

Supporting Inconsistent World Views

(Position Paper)

Robert J. Walker
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, Alberta T2N 1N4, Canada
rwalker@cpsc.ucalgary.ca

ABSTRACT

When composing components that have been independently developed, coordinated pre-planning to ensure interoperability is not always an option. Without pre-planning, mismatches in architectures, interfaces, or protocols tend to occur that prevent the composition of components. To enable the composition of mismatched components, components must be permitted to possess *inconsistent world views* on the constituents of a system, their interfaces, and protocols. At composition time, the conflicts in these world views are reconciled. Each component can express only those details about the system that are important for its local operation. As a result, a component can interact with its context of operation while remaining oblivious to the concrete details there. This position paper outlines how a technique called *implicit context* can be used to support and reconcile inconsistent world views. No other existing AOP approaches are capable of this.

1. INTRODUCTION

Components cannot be treated like “LEGO blocks” in general [21]. Components often need to be coordinated in ways not supported by the standards for which they are developed, such as observing system-wide policies for security or fault handling. In addition, standards, languages, and protocols either evolve over time, or they stagnate and become obsolescent. This results in components adhering to different, mismatched standards. And in situations where there was no attempt to coordinate the interaction of components in the first place, the likelihood of mismatch is all the greater [10].

Instead, we need technology that allows us to compose independently-developed components, without resorting to coordinated pre-planning or ad hoc invasive modifications. Such components will tend to possess *inconsistent world views* about their contexts of operation. While one might expect aspect-orientation to aid us in dealing with inconsistent world views, standard approaches (such as that of AspectJ [16]) do not suffice.

Rather than attempt to eliminate such inconsistencies a priori, we wish to reconcile any conflicts after the fact. This reconciliation is

performed through the use of *implicit context* [26, 25], a technique related to aspect-oriented programming.

2. MOTIVATION

Consider a system in which there exists a module (the `Client`) that needs to obtain a new instance of a class (the `Product`) during the conduct of its activities. In this system, the `Product` class provides a constructor with which new instances may directly be created, and the `Client` uses this constructor.

Imagine that this original system is modified to make use of the Abstract Factory design pattern [9]. The `Product` class becomes an abstract base class that is subclassed on the basis of platform-specific details. Since the `Client` is not concerned with these platform-specific details, Abstract Factory allows us to hide the choice of which subclass to instantiate. An instance of `AbstractFactory` is passed to the `Client` through a newly added parameter and the `Client` requests of this `AbstractFactory` that a new instance of `Product` be created. This request is serviced polymorphically to return an instance of the appropriate subclass of `Product`.

The `Client` modules in the two versions of this system have identical purposes. Rather than maintain a separate version of `Client` for every kind of system that might be encountered, allowing the `Client` to ignore the differences in concrete details of the instantiation protocol would be a win. We could have a single version of `Client`, and it could express the services it needs of the system around it as simply as possible.

Therefore, we wish to implement `Client` as shown in Figure 1. This implementation expresses, in as simple a fashion as possible in Java, that the `Client` requires access to a new instance of `Product`. The perspective of `Client` thus forms one world view.

Externally to the `Client`, the system could make use of the Abstract Factory design pattern or not as appropriate. This is a second world view. In the former case, the world view held by `Client` and the world view held by the remainder of the system are consistent; in the latter case (shown in Figure 1), they are inconsistent.

If we had a means to reconcile the inconsistencies in world views, we would have a highly reusable `Client` component that did not require coordinated pre-planning to be used in these differing systems. We now consider the model of implicit context, a means of such reconciliation.

3. IMPLICIT CONTEXT

Implicit context is based upon three concepts: *boundaries* between conflicting world views; *contextual dispatch* which is used to alter communications passing across boundaries; and *communication history* which is used to retrieve previous state when performing

```

public class Client {
    public void someMethod() {
        Product p = new Product();
    }
}

// --- inconsistency ---

public abstract class Product { }

public class MSWindowsProduct
    extends Product { }

public class MotifProduct
    extends Product { }

public abstract class AbsFact {
    public abstract Product makeProduct();
}

public class MSWindowsFact
    extends AbsFact {
    public Product makeProduct() {
        return new MSWindowsProduct();
    }
}

public class MotifFact
    extends AbsFact {
    public Product makeProduct() {
        return new MotifProduct();
    }
}

public class Main {
    public static void main(String[] args) {
        Properties.initialize();
        AbsFact factory =
            Properties.getFactory();
        Client c = new Client();
        c.someMethod(factory);
    }
}

```

Figure 1: The system when inconsistent world views are permitted. The Client attempts to directly instantiate Product, even though the latter turns out to be abstract. The system external to Client expects that instances of Product will be created through the interface provided by AbsFact.

contextual dispatch.

