Interface-Oriented Programming

L. Robert Varney

varney@cs.ucla.edu

University of California, Los Angeles
Computer Science Department Technical Report
TR-040016

March 29, 2004
Revised: September 17, 2004

Abstract

The predominant design of object-oriented programming languages today provides insufficient support for interface abstraction and implementation inheritance, a limitation that forces software components to be unnecessarily biased towards particular implementations of other components. This scatters implementation bias across a system and impairs its ability to evolve.

In this paper we trace the origin of implementation bias to concrete instantiation dependencies, points in a program text where objects are explicitly or implicitly instantiated in terms of class name references. We then propose interface-oriented programming as a solution, a new approach to language design and software engineering that can eliminate implementation bias by strictly separating interfaces from implementations and decoupling the inheritance mechanism from the implementation binding mechanism.

In interface-oriented programming, all program dependencies are expressed through abstract interfaces, and interface clients are bound with implementations using non-deterministic instantiation, a mechanism that automatically selects an implementation of an interface from among a set of alternatives. Partial and incremental implementation of abstract interfaces in terms of other assumed interfaces is allowed, providing a form of mixin-inheritance that separates interface inheritance from implementation inheritance, but with support for constructors, and without the need for explicit composition. Composition of these separately compiled, partial implementations is handled by a new form of program linking called representaton inference. The representation inference mechanism uses type constraints and method constraints to infer compatible compositions of available partial representations for an interface.

Contents

1 Introduction

Separation of concerns [Par72] is a basic strategy of software engineering, a way to isolate design decisions and engineer software for ease of reuse and maintenance. Software development practices that support this strategy, such as data abstraction [Gut77] and object-oriented programming [Str87b], have equipped software developers with powerful tools for modular design. Despite these advances, large-scale software evolution and reuse remain problematic. Waldo [Wal98] claims that the state of software reuse isn't as bad as it looks, however, observing that programmers often take the rich set of libraries they depend on for granted. But he also asserts that we are usually forced to reuse general-purpose code unchanged, and even in stable, general-purpose domains it is difficult to mix and match code from multiple suppliers. As Ran states [Ran99], software still isn't built from Lego blocks, and Szyperski's vision of component software [Szy98] remains unrealized.

Biggerstaff [Big94] has argued that the source of the problem is the concreteness of components as expressed in conventional programming languages, a problem we refer to here as *implementation bias*. As the size of components in terms of implemented features increases, the number of components needed in the library to provide all combinations of features explodes. Also, the likelihood that any particular feature combination will be useful diminishes as component size increases. Attempting to concurrently address both of these problems demands several implementations for each abstraction, and each implementation becomes a monolithic, hand-optimized combination of features. This problem, called the feature combinatorics problem by Batory, et al. [BSST93] fundamentally complicates library development given the current technology of object-oriented software development. Biggerstaff, Batory, and others have suggested that a different approach is needed, one that generates components from lower-level building blocks. But the solution is not to stock libraries with prefabricated components covering all forseen feature combinations, the solution is to factor abstractions and features into recomposable precursor components, and generate components with specific feature combinations as needed.

Finding the right way to express such building blocks has been the subject of much research, and numerous strategies for controlling implementation bias and separating concerns have been created. Proposals include abstract factories and exemplars ([Cop92], [LS94], [Rie95], [GHJV95]); service facilities [SWB02]; parameterized components ([BO92], [BG96], [BLS03]); mixins and mixin layers ([BC90], [SB98], [FKF98], [ALZ00], [CL01], [CBML02],[Sre02]); subject-oriented programming ([HO93],[OKH+95]); multi-dimensional separation of concerns and hyperspaces ([TOHS99],[TO00],[OT00]); features, families, collaborations, traits, and teams ([Pre97], [Ern01], [MO02], [MO03], [SDNB03], [Her03], [VH03]); generative programming ([CE99], [CE00]); and aspect-oriented programming ([KLM+97], [KHH+01]).

