On the Separation of Concerns in Program Families

Adrian Colyer[†], Awais Rashid[‡], Gordon Blair[‡]

[†]IBM UK Limited, MP 146, Hursley Park, Winchester, England
adrian_colyer@uk.ibm.com

[‡]Computing Department, Lancaster University, Bailrigg, Lancaster, England
{marash | gordon} @comp.lancs.ac.uk

Computing Department, Lancaster University, Technical Report Number: COMP-001-2004

ABSTRACT

Aspect-orientation can help to separate concerns in software. One of the goals of this separation is to promote flexibility and configurability; this is especially true when constructing program families (and product-lines). This paper introduces a set of principles that instruct in the creation of flexible, configurable, aspect-oriented systems. We illustrate the principles through their application to a software product-line. The principle of dependency alignment serves as a guideline for structuring concern implementation in modules, eliminating unwarranted dependencies between concerns. The principles of orthogonal and weakly orthogonal aspects instruct in the design of aspects that are included in some system configurations, but not in others. We show how these principles scale to larger systems and larger concern implementations.

1. INTRODUCTION

Flexibility and configurability are important properties of any software system expected to evolve over time, and these attributes are highly prized in recent methodologies such as Agile Development [1] and Extreme Programming [2]. Aspect-oriented software development (AOSD) meanwhile, is about the separation of concerns in software systems, and the encapsulation of those separated concerns in modules. In particular, the main thrust of AOSD is in separating and modularizing crosscutting concerns – whose implementation cannot be neatly modularized using object-orientation alone.

The motivating problem driving the work presented in this paper is the creation of enhanced flexibility and configurability in software systems through the application of AOSD techniques. If configurability is used to control the inclusion (or exclusion) of certain features, then we can view the different variants of the system that result as members of a program family.

The first author undertook extensive refactoring of components from an IBM® middleware product-line, using aspect-oriented software development tools. This was done in part to assess the ability of AOSD to extend the set of potential program family members [3] in the product-line, by enabling the creation of variants based on (or excluding) the newly separated concerns. A variety of concerns were separated using AspectJ, including both homogeneous crosscutting concerns and heterogeneous ones. By homogeneous crosscutting concern, we mean a concern in which the same or very similar behavior needs to occur at multiple points in the control flow of a software system. Examples of homogenous concerns investigated include tracing and logging, first-failure data capture, and performance monitoring instrumentation [4, 5]. By heterogeneous crosscutting concern, we mean a concern that impacts multiple points in a software system, but where the behavior that needs to occur at each of those points is different. An example of a heterogeneous concern is the support for a use-case that spans multiple modules within a software system [6].

When refactoring in this way to create flexibility, we have to address two important questions:

1. What module breakdown will result in the most flexible and configurable system, and
2. How does one know whether a concern (or feature) implementation can be safely added to, or removed from, the software system?

The rest of this paper provides some insights into the answers to these two questions. In section 2 we address the relationship between a concern and the artifacts produced in the software construction phase. Section 3 discusses the reduction in flexibility caused by dependencies between concerns, and shows how aspect-orientation can help to better align or even eliminate dependencies. We introduce the principle of dependency alignment, which serves as a guideline when decomposing a system into modules. Section 4 focuses on individual aspects, describing the design criteria that must be followed if an aspect is to be included in some system configurations, but not in others. The principles of orthogonal and weakly orthogonal aspects are presented. Section 5 demonstrates the application of these principles to larger systems and concerns, giving the design criteria for any concern implementation that constitutes a configurable option in a software system. If a micro-kernel architecture is used, then this will include all concerns outside of the kernel. In section 6 we show how the principles apply to the creation of software product-lines.

Whilst these principles are drawn from work on a large middleware system, and many of the examples are hence from the middleware domain, we believe that the results are generally applicable to the development of any aspect-oriented system seeking to attain the attributes of flexibility and configurability.

2. WHAT IS A CONCERN?

Ossher and Tarr define the separation of concerns as "the ability to identify, encapsulate and manipulate only those parts of software that are relevant to a particular concept, goal, or purpose" [7]. A concern can therefore be considered "the part of a software system relevant to a particular concept, goal, or purpose." In this paper, we limit ourselves to the software construction phase and will consequently use a more specific definition of a concern. We consider the embodiment of a concern to be the collection of all types and type members relevant to a particular concept, goal, or purpose addressed by the software, coupled with any input or output data sets (or parts thereof) relevant to that same concept.