Contextual dispatch may be performed on communications as they cross a boundary between conflicting world views. Rather than selecting the implementation of a method on the basis of the run-time type of a given object, we select the method or methods to execute on the basis of the context in which the communication occurs—both static properties and dynamic properties of the system. Contextual dispatch can be used to fill-in concrete details in an abstract service request: a message can be rerouted to a concrete implementor of the service, with additional parameters filled-in.

Trouble can arise when, for example, the source of such additional parameters is not immediately obvious. In such a situation,

communication history can be used. Communication history is, conceptually, a record of all communication that has occurred in the system, including all arguments passed and values returned. Through queries on communication history, we can retrieve state information of interest: the last passed argument of a particular type, for example. Communication history queries have the advantage that they can be expressed to minimize our assumptions about the system, permitting the use of a module in many different contexts of operation. Ideas similar to communication history have been used in a variety of specification techniques [2, 7, 13, 3, 15].

Implicit context has been used to ease software evolution and software reuse of a variety of systems, including the Java Swing library [26], a File Transfer Protocol server, and the Eclipse integrated development environment [25]. To see how implicit context can be used to reconcile differences in inconsistent world views, we consider its application to the Abstract Factory design pattern example.

3.1 Integration through Implicit Context

Consider the inconsistent system shown in Figure 1. There are two problematic communications between the conflicting world views. First, when Main calls Client, it attempts to pass an argument of type AbsFact; however, the method declared by Client does not take a parameter of this type. Second, Client attempts to directly instantiate Product; however, the system external to Client sees Product as abstract, and requires instantiation of its subclasses through an instance of AbsFact.

To cope with these conflicts, we begin by considering there to be a boundary surrounding Client that separates the inconsistent world views. When the problematic communications described above cross this boundary, they must be altered through contextual dispatch. In my prototype tool for applying implicit context, contextual dispatch is performed in snippets of code called *boundary maps*. Sets of boundary maps are organized into constructs called *mapsets*, each of which is explicitly associated with a given boundary.

The mapset, called AbsFactMaps, used to reconcile the inconsistent world views in our example is shown in Figure 2. It contains two boundary maps, one to deal with each of the two problematic communications described above. The first boundary map intercepts incoming calls on the method someMethod where an argument of type AbsFact is being passed. This call (along with its argument) is discarded, and replaced with a call to someMethod

```

mapset AbsFactMaps {
    void map(): in(someMethod(AbsFact)) {
        someMethod();
    }

    Product map(): out(Product.new()) {
        AbsFact factory = (AbsFact)History.
            lastInstancePassed(AbsFact.class);

        return factory.makeProduct();
    }
}

apply AbsFactMaps to Client;

```

Figure 2: The mapset and apply statement used to reconcile the conflicts between the two inconsistent world views in the Abstract Factory example.

taking no arguments. The second boundary map is more complex; it is used to intercept outgoing calls to instantiate `Product`. A query is made to communication history to find the last instance of `AbsFact` that has been passed in any communication.¹ This instance is then used to create and return an instance of `Product`. The `apply` statement shown below the mapset is used to associate this mapset with the boundary of `Client`.

After the application of the mapset, the `Client` is composed with the remainder of the system. However, from the perspective of the `Client`, the system does not use the Abstract Factory design pattern and `Product` is a concrete class. From the perspective of the rest of the system, the Abstract Factory design pattern is in force, and `Client` supports it. The inconsistencies have been reconciled.

Implicit context uses a model of *local execution*. This model describes how the local semantics of a program fragment can be interpreted regardless of the context in which that fragment exists. This local execution is translated to more global views. The model is based upon the translation between local execution traces and pattern matching against these traces. The details of this model, and its use in efficient support of communication history, lie beyond this position paper (see [25]).

We proceed to consider how AspectJ might be applied to this example.