Yet existing approaches for controlling implementation bias only partially address the problem, tending to move implementation dependencies around instead of eliminating them. In particular, abstract factories replace direct instantiation dependencies with indirect ones, yielding dependencies on factory implementations. Similarly, parameterized components and mixins introduce parameters for abstract base classes or contained objects, shifting the responsibility to correctly instantiate these objects to the supplier of the actual arguments.

And available methods for design decomposition are also insufficient, physically grouping entities that are logically separable, and often creating new problems when separated concerns are reintegrated. The nesting of collaboration-oriented designs forces designers to co-locate specifications of independent concepts, whereas separately specified partial elements must somehow be recombined to create consistent wholes. Some combination methods require that factors be composed manually and then checked for consistency, and other methods provide automated support for composition but without any guarantees about what the compositions mean. An example of the latter approach is aspect-oriented programming (AOP). While AOP

improves design factorization, aspect weaving can violate encapsulation and understanding how composed aspects interact is challenging [Ald04].

As a result of these limitations, the methods available today present us with an unfortunate tradeoff in component design. On the one hand we can choose to encapsulate implementation choices in a component for the sake of interface clarity, possibly sacrificing some flexibility and performance. Or we can expose implementation choices through interface parameters to retain flexibility, accepting less interface clarity and expecting clients to decide what implementations to use. Despite the apparent flexibility of the latter approach, the client must still select consistent sets of values for parameters and has no option but to embed this choice somewhere. Furthermore, choosing to expose certain interface parameters and not to expose others implicitly limits the set of possible implementations of that interface. It appears that no matter what approach we take, ultimately we are forced to bury implementation choices in code that logically should depend on something more abstract. This practice spreads from one component to another like a contagious disease, an infectious scattering of inappropriately concrete implementation choices that impairs our ability to comprehend, evolve, and reuse the system and its parts.

We contend that the root of this dilemma is weak separation of interfaces from implementations in the underlying programming language, and that this coupling reduces the usefulness of other incremental programming techniques such as composition and inheritance. We also maintain that existing techniques for incremental programming still force too much design information to be expressed in one place, and that any approach for separation of concerns must also provide automatic support for safe and consistent integration of concerns. A key element of the solution to all of these problems is interface abstraction.

An effective and scalable solution, one that solves problems and doesn't just move them around, requires that all program dependencies be defined in a strictly interface-oriented fashion, a seemingly radical practice that is impossible given the programming languages available today. But abstract interfaces should be mandatory and ubiquitous, not optional constructs usable only in certain locations. A realization of this principle would also demand that mechanisms for incremental programming, such as inheritance, be decoupled from implementation binding. Most striking of all, perhaps, is that a strictly interface-oriented approach forces programmers to delegate to some other agent in the system the responsibility for two tasks normally in their purview, the tasks of (1) composing complete implementations from incrementally defined ones, and (2) selecting suitable complete implementations of an interface from among several alternatives.

In this report we present a new model of program development called *interface-oriented programming* (IOP) that adheres to these principles and resolves the dilemma we have described. In IOP, all program interdependencies are expressed in terms of abstract interfaces, separating the client of an interface from its implementation. Partial implementation of interfaces as well as incremental specialization and adaptation via interface-oriented inheritance are encouraged, separating independent segments of an implementation from each other (be they portions of a single object or participants in a collaboration). Assembly of complete components from partial ones is automatic and safe, based on locally-defined, interface-oriented type constraints. And rather than try to check the consistency of manually crafted compositions of components, IOP uses a new form of program linking called representation inference to automatically generate a set of candidate component assemblies from available parts based on their local constraints, and employs interface-oriented, non-deterministic instantiation to select whole assemblies when needed.

Although other researchers (e.g., Snyder [Sny86]) have called for the separate treatment of interface and implementation, and for the separate handling of subtyping and subclassing, no programming language in widespread use today separates these issues sufficiently to solve the information distribution problems we've described and to adequately support evolution. The separate treatment of abstract interfaces as first-class entities in some of the latest programming languages (e.g., Java [AGH00] and C# [ADM01]) is superficial, as these languages merge the interface and implementation hierarchies and all too often force programmers

to identify concrete implementations by name in places where weaker, more abstract dependencies should suffice. In addition to presenting IOP we also describe an experimental programming language called ARC that implements the concepts of IOP and accomplishes a strict separation of interface and implementation that is lacking in all programming languages we are aware of.

