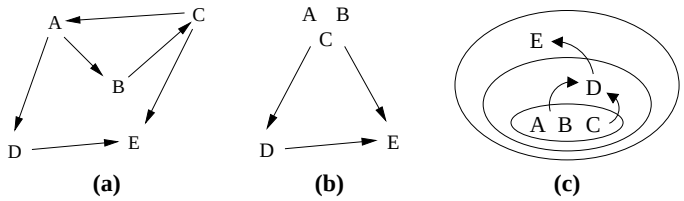


Fig. 2. A *doesReference graph* for a program P **(a)** before and **(b)** after grouping strongly connected objects. An arrow from A to B indicates that object A stores a reference to object B. **(c)** shows one possible scoping structure, where E is in the outermost scope, D is in a sub-scope, and A, B, and C are in a sub-scope of D's scope. Object references in RTSJ may only point outward to enclosing scopes.

to enclosing scopes; scopes are at least as long-lived as those that nest within them, so these outward-pointing references are considered safe.

If a Java program were simply moved to an RTSJ platform, then by default all objects would be allocated in the garbage-collected heap, which offers no guarantees for real-time activities. To generate an RTSJ scope hierarchy out of Java's flat memory model, we work backwards, tracking which objects reference which others, to build a set of scope nesting structures that do not violate the object referencing regulations of RTSJ.

We have developed a *reference-probing aspect* to determine these legal RTSJ `ScopeMemory` assignments. The probe defines as join points [2] the object instantiations and assignments of interest to us and builds a *doesReference graph* to track which objects refer to which other objects. The *doesReference graph* may contain strongly connected objects; these objects are grouped together as they must share a scope. This collapses the more general graph into a directed acyclic graph (DAG).

For example, if object A references object B, the DAG contains an edge from A to B. Then B's scope must be at least as long-lived as A's. There are two legal scoping hierarchies in this instance: that where B is in an outer scope and A is in an inner scope, and that where A and B are both in the same scope. A simple (but nonoptimal) algorithm for determining suitable scopes is to perform a topological traversal of the DAG, which corresponds to a right-to-left preorder traversal of any depth-first spanning tree of the DAG. Figure 2 shows an example illustrating this procedure.

3 Join Point Discovery

The join points in the original program at which we need to inject instructions to enter these scopes must also be determined. Consider Figure 3, a possible

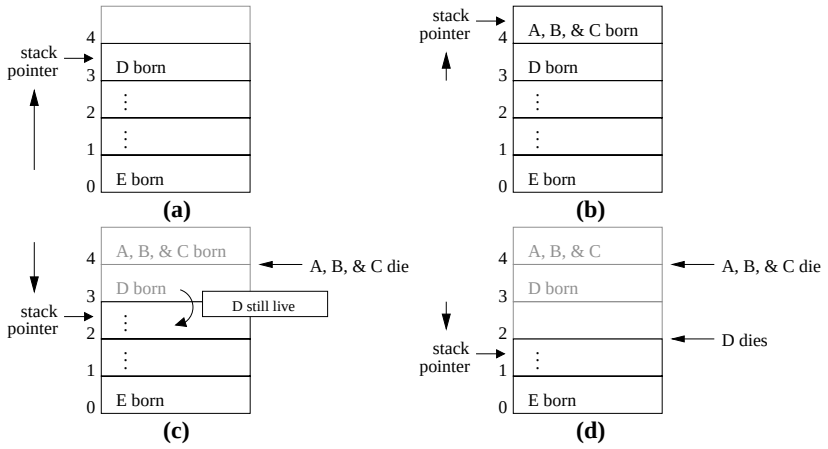


Fig. 3. A view into the execution stack of program P at different points in P's execution. **(a)** E is born first, in frame 0. In frame 3, D is allocated, then **(b)** A, B, and C (which refer back to D) are allocated in frame 4. A, B, and C are not returned or thrown from the execution scope that generated them, and we know from Figure 2 that there are no references to them, so they must die upon exiting their birth stack frame—this indicates that we can close their associated memory scope when frame 4 pops. **(c)** However, D escapes the execution scope of its birth (it was returned or thrown), and it is still live in frame 2. **(d)** Therefore, D's scope must not close until frame 2 is popped and D is known to be dead.

execution stack for the program P whose reference behavior is described by Figure 2. First, E is constructed, then in some later stack frame D is constructed (and stores a reference to E). If we can determine that D is always dead at the time a particular stack frame pops, we can close its scope at that point. This is demonstrated in the figure.

We can reason about the frame events of Figure 3 by engineering advice on method and constructor join points. To determine the points at which objects are dead, we weave a *liveness-probing aspect* into program P. This aspect inspects method and constructor executions to determine when objects are born and when they become unreachable.

By combining the *doesReference* scope information harvested by the *reference-probing* aspect and the frame birth and death data from the *liveness-probing aspect*, we can discover a scoping hierarchy that respects both RTSJ's requirements on object references and the observed execution flow of P.

4 Conclusion

Our approach has the following advantages:

1. *The memory concerns of the system are described and enforced in a modular fashion.* The memory concerns are described through the use of aspects, rather than sprinkling memory instructions throughout the code via `newInstance` invocations. This is a particularly important issue for objects in real-time Java packages that want or need to manage their own memory concerns under RTSJ. Without aspects, the user would have the responsibility of placing packaged objects in the correct type of memory area.
2. *Automation of* `ScopeMemory` *detection and management lowers development costs.* This dynamic analysis approach can be used to find a memory-efficient scoping structure, and the resulting, automatically generated memory aspect is easily tested by weaving it in to user code. Human-readable descriptions of object behavior can be generated that allow a useful view into the system and point out inefficiencies or unintended design consequences in the system.
3. *The introduction of aspect code into the target program introduces real-time predictability.* Because the scoping hierarchies are computed and the necessary join points are discovered offline, the aspect that enforces the runtime use of RTSJ scoped memory areas consists mainly of a table lookup. This can be done in bounded time, and we expect the translated program's performance to be more predictable (and suitable to real-time environments) than the original program.
4. *User source files are unchanged.* The real-time modifications are completely described in separate aspect source code; our aspect weaves into the user's source in order to modify it. For large source code trees, disk space requirements can be dramatically smaller. Additionally, one source code tree is sufficient for both Java and `ScopeMemory`-enhanced RTSJ code.

In summary, we have proposed a mechanism for automating the creation of RTSJ memory scopes. The expression of those scopes is accomplished via aspects, as is the offline dynamic analysis to determine the scopes. At present we have pieces in place to perform the analysis and to create scopes that are respectful of object references from program runs. Our future plans call for investigating tradeoffs between various scope nesting structures, in terms of footprint and the overhead incurred for managing the scopes.

References

1. Bollella, Gosling, Brosgol, Dibble, Furr, Hardin, and Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.
2. Gregor Kiczales. Aspect-Oriented Programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, June 1997.
3. The AspectJ Organization. Aspect-Oriented Programming for Java. `www.aspectj.org`, 2001.

Middleware Architecture Design Based on Aspects, the Open Implementation Metaphor and Modularity

H.-Arno Jacobsen
University of Toronto
Department of Electrical and Computer Engineering and
Department of Computer Science
jacobsen@eecg.toronto.edu *

1 Introduction

A "middleware system" constitutes a set of services that aim at facilitating the development of distributed applications in heterogeneous environments. The primary objectives of a middleware are to foster application portability and distributed system interoperability. At least conceptually, the "middleware layer" comprises a layer below the application and above the operating system and network substrate. Common middleware platforms include CORBA, DCOM, and the Java suite of protocols.

The key design principal underlying middleware design has been to hide and encapsulate system details behind common abstractions and offer various dimensions of transparency to the application developer (e.g., with respect to object location, data access, and service implementation language). As systems code (or near systems code) standard middleware has not been designed with *extensibility*, *modularity*, *configuration*, *openness*, and *customization* in mind. It is, for instance, impossible to enhance a platform with techniques that address network awareness, mobile awareness, and dynamic adaptability, or to exploit application level semantic for system operation. Commonly, the argument against such a design is a postulated sacrifice of performance.

Some of these issues have been partially recognized by bodies defining middleware specifications, more notably the OMG, who has introduced a number of features addressing this dilemma. This includes, for instance, message and method level interceptors, hooks for custom marshalling (restricted to value types), open stub-to-orb interface (restricted to the Java language mapping), and pluggable transports (restricted to the real-time CORBA profile, although often implemented by standard ORBs). These enhancements leverage part of the problem, but leave much to be desired. Extending ORBs with language mappings, protocol mappings, object adaptors, general custom marshalling, smart

* This work is supported by NSERC. Position Paper in Workshop on Aspect-Oriented Programming and Separation of Concerns, Lancaster, UK, August 2001.

proxies, or techniques to achieve network awareness and dynamic adaptability to run-time conditions is still not a given (see [8, 5] for a more detailed discussion of some of these issues.)