When working with existing software systems, the first step is concern modeling [8]. Concerns are *identified*, and we can *manipulate* them (given appropriate tools) – but they may not necessarily be *encapsulated*. The second step is to encapsulate and separate the concerns lexically too – using for example a combination of regular object-oriented techniques, and aspect-oriented ones such as aspects in AspectJ [9] or Java™ classes composed by the CME's composition engine [10]. In order to achieve flexibility in system composition and evolution, and especially if the intention is to add or remove concerns as features in a program family, then lexical encapsulation is necessary but not sufficient. It is also required to consider the dependency relationships between concerns.

3. THE PRINCIPLE OF DEPENDENCY ALIGNMENT

Concern A is considered dependent on concern B if there is a build time or runtime (uses) dependency between them [11]. Dependency relationships between concerns seriously compromise system flexibility, and the ability to create configuration options. Consider a system with n features: a perfectly flexible architecture would permit the creation of $p(n)$ variants, where $p(n) = 2^{n-1}$. Adding even a single dependency, such that feature a can only be included when feature b is also present reduces the number of valid program variants by 25%. Table 1 illustrates this effect. The flexibility coefficient f, where $0 <= f <= 1$, is a measure of the flexibility in the system afforded by the separation of its concerns. It is calculated as the ratio of the number of program family members that can be built, to the theoretical maximum number of members that could be built if the concerns were perfectly separated.

AOSD helps to reduce dependencies between concerns through the inversion of control. Suppose there are two concerns A and B. B legitimately depends on A, and in addition the requirements of concern B are such that it needs to trigger some additional behavior at well-defined points in the control flow of concern A. In an object-oriented solution, calls to B would need to be inserted at the appropriate points in A, creating a dependency from A to B that exists solely for the benefit of concern B. Aspect-orientation allows for the reversal of this dependency inversion, concern B itself can state which events in A it is interested in.

Note that an alternative object-oriented solution would be to introduce an observer interface on which both concerns A and B depend. This solution requires pre-planning, adds complexity in concern A to track observers (which exists solely for the benefit of concern B), and obscures program intent if used too widely.

Table 1 System flexibility with a single dependency

n	$p(n)$	#eliminated variants	f
1	1	n/a	n/a
2	3	1	0.667
3	7	2	0.714
4	15	4	0.733
5	31	8	0.742
6	63	16	0.746
7	127	32	0.748
8	255	64	0.749
9	511	128	0.750
10	1023	256	0.750

The presence of a 'uses' relationship between two concerns, in which the direction of the 'uses' relationship is not aligned with the requirements of the using party, is a 'smell' [12] indicating some (aspect-oriented) refactoring may be advisable. Our analysis leads us to present the following general guideline:

Principle of Dependency Alignment:

Any ‘uses’ relationship from concern A to concern B should be in direct support of a feature or function that is part of concern A.

Many object-oriented systems are littered with violations of this guideline. Listing 1 is based on an excerpt from an Enterprise JavaBean™ container in the middleware product-line:

Listing 1 Entity Bean Passivation

```
01 try {
02   if (!removed)entityBean.ejbPassivate();
03   setState(POOLED);
04 } catch( RemoteException ex) {
05   FFDCEngine.processException(ex,
06      “EBean.passivate(),”237”,this);
07   destroy();
08   throw ex;
09 } finally {
10   if (!removed &&
11       statisticsCollector != null) {
12     statisticsCollector.
13         recordPassivation();
14   }
15   removed = false;
16   beanPool.put(this);
17   if (!Logger.isEnabled) {
18     Logger.exit(tc,”passivate”);
19   }
20 }
```

The core concern of entity bean passivation is tangled with logic for first failure data capture (lines 05-06), statistics gathering (lines 10-14) and logging (lines 17-19). This introduces a dependency from the passivation concern on the first failure data capture concern, which exists for the benefit of the first failure data capture concern. It creates a dependency from the passivation concern to the statistics gathering concern, which exists for the benefit of the statistics gathering concern. It also creates a dependency from the passivation concern to the logging concern, which exists for the benefit of the logging concern. All three of these dependencies are in violation of the dependency alignment principle. They prevent us from creating a product-line member that includes passivation, but does not include first failure data capture, statistics gathering and logging. Using aspect-orientation, we were able to remove these dependencies, creating a cleaner more comprehensible passivation implementation:

Listing 2 Refactored Entity Bean Passivation

```
try {
  if (!removed)entityBean.ejbPassivate();
  setState(POOLED);
} catch( RemoteException ex) {
  destroy();
  throw ex;
} finally {
  removed = false;
  beanPool.put(this);
}
```

The statistics gathering dependency was inverted so that the statistics gathering concern depended on the passivation concern in order to record statistics. The first failure data capture and logging dependencies were eliminated altogether in this instance since the aspect implementations were able to use pointcut expressions not tied into any details of passivation. By using the aspect-oriented approach, we greatly increased the flexibility in our product line.

4. ORTHOGONAL ASPECTS

In this section, we focus in on individual aspects, and the design guidelines that enable the inclusion or exclusion of an aspect to be made a configurable option in a software system.

Barbara Liskov's substitution principle [13] gives the conditions under which a derived class can be considered a genuine subtype of its super-type. The principle is stated as follows:

"*If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*"

The subtype S may override method behavior from the super-type T, and may introduce new fields, methods and constructors, but must do so in a way that honors the contract that T has with its clients. From the perspective of the clients of T, the potential effects of an aspect A on T are similar to those of a subclass: the aspect may override method or constructor behavior (through advice) and may introduce new fields, methods and constructors (through inter-type declarations). We divide aspects that affect T into two categories: those that are ever-present with respect to T, such that in any system including T they will be included also, and those that may be present in some systems including T, but not in others. Aspects whose inclusion is optional are sometimes known as *orthogonal* or *non-functional*. Orthogonal aspects need to obey a substitution principle that is a corollary of Liskov's:

Principle of orthogonal aspects:

"Let A(T) represent the behavior of T in the presence of aspect A.

If for each object o1 of type A(T), there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2, and

If for each object o1 of type T, there is an object o2 of type A(T) such that for all programs P defined in terms of A(T), the behavior of P is unchanged when o1 is substituted for o2,

then A is an orthogonal aspect with respect to T."

The first clause states that the behavior of all programs P defined in terms of T should be unchanged when the aspect A is present in the system. In other words, it is safe to add the aspect A to the system from the perspective of T. The second clause states that the behavior of all programs P defined in terms of A(T) should be unchanged when the aspect A is not present in the system. In other words, it is safe to remove the aspect A from the system from the perspective of T. The aspect A is restricted to introducing behavioral changes outside of the specification of T. For example, if the specification of T makes no statement about tracing, adding an aspect that traces the execution of methods in T is perfectly acceptable. Adding an aspect that strengthens the pre-condition for a method in T (for example, by requiring a non-null parameter value) is not acceptable as it violates the first clause. In Liskov substitutability, a subtype of T is allowed to weaken the pre-conditions and strengthen the post-conditions of any of T's methods (for example, handling a null-parameter value where previously it was not accepted). Note that the orthogonal aspects principle does not permit A(T) even this freedom: to do so would break the second clause and make it unsafe to remove the aspect from the system.

Consider the class T and aspect A in Listing 3:

Listing 3 Orthogonal Aspect?

```
public class T implements Serializable {
  private int t;
  public T() {…}
  public void foo() {…}
}

```

```
public aspect A {
  private int T.a;

  after(T t) returning() :
     execution(T.new(..)) && this(t) {
     t.a = …;
  }
  …
}
```

At first glance, A appears to be orthogonal to T: it declares only private state, and does not affect the pre- or post-conditions of any of T's methods or constructors. However, this example demonstrates one of the most subtle ways of violating the orthogonal aspect principle. Since T is declared serializable, the serialized form of T becomes part of its external interface. The declaration of the private field T.a in A causes the serialVersionUID (a constant that the Java runtime calculates to ensure version consistency of serialized objects) to alter. The consequence is that A is not orthogonal to T, as objects written by a program built in terms of T cannot be read by a program built in terms of A(T) and vice-versa. The point of course is that when considering orthogonality, the full visible behavior of the system must be taken into account. The aspect and class in this example were deliberately chosen to illustrate a worst case scenario. In AspectJ v1.1.1 and prior, it is not possible to write an orthogonal aspect with respect to some type T, if the aspect makes inter-type declarations for T, and T uses the default serialization mechanism. A secondary lesson is that it is inadvisable to depend on the default serialization mechanism (see e.g. [14]), especially in product-line development, and certainly in aspect-oriented product-lines.

