Dynamic Aspect-Oriented Programming in an Untrusted Environment

Doug Palmer
CSIRO Mathematical and Information Sciences
GPO Box 664, Canberra, ACT, 2601, Australia
Doug.Palmer@csiro.au

Abstract

Current component-based architectures allow great flexibility in component composition but little in terms of component customisation and extension. Dynamic aspect-oriented programming, using wrapper technologies, offers a way of allowing components to be individually customised and extended. Dynamic aspect-oriented programming can be used in a public computing environment to allow richer interaction between components that are interacting on an unplanned and temporary basis. Mobile code, in the form of wrappers, can be used to extend and interact with other components. In a public environment, mobile code represents a considerable security risk, particularly as the wrappers are used to modify the behaviour of an existing component. A security mechanism, based on explicit contracts, can be used to detect and correct the effects of any hostile code that damages the behaviour of a component.

1 Introduction

Current distributed architectures make heavy use of components: independent binary units that can be composed to form larger systems. Component-based architectures tend to allow considerable flexibility in the structuring of component relations, but not much in the way of adjusting or customising the components themselves.[1] Components tend to present a static interface and semantics. The use of the component is, therefore, reliant on a sufficiently rich interface and architecture being built into the component at design time.

Aspect-oriented programming (AOP)[2] is a continuation of the principles inherent in object-oriented programming. AOP is usually used to allow the separation of concerns within a program or object, such as business functionality, security, transactional behaviour, etc. where these concerns cut across the normal class hierarchy of an object-oriented program. The concerns are *woven* into a program. This weaving usually occurs at compile-time — for example, AspectJ.[3] However, AOP can also be used to model and control the process of customisation and extension of existing components; an extension is essentially another aspect to be added to the component. Dynamic aspect-oriented programming (DAOP)[1, 4] allows components to be extended dynamically at run time.

Dynamic aspect-oriented programming is relatively straightforward in a trusted environment, where (with the exception of bugs) aspects can be assumed to be safe. In an untrusted public computing environment, new aspects cannot be assumed to come from a benign source. There need to be mechanisms that ensure that the underlying component is not damaged by the addition of potentially hostile aspects.

This paper is structured as follows: an example, drawn from the CSIRO Smart Spaces project, is presented in section 2; the security aspects of untrusted DAOP, based on wrappers is discussed in section 3; a security mechanism, based on the concept of a *validator wrapper* is presented in section 4.

2 A Motivating Example

The *Smart Spaces* project draws on a number of areas of expertise within CSIRO with the intention of creating spaces filled with sensors and other "smart" devices. We use the example of a smart shop, drawn

from the Smart Spaces project. In a smart shop, someone entering the shop immediately communicates their shopping and comfort preferences to the shop, using their own personal computing devices. In turn, the shop communicates catalogue and other information to the customer. As the customer moves through the store, a mixture of local devices and the main shop backbone adjust the store environment, highlight interesting items and records anything that has been picked up as a potential sale.

The levels of customised behaviour in this example can be very high. The shop itself represents a particular way of doing business and will be customised to reflect that; entering the next shop is likely to result in radically different behaviour. The customer's own devices will also be highly personalised. The interaction between the customer, the shop and the network of sensors and other devices in the shop is, therefore, likely to be both complex and highly flexible. Building a static component architecture that accommodates such flexibility is, of course, possible but more dynamic architectures allow the development of more tractable solutions.

A simple example involves a connection to the customer's display shown in figure 1. A catalogue of products can be shown on the display. When the customer moves their pen over a product in the catalogue, the shop is notified and contrives to highlight the product in the shop. We have the following interfaces:

```
interface Select {
  void setSelection(Object selectee);
};

interface Select {
  void turnOn();
  void turnOff();
};
```

Both the **Display** and **Assistant** components implement the Select interface. On the **Display** component, the `setSelection` method is invoked when an object is selected on the display. When a method is invoked, the **Assistant** component would also like to be informed of the change. A typical static component would need to provide some sort of notification interface, so that **Assistant** could register to be notified of events. As an alternative, **Assistant** could supply an aspect to **Display** with the following semantics, (given in a made-up language):

```
aspect SelectListener extends Select {
  private Select listener;

  void setSelection(Object selection) {
  } {
    this.listener.setSelection(selection);
  }
};
```

This aspect wraps the `setSelection` method with pre- and post-invocation actions. In this case, the pre-invocation action is null and the post-invocation action is to call `setSelection` on the **Assistant**.

When the customer leaves the shop, the aspect can be removed.