Moreover, middleware systems are extended to address requirements of the most diverse application domains and new middleware services supporting application transcending requirements are constantly added. The application domains addressed by the CORBA platform alone, range from (distributed) business applications (standard CORBA), real-time systems (Real Time CORBA), and scientific and high performance computing (Data Parallel CORBA), to wireless and embedded systems (Minimum CORBA). The service sets extend from fault tolerance, security, and object group management to synchronous and asynchronous communication, to just name a few.

Middleware platform families are evolving to support these different application domain requirements within the same *platform line* (i.e., essentially product line). Different profiles (i.e., specialized subsets of the core model) are defined to address service sets and specific domain requirements. Similar efforts are emerging for Java-based middleware, and to some extend for DCOM-based systems. This proliferation as well as the aim to support a vast functional spectrum within one environment is leading to the co-existence of middleware platforms with many non-orthogonal features and considerable functional overlap.

To summarize, the key problem with current middleware systems is non-extensibility, proliferation with respect to supported features and application domains, and a too coarse-grained service model exhibiting functional redundancy.

While we cannot offer a full solution to these problems, we intend to outline our position towards a new architecture for middleware systems. We first discuss a number of principles to address shortcomings in current middleware architectures and then present potential solutions to the above outlined problems. The new middleware architecture will be based on three key concept, *modularity*, *aspect orientation*, and the *open implementation* metaphor. We also summarize related work that has addressed some of these issues.

2 Our Position: Towards an Improved Middleware Architecture Model

Our position is that, for the above outlined reasons, the design of a middleware platform should be re-thought from ground up and the hypothesis of whether or not an efficient modular platform cannot be developed, must be tested. We first discuss a set of principles a new middleware platform design should be based upon and then discuss how to achieve this with emerging development concepts.

Middleware Design Principles

1. *Micro-kernel like architecture:*

Current middleware architectures favor one primary communication style – synchronous RPC – as basis and build other styles, as secondary add–on services next to it[1] (e.g., asynchronous communication, non-continuous operations, reliable / unreliable multicast, publish / subscribe, group communication etc.) A major drawback of this approach in current platform implementations is the overlap of the different models, the replication of core functional units that are part of all communication style, the subtle difference in the models (as they were incrementally developed over years), and the complexity of the final programming model supporting several non-orthogonal overlapping features.

To address these issues, a new middleware architecture should provide a model in which all communication styles are equally placed side-by-side and modularly broken down into functional units. Different communication styles are incrementally obtained from the composition of underlying functional units. In this model no one style is favored in the architecture over another one, functional redundancy is avoided, and consistency is maintained. We refer to this as a micro-kernel like design to emphasize that there should only be a very minimal core onto which different communication styles are to be configured.

2. *Open and modular middleware platform design:*
   All platform functionalities, including the non-application exposed middleware service layers, should become accessible, open and modular building blocks. This refers to platform units such as communication sub-system, stubs, skeletons, ORB transport layer, interface repository, implementation repository, and all parts of the object adapter.
3. *All functional units in the platform should be accessible, customizable, and replaceable by custom implementations:*
   A functional unit refers to one step in the execution chain of a service provided by the middleware (e.g., this could be one of the building blocks referred to in the item above). The exact extend of a step depends on the concrete platform design and the service offered. For a synchronous method invocation the functional units may be data structure packing, argument marshalling (unpacking and unmarshalling respectively), system-level RPC (possibly broken down in its constituent protocols), and call dispatching. These units should not be too coarse, to enable reuse, ease of customization, and low level access to their end-points (needed as join points for aspects). Different functional unit implementations, specialized for various environmental conditions and usage patterns, may be included with the platform to be selected at configuration time or provided as custom extensions. This unit breakdown is also required to enable very low-level aspects to be woven around unit end-points. Platform aspects such as, exception management, monitoring, authentication, encryption / decryption, and selective type support will need this modular break down as well.

[1] Not around or on top of it, i.e., these services are commonly stand alone implementations.

4. *Fine-grained configuration of platform:*
   To date middleware systems are not configurable at all. For the most part, a one size fits all approach is taken. Under fine-grained configuration we understand the specialization of a platform instance for one particular application domain and, potentially, for one particular application in this domain. A higher level specification of the aspects the platform requires and the interfaces (e.g., required services, types, and exceptions) of the application guide the synthesis of the platform instance.
5. *Independence of accompanying platform tools:*
   By platform tools we mean stub-/ skeleton-generator and meta-data repositories (IDL and implementation repositories). Current middleware platforms tightly integrate these tools into the platforms. This refrains from third party implementations of these components, use of standard technology, or the use of these components for purposes they were not immediately intended for. These tools should be separable form the platform, build on open interfaces, and follow an extensible design themselves.

Towards Realizing these Principles:

We think that all of the above principles can be achieved by defining an open framework of a middleware platform, by using techniques from aspect-oriented programming for the generative configuration of middleware platform instances, and applying the open implementation metaphor to functional unit design to obtain customization at a very low level.

The Role of an Open Middleware Platform Framework: The framework lays out the middleware architecture, defines all platform interfaces, defines execution chains and access points (i.e., interfaces to individual steps in an execution chain), and defines how they inter-relate. This framework is the principal prerequisite for all the rest. It defines the level at which aspects can intervene and their granularity. It also impacts the possible design choices for all functional units.

The Role of Aspects: An aspect is a system feature that cross-cuts the implementation of the system and is manifest at multiple loci in the code [9]. Examples of aspects in the context of middleware are:

- exception management (raising, propagation, handling)
  If an exception were to be defined as a system aspect and a middleware platform could be configured for a particular application (domain), the exception aspect could be configured in or configured out, depending on the application semantic and runtime environment.[2]

[2] When the Minimum CORBA (embedded system profile) standard was conceived a discussion of whether or not to include exception handling came up. Opponents

- synchronization management and concurrency control
  Synchronization constraints have often been used to illustrate aspect oriented programming techniques [9].
- interface definition language extensions
  Many extensions for interface definition languages have been proposed. This includes, for example, quality of service annotations, real-time constraints, assertions, pre and post conditions, and behavioral annotations. All of these constitute aspects.
- access control and security
  Access control (e.g., object-based, method-based, interface-based) and security (e.g., authentication and encryption / decryption) constitute examples of middleware aspects that may need to be blended in or out depending on the execution context.
- computing and network resource monitoring
- individual platform types
  A middleware platform supports a certain type model, ranging from basic types (e.g., int, float, char) to very sophisticated dynamically managed types (e.g., type ANY in the CORBA model). The processing of a type intervenes at different levels of the platform (e.g., in marshalling, in packaging, in transport etc.) In that sense a particular type constitutes an aspect of the middleware.

The above list comprises concrete aspects, there also exists a number of more abstract aspects, that will require a more refined decomposition. These include real-time, quality of service, and fault tolerance aspects.

Aspects could intervene at several stages in middleware platforms. For one, they could be used to extend the platform with certain features (cf. list above), but, more interestingly, aspects could be applied to configure a platform instance for a particular application domain and specific application.

The great benefit of this is that the proliferation of static platform profiles would disappear and, as new requirements (domain and feature) arise, simply more aspects need to be added. Clearly, this approach is entirely based on a modular open middleware platform framework that permits to define access points at which aspects can intervene.

The Role of the Open Implementation Metaphor: The open implementation metaphor reveals, at a function's interface, details about its implementation [10]. A client of the interface may influence the underlying implementation according to its usage pattern (this may be decided statically or dynamically depending on the sophistication of the open implementation based design of the function's implementation.) This concept can be applied to customize functional

argued that in the context of embedded systems exceptions were not needed and the primary design goal would be a small memory footprint. Similarly, other features of the platform where at scrutiny (e.g., various types of the CORBA type model, dynamic invocation support etc.)

units of the middleware platform. OI suggests several different levels of doing that, ranging to client provided implementation of the unit as final instance.

3 Related Work

Related work on the topics discussed in this position paper can be broadly classified into approaches that provide *customization* through static or dynamic policy selection, *reflection* to adapt middleware internals to changing runtime conditions, and *configuration* based on various forms of aspect definitions. Much of the discussed projects use several of these techniques. We briefly discuss some of them below.

Astley *et al.* [1] achieve middleware customization through techniques based on separation of communication styles from protocols and a framework for protocol composition. Further aspects that cross cut the system implementation are not explicitly addressed.

Several projects exploit reflective programming techniques to allow the middleware platform to adapt itself to changing runtime conditions. This includes projects such as openORB [2], openCORBA [12], and dynamicTAO [11]. Recent progress in this area has been summarized in a reflective middleware workshop [3].

LegORB[4] [13] and Universally Interoperable Core (UIC)[5] are middleware platforms designed for hand-held devices, which allow for interoperability with standard platforms. Both offer static and dynamic configuration and aim to maintain a small memory footprint by only offering the functionality an application actually needs. Customizable functions range from the transport protocol to method dispatching and marshalling. Both platforms do not support the notion of aspects as code cross cutting concerns. Aspects in the sense of LegORB and UIC are functional units supporting application-level requirements.

Similarly, Jonathan[6] constitutes an open middleware framework that can be customized with respect to a large number of functions. Jonathan aims to embrace several standard middleware platforms and offer customization according to application needs. It can be configured to use IIOP or RMI.