A very simple first-failure data capture aspect F, as shown in Listing 4, is orthogonal to T as long as the analyzeFailure() routine has no observable side-effects.

Listing 4

```
public aspect F {
   after() throwing(Throwable t) :
     execution(* T.*(..)) {
     FFDC.analyzeFailure(t);
   }
}
```

(Note, you wouldn't normally write such an aspect to explicitly mention T).

4.1 Weakly Orthogonal Aspects

In practice, it is often the case that a collection of types are grouped together in a concern and added or removed as a unit. In this situation, a more relaxed form of the orthogonal aspect principle is appropriate:

Principle of weak orthogonality:

"Let A(T) represent the behavior of T in the presence of aspect A.

If for each object o1 of type A(T), there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2,

then A is a weakly-orthogonal aspect with respect to T."

Any existing programs written to work with T should continue to work in the presence of the aspect A (it is safe to add A to the system from the perspective of T). In contrast to strong orthogonality, other types in the same concern as A may depend on the behavior of A(T). In other words, it is not safe to remove A from the system from the perspective of T, unless all the dependent types in the same concern as A are removed also.

In the example shown in Listing 5, A is weakly orthogonal to T. No client of T can be dependent on the foo() method that A declares, but clients of A(T) may be.

Listing 5 Weakly Orthogonal Aspect

```
public class T {}

public aspect A {
  public int T.foo() {…}
}
```

If we change the example slightly as in Listing 6, this is no longer the case:

Listing 6

```
public class S {
  public int foo() {…}
}

public class T extends S {}

public aspect A {
  public int T.foo() {…}
}
```

Clients of T may be dependent on the implementation of foo() that T inherits from S. In the presence of A(T) those same clients see a different implementation of foo() that may not be behaviorally equivalent.

5. LARGER CONCERNS

Some concerns can be fully implemented by a single aspect, but in our experience, many concerns are a mixture of classes and aspects. In the middleware product-line example from section 3, the full first failure data capture concern would contain one library aspect, plus one concrete sub-aspect for every component requiring first-failure data capture. In addition, there is a set of diagnostic modules implemented as Java classes, plus an analysis engine. This whole collection of types forms a concern that we may wish to make a configurable option in the system. Even a simple tracing and logging concern will contain some aspects together with a library such as Log4j [15].

If S represents the set of all types in all concerns in the software system, and C represents the set of types in one of those concerns, then the concern C can be safely added to or removed from the system if the following conditions hold:

1. There are no dependencies from the set of types in $S\backslash C$ to any of the types in C.
2. For all programs defined in terms of types in $S\backslash C$, for all aspects A in C, and for all types T in $S\backslash C$, A is orthogonal to T.
3. For all programs defined in terms of types in S, for all aspects A in C, and for all types T in $S\backslash C$, A is weakly orthogonal to T.

The first condition simply says that a concern cannot be removed if an element outside of the concern depends on it. The second condition says that programs defined outside of the concern must continue to work whether the concern is present or absent. The final condition says that all programs must continue to work in the presence of the concern, but that some programs may fail in its absence. Putting the second and third conditions together, we see that the set of programs that may fail are restricted to those that use types in C.

Sometimes concerns themselves are inter-dependent such that they represent a single configurable option. The conditions under which this can be done safely are the same as those given above, with the set of concerns in question treated as if they formed a single larger concern.