3 Security

Since both the **Display** and **Assistant** components from section 2 are exposed, almost any aspect can be supplied to be integrated into the components. As a consequence, such exposed components are vulnerable to hostile code wrapping methods and causing unexpected behaviour. The security aspects of such exposed components, therefore, extend beyond simple access security into testing for program correctness and code verification.

Dynamic aspect-oriented programming needs a technology that allows the easy insertion and removal of aspects in a running system, while maintaining object identity. Wrapping systems, where objects can be wrapped in decorators[5, 6] provide a convienient mechanism for building DAOP platforms. Both JAC[4] and Lasagne[1] use wrappers. From the point of view of an untrusted environment, wrappers have a number of advantages:

- They do not contaminate the core component, making insertion and removal of aspects relatively straightforward.
- They can be structured so that pre- and post-invocation of the underlying method can be separated and analysed independently.
- Individual components can be modified on a per-component basis, rather than a per-class basis.

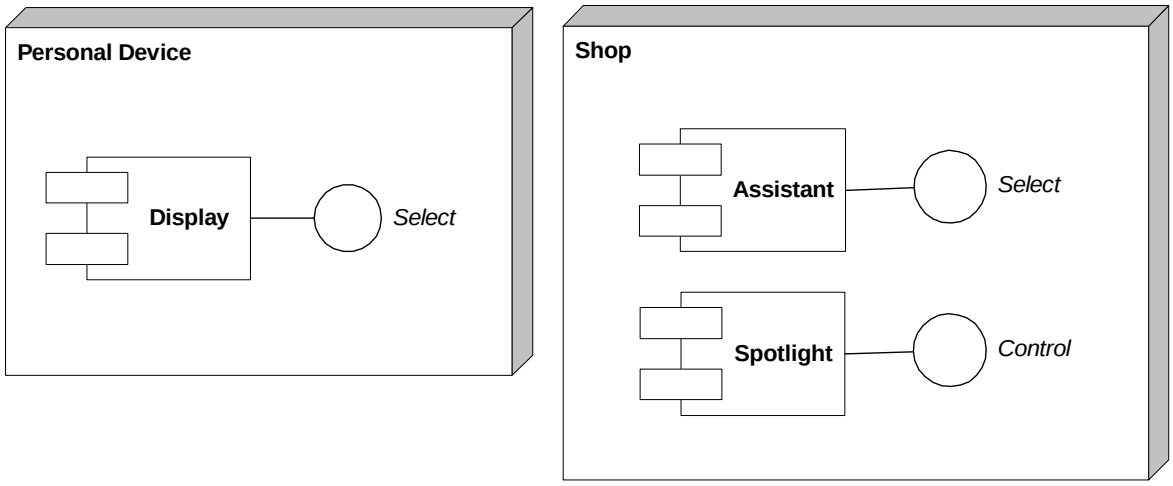

Figure 1. Example Shop Components

Similarly, different clients can use different wrappers on the same component.

- They can be ordered, so that the wrapping sequence is always known.
- They can be identified, so that hostile aspects can be identified and cleanly removed.

This section does not discuss the question of basic identity validation, such as code-signing[7]. Mobile code in the form of aspects will need to be signed to ensure that the code has not been tampered with before delivery.

3.1 Component Contracts

In order to determine whether an aspect is hostile or not, there needs to be some concept of what constitutes *correct behaviour* for a component. Any aspect that interferes with the correctness of an underlying component can then be deemed hostile.

To ensure correct behaviour, a component needs to be checked against a declarative model of the component's expected behaviour. This forms the basis of a design by contract approach to correctness.[8]. A component's *contract* is a set of assertions, invariants and pre- and post-conditions that establish the range of acceptable behaviour. When a message is sent to an aspect-extended, the component needs to test its contract before the message returns to ensure that the contract still holds.

As an example, the `setSelection` in the **Display** component may have the following post-condition, which ensures that the method has behaved as expected.

```
void setSelection(Object selectee)
  postcondition "Selection not set" {
    this.getSelection() == selectee;
  }
```

3.2 Inner and Outer Wrappers

Wrappers can be divided into two classes: *inner* and *outer*. Outer wrappers are wrappers that need to be used when a method is called from an external source. Inner wrappers need to be used whatever the source of the method call.

Outer wrappers represent the sort of wrappers normally used to add aspects such as security, transactional management, etc. to a component. In the example in section 2, an attempt by the **Assistant** component to invoke the `setSelection` method on the **Display** component would need to go through the outer wrappers. It is unlikely that the display would accept changes from an external source and an access control exception would be raised.