The effectiveness of the open implementation metaphor for the design of a light-weight thread package is demonstrated by Haines [4]. The renewed design leads to a more efficient and portable package. While this approach is not primarily addressing middleware platform issues per se, it may also prove effective for the design of functional units within a platform.

Brodsky *et al.* [3] demonstrate very elegantly the use of aspect oriented programming for customization and extensibility of a distributed file system middleware with fault tolerance features.

Jacobsen and Krämer [8] show how to weave interface level specification of synchronization constraints into stubs and skeletons generated by standard IDL

[3] http://www.comp.lancs.ac.uk/computing/rm2000/
[4] http://devius.cs.uiuc.edu/2k/LegORB/
[5] http://www.ubi-core.com/
[6] http://www.objectweb.org/jonathan/jonathanHomePage.htm

compilers. This work is extend in Jacobsen and Krämer [6] to also account for other aspects defined at the interface level (e.g., QoS annotations, behavioral annotations, and pre and post conditions). A processing framework based on the extended markup language (XML) for this approach is presented in [7].

References

1. M. Astley, D. C. Sturman, and G. A. Agha. Customizable middleware for modular software. *ACM Communications*, 44(5), May 2001.
2. G. S. Blair, G. Coulson, A. Andersen, M. Clarke, F. M. Costa, H. A. Duran, R. Moreira, N. Parlavantzas, and K. B. Saikoski. The design and implementation of OpenORB version 2. *IEEE Distributed Systems Online Journal*, 2(6), 2001.
3. Alex Brodsky, Dima Brodsky, Ida Chan, Yvonne Coady, Jody Pomkoski, and Gregor Kiczales. Aspect-oriented incremental customization of middleware services. Submitted.
4. Matthew Haines. An open implementation analysis and design for lightweight threads. In *OOPSLA*, pages 229 – 242, 1997.
5. H.-A. Jacobsen. Programming language interoperability in distributed computing environments. In Lea Kutvonen, Harmunt Knig, and Martti Tienari, editors, *Second IFIP Working Conference on Distributed Applications and Interoperable Systems II (DAIS)*, Helsinki, Finland, June 1999. Kluwer Academic Publisher.
6. H.-A. Jacobsen and B. Krämer. Design patterns for synchronization adaptors of CORBA objects. *Special issue of L'OBJET Journal on "Object Orientation and Formal Methods*, 2000. Hermes Publisher.
7. H.-A. Jacobsen and B. Krämer. Modeling interface definition language extensions. In *37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-37)*, Sydney, Australia, 20-23 November 2000.
8. H.-A. Jacobsen and B. J. Krämer. A design pattern based approach to generating synchronization adaptors from annotated IDL. In *IEEE Automated Software Engineering Conference (ASE'98)*, pages 63–72. IEEE Computer Society, September 1998.
9. G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv*, 28(4), Dec 1996.
10. Gregor Kiczales, John Lamping, Christina Videira Lopes, Chris Maeda, Anurag Mendhekar, and Gail Murphy. Open implementation design guidelines. In *international conference on Software engineering*, pages 481 – 490, 1997.
11. Fabio Kon, Manuel Román, Ping Liu, Jina Mao, Tomonori Yamane, Luiz Claudio Magalhães, and Roy H. Campbell. Monitoring, Security, and Dynamic Configuration with the dynamicTAO Reflective ORB. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'2000)*, number 1795 in LNCS, pages 121–143, New York, April 2000. Springer-Verlag.
12. T. Ledoux. OpenCorba: A reflective open broker. *Lecture Notes in Computer Science*, 1616:197–??, 1999.
13. Manuel Roman, M. Dennis Mickunas, Fabio Kon, and Roy Campbell. LegORB and ubiquitous CORBA. Reflective Middleware Workshop. Held in conjunction with Middleware 2000. http://www.comp.lancs.ac.uk/computing/rm2000/, 7th-8th April 2000.

Aspects of Exceptions at the Meta-Level (Position Paper)

Ian S. Welch[1], Robert J. Stroud, and Alexander Romanovsky

Department of Computing, University of Newcastle upon Tyne,
United Kingdom NE1 7RU
{i.s.welch, r.j.stroud, alexander.romanovsky}@ncl.ac.uk
http://www.cs.ncl.ac.uk/research/dependability/reflection/

1 Introduction

This paper describes the design and usage of a metaobject protocol that explicitly includes support for handling exceptions. We do not propose implementing exception mechanisms anew [3, 5] or proposing a unified meta-level software architecture for exception handling [4]. To make our discussion concrete we describe an extension of the Kava [9] metaobject protocol that includes exceptions as first class values, provide examples of Kava's use, and compare Kava with related Java extensions.

We believe that insufficient attention has been paid to exceptions by designers of metaobject protocols for object-oriented languages. Most metaobject protocols provide a way of intercepting method execution but these protocols are usually discussed solely in terms of arguments and results. Signalled exceptions are rarely discussed. However, in order to successfully implement non-functional requirements using metaobject protocols it is important that exceptions are explicitly considered. For example, consider using a metaobject protocol approach to implement distributed objects. It is not sufficient just to convert method calls into remote method calls, exceptions must be converted into remote exceptions as well. Therefore, a metaobject protocol should be designed to treat method arguments, method return values and exceptions equally. This means that if the behaviour of method execution is reflected upon, then any signalled exception should be reified and be manipulable at the meta-level. Note that such a "exception-aware" metaobject protocol should not lead to base-level programmer's expectations being confounded, as doing so would make both programming and verification very difficult. For example, the exception model should not be able to be changed dynamically, say from a termination model to a resumption model (as opposed to [2]).

2 Meta-Level Requirements

What facilities should a "exception-aware" metaobject protocol have? We propose that such a metaobject protocol needs two facilities: meta-level interception

of exceptions signalled from the base-level, and meta-level raising of exceptions at the base-level.

Meta-level interception is required to handle new exceptions introduced as a side-effect of the implementation of non-functional requirements. It may also be required to reinterpret existing base-level exceptions in the context of new non-functional requirements. An example of this was given in the introduction where distribution requires that local exceptions are reinterpreted as remote exceptions.

Meta-level raising of exceptions at the base-level is required to allow metaobjects to raise new types of exceptions and maintain the transparency of the meta-layer. For example, a metaobject may enforce a security policy by raising a security exception whenever the security policy is violated. Since we normally wish to implement non-functional requirements transparently this exception should appear to be raised at the base-level. If it appears to have been raised by the metaobject then the meta-level becomes visible to any clients of the base-level and then transparency is shattered.

These two features allow the metaobject protocol to support the following mappings between exceptions and values: from one exception to another, from one exception to a value, or from a value to an exception. In the remainder of this section we provide examples of how these mappings can be used.

Exception to exception. Adding debugging information to exceptions requires that one exception is mapped to another. Here, we want to add meta-information to an exception such as the time it was signalled. An extended version of the exception class could be defined that encapsulates the base-level exception and the meta-information. At the meta-level the signalled exception is replaced by an instance of the extended exception class.

Exception to value. Logging and then ignoring an exception requires that an exception is mapped to a value. Here, the base-level exception is suppressed and the method terminates normally returning a value specified at the meta-level.

Value to exception. Assertion checking [7] requires that a value is mapped to an exception. Here, a value of a member variable or argument of the method causes an exception to be raised. This exception will appear to be raised at the base-level to preserve transparency.

3 Kava

Kava is a reflective Java implementation [9]. It uses byte code transformations to make constrained changes to the binary structure of a class in order to provide a metaobject protocol that brings object execution under the control of a meta-level. These changes are applied at the time that classes are loaded into the runtime Java environment. The meta layer is made up of metaobjects that are written using standard Java. The binding between classes and metaobject classes are specified in an XML configuration file called a *binding specification*. Kava

brings the sending of invocations, initialisation, finalization, state update, object creation and exception signalling under the control of a metaobject protocol.

When a meta-level programmer creates a new metaobject class, the programmer extends the default metaobject class and overrides those methods that control the behaviours the programmer wishes to redefine. In Kava we define *around* style meta methods, so for each behaviour there is a *before* and *after* method.

The following listing shows the methods relating to overriding method execution in the metaobject class interface,

```
public interface IMetaObject {
  ...
  public void beforeMethodExecution(IMethodExecution context)
    throws Exception;
  public void afterMethodExecution(IMethodExecution context)
    throws Exception;
}
```

A context object is passed as an argument to each of the meta-level methods. The context reifies the context of the metainterception as a context object that implements the `IMethodExecution` interface. In earlier versions of Kava exceptions that were raised during the execution of a method were not included in the context. Now, any exceptions that have been raised is included in the context in addition to reified method, its actual parameters, and the result of the execution of the method. In addition to reifying exceptions the context API has been extended to support the reflection of the exception back to the base-level and the overriding of the exception signalling at the base-level.

We are currently examining how this extended metaobject protocol can be used to implement Java language extensions such as multi-level handlers for exceptions (statement, block, method, class and exception level), design by contract, and n-version programming).

4 Examples

In this section we show how to use Kava to implement the examples described in the meta-level requirements section.