Whilst we modeled concerns in our system with a view to creating separable components, concerns were not necessarily disjoint: there were multiple occasions where the same types or type members belonged to more than one concern simultaneously. On the surface, it seems counter-intuitive that this should be so, but in practice it worked out very naturally. As an example, we defined an Enterprise JavaBeans™ (EJB) concern that captured everything concerned with supporting EJBs within an application server in the product-line. We also defined a "Debug" concern that captured everything concerned with supporting debugging of enterprise applications running in the application server. At the intersection of those two concerns was a class used to assist in the remote debugging of EJBs. This class (and some associated behavior that surrounded it) should only be included in a system that has both EJB and debugging features. Similar intersections occur between the debugging concern and the web container, and between the EJB concern and many other concerns in the system. Perhaps the concern model should be normalized to split out these intersection points into separate concerns, but we found it useful to be able to talk about the "EJB concern" and the "Debug concern" and have this mean everything associated with that particular concept. The requirement for any behavior that fits at such a concern intersection is that it be separable in both directions – for example, we may want to build a system that supports debugging but not EJBs, or that has EJB support but not debugging. These natural concern interactions mean that including support for a concern in a product variant is not simply a matter of including its files in a build, but instead needs to be done in consideration of which other concerns are also to be included. We used simple exclusion lists as a first pass solution to this in our ant-based [16] build system. We set build properties to indicate which concerns should be included in a build. Each build component was then associated with a primary concern, and could optionally contain one or more "*feature.<concern-name>*" files. Each of these feature files contained a list of files within the build component that should be excluded from the build unless the corresponding feature property was set. For example, the debug component contained a file "feature.ejb" that caused the EJB debugging classes to be excluded unless the EJB feature was enabled.

6. APPLICATION TO PRODUCT-LINES

Creating flexibility and configurability in a software system is a pre-requisite to using that system as the basis for a software product-line. Throughout this paper, the examples used to illustrate the principles are grounded in a middleware product-line. We used the product-line as the basis for a case study in improving these attributes of a system through the application of aspect-oriented techniques. A product-line member that adds a set of features to a base system must do so in such a way that those features are orthogonal to the base system. Figure 1 shows a simple product-line with two members: a base product-line member that provides features f1 and f2, and a product A that builds on the base to add features f3 – f6.

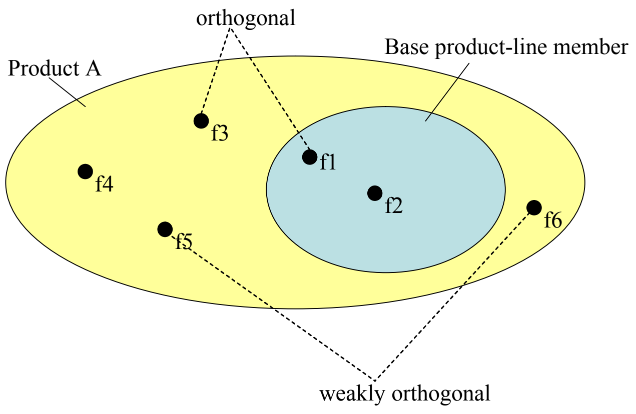

Figure 1 Product-Line with Two Members

Feature f3 must be orthogonal to f1 (f1 must continue to function correctly in the presence or absence of f3). Feature f5 must be weakly orthogonal to f6 (f6 must function correctly in the presence of f5).

Consider what happens when we add a new product B to the product-line, as shown in Figure 2. Product B builds on the base, and adds features f6 – f8. The addition of this product to the product-line changes the nature of the relationship between features f5 and f6. Since a product now exists that includes f6 but not f5, it is no longer enough that f5 be weakly orthogonal to f6. Instead f5 must be (fully) orthogonal to f6: f6 must function correctly in the presence *or absence* of f5. Since we can't predict accurately in advance which products we may ultimately want to build in the product-line (and which features will be included in those products), it is best to make all features orthogonal whenever possible.