4. INTEGRATION THROUGH ASPECTJ

Let us consider how we can reconcile the inconsistent world views in Figure 1 using AspectJ. We declare an aspect to attempt the reconciliation; the result, called `AbsFactAspect`, is shown in Figure 3.

```
aspect AbsFactAspect {
    pointcut incall(AbsFact factory):
        call(void Client.someMethod(AbsFact)
            && args(factory));

    pointcut outcall(): call(Product.new());

    around(AbsFact factory):
        incall(factory) returning void {
            Client.someMethod();
        }

    around(AbsFact factory):
        outcall() && cflow(incall(factory))
        returns Product {
            return factory.makeProduct();
        }
}
```

Figure 3: The putative aspect used to separate concrete details of the instantiation protocol.

In this aspect are defined two pointcuts. The first one (`incall`) captures attempts within the system to send an instance of `AbsFact` to the `Client`. Events that are calls to `someMethod` on `Client`, where an instance of `AbsFact` is passed as an argument, are intercepted. The second pointcut (`outcall`) captures

¹The query syntax is weak and due to be replaced in an improved tool. The new syntax will provide for specifying patterns to match against communication history in a construct reminiscent of AspectJ's pointcut designators.

attempts by the `Client` to directly instantiate the `Product` class.

These pointcuts are used in two pieces of around advice. In the first, the attempt to call the `Client` module while passing the instance of `AbsFact` is replaced by a call to the `Client` module without this instance. In the second, the attempt to directly instantiate the `Product` class is replaced with a call to the `AbsFact`, requesting that it construct an instance of (the appropriate subclass of) `Product`. The instance of `AbsFact` to be used is retrieved via the exposure of calling context by the `cflow` pointcut designator (PCD). This PCD (roughly) finds all events that occur while a given method execution remains on the stack.²

The system would not compile with this aspect. The AspectJ compiler would complain about several factors with the proposed design, such as that the `Product` class is abstract but its constructor is being called—even though that call is completely replaced by the advice. This is not to say that there are flaws in AspectJ, its syntax, or its implementation; it is simply that the design we have attempted cannot be expressed in the AspectJ-style of AOP. One might claim that our particular design for this problem is flawed; however, others have also found that the Abstract Factory design pattern is not composable in the AspectJ-style of AOP [11].

The semantics of the AspectJ-style join point model have been described elsewhere [27]. To summarize and simplify, this model is built around the concept of the execution stack. Join points occur when the execution stack changes, or is about to change. For example, an execution pointcut designator describes those events when a method is executed—roughly speaking, on top of the execution stack. The `cflow` pointcut designator selects stack events that occur when a given method execution exists somewhere on the stack, i.e., those events that occur within the control flow of a given execution. Of course, some join points do not correspond to actual events on the Java Virtual Machine execution stack, and others can be statically determined and dealt with through source code instrumentation; however, such special cases and optimizations are still conceptually consistent with the basic execution stack model.

For the Abstract Factory example, the basic execution stack model cannot support our desired solution because calls and executions are tightly coupled events under this model. The execution stack does not support calls as separate entities; therefore, we cannot utilize information, such as the Abstract Factory object passed in the call to the `Client`, that does not occur in the execution stack. Also, any call is treated as though it has a receiver of the corresponding signature; therefore, we cannot make calls, such as the direct instantiation of the `Product` class, unless that receiver exists—even if the call will never arrive there. It is true that AspectJ provides a `call` pointcut designator, but its use lies in aiding the quantification of events [8] and the scope over which those events occur.

5. RELATED WORK

There have been various attempts at reconciling mismatches in independently developed components. Only implicit context supports localized, inconsistent world views.

Hölzle [14] describes the problems with simple techniques such as adaptors and wrappers.

Type adaptation [28] provides for a dynamic, protocol-based adaptation between two modules. This adaptation is based on the formal specification of the communication protocols of the two modules being composed. It can possess memory, but cannot make

²Alternatively, we could have stored the instance of `AbsFact` within a local variable in the aspect. This would not affect the difficulties encountered.

reference to the system at large (to obtain additional parameters, or to determine the recipient of a call), nor to explicit communication history. Implicit context permits a partial, incremental adaptation between multiple modules; unbound interactions can later be adapted through the addition of boundary maps. Communication history permits us to specify partial traces of behaviour; specifying complete state machines is tedious and error-prone, as noted by Yellin and Strom. Implicit context remains to be formalized to the extent of the type adaptation work.