First, we show how an exception can be intercepted and converted to another type. Here, the exception is converted to an instance of an exception class used for debugging which encapsulates the base-level exception, the key to this is using the `setException` method to change the exception raised at the base-level,

```
public DebugMetaObject extends MetaObject {
  public void afterMethodExecution(IMethodExecution context)
    throws Exception {
    if (context.isExceptionRaised()) {
      context.setException(new DebugException
        (context.getException())); }}}
```

The next example shows how an exception can be mapped to a value and the exception raising at the base-level suppressed. This metaobject is used to log and suppress `IOExceptions` exceptions. It checks that the base-level method exited because an exception was raised. The base-level exception is suppressed through the use of the `overrideException` method,

```
public LogMetaObject extends MetaObject {
  public void afterMethodExecution(IMethodExecution context)
    throws Exception {
    if (context.isExceptionRaised()) {
      Exception e = context.getException();
      if (e instanceof java.io.IOException) {
        context.overrideException();
        log(e); }}}}
```

The final example shows a mapping from a value to an exception. We want to check that a method never returns a `null` value. First, we check using the convenience method `getReturnType` returns an object reference, then we check that the value of that reference is not `null`. If it is `null` then we throw an `AssertionFailed` exception,

```
public AssertMetaObject extends MetaObject {
  public void afterMethodExecution(IMethodExecution context)
    throws Exception {
    if (context.getReturnType() == Type.OBJECT) {
      if (context.getReturnValue() == null) {
        throw new AssertionFailed(); }}}}
```

5 Related Work

Although exceptions are an integral part of the Java language there has been little explicit attention paid to them by the Java reflection community with the exception of Garcia et. al. [4]. Garcia et al. have proposed a unified meta-level software architecture for sequential and concurrent exception handling that is described using a set of design patterns. The patterns cover: Exceptions, Handler, Exception Handling Strategy, and Concurrent Exception Handling Action. They are attempting to codify "best practice" with regard to the implementation of reflective exception handling. They have made an implementation using a custom Java VM (Guaraná [8]) which means it is non-portable. In contrast, our work is more narrow in focus but has resulted in a portable implementation.

Explicit support for exceptions has been introduced into some Java implementations of portable compile-time Java extensions for programming using advanced separation of concerns. Below we describe the approach to exceptions taken with AspectJ[1] [6] and ComposeJ [10].

[1] the version described here is 0.8

AspectJ allows programmers to use aspect-oriented programming techniques in Java. AspectJ like Kava can be used to map exceptions and values to each other. An `around` advice applied to a `receptions` pointcut can be used to implement the mapping of an exception to exception, exception to value, value to exception. This is because `around` advice selectively pre-empts the normal computation at the specified join point. AspectJ also has two new features related to exception handling. First, the advice `after throwing` allows aspects to be invoked when an exception is thrown (in Java `throw` is used to raise an exception). This allows extra code to be executed when an exception is signalled but does not allow the signalling to be overriden. This is roughly equivalent to the interception feature in Kava. Second, aspects can be woven into existing exception handler code through the use of the `handles` pointcut. This allows extra code to be invoked when an exception is handled, and it allows handling code to overriden. This is feature is not supported in our metaobject protocol as we currently choose to intervene only at the level of a method rather than within `try ... catch ... finally` clauses.

ComposeJ allows programmers to use composition filters in Java. Composition filters [1] allow messages sent and received by objects to be intercepted and manipulated. Filters can be composed with other filters to implement complex non-functional behaviour. There are different types of filters in the model, one of which has explicit support for exceptions. The `Error` filter allows predicates on base-level state to be evaluated and an exception to be raised that causes the system to halt. This allows the implementation of assertions and contracts in Java. In the current version it is not clear if signalled exceptions are considered to be message or not. If they are then other filters such as `Dispatch` could be used to implement mappings that are similar to Kava.

Kava could implement the same functionality as the `Error` filter. It cannot add behaviour to exception handlers like AspectJ although we believe that many useful extensions for dependability can be developed without that capability. In terms of implementation Kava differs from AspectJ and ComposeJ in that it is a load-time extension to Java and can be used to add non-functional behaviour to compiled code. This makes it useful for dealing with mobile or third-party code where the API may be understood but the source code might not be available.

6 Conclusions

Metaobject protocols must be "exception aware" so that they can be used to implement a wide range of non-functional requirements. Such a metaobject protocol requires two features to support the successful implementation of non-functional requirements. The first feature is the ability to intercept exceptions signalled from the base-level, and the second feature is the meta-level raising of exceptions at the base level. These two features allow the metaobject protocol to implement mappings between exceptions and values that can be used to improve the dependability of applications.

There is one reflective Java implementation that is "exception aware" but it is non-portable. There are portable extensions to Java that introduce "exception awareness" for advanced separation of concerns but these require access to source code. Our implementation in Kava is portable and applies reflection and load-time. This allows Kava to be used for a wide range of applications such as mobile code or third-party code.

Acknowledgements

We would like to acknowledge the financial support of the ESPRIT projects: MAFTIA project (IST-1999-11583), and DSOS project (IST-1999-11585).

References

1. M. Askit, L. Bergmans, and S. Vural. An Object-Oriented Language-Database Integration Model: The Composition-Filters Approach. In *ECOOP*, volume LNCS 615, pages 372–395. Springer-Verlag, 1992.
2. A. Burns, S. Mitchell, and A. J. Wellings. Mopping up Exceptions. In *ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems*, pages 365–366, 1998.
3. Christophe Dony. Exception Handling and Object Oriented Programming : Towards a Synthesis. In *Proceedings of ECOOP/OOPSLA'90*, pages 322–330, Ottawa, Canada, 1990.
4. Alessandro F. Garcia, Delano M. Beder, and Cecilia M. F. Rubira. Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling. *The Computer Journal (Special Issue on High Assurance Systems Engineering)*, 2001.
5. M. Hof, H. Mossenbock, and P. Pirkelbauer. Zero-Overhead Exception Handling Using Meta-Programming. 1338:423–431, 1997.
6. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffery Palm, and William G. Griswold. An Overview of AspectJ. In *ECOOP 2001*, volume LNCS 2072, pages 327–353, Budapest, Hungary, 2001. Springer-Verlag.
7. B. Meyer. Design by Contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice-Hall, 1991.
8. Alexandre Oliva and L. E. Buzato. The Design and Implementation of Guaraná. In *Usexnix COOTS*, pages 203–216, San Deigo, California, USA, 1999. Usenix.
9. Ian Welch and Robert Stroud. Kava – Using Byte-Code Rewriting to Add Behavioral Reflection to Java. In *6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS 2001)*, pages 119–130, San Antonio, Texas, 2001.
10. J. C. Wichman. ComposeJ: The Development of a Preprocessor to Facilitate Composition Filters in the Java Language. Master's thesis, University of Twente, 1999.

Fault tolerance AOP approach[1]

José Luis Herrero[1], Fernando Sánchez[1], Miguel Toro[2]

[1] Computer Science Department
University of Extremadura.Spain
{jherrero, fernando}@unex.es

[2] Computer Science Department
University of Sevilla.Spain
mtoro@lsi.us.es

Abstract.

Object oriented systems are composed by a collection of interacting objects. Distributed object oriented systems consider that all these objects can be located at different computers connected through a network. Reliability and availability are very important trends in the development process of these kinds of systems. In order to improve these features, object replication mechanisms have been introduced. Programming replication policies for a given application is not an easy task, and this is the reason why transparency for the programmer has been one of the most important properties offered by all replication models. However, this transparency for the programmer is not always desirable. There are situations in which programmers need to manipulate by hand the replication properties. In this paper we present a replication model, JReplica, based on Aspect Oriented Programming (AOP). JReplica allows the separated specification of the replication code from the functional behaviour of objects, providing not only a high degree of transparency, as done by previous models, but also the possibility for programmers to introduce new behaviour to specify different fault tolerance requirements. Derived from the use of AOP, JReplica also obtains two important added benefits: the possibility of obtaining an ORB independent replication and the possibility of reusing entire replication policies. Moreover, the replication aspect has been introduced at design time, and in this way, UML has been extended in order to consider replication issues separately at the moment of designing fault tolerance systems.

Introduction

This work tries to introduce replication in object orientation by means of a new aspect. For this purpose, a new language called JReplica has been developed. This language tries to capture the relevant aspects of replication, encapsulating them into a

[1] This work has been developed with the support of CICYT TIC 99-1083-C02-02

component or group of related components favouring the reusability and dynamic adaptability of replication policies. This language also favours the use and reuse of replication policies independently from the middleware used to communicate objects.

The work is not limited to the definition of this new language. AOP ideas have been translated to the design level. In this way, the semantic of UML has been extended in order to represent replication properties. From a given design, the same for whatever middleware, a visual tool is able to generate code. The rest of the paper is as follows: section 2 explains the different approaches to introduce replication in object orientation. Our proposal is introduced in section 3. Section 4 shows related works. Finally, future works are outlined in section 5.