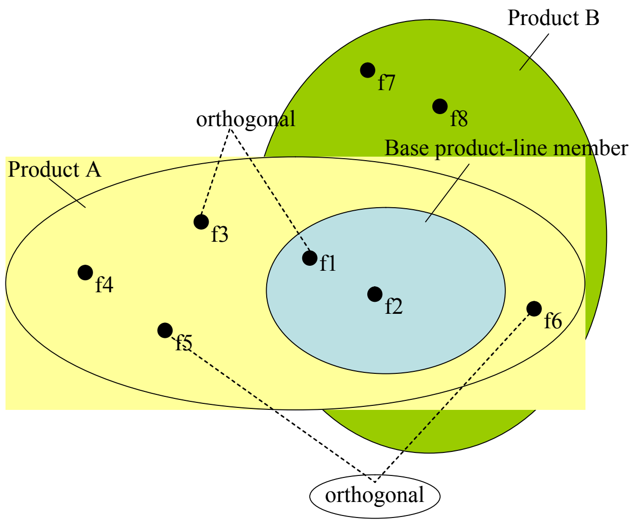

Figure 2 Product-Line with Three Members

7. RELATED WORK

In the Atlas system case study [17], Kersten and Murphy discuss different kinds of aspect-class association, and recommend that aspects be aware of classes and not vice-versa. Filman [18] introduces the concept of obliviousness (in programmers) to the presence of an aspect. Both of these ideas are related to the principle of dependency alignment, but whereas they give guidelines for the association between aspects and classes, the dependency alignment principal instructs on when concerns should be separated in the first place (whether by aspect-orientation or some other technique). The principles of orthogonal and weakly-orthogonal aspects give the conditions under which a programmer may be safely oblivious to an aspect (as opposed to ignorant of it). Nordberg [19] analyses principles for dependency management in object-oriented systems, and shows how aspect-oriented solutions can help better align dependencies in many common situations. The work is complimentary to that presented in this paper.

Lieberherr presents the Law of Demeter [20], which states that an operation in a class should only call operations on immediate subparts of the class, any arguments passed to the operation, and any objects created by the operation. Many but not all violations of the principle of dependency alignment also violate the Law of Demeter.

Hunleth et al. have studied the use of AspectJ for feature and footprint management in a CORBA event channel, using aspects to introduce features incrementally and as independently as possible [21, 22]. The relationships between features (concerns) were modeled so that only valid system combinations were produced (4,596 combinations or variants were meaningful out of a total possible set of nearly 8 million). To handle the situation where the addition of a feature may require parameters to be added to a method, they introduce a notion called the “Encapsulated Parameter Pattern.” Using this technique, all the parameters to a method are bundled into a class, allowing extension of the method parameter set by inter-type declarations for the parameter class.

Yvonne Coady and Gregor Kiczales undertook a retrospective refactoring on portions of the FreeBSD operating system using AspectC [23], and then replayed actual changes that had occurred in the system code to see how the aspect-oriented design fared under the same modifications. They found that with the aspect-oriented design changes were better encapsulated, redundancy was reduced, *the creation of alternate system configurations became easier, and extensibility aligned with an aspect was more modular*.

Neither of these works presents guidelines for constructing a program family using aspect-orientation in order to maximize flexibility.

Zhang and Jacobsen have undertaken aspect-oriented refactoring in a series of ORB implementations [24]. Their most recent work presents a set of principles for horizontal decomposition of software systems, in which features implemented using aspects are added to a minimal core [25]. This includes the ‘aspect-directional’ principle that states that the core should not be aware of any aspects that may be applied to it. Our work presents the conditions under which this can be true.

In the product-line literature, the Software Engineering Institute’s “Framework for Software Product Line Practice” [26] contains a section on architecture definition and discusses build-time parameters, inheritance and delegation (in object-oriented systems), and component replacement as means of achieving variation in a product-line. A more exhaustive treatment can be found in [27]. The SEI Product Line Framework makes mention of aspect-orientation as a possible implementation technique, but no guidance is given as to how it might best be applied. Griss [28] makes the case for feature-driven, aspect-oriented product-line development, but focuses only on the modularization of concerns that cut across features making no recommendations for the design of such concerns.

Don Batory’s work on Feature-Oriented Programming and the GenVoca design methodology [29] for creating application families and architecturally extensible software is very close to some of the concepts in aspect-orientation. GenVoca uses a process of refinement to add features to a program, and equations are used to define program family members. A feature of the work is in the specification of a simple algebraic model for composition. The approach has been applied to generating product-lines of product families [30], and scaled to apply to artifacts other than code [31].

Other groups have studied the problem of flexible architectures for middleware program families, but do not explicitly consider aspect-orientation in their work. Example projects are Flexinet [32] and FlexiBind [33], the ArticBeans project [34], OpenCOM [35] and Open-ORB [36].

The ACE ORB (TAO) [37, 38] is an extensible and configurable ORB that makes use of the strategy pattern to separate out aspects of the internal ORB engine. A configuration file is read at start-up time to determine which strategies to load. Research at the University of Illinois has extended the component configuration capabilities of TAO over a series of projects. DynamicTAO added on-the-fly reconfiguration of strategies via a "ComponentConfigurator" framework [39, 40]. Dependencies between components are explicitly represented, and the representation can be inspected and adapted to enable implementation of components that can configure themselves and adapt to changing conditions. Dependencies are divided into prerequisite (or 'requires') load-time dependencies and dynamic dependencies between loaded components in a running system.