Generic programming [19, 20], most famously exemplified by the C++ Standard Template Library [23], is concerned with finding the most abstract representation of efficient algorithms. Generic programming is realized concretely through templates: generic algorithms expressed in terms of formal, generic parameters; concrete versions of these algorithms can be generated for concrete contexts by binding these generic parameters to concrete values or types. Although a generic module is defined relative to a set of formal parameters, the manner in which those parameters can be bound to external entities is constrained. Arbitrary functionality cannot be performed to map one of these references to the external perspective.

Flexible packaging [6] focuses on separating the details about a module's interaction from the module itself. Flexible packaging separates a module's functionality and its interactions, called its packaging, into distinct entities: a ware and a packager. A given ware can be packaged to work in different environments, such as a plug-in for a web browser or a command-line filter. Although the interaction is generic, a ware must explicitly define its interaction with packagers. Thus, there is no support for crosscutting concerns, or inconsistent world views.

Mixin layers [22] take a set of collaborating mixins and places them in a parameterized module. The layers are then stacked vertically by instantiating each with the layer below it; within the definition of the mixin layer lie the more fine-grained bindings between the lower mixin layer (specified as a formal parameter) and the current one. This technique requires a high degree of pre-planning and standardization, since each mixin layer is dependent upon particular names being present within the vertical dimension of composition. In contrast, implicit context can be used to compose independently developed collaborations, and does not depend upon there being a strict hierarchy.

In addition to AspectJ, there are a number of other approaches to aspect-oriented programming. Composition filters [1] provide a means for intercepting and rerouting messages at run-time. This permits functionality to be added or replaced when a dynamic event occurs. The original destination for a message must exist, however, in order for the message to be sent, even if it will ultimately be intercepted. Thus, the composition filters approach requires a globally-consistent frame of reference.

Adaptive object-oriented programming (typically shortened to "adaptive programming") [17] permits modules to possess pattern-based knowledge of the class hierarchy in a system. Such patterns are used to specify where in the system to locate information of interest, while stating as little as possible about the system. Adaptive programming attempts to specify all support for crosscutting concerns in terms of traversals of the class hierarchy; that this is an effective and comprehensible means of such specification is unclear. It does not provide support for inconsistency in local world views.

Subject-oriented programming [12] allows separate class hierarchies (called *subjects*) to be composed; messages sent to an object from within one of these subjects can cause the invocation of behaviour within other subjects. However, subjects are heavyweight

modules. Each must define a self-consistent and complete world view, rather than the sorts of fragments available to implicit context.

Theme/UML (earlier called subject-oriented design) [4, 5] builds on the ideas of subject-oriented programming and generic programming. It is constrained to the sorts of limited inconsistency permitted by subject-oriented programming and generic programming. Recently, an implementation mechanism has been introduced that provides Theme/UML-like support; it possesses corresponding limitations in tolerating inconsistency [18].

Hyperspaces [24] are an extension to the ideas of AOP in general, and of subject-oriented programming in particular. They are intended to permit "multidimensional separation of concerns." The idea centres around the proposition that no single modularization exists that will suit all purposes. Instead, sometimes one will desire a functional view of the system and other times one will desire a class-based view, for example. Hyperspaces have been partially realized in the Hyper/J tool: portions of existing systems (i.e., individual Java classes or methods) can be removed and composed together independent from the remainder. However, those pieces have to have been compiled in the first place, so Hyper/J does not support program fragments with independent world views in the manner of implicit context; thus, it remains a relatively heavyweight solution.

Our previous work on implicit context [26] failed to recognize the significance of independent world views. We had not constructed the model of local execution and execution traces, nor had we made any connection between independent world views and communication history. As a result, communication history had the appearance of being arbitrary in purpose and function. Details of this model appear elsewhere [25].

6. CONCLUSION

Components that are developed independently, without benefit of coordinated pre-planning, tend to be mismatched in terms of the external modules, interfaces, and protocols they expect of the system around them. Rather than trying to eliminate such mismatch a priori, we wish to cope with it at composition time through the support of inconsistent world views. We have demonstrated the efficacy of such support through a motivational example involving the Abstract Factory design pattern.