The remainder of this report is organized as follows. Section 2 characterizes the problems of implementation bias and non-incremental expression in the context of established methods for object-oriented design, clarifying the motivation for our solution. Section 3 proposes a new strategy and a set of principles we adopt for interface-oriented program design, and presents the interface-oriented constructs of the ARC programming language, the underlying mechanisms needed to support them, and examples of their use. We review related work in section 4, discuss the status and future directions of our research in section 5, and conclude in section 6.

2 The Problem of Implementation Bias

Reusability of a software component is strongly determined by its implementation bias, the degree to which the expression of the component's implementation contains information not logically implied by the intent of its design. When a component embeds excessive implementation dependence it becomes difficult to reuse in new contexts. This bias can be reduced by interface parameterization, but cluttering interfaces with parameters just forces the issue onto the client.

In this section we explore the phenomenon of implementation bias, a phenomenon that appears to occur whenever a design is expressed in one of the mainstream programming languages of today such as Java, C++, or C#. First, we show by example how various forms of implementation bias commonly occur and how existing programming techniques and language constructs are unable to adequately control it. (Our examples are written in a Java-like language, although occasionally we use extensions that are not strictly Java.) Second, we isolate the underlying cause of implementation dependence and enumerate a series of distinct problems that arise because of it.

2.1 Forms of Implementation Bias

The most basic form of bias in object-oriented languages occurs when class names are used directly instead of interfaces. Abstract classes in C++ were an early approximation to interfaces, and languages today such as Java and C# provide built-in interface constructs in addition to classes. So instead of just defining a class:

```
class A {
    void a1() { ... }
    ...
}
```

we can define a class which implements an interface:

```
interface A {
    void a1();
    ...
}
class A_impl implements A {
    void a1() { ... }
    ...
}
```

It might appear that implementaton bias can be avoided by avoiding class names, and using interfaces names instead. This would indeed insulate clients from underlying implementation classes, but to use an interface we first have to acquire an object that implements it. And, since interfaces cannot themselves be instantiated (and provide no means for declaration of constructors), creating an implementation requires naming a class:

```
void foo() {
    A a = new A_impl();
    ...
}
```

Although logically speaking, function `foo()` might not depend on anything more than `A`, we have nevertheless embedded the choice of `A_impl`. That choice cannot be changed without changing `foo()`, hindering its reusability.

To fix this we could encapsulate the choice behind a factory object that implements an abstract factory interface [GHJV95]. The factory object can then be used to create an object that implements `A`:

```
void foo() {
    AFactory aF = new AFactory_impl();
    A a = aF.create();
    ...
}
```

Here again, a choice of implementation must be made, in this case for the factory interface `AFactory`, and our choice is again embedded in `foo()`.

A natural reaction to this situation is to factor the implementation choice out of the body of the function and defer it to a parameter in the interface. In particular, we could add a parameter to `foo()`'s interface that can be bound externally, specifying either the implementation of `A`:

```
void foo( A a ) {
    ...
}
...
obj.foo( new A_impl() );
```

or the implementation of a factory for `A`:

```
void foo( AFactory aF ) {
    A a = aF.create();
}
...
obj.foo( new AFactory_impl() );
```

But adding such parameters merely moves the implementation dependency to the caller of `foo()`. And, not only is the caller of `foo()` burdened with the implementation decision, it is now aware of and coupled to `foo()`'s internal use of `A`. While factories can be useful if they replace multiple object instantiation decisions with one factory instantiation decision, the factories themselves must still be instantiated somewhere. Parameters[1] also bias interfaces and increase coupling between clients and implementations.