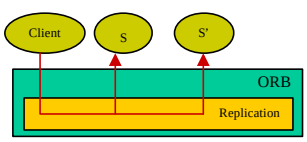
Fault tolerance approaches

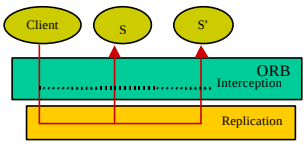
There has been proposed different approaches to introduce fault tolerance in object oriented systems. These models are the followings:

1. **Integration approach :** In this approach, replication is integrated inside the model. Replication is coded inside the ORB. In this way, each ORB must be modified in order to provide fault tolerance. Electra [Maf95], Orbix+Isis [II94] are two models that are based on this approach (figure 1).

2. **Interception approach :** In this model, every message is intercepted and redirected to a replication toolkit. This new tool is in charge of providing fault tolerance. The ORB must be modified introducing the interception mechanism. Eternal [Mos98] is an example of this approach (figure 2).

3. **Service approach :** A new replication service is added to the ORB. This service provides mechanisms for object replication. OGS [Fel98] and the new Corba Fault Tolerance specification [OMG00] are based on this approach (figure 3)





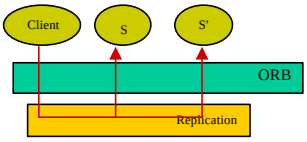


Fig. 1. Integration approach **Fig. 2.** Interception Approach **Fig. 3.** Service Approach

All these models introduce new elements to provide fault tolerance through replication. Transparency is the most important property achieved. In this way, programmers do not have to take care about replication, and they do not need to define any protocol to develop fault tolerance applications because replication is obtained automatically by the model. However, all these models can be considered too restricted in the following sense:

- **Close:** A totally transparent system doesn't allow programmers to change replication mechanisms. Replication properties can not be established, such as the replication granularity, or the moment when replication protocols must be executed. These properties are defined automatically by the model and they are the same for every system. In this way, programmers can not take advantage from system requirements.

- **ORB dependent:** Replication depends on the ORB implementation. Any replication policy must be coded into an individual ORB, and it can not be reused in a different ORB. There's no way to port the same replication policy to other ORBs.

Although transparency is a good property to be achieved, it is not always necessary, moreover, sometimes it is not advisable. Sometimes the nature of the problem may require establishing the replication properties and behaviour by the programmer. Even more, if requirements guide the replication behaviour, the system could take advantage of them, and system performance could be increased. If the replication model is totally transparent, there is no way to define fault tolerance applications according with system requirements.

Proposal

The model here proposed is based on the paradigm of Aspect Oriented Programming (AOP). Our research group has gained experience with AOP during the last few years working with the synchronization, coordination and distribution aspects [Mur99, San00]. Here we go one step further introducing the replication[2] aspect as a new non-functional property of the object. With this separation transparency is granted because replication policies can be reused among applications with no changes. In addition, programmers can get control over the replication policy using the specific replication language provided: JReplica.

In order to accomplish this separation, the model is based on reflective architecture. The following two levels have been defined:

- **Object level:** Object functionality is defined at this level. Object code does not refer to any replication mechanism, it just only defines object functional behavior using whatever language.

- **Replication level:** Replication policies are defined at this level using JReplica.

The model is based on the same concept that the interception model, that is, all the messages arriving at an object are intercepted and redirected to another entity (figure 4). The interception and replication level is located just before the ORB, that is,

[2] Although there are different replication strategies, in this paper we only focus on the backup replication model due to it being suitable for deterministic and non-deterministic objects.

before sending and receiving messages to and from the ORB, they are intercepted. It is in this moment when new actions for replication can be added. Figure 5 illustrates this fact. The order of messages is the following:

1. A message arrives at the target object.
2. After the message is executed, the new state is sent to the replication component of the target object.
3,4. This replication component propagates the new state to the rest of replication components.
5,6. Every replication component updates the state of the replica objects.

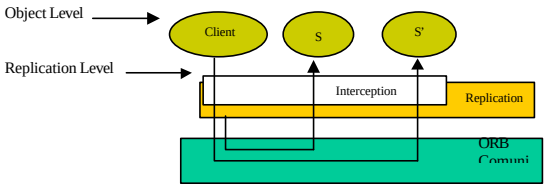


Fig. 4. Proposed model

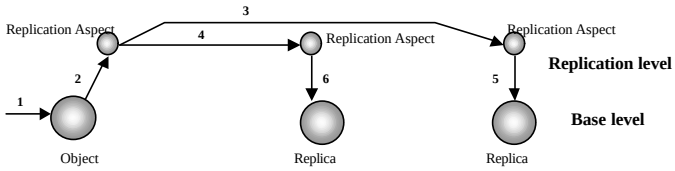


Fig. 5. Message order

1. Those benefits derived from the **use of AOP**, mainly modularity, reusability of code and adaptability of applications.
2. **ORB Independence:** Replication algorithms are independent from de ORB. In a previous work [San00] different distribution protocols were defined as a separated aspect providing a dynamic, adaptable and transparent object distribution. Now, as the replication module is defined outside the ORB, the combination of distribution and replication aspects offer the possibility of reusing the same replication policy in different ORBs. Figure 6 illustrates this fact.
3. **Open**: Thought replication algorithms are hidden and separated from object behaviour, replication properties and behaviour can be defined. Reflective mechanisms can communicate the object level with the replication level. This communication provides the way to introduce new replication actions.

JReplica: Java Fault Tolerance Language

JReplica is a language with the only purpose of defining replication policies. Its syntax is based on Java. It introduces new primitives, which are shown in figure 7.

This Java extension introduces two main elements:

1. **Replication Policy:** A new entity called *Disguise*[3] *Replication* defines the replication aspect. This entity is divided into the following parts:
   - **Attributes:** Represent the information that defines the replication policy.
   - **State:** Represent the set of replication states.
   - **Operations:** Represent methods that can manipulate the replication state.
   - **Guard:** Represent a condition that must be true before replication. If this condition is false, replication won't be executed.
   - **Before Replication:** Represent the set of actions that must be executed just before replication.
   - **After Replication:** Represent the set of actions that must be executed just after the replication is executed.
   - **Error:** Represent the set of actions that must be executed when a replication error appears.
2. **Composition:** A class can be composed with different aspects, this means that every object will extend its functionality with replication mechanisms.

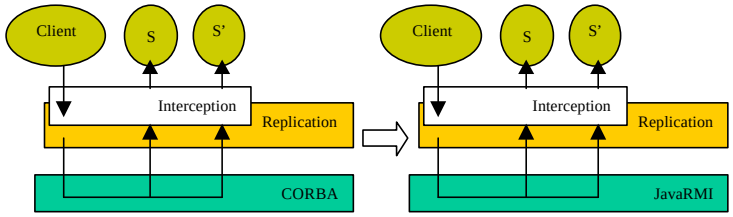


Fig. 6. Replication policies reuse

```
Class  <name>
{ ........
}

.........................
x=new C1
Compose x with R;
y=new Replica of x;
.........................
```

```
Diguise Replication <name>
{
 Attributes: .....
 Operations: .....
 State:  .....
 Guard: .....
 Before Replication: ....
 After Replication: .....
 Error: .....
}
```

Fig. 7. Jreplica replication primitives

Representing Replication at Design Level

Replication policies now can be defined with JReplica language. This language helps programmers to define easily replication properties in object oriented systems. But we

[3] The word disguise comes from the original model: Disguises Model.

consider that replication must be introduced at earlier stages of object life cycle, more concretely at design level. In this way, UML [UML99] is used as the modeling language due to it being a standard. As UML does not provide mechanisms to represent replication, its semantic has been extended in order to express replication properties and behavior.

UML semantic can be extended with the introduction of new stereotypes. At this point, we have considered that replication policies can be designed separately and independently, in the same way as has been explained at the implementation level. As such, the aspect concept is introduced in UML to express the AOP philosophy. The replication aspect is represented with a new stereotype, called <Replication>. This new stereotype is shown if figure 8.

The replication stereotype represents a particular replication policy. Information is represented as follows:

- **Stereotype Attributes:** Represent the information that defines the replication policy.
- **Stereotype Methods:** Define the set of methods that can manipulate the replication state.

As it can be shown, there are other elements that can not be represented in this stereotype. The dynamic behaviour of replication can not be represented in a normal class diagram. Statechart diagrams represent dynamic behaviour. So the solution goes by attaching a statechart diagram to this replication stereotype. In this way, replication static properties and dynamic behaviour can be designed. The dynamic behaviour of replication policies can be represented in a statechart diagram as it is shown in figure 9.

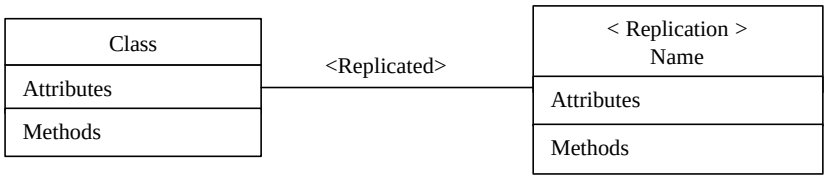


Fig. 8. UML extension

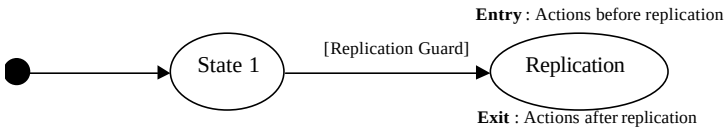


Fig. 9. Statechart Representation

The elements that are represented in this statechart diagram are:

- **State:** Each replication state is represented by an state. There is a special state called *Replication* that represents the moment when replication is to be executed.
- **Guard:** Guards are represented in the transition of each state.
- **Before Replication:** The set of actions that is executed just before the replication begins is represented in the entry actions of the Replication state.
- **After Replication:** The set of actions that are executed just after the replication ends are represented in the exit actions of the Replication state.
- **Error:** Replication errors are represented as a new state.