Outer wrappers do not need to strictly observe the component contract, since they effectively represent

behaviour external to the component. In some cases, eg. access control, it is simply not possible to maintain the component's contract, since a security violation will raise an exception, leaving the operation unperformed.

Inner wrappers represent modifications to the basic functionality of the component. The aspect added to the **Display** component in the example is an inner wrapper. The **Assistant** is interested in all calls to `setSelection`, including those from trusted components within the customer's personal device.

Inner wrappers can fundamentally modify the behaviour of a component, possibly to the detriment of the component's other clients. As such, a component with an inner wrapper must ensure that the component's contract is being observed.

4 Validating Contracts

The natural way to allow contract validation within a wrapped system is to treat the validator as another wrapper. The validator wrapper can then test calls exiting from the inner wrappers. The process is illustrated in figure 2. A series of wrappers, both inner and outer, surround a component. A call first passes through the stack of pre-action modifications. As the call passes through the validator wrapper, the validator can store any information that it needs to ensure validation. As the call returns, it passes through a stack of post-actions. As the return passes through the validator, the validator can then check to ensure that the component's contract is still sound.

4.1 Instrumenting Wrappers

In general, the validator will need to check only a part of the component's contract with each call. In many cases, no checking need be done at all.

To allow the validator to intelligently check wrappers, wrappers will need to be *instrumented*. An instrumented wrapper can be trusted to inform the validator if a potentially contract-breaking operation takes place.

The instrumenter is similar in spirit to the standard Java bytecode verifier,[9] with an additional re-write component. A dataflow analysis needs to be performed to identify the following possibly unsafe actions:

- An additional message is sent to the component.
- A variable which represents an argument is assigned to, or the variable is not returned intact in the pre-action phase.
- A message is sent to one of the method's arguments. There is a potential for variable aliasing, both directly by assignment and indirectly through object construction, that needs to be covered by the analysis.

In addition, exceptions may not be thrown from the pre-action parts of inner wrappers. Otherwise, it is possible to block access to a method.

Messages sent to external components can be ignored. If they, in turn, result in further messages being sent back to the component, these messages will act as ordinary interactions with the component. Distributed deadlock, however, needs to be prevented; it is not clear how this requirement can be cleanly achieved, although prevention is necessary if hostile code is to be prevented from deadlocking a component. A simple solution to deadlock and other forms of hanging[1] is to use a time-out on any external, untrusted interactions. Any aspect that causes a time-out causes a contract violation.

If a potential contract violation is identified, the wrapper can be modified to signal the need for a contract check to the validator. Along with the need for a contract check, the requesting wrapper and, if possible, the potential violation can also be supplied to the validator. The wrapper needs to be identified, so that contract violations can be remedied (see section 4.2).

The system that plays host to the wrapper must, ideally, perform the verification and instrumentation of the wrapper, to ensure security. Instrumentation may be a complex task and it is possible that some of the devices described in the Smart Spaces project would not have the necessary resources. In this case, a trusted proxy instrumenter would need to be used. In the example in section 2, it may be necessary to have the main server in the shop instrument wrapper destined for such devices as spotlights and temperature sensors.

[1] A hostile component could simply just not return from a call, causing the client to hang.