8. SUMMARY

Flexibility and configurability are important attributes of any large software system expected to evolve over time. This is especially true when constructing program families or product-lines.

We showed how dependencies between software parts hamper flexibility by restricting the number of valid combinations that can be put together. A particularly unpleasant type of dependency that arises in object-oriented systems is an inverted dependency that causes a part to depend on a third-party, *for the exclusive benefit of the third-party*. The dependency alignment principle states that a part should only depend on other parts in direct fulfillment of its own requirements. Violations of the dependency alignment principle are commonplace in object-oriented software. We believe this is because object-oriented techniques lack a mechanism for implicit flow of control. Aspect-oriented techniques provide for implicit control flow through the combination of advice, pointcuts, and a joinpoint model. We demonstrated that aspect-orientation can be used to remove inverted dependencies from a software system, and in so doing also improved the clarity of the source code from which the dependencies were removed.

With the dependency graph straightened out, there remains the question of whether the inclusion or exclusion of a part can be made a configurable option in the system, without causing unwanted side-effects. The principles of orthogonal and weakly orthogonal aspects address this question with respect to the addition (or removal) of a single aspect in a system. As with Liskov's substitution principle, we showed that an answer to the question cannot be found by studying an aspect in isolation, but depends on the impact an aspect has on the observable behavior of programs to which it is introduced. We presented examples where the same aspect either obeyed or did not obey the principles, based on small changes to the program. This has implications for the principle of obliviousness when designing aspects for use in some system configurations but not others.

The implementation of concerns (the part of a software system relating to some feature, goal, or purpose) often involves not just a single aspect, but a whole collection of types. The principles of orthogonal aspects can be applied to derive the conditions under which such concern implementations can be included in (or excluded from) program family members.

In our future work we plan to apply these principles to the construction of an aspect-oriented middleware platform with a micro-kernel architecture, which will provide opportunity for further evaluation and assessment.

9. ACKNOWLEDGMENTS

IBM is a trademark or registered trademark of International Business Machines Corporation in the United States, other countries, or both.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc., in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

10. REFERENCES

[1] A. Cockburn, *Agile Software Development: Software Through People*: Addison-Wesley, 2002.

[2] K. Beck, *EXtreme Programming EXplained*: Addison-Wesley, 1999.

[3] D. L. Parnas, "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, vol. SE-2, 1976.

[4] R. Bodkin, A. Colyer, and J. Hugunin, "Applying AOP for Middleware Platform Independence," *Practitioner Reports, 2nd International Conference on AOSD*, 2003.

[5] A. Colyer, A. Clement, R. Bodkin, and J. Hugunin, "Using AspectJ for Component Integration in Middleware," *Practitioner Report, OOPSLA 2003*, 2003.

[6] I. Jacobson, "Use Cases and Aspects - Working Seamlessly Together," *Journal of Object Technology*, 2003.

[7] H. Ossher and P. Tarr, "Using Multidimensional Separation of Concerns to (re)shape Evolving Software," *Communications of the ACM*, vol. 44, pp. 43-49, 2001.

[8] S. M. Sutton and I. Rouvello, "Modelling of Software Concerns in Cosmos," presented at 1st International Conference on Aspect-Oriented Software Development, Enschede, The Netherlands, 2002.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "Getting Started with AspectJ," *Comm. ACM*, vol. 44, pp. 59--65, 2001.

[10] H. Ossher, P. Tarr, and W. Harrison, "Concern Manipulation Environment (CME): A Flexible, Extensible, Interoperable Environment for AOSD." http://www.research.ibm.com/cme/, 2003.

[11] D. L. Parnas, "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, vol. 5, pp. 128-38, 1978.

[12] M. Fowler, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

[13] B. Liskov, "Data Abstraction and Hierarchy," *SIGPLAN Notices*, vol. 23, 1988.

[14] J. Bloch, *Effective Java - programming language guide*: Addison-Wesley, 2001.

[15] Apache Software Foundation, "Log4j." http://jakarta.apache.org/log4j/docs/.

[16] "Apache Ant." http://jakarta.apache.org/ant: The Apache Jakarta Project.

[17] M. Kersten and G. C. Murphy, "Atlas: A case study in building a web-based learning environment using aspect-oriented programming," *Proceedings of the 7th Conference on Object-Oriented Programming, Systems, Languages, and Applications*, vol. 34, pp. 340-352, 1999.