A tool is being developed in order to generate JReplica code starting from this extension of UML. In this way Replication aspect has been introduced from design to implementation level. This tool is based on other one we have developed for the synchronization aspect [Her00].

Related works

There are several models that provide replication mechanism to achieve fault tolerance. In [Nar00], a new interception mechanism called Aroma is introduced in the Java RMI architecture. Other models are based on the introduction of separated entities that implement replication protocols. The Cadmium Model [Bag99] defines a couple of new entities called *Stub* and *Scion*, which are attached to a client and a server respectively and offer replication mechanisms. In [Bru95] a new replication entity and a consistency manager are introduced, both separated from the object. In AspectIX [Gei98] a single object is divided into fragments, all of which have a different purpose. One of these fragments offers replication facilities. The GARF [Gar95] model defines two different entities in order to introduce replication, they are called e*ncapsulator* and *mailer*. A two level reflective architecture was defined for Java in [Kle96]: object functionality is defined in the first level, while replication protocols are established in the second one. All these models only take into account the implementation level, they are focused on replication protocols and the definition of a framework that provides fault tolerance, ignoring the design phase.

A new pattern [Gon97] has been defined in order to provide support for the representation of replicated objects. Moreover, a new language that helps programmers to build fault tolerance systems has been defined in [Fab97, Fab00]. This proposal is based on the concept of separation of concerns and extends AspecJ language [Lop97] with replication primitives. It is possible to define the attributes that need replication and what to do when a replication error happens. But there is no way to express new replication actions or when replication must be executed. Although these models help programmers to implement fault tolerance systems, it is necessary to introduce mechanisms that help software engineers to design this kind of requirements.

Future works

Future works will consider extensions to JReplica in order to express more complex replication mechanisms. The current version showed us the suitability of the model.

References

[Bag99] Aline Baggio. *Adaptable and Mobile-Aware Distributed Objects.* PhD Thesis, Université Pierre et Marie Curie and INRIA, Paris, France, June 1999.

[Bru95] Georges Brun-Cottan and Mesaac Makpangou. *Adaptable Replicated Objects in Distributed Environments*. BROADCAST TR No. 100. Appeared in the proceedings of the 2nd BROADCAST Open Workshop, Grenoble, July 1995.

[Gar95] B. Garbinato, R. Guerraoui, and K. R. Mazouni. *Implementation of the GARF replicated object platform.* Distributed Systems Engineering Journal, 2:14-27, 1995.

[Fab00] Johan Fabry. *A Framework for replication of objects using Aspect-Oriented Programming. Phd Thesis 1998. University of Brussel* .

[Fab97] Johan Fabry. *Replication as an Aspect - The Naming Problem.* ECOOP Workshops 1998: 424-425.

[Fel98] Pascal Felber. *The CORBA Object Group Sevice. A Service approach to object groups in CORBA.* Phd Thesis 1998. University of Lausanne.

[Gei98] Martin Geier, Martin Steckermeier, Ulrich Becker, Franz J. Hauck, Erich Meier, Uwe Rastofer. *Support for mobility and replication in the AspectIX architecture.* Object-Oriented Technology, ECOOP'98 Workshop Reader, LNCS 1543, Springer, 1998; pp. 325-326.

[Gon97] Teresa Gonçalves and António Rito Silva. Passive Replicator: *A Design Pattern for Object Replication.*Second European Conference on Pattern Languages of Programs. July 1997.

[Her00] J.L.Herrero. Introducing separation of concerns at design time. PhDOOS Workshop, European Conference on Object-Oriented Programming (ECOOP'2000).

[II94] IONA and Isis. *An Introduction to Orbix+Isis*. IONA Technologies Ltd. and Isis Distributed Systems, Inc., 1994

[Kle96] Jürgen Kleinöder, Michael Golm. *Transparent and Adaptable Object Replication Using a Reflective Java*. Tech. Report TR-I4-96-07, Universität Erlangen-Nürnberg: IMMD IV, Sept. 1996

[Lop97] C.V. Lopes. *D: A Language Framework for Distributed Programming*. Phd Thesis 1997. University of Northeastern

[Maf95] S.Maffeis. *Run-Time support for object-oriented distributed programming*. Phd Thesis, University of Zurich, 1995.

[Mos98] L.E.Moser, P.M. Meliar-Smith and P. Narasimhan. *Consistent object replication in the Eternal system.* Theory and Practice of Object Systems, 81-92, 1998.

[Mur99] J.M. Murillo, J. Hernández, F. Sánchez, L.A. Álvarez. *Coordinated Roles: Promoting Reusability of Coordinated Active Objects Using Events Notification Protocols.* In Coordination Languages and Models. Springer-Verlag, LNCS 1594, April, 1999

[Nar00] N. Narasimhan, L.E. Moser and P. M. Melliar-Smith. *Transparent Consistent Replication of Java RMI Objects.* 2nd Intl. Symposium, Distributed Objects & Applications (DOA 2000).

[OMG00] OMG TC document ptc/2000-03-04. *Fault Tolerant CORBA*. Draf Adopted Specification. 2000.

[San00] F. Sánchez, J.Hernández, J.M.Murillo, J.L.Herrero, R.Rodríguez. *Adaptability of Object Distribution Protocols Using the Disguises Model Approach.* 2nd Intl. Symposium, Distributed Objects & Applications (DOA 2000).

[UML99] Object Management Group. Unified Modeling Language, version 1.3. http://www.rational.com/uml/resources/documentation

Transferring Persistence Concepts in Java ODBMSs to AspectJ Based on ODMG Standards

Arno Schmidmeier

Sirius Software GmbH,
Oberhaching

Abstract: *Aspects are abstractions that capture and localise crosscutting type concerns. Although the persistence of aspects has received an increasing interest among researchers in software engineering, basic distinctions between persistent and transient aspects, and their relationship to weaved objects, are lacking clarification. This paper introduces definitions and illustrates how an existing object oriented database management system (ODBMS) can be used as an aspect oriented database management system (ADBMS) based on such definitions and previously established ODMG standards.*

Introduction

Implementation approaches dealing with typical usage scenarios of an object oriented database raise the importance of persistent aspects [1]. The capability to store aspects in an ODBMS requires extending the ODBMS to an ADBMS. This can be achieved by enhancing the persistent object model of an object oriented programming language to that of a general aspect oriented programming language (GAPL), which enhances the first programming language and can be compiled back to it. At the design level, enhancing the persistent object model consists of two major steps:

1. Porting the concept of persistent capable classes to persistent capable aspects
2. Extending the concept of persistence through reachability to encompass aspects.

Once these steps are accomplished, it is relatively easy to write persistent capable aspects. The GAPL compiler enables translating the persistent capable aspects into persistent capable classes, which can then be stored in the ODBMS. As a result, existing commercial ODBMSs can be reused with little or no modifications. The approaches discussed in this article are based on AspectJ, versions 0.8beta1 to beta3 [2], the standards outlined in ODMG 2.0 [3], and compliant database Objectivity 6.0 [4]. A similar implementation of this approach is discussed in [1].

Analysis of the Object and Aspect Models of AspectJ

AspectJ offers, in addition to the java type construct, the aspect type construct. According to the aspect language reference: *"An aspect is a crosscutting type defined by the aspect declaration. The aspect declaration is similar to the class declaration in that it defines a type and an implementation for that type. It differs in that the type and implementation can cut across other types (including those defined by other aspect declarations), and that it may not be directly instantiated with a new expression. Aspects may have one constructor definition, but it must be of a nullary constructor throwing no checked exceptions."* [6]

Instances of an aspect class are called aspect instances[1], which only the AspectJ runtime environment is capable of generating. The aspect class is instantiated based on the aspect signature. However, the standard aspect signature 'of eachJVM()' can be omitted. In this case, one instance is generated inside each Java Virtual Machine where the aspect is used. 'of eachJVM()' realises a kind of a singleton [10] pattern for an aspect.
If a user wants to have more instances of an aspect, the aspect class must be declared using 'of eachobject(PCD[2])', of 'eachcflow(PCD)', or 'of eachcflowbelow(PCD)'. In the first case, a new aspect instance is created for every object associated with the pointcut P. If an aspect class A is defined 'of eachcflow(P)', then one object of type A is created for each flow of control at the join points of pointcut P. If an aspect class A is defined 'of eachcflowbelow(P)', then one object of type A is created for each flow of control below the join points of pointcut P. Except the difference in generating aspect instances, aspect instances and aspects classes behave like objects and classes. Aspect classes not only can extend both java classes and aspects classes, but also can implement interfaces. Aspect instances can be used anywhere a java object is expected. The AspectJ compiler transforms an aspect class into a java class.