Other forms of implementation dependence arise when trying to incrementally define a component implementation in terms of other components, either via aggregation or some form of inheritance. When instantiating a compound object, for example, implementations must be specified for the aggregated object references, either embedded within or as arguments to the compound object's constructor. Similarly, when instantiating an object of a class derived by inheritance from one or more base classes, the constructor implicitly instantiates the base objects whose implementations are embedded by name in the inheritance relation.

In our next example we define a class that implements three abstract interfaces by aggregating implementations for two of them, `A` and `B`, and locally defining the implementation of the third interface, `C`:

[1] Global parameters do not bias the interfaces themselves but still force clients to make binding decisions.

```
interface A { ... }
interface B { ... }
interface C { ... }

interface AX extends A { ... }
interface BX extends B { ... }

class C_impl implements A, B, C {
    AX aPart;
    BX bPart;
    public C_impl( AX a, BX b ) {
        aPart = a;
        bPart = b;
    }
    public void aMethod() {
        aPart.aMethod(); // forward
    }
    public void bMethod() {
        bPart.bMethod(); // forward
    }
    public void cMethod() {
        ... // local implementation
    }
}
...
C_impl c = new C_impl( new AX_impl(), new BX_impl() );
```

By parameterizing the constructor we defer the selection of implementations for `A` and `B` to the caller of the `C_impl` constructor. Other than the dependence on `C_impl`, which we have already discussed, there are several additional problems with this approach. The first is that we have to make multiple implementations choices. The second has to do with the use of `AX` and `BX` in the constructor interface instead of `A` and `B`. In this situation the implementation of `C`, even though it implements `A` and `B` externally, uses more specialized interfaces internally, namely `AX` (which extends `A`) and `BX` (which extends `B`). However, since we do not want `C_impl` to depend on particular implementations of `AX` and `BX`, we have to use either embedded factories or parameters. In this case, the decision to use `AX` and `BX` is exposed in constructor parameters, forcing a dependency between clients of `C_impl` and an aspect of its internal design, an aspect that deserves to be hidden.

Without some form of mixin inheritance, aggregation with constructor parameterization is the only way to program incrementally in terms of abstract bases. But such compound objects also introduce their own problems, such as the indirection of forwarding, wrapper identity problems [HÖ93], and the resulting complications to method overriding. While implementation inheritance does not suffer from the identity and overriding problems, it accomplishes this by depending on specific base classes, and with single inheritance, only on one base class at a time.

Things get a little cleaner with mixin-inheritance [BC90], a form of incremental programming based on inheritance of abstract base classes. Mixins are not supported in Java, although various proposals for doing

so have been made (e.g., [FKF98], [ALZ00]). If mixin-inheritance were available we could avoid the identity and overriding problems while still programming incrementally in terms of abstract bases. A key distinction with traditional inheritance is that mixin-inheritance does not embed the name of an implementation class in the inheritance relation, the base is defined either as a named interface or an unnamed set of features assumed to be provided by some interface, and is bound with an implementation class when the mixin class is composed:

```
mixin class C_impl extends AX, BX, implements A, B, C { // NOT JAVA!
    public C_impl() {
        ...
    }
    ...
}
...
// Pretend "+" is mixin composition
C_impl c = new C_impl() + AX_impl() + BX_impl();
```

In this example the constructor for `C_impl` no longer appears parameterized, but it actually is, since we cannot create a `C_impl` without composing it with other mixins that resolve its abstract bases to concrete implementations. Since mixins are defined in terms of named interfaces that lack constructors, or even in terms of unnamed interface features, mixins typically do not provide adequate support for merging of constructors.

Unlike compound objects, mixin-inheritance merges separately defined implementation fragments into a single class, and when instantiated this class creates an object with a merged identity as opposed to a network of interlinked objects with separate identities. This merger in identity also parallels the subtype relationship that exists between a mixin-inherited object and its inherited parts, unlike in the compound object case. Since `c` is instantiated by mixin composition, not only is the use of `AX_impl` and `BX_impl` exposed to the composer of `c`, the object `c` can actually be used polymorphically by the client of `c` as an `AX_impl` or a `BX_impl`. While this polymorphism is useful internally when implementing methods in terms of a base implementation, when exposed externally it violates encapsulation, in this case allowing `C_impl` to be used inappropriately by clients. In essence, mixin composition is equivalent to public implementation inheritance.