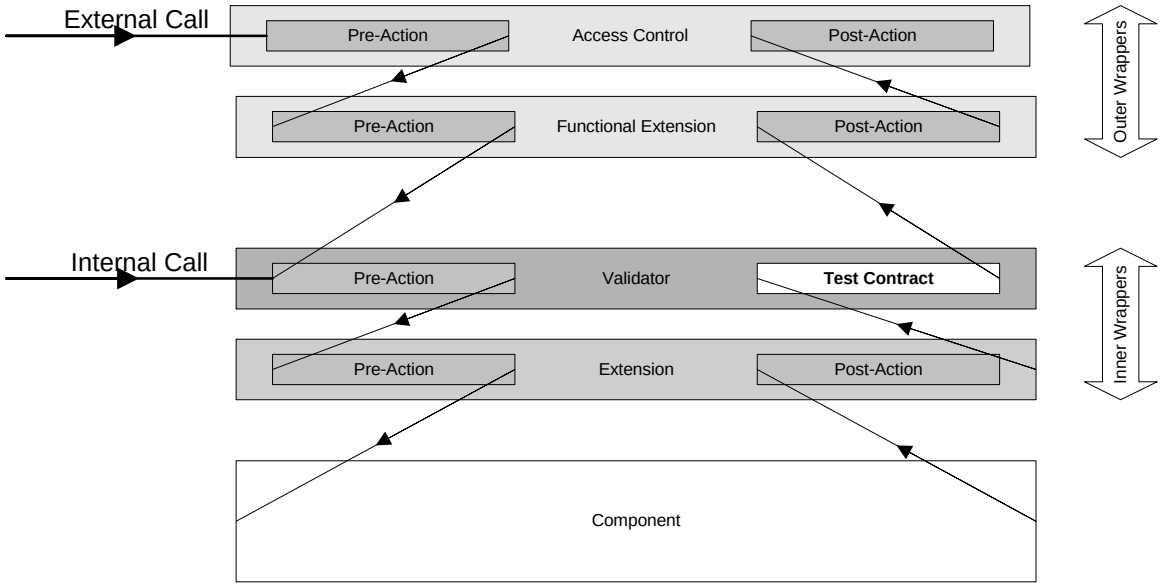

Figure 2. Contract Validation

4.2 Handling Contract Violations

When a contract violation is detected the operation that has been invoked needs to be rolled back and the offending wrapper(s) removed from the wrapper stack. In general, contract violations can be viewed exceptional events akin to program crashes; recovery can be similarly brutal.

Rolling the operation back requires both the recovery of the component's state and the rolling back of any actions taken by the component. In some cases, where the operation is wrapped in a transactional context, rollback is a relatively straightforward affair. In more extreme cases, the component and its clients may need to be recovered from a snapshot or simply shut down and restarted.

Removing a (potentially) hostile wrapper requires the offending wrapper to be identified from the stack of wrappers in the system. The process of instrumentation, discussed in section 2 allows a *candidate set* of wrappers to be identified when a contract violation occurs. Only those wrappers which indicate potential contract violations of the appropriate part of the contract form part of the candidate set.

Ideally, the candidate set consists of a single wrapper, which can then be *ejected*: removed from the component, along with any other wrappers from the same source. If the candidate set consists of multiple wrappers, then either all of the candidate wrappers can be removed or each wrapper can be removed in turn and the call restarted, until the offending wrapper is uniquely identified. Once a wrapper is ejected, the call can be restarted, if desired.

5 Conclusions

Dynamic aspect-oriented programming offers the possibility of a more flexible, interactive style of distributed architecture. In an untrusted environment, however, DAOP can allow hostile code to damage existing components. A security mechanism based on explicit contracts, contract testing and recovery allows the safe use of DAOP in an untrusted environment.

References

[1] Eddy Truyen, Bart Vanhaute, Wouter Joosen, Pierre Verbaeten, and Bo Nørregaard Jørgensen, "Dynamic and selective combination of extensions in component-based applications," in *Proceedings of the International Conference on Software Engineering (ICSE 2001)*. May 2001, IEEE Computer Society.

[2] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Christina Videra Lopes, Jean-Marc Loingtier, and John Irwin, "Aspect-oriented programming," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001)*, Mehmet Aksit and Satoshi Matsuoka, Eds. June 1997, vol. 1241 of *LNCS*, pp. 220–242, Springer Verlag.

[3] Gregor Kiczales, Erik Hilsdale, Mik Kersten, Jeffrey Palm, and William G. Griswold, "An overview of AspectJ," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2001)*, Jørgen Lindskov Knudsen, Ed. June 2001, vol. 2072 of *LNCS*, pp. 327–353, Springer Verlag.

[4] Renaud Pawlak, Lionel Seinturier, Laurence Duchien, and Gerard Florin, "JAC: A flexible solution for aspect-oriented programming in java," in *Metalevel Architectures and Separation of Crosscutting Concerns, REFLECTION 2001*, Akinori Yonezawa and Satoshi Matsuoka, Eds. Sept. 2001, vol. 2192 of *LNCS*, pp. 1–24, Springer Verlag.

[5] Martin Büchi and Wolfgang Weck, "Generic wrappers," in *Proceedings of the European Conference on Object-Oriented Programming (ECOOP 2000)*, Elisa Bertino, Ed. June 2000, vol. 1850 of *LNCS*, pp. 201–225, Springer Verlag.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design Patterns*, Addison-Wesley, 1995.

[7] Gary McGraw and Edward W. Felten, *Securing Java*, Wiley, 1999, `http://www.securingjava.com/`.

[8] Bertrand Meyer, *Object-Oriented Software Construction*, Prentice-Hall, second edition, 1997.

[9] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification*, Addison-Wesley, second edition, 1999, `http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.ht%ml`.