Transferring the Persistence Model of Java Classes to Aspects.

For such a transfer to take place, a definition can be made similar to that which is defined for java by the ODMG. Mainly, *persistent capable aspects classes are aspect classes, whose instances can be stored in an aspect oriented database*. All aspect classes are persistent capable if the following conditions are met:

1. The class either implements a specific interface or extends a specific root class.
2. All attributes must be either:
    a. An atomic data type
    b. A persistent capable class
    c. A persistent capable aspect
    d. An atomic data structure
    e. An array consisting only of elements, which fulfil a, b, c, d or e.
    f. or, marked as transient, static or final.

All *aspect classes*, which are not persistent capable, are transient *aspect classes*. Instances of transient aspect classes cannot be stored in the aspect oriented database. All aspect instances, which are stored in the persistent storage, are called *persistent aspect instances*. All other aspect instances are called *transient aspect instances*. Transient aspect instances can become persistent aspect instances, like transient objects can become persistent objects, by either storing the aspect directly in the database (e.g. clustering) or achieving persistence through reachability.

Extending the Concept of Persistence through Reachability

It is necessary to extend the concept of persistence through reachability, which takes into account aspect classes, in addition to the existing mapping of java classes drawn by the ODMG [3], [5].

[1] For clarity, we use the term *"aspect class"* for an aspect and the term *aspect instance* for an instance of an aspect class in this paper.
[2] PCD Pointcut discriminator

To be made persistent at the end of a transaction, all transient aspect instances and objects, *must* be:

1. An instance of a persistent capable class or an instance of a persistent capable aspect instance.
2. Directly or indirectly referenced from a persistent aspect instance or from a persistent object.

This concept is called: *persistence through Reachability for aspects and classes (*or in the context of aspects, simply,*: persistence through Reachability.)* A persistent aspect instance remains persistent, till it is removed explicitly from the database, or till it is removed from the database by the database garbage collector. So it is obvious, that the lifecycle of a persistent aspect instance, can easily extend the lifecycle of some Java virtual machines. Additionally, one can use the same aspect instance in several Java virtual machines at the same time.
Persistence through reachability, as just defined, allows an aspect instance to get stored alone without the object instance for which it was bound; and, vice versa, an object gets stored without the aspects, which it was bound to it.

Some usage patterns:
A new instance of an persistent capable aspect class A is generated at any time, when a public method foo() is called in any class. The constructor of the aspect class A stores the aspect instance in the database. One could use explicit techniques, (e.g. clustering or using named roots) or apply persistence through reachability. The aspect instance will be made persistent independent from that fact, if the object that the aspect binds is persistent capable. When an instance of such an aspect is loaded from the database to a different JVM, it is quite clear, that it is not bound to any object anymore, if no reweaving takes place.
A more common usage pattern is, that only instances of transient aspect classes are bound to an object of a persistent capable class. If this object is made persistent, the aspects instances could not be saved.
In some other cases the weaving relationship shall be counted as a reference according the newly established definitions for persistence through reachability. For example, when an object is made persistent all aspect instances bound to this object should be made persistent as well, and vice versa.

Persistence of Weavings

The usage patterns above illustrate the need to further define such patterns and specify which of these patterns are to be supported by the runtime environments of the GAPL and the ADBMS.

Definitions:
Let O is any transient object and A is a transient aspect instance weaved to O.

If the weaving between O and A fulfils the following conditions:

> If O is made persistent, then A is automatically made persistent too *and*
> If A is made persistent, then O is automatically made persistent too,

…the weaving is considered a *persistent weaving*.

When weaving between O and A meet these conditions:

> If O is made persistent, then A is automatically made persistent **or**

If A is made persistent, then O is automatically made persistent.

…the weaving is called a *partial persistent weaving*.

All other weavings are considered *transient weavings*.

Based on further investigations, it is necessary to differentiate the transient weavings even more. If a persistent object or aspect with a transient weaving is loaded into a JVM and the weaves could be re-established, the transient weaving is considered a *rebindable transient weaving*, or in short a *rebindable weaving*. If the reweaving is not initiated by the programmer or by another "user"-aspect, (e.g. from the database runtime or from the runtime of the GAPL), the weaving is called *transient, automatic rebindable weaving* or, in short, *automatic, rebindable weaving*. If a reweaving is not possible we speak from a *lost through persistence weaving*.

it is possible that some weavings can be persistent, some other weavings of the same persistent object might be rebindable transient, and some other are lost through persistence weavings.

Any ODBMS, supporting at least *lost through persistence weaving*s and (partial) *persistent weaving* can be called an ADBMS from an *aspect* point of view.

Experiences

Sirius Software's Research and Development is currently using Objectivity 6.0 [4] with AspectJ (version 0.8beta1 through 0.8beta3) for the ADBMS, and its GAPL based on the concepts demonstrated in this article. Neither were the AspectJ compiler, nor the AspectJ and Objectivity runtimes changed. *Partial persistent weavings*, *lost through persistence weaving*s and *automatic rebindable transient weavings* are supported out of the box and heavily used.
In the last case, the class is persistent capable, while the aspect class is transient and from a 'of eachJVM' type.
An automatic rebindable weaving for transient aspect classes of the type 'of eachobject()' can be realized by modifying the AspectJ compiler. Currently, the AspectJ compiler does not allow the user to directly invoke the automatically generated methods responsible for initiating the (re)weaving.
Therefore, if rebinding is necessary for transient aspects of type 'of eachObject()' the Java Reflection API is used to bypass these restrictions.
The combination of Objectivity and AspectJ version 0.8beta3 does not currently support rebindable weavings where the aspect class is persistent capable and the class is transient.

It was further discovered in a real world example, that no aspect class of the type 'of eachcflow()' and 'of eachcflowbelow()' is currently stored in the database. Moreover, no aspect instances of the type 'of eachJVM()' which is stored in the database. However, these transient aspect instances are often used in rebindable weavings. Aspects instances of the type 'of eachobject()' are quite often stored in persistent database, most of which are stored through *persistent weavings*.

Sirius Software is due to release a proof of concept in the near future (as well as detailed website postings [7]) that will further substantiate, based on experiences, the feasibility of ADBMSs through existing commercial ODBMSs [8], [9].

Conclusions

Concrete definitions of persistent and transient aspects are required to establish and realize the concept of persistence *of aspects* in object-oriented programming. The definition in this article proved to be a solid foundation for a common wording of Sirius developers and architects in discussing and designing persistency in aspects.
The common wording is a prerequisite for pattern hatching in such an environment. It is still important to examine which patterns of persistent aspects are needed, as well as, the types of rebindable weavings that are really required to support these patterns, when applied in real world projects. The weaving relationship between an aspect and an object does not necessarily establish, or require, persistency. A dynamic weaving support from a GAPL can further propogate the use of persistent aspects. In fact, the Java Mapping of the ODMG can be easily extended to Java based GAPLs like AspectJ and closing the gap between existing commercial ODBMSs and future ADBMSs.

Biography

Arno Schmidmeier (arno.schmidmeier@sirius-eos.com) is the Chief Scientist at Sirius Software GmbH. Prior to his current position he architected the EOS ® SLM Solution. He is the technical representative of Sirius Software in the TMF. He is also an independent member of the Java Specification Request 0090 'OSS Quality of Service API'.

References

[1] "On to Aspect Persistence", Awais Rashid, Proceedings of Second International Symposium on Generative and Component-based Software Engineering GCSE 2000 (part of Proceedings of NetObjectDays2000), pp. 453-463 (Also to appear in post symposium proceedings published by Springer-Verlag)
[2] AspectJ Home Page, http://aspectj.org/, Xerox PARC, USA
[3] Cattell, R. G. G., et al., "The Object Database Standard: ODMG 2.0", Morgan Kaufmann, c1997
[4] Objectivity Home Page, http://www.objectivity.com/, Objectivity inc. Mountain View, USA
[5] Cattel, R.G.G., et al, "The Object Database Standard: ODMG 3.0", Morgan Kaufmann, 2000
[6] Gregor Kiczales, Erik Hilsdale, et al, "Language Semantics", http://aspectj.org/doc/primer/ref/semantics.html,
[6] "Jasmine 1.21 Documentation", Computer Associates International, Inc., Fujitsu Limited, c1996-98
[7] Sirius Research AOD Page, http://www.sirius-eos.com/
[8] Versant Home Page, http://www.versant.com/, Fremont CA, USA
[9] Fast Objects by Poet, http://www.fastobjects.com/
[10] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995
[11] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, William G. Griswold. An Overview of AspectJ. To appear in ECOOP 2001, 2001

Alternatives to Aspect-Oriented Programming?