The implementation dependence problems we have seen thus far can be described as our inability to separate a client of an interface from its implementation, or the specializer of a base class from its base implementation. A different problem occurs when we are unable to separate independent portions of the implementation of some interface. Different groups of methods may embed different design decisions and encapsulate access to different instance variable subsets, but our ability to program these method groups incrementally is limited.

The mixin-inheritance approach supports incremental programming in terms of abstract bases, but if we choose to split up the implementation of methods, that split becomes apparent in the set of classes, interfaces, or mixins that need to be composed. Consider the following example showing an interface and two partial implementations:

```
interface W {
    void w1();
    void w2();
}
```

```
class W_impl1 implements W { // NOT JAVA!
    void w1() { ... }
}
class W_impl2 implements W { // NOT JAVA!
    void w2() { ... }
}
```

We are not allowed to do this in Java because classes `W_impl1` and `W_impl2` do not implement all of the methods in interface `W`. We would need a mixin-like approach, with mixins that can both extend and implement the same interface. Even in this case, however, the client must still be aware of the particular pieces, and instead of having to select one implementation for one interface, several compatible pieces must be combined.

```
class W_impl1a extends W implements W { // NOT JAVA!
    void w1() { ... }
}
class W_impl2a extends W implements W { // NOT JAVA!
    void w2() { ... }
}
// Pretend "+" is mixin composition
W w = new W_impl1a() + W_impl2a();
```

This combination process gets more complicated when the implementation pieces make local assumptions such as extending different sets of base interfaces, all of which must be properly instantiated by the client, who really only cares about interface `W`.

It has been argued that class-based object-oriented programming is limited in its support for large scale software development, multi-object collaborations, and handling of cross-cutting concerns. Languages and language-extensions for component-oriented development [Szy98], collaboration-based design (e.g., [MO02], [Her03]), and aspect-oriented programming [KLM+97] have been suggested to address each of these limitations. However, while each of these approaches addresses a somewhat different problem than ours, each proposed solution actually introduces new problems similar to the ones we have just discussed. We conclude this section by evaluating these ideas with respect to the problem of implementation bias.

In component-oriented design, for example, a component exposes one or more provided interfaces and expects that the user of the component will bind one or more required interfaces, a model we refer to as the *dual-ported component* model. Unfortunately, required interfaces are just another form of implementation parameterization and suffer from all of the same problems, namely that (1) the component interface is biased towards implementations that can be realized in terms of the required interfaces, (2) clients (or at least composers) are coupled to an improperly exposed implementation detail, and (3) the binding decision is transferred to the user of the component.

Collaboration-based design is based on the realization that individual classes and objects are not the only unit of encapsulation, that often a group of distinct objects need to collaborate and share implementation details. Most approaches to collaboration-based design use some form of class nesting, with a top-level class capturing the collaboration, and contained classes representing the collaborators. Such collaboration classes suffer from the same implementation bias problems as individual classes. Abstract roles for collaboration participants have been introduced to enhance the separation of collaborator interfaces from their implementations. One approach derives a concrete collaboration from one with abstract roles, forcing different collaborator implementations to be co-specified in the derived collaboration. This co-location of collaborators

suffers from the feature combinatorics problem ([BSST93], [Big94]), requiring specification of a new derived collaboration whenever we want to change one of the collaborator implementations. Another approach indirectly binds abstract roles with separately specified role implementations, but this either suffers from the same identity and overriding problems that compound objects suffer from, or requires dynamic lifting and lowering.