[18] R. Filman and D. Friedman, "Aspect-Oriented Programming is Quantification and Obliviousness," *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.

[19] M. E. Nordberg, "Aspect-Oriented Dependency Inversion," *Workshop on Advanced Separation of Concerns, OOPSLA 2001*, 2001.

[20] K. Lieberherr, I. Holland, and A. J. Riel, "Object-oriented programming: An objective sense of style.," *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, vol. 11, pp. 323-334, 1988.

[21] F. Hunleth and R. Cytron, "Footprint and Feature Management Using Aspect Oriented Programming Techniques," presented at LCTES 02, Berlin, Germany, 2002.

[22] F. Hunleth, R. Cytron, and C. Gill, "Building Customizable Middleware using Aspect Oriented Programming," presented at OOPSLA 2001 Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa, Florida, 2001.

[23] Y. Coady and G. Kiczales, "Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code," *Proceedings 2nd International Conference on Aspect-Oriented Software Development*, pp. 50-59, 2003.

[24] C. Zhang and H.-A. Jacobsen, "Quantifying Aspects in Middleware Platforms," *Proceedings 2nd International Conference on Aspect-Oriented Software Development*, pp. 130-139, 2003.

[25] C. Zhang and H.-A. Jacobsen, "Resolving Feature Convolution using Horizontal Decomposition in Middleware Systems," *University of Toronto, Computer Systems Research Group Technical Report Nr: 475*, 2003.

[26] L. Northrop, "A Framework for Software Product Line Practice." http://www.sei.cmu.edu/plp/framework.html: Software Engineering Institute, 2003.

[27] M. Svahnberg and J. Bosch, "Issues Concerning Variability in Software Product Lines," *Proceedings of the Third International Workshop on Software Architectures for Product Families*, pp. 50-60, 2000.

[28] M. L. Griss, "Implementing Product-Line Features by Composing Component Aspects," *Proceedings of the First Software Product Line Conference*, pp. 271-288, 2000.

[29] D. Batory and B. J. Geraci, "Composition Validation and Subjectivity in GenVoca Generators," *IEEE Transactions on Software Engineering*, pp. 67-82, Feb. 1997.

[30] D. Batory, R. E. Lopez-Herrejon, and P. M. Martin, "Generating Product-Lines of Product-Families," *Proceedings 2002 Automated Software Engineering Conference*, pp. 81-92, 2002.

[31] D. Batory and J. N. Sarvela, "Scaling Step-Wise Refinement," *International Conference on Software Engineering (to appear)*, 2003.

[32] R. Hayton, "FlexiNet Architecture Report," ANSA Phase III report 1999.

[33] O. Hanssen and F. Eliassen, "A Framework for Policy Bindings," *Proceedings of the International Symposium in Distributed Objects and Applications (DOA'99)*, 1999.

[34] A. Anderson, G. Blair, V. Goebel, R. Karlsen, T. Stabell-Kulo, and W. Yu, "Arctic Beans: Configurable and Reconfigurable Enterprise Component Architectures," *IEEE Distributed Systems Online*, vol. 2, 2001.

[35] G. Coulson, G. Blair, M. Clarke, and N. Parlavantzas, "The design of a configurable and reconfigurable middleware platform," *Distributed Computing*, pp. 109-126, 2002.

[36] G. Blair, G. Coulson, P. Robin, and M. Papathomas, "An Architecture for Next Generation Middleware," *Proceedings IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing: Middleware '98*, pp. 191-206, 1998.

[37] D. Schmidt, "The ADAPTIVE Communication Environment: An Object-Oriented Network Programming Toolkit for Developing Communication Software," presented at 12th Annual Sun Users Group Conference, San Francisco, CA, 1994.

[38] D. Schmidt and C. Cleeland, "Applying a Pattern Language to Develop Extensible ORB Middleware," *IEEE Communications Magazine*, vol. 37, 1999.

[39] F. Kon and R. H. Campbell, "Dependence Management in Component-Based Distributed Systems," *IEEE Concurrency*, vol. 8, pp. 22-36, 2000.

[40] F. Kon and R. H. Campbell, "Supporting Automatic Configuration of Component-Based Distributed Systems," *5th USENIX Conference on Object-Oriented Technologies and Systems*, pp. 175-187, 1999.