David Bruce Nick Exon

Distributed Technology Group
QinetiQ ltd
St Andrews Road, MALVERN WR14 3PS, UK

dib@QinetiQ.com njexon@QinetiQ.com

Abstract. In the interest of stimulating debate, we present a broad, all-inclusive view of aspect-oriented programming (AOP) and related initiatives. In particular, we contrast the compelling vision used to 'sell' the idea of AOP with the more finicky nature of its realizations to date. We outline a scenario that we feel AOP ought at least aspire to address, and ask whether the direction it is currently following is likely to take us as far as we really need to go.

Separation of concerns: there is no alternative!

In the general software world, there would appear to be an irresistible trend towards the widespread use of components. There are countless articles, both in the technical literature and the computing press, that expound various reasons for this, and we have neither the space nor the inclination to re-iterate them all here. We will simply note that much the same forces are at work in computer simulation, our core research focus in recent years. Indeed, amidst the inevitable debates on the desirability or otherwise of 'federating' simulation components for large-scale simulations, Richard Weatherly in particular has noted on several occasions that "there is no alternative!".

The view of many researchers is broadly similar for aspect-oriented programming (AOP) [1,2]. The limitations of traditional, time-honoured but fundamentally quite basic techniques for software construction are increasingly making themselves felt, most notably through our inability to construct and evolve programs at the pace demanded by modern times. Proper separation of concerns [3,4] plays a vital rôle here, but the abstraction and composition mechanisms we have today far from suffice. What AOP and related initiatives can offer are exciting glimpses of how we might be able to articulate and encapsulate hitherto-elusive concepts in qualitatively new ways. Thereby springs a much-needed sense of hope; surely there can be no alternative?

An anecdotal non sequitur

Back in autumn 2000, the first author gave an internal talk to our group, entitled "Aspect-oriented programming and AspectJ". This talk first examined the nature of computer software, and some of the fundamental problems caused by inadequate

separation of concerns. It went on to present the vision motivating aspect-oriented programming, that one could provide independent specifications for each distinct concern and then 'weave' them together to build the resulting system. The talk finally touched on some of the prototype AOP systems/tools that were available at that time. In particular, it outlined Xerox PARC's AspectJ [5,6,7] and the sorts of things that that language lets you do — specify program 'pointcuts', add or modify functionality through before/after/around advice, extend classes using introduction, and associate 'aspect' classes with objects, pointcuts, etc.

Several of our colleagues pointed out — some there and then, others later — that the early part of this talk was fine, as was the later part, but the two seemed a void apart. The idea of AOP was great, they said, and AspectJ made perfect sense in its own right … but the former was a grand conceptual vision while the latter focussed on low-level details.

Since then, the authors and various other members of our group have experimented with AspectJ from time to time. This continued exposure to AspectJ has done little to bridge the gap, however; if anything, it's reinforced it! *(We should note that we're not picking on AspectJ in particular here; it's just the most prominent example, and the one that we've had most experience with. Other AOP and related systems seem broadly comparable in this regard. More on this later.)*

So, maybe our colleagues' gut-reactions were founded; maybe there is something missing? But what? Could it just be that we're being dumb? We'd like to think not! The Xerox PARC team behind AspectJ acknowledge that its documentation often lags behind their implementation efforts, but it would be churlish as well as disingenuous to suggest that that's the problem. Aspect-oriented programming is, of course, a relatively new field, so it is only natural that the community at large will take some time to learn and communicate good design principles; perhaps we just need to wait? (It is worth noting here that excellent tutorials such as [7] are now starting to emerge.) One final option remains: it could be that AspectJ et al. really are too low-level for our ambitions — or, turning that around, that we're guilty of expecting too much.

A multi-dimensional functionality thought experiment

We have spoken of aspirations, but given few details. What sort of thing do we have in mind?

One way to articulate such matters is by means of a 'thought experiment' — in this case taking inspiration from the military simulation domain. Our intent is to show the breadth of multi-dimensional functionality in what for that domain is a relatively simple problem, and to stimulate thought about how software construction techniques influence its subsequent evolution.

Consider a computer generated forces (CGF) assault on an enemy position. The requirement is for a simulation (component) to plan and execute an operation:

- according to some specified scenario (location, time, military resources, opposition, …)
- using a given form of reasoning process (broad agents, rule based, scripted, …)
- following particular doctrinal principles (tactics, rules of engagement, …)

- inside certain computational resource limits (time, memory, …)
- implementing a particular style of simulation (training, analytic, predictive, …)
- visualized as required (immersive VR, plan view display, statistical summary, …)
- within acceptable validity tolerances
- …

Each of the above represents a design decision that could — in principle at least — be changed independently. To get a sense for how hard it is to plan ahead for all possible eventualities, think about how you might go about coding such an example. What abstraction mechanisms would you use to structure it? How well could your approach cope with this range of changes, and with other possible variations that you can think of?

Clearly, some of the flexibility that we and our customers demand can be accommodated using conventional methods (e.g., parametrization of scenario). Aspects as we currently know them might well serve for others (e.g., at least for some forms of visualization). However, entirely new techniques would also seem to be required (e.g., for separating out elements of 'intelligent' behaviour such as doctrine — especially if this is to be in some way independent of the reasoning approach).

So what's the point?

Having started by arguing the case for aspect-oriented programming, we conclude by turning about-face to knock our strawman down — if only on a technicality.

Although there may be no alternative but to pursue such mechanisms, that does not make the future entirely predestined and inevitable. We actually have a lot of choice. It is not the choice of whether to adopt something like AOP, but the more exacting choice of how best to adopt it. This might not be the choice we thought that we had, but it's actually a pretty good one; being so wide-ranging and open-ended, it gives plenty of room for manoeuvre.

In other words, the principle seems sound, but the practice still needs a good deal of refinement. The real question is whether the mechanisms that we know about now (e.g., those in AspectJ, in other variations on the theme such as HyperJ [8], or even those investigated in related initiatives such as Minsky's law-governed regularities [9] or Microsoft's intentional programming [10,11]) suffice to satisfy the aspirations that we already have, and those that we are going to formulate over the coming years.

Our honest answer is that we don't know, but on balance we remain sceptical.

We therefore challenge the research community to join us in looking for new forms of program abstraction, composition and transformation — which may or may not end up resembling (or being called) aspect-oriented programming — that address both the precisely formed targets of academic fascination and the less easily characterized problems that software developers really face. Bridging the gulf between conceptual levels, and exploring the full life-cycle viability of aspect-oriented programs, are but two of the more interesting that immediately spring to mind.

Aspect-oriented programming as we know it now is doing a grand job of exploring interesting territory; we simply urge that the research community widen its horizons, to see what else remains uncharted.

References

1. Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopez, John-Marc Loingtier and John Irwin, "Aspect-Oriented Programming", pp. 220 ff. in Proc. *11th European Conference on Object-Oriented Programming* (Jyväskylä, Finland, 9–13 June 1997) — published by *Springer-Verlag* as *Lecture Notes in Computer Science* no. 1241 (Mehmet Akşit and Satoshi Matsuoka, editors).
2. "Aspect-Oriented Programming".
   Xerox PARC website, URL: http://www.parc.xerox.com/csl/projects/aop/.
3. D. L. Parnas, "On the Criteria To Be Used in Decomposing Systems into Modules", pp. 1053–1058 in *Communications of the ACM*, Vol. 15, No. 12, December 1972.
4. Edsger W. Dijkstra, "A discipline of programming", *Prentice-Hall*, 1976.
   [See in particular "In Retrospect" (chapter 27; pp. 209–217), and also "note 1" (p. 203).]
5. "AspectJ: Crosscutting Objects for Better Modularity".
   Xerox PARC website, URL: http://aspectj.org/.
6. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, "An Overview of AspectJ", pp. 327 ff. in Proc. *15th European Conference on Object-Oriented Programming* (Budapest, Hungary, 18–22 June 2001) — published by *Springer-Verlag* as *Lecture Notes in Computer Science* no. 2072 (Jørgen Lindskov Knudsen, editor).
7. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, "Getting Started with AspectJ", tutorial article submitted for a special theme section on Aspect-Oriented Programming to appear in *Communications of the ACM*, Vol. 44, No. 10, October 2001.
   (Available on-line at URL: http://aspectj.org/doc/gettingStarted/index.html.)
8. "HyperJ™: Multi-Dimensional Separation of Concerns for Java™".
   IBM Research website,
   URL: http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm.
9. Naftaly H. Minsky, "Law-Governed Regularities in Object Systems. Part I: An Abstract Model", pp. 283–301 in *Theory and Practice of Object Systems*, Vol. 2, No. 4, 1996.
10. Charles Simonyi, "The Future Is Intentional", pp. 56–57 in *IEEE Computer*, Vol. 32, No. 5, May 1999. [One of nine contributions to "Software [R]evolution: A Roundtable", Kirk L. Kroeker (editor), pp. 48–57 of that issue.]
11. "Intentional programming".
    Microsoft Research website, URL: http://www.research.microsoft.com/ip/.

Acknowledgements

One of the authors has twice had the privilege of seeing Gregor Kiczales' excellent presentations on aspect-oriented programming. Our colleague David Allsopp offered some particularly thoughtful observations. This work has been supported by the UK Ministry of Defence under Corporate Research TG10 project 5.4.4, "Re-usable Simulation Components for Synthetic Environments".