Finally, aspect-oriented programming attacks so-called "non-functional" concerns that cross-cut "functional" components by isolating such concerns into aspects and weaving them back in a separate step. From one perspective, aspects are just like classes, and aspects can be defined by inheriting other aspects. Aspect inheritance, however, suffers from the same implementation bias as class inheritance. More importantly, the linkage between aspects and components is established based on low-level statement patterns, not on an identifiable interface or contract. This violates the encapsulation of components and strongly couples aspects to component implementation details. Furthermore, unlike classes that may implement interfaces, aspects are implementations without interfaces, and as a result there is no notion of multiple implementations of the same aspect interface.

2.2 Decomposing the Problem

According to generally accepted principles of good design, components of a complex system should be designed to have limited knowledge of the rest of the system. Designing systems this way makes it easier to change one component without affecting another, and to reuse components unchanged in new contexts. As we have just seen, however, currently available programming languages limit our ability to decouple a component from its environment, leading to distribution of implementation decisions inappropriately throughout the system, and violating our basic principles of good design.

We argue that the root cause of this problem is the manner in which programming languages handle *instantiation dependencies*. We say an instantiation dependency occurs when one program fragment refers to the name of another for the purpose of instantiating a whole or partial object. This occurs in two basic ways in object-oriented languages, (1) when a whole object is dynamically created via the `new` operator, or (2) when a derived class implicitly instantiates its base. Although other dependencies besides instantiation dependencies also occur, such as when we define types for variables, arguments, and return types, these can be handled adequately with abstract interface types. Problems begin with instantiation dependencies, dependencies that cannot be expressed with an appropriate level of interface abstraction using the programming languages of today.

The term *implementation bias* refers to a class of problems related to overdependence on particular implementations. To understand implementation bias it is important to separate two distinct notions, (1) the intended design of a piece of program text along with what it demands from its underlying environment for its correctness, versus (2) what the program text actually says. To implement an abstract interface an implementation must adopt some approach and make some choices - this design intent is a necessary and appropriate choice of an implementation strategy and is *not* what is meant by implementation bias. The expression of that design intent in some programming language, however, may be forced to make certain non-essential choices which go beyond the implementation strategy and are not strictly needed for correctness. These choices, which programming languages often force us to make, produce implementation bias.

We contend that instantiation dependencies are the primary source of implementation bias. When faced with an instantiation dependency, we are generally forced to decide between embedding the instantiation decision in a component, delegating to a factory that itself must be instantiated, or parameterizing the component's interface. While interface parameterization can be helpful when used sparingly, it isn't scalable, because when components consistently delegate implementation decisions through interface parameters, the

parameter list grows in proportion to the size of implementation closure of a component. Parameters used internally as aggregated references as opposed to mixin-inherited bases may also produce identity and overriding problems and the overhead of method call forwarding. Finally, the mere presence of implementation parameters in an interface implicitly restricts the set of implementations to those that can be realized in terms of the exposed implementation parameters, and couples clients to the parameters' existence.

Thus, parameters make interfaces harder to use and restrict their possible implementations, whereas lack of parameters commits us to implementation choices and limits our flexibility. In practice this means that we manually parameterize components to preserve implementation flexibility where we can anticipate its importance and justify its cost, and we embed implementation decisions everywhere else and live with the consequences. Given today's programming languages, this dilemma is inescapable.

We now characterize this dilemma more precisely. First, we classify the kinds of decisions made in designing and implementing a software component, and then we define the specific problems that can arise when the resolution of abstract design decisions is not congruent with their expression in a concrete implementation language.

Our classification distinguishes between the *essential* and *non-essential* decisions that are made in the expression of a design. Essential decisions are necessary and sufficient to elaborate our design intent, whereas non-essential decisions embellish our implementation in ways not strictly required by the intent of our design and may be artificially introduced by its expression in some language. Our goal in implementing a design is to express *essential* decisions only, deferring all other decisions to another time, place, or agent in the system.

Various forms of implementation bias prevent us from achieving this goal. Our first problem captures the most basic form of implementation bias, bias that occurs when the programming language forces an arbitrary choice of implementation when the intent of our design demands only the use of an abstract interface.