Implicit context permits each module to possess a localized, inconsistent view of its context of operation. The differences in these views must be reconciled when communication occurs between the views. Implicit context provides the mechanism of contextual dispatch to intercept communication as it travels from one world view to another. This communication can then be altered, replaced, or discarded according to the situation; in the meantime, the modules involved in the communication remain oblivious as the translation occurs transparently. While implicit context is related to aspect-oriented approaches, none of these provides support for inconsistent world views.

7. REFERENCES

- [1] M. Akşit, L. Bergmans, and S. Vural. An object-oriented language–database integration model: The composition-filters approach. In *ECOOP '92: European Conference on Object-Oriented Programming*, volume 615 of *Lecture Notes in Computer Science*, pages 372–395, 1992.
- [2] W. Bartussek and D. L. Parnas. Using assertions about traces to write abstract specifications for software modules. In *Information Systems Methodology*, volume 65 of *Lecture Notes in*

- Computer Science*, pages 211–236, 1978.
- [3] M. Broy and K. Stølen. *Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement*. Springer-Verlag, 2001.
 - [4] S. Clarke, W. Harrison, H. Ossher, and P. Tarr. Subject-oriented design: Towards improved alignment of requirements, design and code. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 325–339, 1999.
 - [5] S. Clarke and R. J. Walker. Composition patterns: An approach to designing reusable aspects. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 5–14, 2001.
 - [6] R. DeLine. Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering*, 27(2):124–143, 2001.
 - [7] M. S. Feather. Reuse in the context of a transformation-based methodology. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, volume 1: Concepts and Models, pages 337–359. Addison–Wesley, 1989.
 - [8] R. E. Filman and K. Havelund. Source-code instrumentation and quantification of events. In *Proceedings of Foundations of Aspect-Oriented Languages, Workshop at AOSD 2002*, pages 1–8, 2002.
 - [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison–Wesley, 1994.
 - [10] D. Garlan, R. Allen, and J. Ockerbloom. Architectural mismatch or why it’s hard to build systems out of existing parts. In *Proceedings: 17th International Conference on Software Engineering*, pages 179–185, 1995.
 - [11] J. Hannemann and G. Kiczales. Design pattern implementations in Java and AspectJ. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173, 2002.
 - [12] W. Harrison and H. Ossher. Subject-oriented programming: A critique of pure objects. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, 1993.
 - [13] D. Hoffman and P. Strooper. State abstraction and modular software development. In *SIGSOFT’95: Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 53–61, 1995.
 - [14] U. Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP ’93—Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 36–56, 1993.
 - [15] R. Janicki and E. Sekerinski. Foundations of the trace assertion method of module interface specification. *IEEE Transactions on Software Engineering*, 27(7):577–598, 2001.
 - [16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP 2001—Object-Oriented Programming*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353, 2001.
 - [17] K. J. Lieberherr, I. Silva-Lepe, and C. Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, 1994.
 - [18] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 52–67, 2002.
 - [19] D. R. Musser. Introspective sorting and selection algorithms. *Software—Practice and Experience*, 27(8):983–993, 1997.
 - [20] D. R. Musser and G. V. Nishanov. A fast generic sequence matching algorithm. Technical Report 97-11, Computer Science Department, Rensselaer Polytechnic Institute, 1997.
 - [21] A. Ran. Software isn’t built from Lego blocks. In *Proceedings of the Fifth Symposium on Software Reusability*, pages 164–169, 1999.
 - [22] Y. Smaragdakis and D. Batory. Mixin layers: An object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology*, 11(2):215–255, 2002.
 - [23] A. Stepanov and M. Lee. The Standard Template Library. Technical report, Hewlett–Packard Company, 1995.
 - [24] P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton, Jr. *N* degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 107–119, 1999.
 - [25] R. J. Walker. *Essential Software Structure through Implicit Context*. PhD thesis, Department of Computer Science, University of British Columbia, February 2003. In submission.
 - [26] R. J. Walker and G. C. Murphy. Implicit context: Easing software evolution and reuse. In *Proceedings of the ACM SIGSOFT Eighth International Symposium on the Foundations of Software Engineering*, pages 69–78, 2000.
 - [27] M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. In *Proceedings of Foundations of Aspect-Oriented Languages, Workshop at AOSD 2002*, pages 1–8, 2002.
 - [28] D. M. Yellin and R. E. Strom. Protocol specifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.