Problem 1 (Forced Implementation Choice) *The* forced implementation choice problem *occurs when a client is forced to make the non-essential choice of a particular implementation in order to express an essential choice of an abstract interface in some language.*

With a language like Java, this problem need only occur with instantiation dependencies. Other dependencies can always be expressed in terms of interfaces, although this is not enforced by Java and is certainly not always how programs are written. But now, given that we have to select an implementation, how do we decide which one to use? That we have no real basis to make this decision is the next problem.

Problem 2 (Optimal Implementation Choice) *The* optimal implementation choice problem *occurs when the best choice of implementation for some interface cannot be determined by local information.*

The first problem forced us to make a choice, and the second problem says we have no good way to make it. The information that dictates the best choice could depend on the context of a component's use, which is generally not completely known to the author of the component. Even if we did have this information for some particular contexts, principles of good design would demand that we ignore it because it's volatile, and our goal is to preserve reusability. But some choice must be made, however suboptimal and context-dependent it may be. If we make it internally without parameterization, the choice is trapped, producing yet another problem.

Problem 3 (Trapped Implementation Bias) *The* trapped implementation bias problem *occurs when we embed a non-essential choice of implementation inside a component, making it difficult to see and to change.*

On second thought it might appear that principles of information hiding would indicate that this trapped decision is a good thing – but it isn't. For example, the decision by a class C to use some interface S (as opposed to some other interface T) may be essential to C's design intent in which case it should be embedded in C. However, the implementation of S is another matter entirely, a non-essential detail that does not belong in C. Granted, hiding the detail within C insulates the clients of C from it, but C should be insulated from it as well, and trapping it there makes matters worse. To avoid trapping this detail we naturally turn to interface parameters, but as we've seen this transfers the decision to the client, creating our next problem.

Problem 4 (Exposed Implementation Parameter) *The* exposed implementation parameter problem *occurs when an internal implementation choice is exposed and transferred to the client via an interface parameter.*

Whether or not a parameter is an implementation detail can only be answered relative to the abstract interface in which the parameter appears. If it has no purpose for the interface client other than to control how the interface is implemented, and is not otherwise needed by the client, it should be considered an implementation parameter. The mere presence of such a parameter implicitly limits the space of possible implementations to those that can be implemented in its terms. Thus, the bias has now spread to the interface itself, as well as to all of the clients who use the interface. The client's obligation to bind these parameters to implementation arguments is the subject of our next problem.

Problem 5 (Implementation Parameter Binding) *The* implementation parameter binding problem *occurs when an interface client is forced to bind multiple interface parameters to actual implementations.*

This may seem like a special case of the forced implementation selection and trapped implementation bias problems, as this problem involves choosing an implementation and trapping it where it cannot easily be changed. But here we also have the additional challenge of ensuring mutual consistency of multiple implementation arguments. On the one hand we can select parameter types to be sufficiently specific, ensuring that all object combinations with proper types are also mutually consistent. Unfortunately, accomplishing this safety may overly constrain the space of legal argument combinations, and in doing so we may expose overly strong types that reveal our internal implementation strategy, further restricting our ability to change the implementation. On the other hand, if we choose weaker parameter types to reveal less of our strategy and allow more legal combinations, we may not be able to prevent combinations that individually type-check but are mutually inconsistent.

This discussion on choosing the strength of implementation parameter abstractions points to a more general issue. Regardless of whether the interface abstractions we choose internally as part of an implementation strategy are exposed as parameters, we still need to choose interface abstractions that suit the strategy. Making this choice properly is the subject of our next problem.

Problem 6 (Strong Abstraction) *The* strong abstraction problem *occurs when the semantic abstraction of a chosen interface is too strong, i.e., when a weaker abstraction (a proper supertype of the chosen interface) would satisfy the demands of the implementation strategy equally well.*

It is convenient to think of the abstraction strength issue as a question of implementation tolerance. The selection of an interface abstraction defines our implementation tolerance, the range of possible implementations our strategy can tolerate, and by "tolerate" we mean "use and still be correct." By using an interface abstraction that is too specific, we unnecessarily restrict the set of possible implementations, tightening the

tolerance and making it overly difficult to select an implementation that fits. We know we have chosen an appropriately weak abstraction when use of any weaker abstraction would not guarantee enough to ensure correctness.

When implementing an interface, the strategy or algorithm chosen determines the ideal strength of a set of supporting abstractions to use in expressing the implementation. However, the scope of the strategy may also be limited in terms of what subset of interface methods are relevant to the strategy. This leads to the question of how interfaces relate to implementation strategies, and how elements of interfaces and implementations should be grouped.

As a general rule, elements that are used together should be grouped, and elements that are used independently or change independently should be separated. Abstract interfaces, for instance, should be organized primarily for the benefit of clients, and are typically arranged in in a subtype lattice. Implementations should be organized to encapsulate design decisions. These grouping considerations lead to a conflict that is captured in the next problem.

Problem 7 (Interface-Implementation Cohesion) *The* interface-implementation cohesion *problem occurs when the ideal grouping of elements in the interface abstraction dimension does not coincide with how the implementations of those elements are ideally grouped in the implementation dimension.*

This problem manifests itself in several ways. One is the natural occurrence of an $n : m$ relationship between interfaces and implementations – an interface can have many implementations, and an implementation can implement many interfaces. There may also be a difference in grouping across different implementations of the same interface. Furthermore, implementing an interface may involve several independent implementation choices that affect different portions of its implementation, a situation that gets more complex with collaborations. If we are forced to represent all choices in one place then we are also forced to restate them all whenever one choice changes. Factoring into base classes can help, but is limited without support for multiple inheritance, and can introduce spurious super/subordinate relationships between concepts that ought to have a peer-to-peer relationship, or no relationship at all. The key point is that the architecture of the separation depends on the implementation strategy. Not being able to separately define independent units of implementation is the subject of our next problem.

Problem 8 (Feature Combinatorics) *The* feature combinatorics problem *occurs when multiple independent design decisions must be made in the same implementation unit. This requires a separate implementation unit to be created for each unique feature combination, causing implementation fragments to be replicated.*

This problem has also been referred to as the library scaling problem (e.g., [BSST93], [Big94]). As components get larger, it is increasingly difficult to manage the feature combinations contained in components because there is inadequate support for specifying them separately and composing them together. As a result, either we accept the replication effort needed to produce a large space of feature combinations, or we get by with a smaller feature space.

These problems can be divided into two groups as follows:

- Problems with making design decisions:
  - Problem 2 - Optimal Implementation Choice
  - Problem 6 - Strong Abstraction

- Problems impairing the appropriate expression of a design:
  - Problem 1 - Forced Implementation Choice
  - Problem 3 - Trapped Implementation Bias
  - Problem 4 - Exposed Implementation Parameter
  - Problem 5 - Implementation Parameter Binding
  - Problem 7 - Interface-Implementation Cohesion
  - Problem 8 - Feature Combinatorics

We contend that the problems of the first kind are about good design – choosing good design strategies to implement an interface, choosing the weakest abstractions which will satisfy the requirements of our design strategy, and *not* making choices which go beyond that strategy. These are issues with the design concepts themselves, not issues that can be resolved in how those concepts are expressed in some language.

In contrast, problems of the second kind are about how design decisions may be expressed in some underlying programming language, and we assert that the task of organizing these choices properly demands better support from the underlying programming language than exists today. In the next section we turn to our recommendations for solving these problems.

3 Towards a Solution

In this section we present interface-oriented programming (IOP) as a solution to the problems just discussed. First, we discuss our general strategy and then propose a more detailed set of principles for interface-oriented program design. Then we sketch the design of an interface-oriented programming language called ARC, and describe the supporting mechanisms required to implement it. Finally, we illustrate IOP through several examples written in ARC. IOP in general and the ARC language in particular were first proposed in the author's PhD dissertation proposal [Var03].

3.1 An Interface-Oriented Programming Strategy

The essence of interface-oriented programming is to express all program interdependencies in terms of abstract interfaces, including those that occur at instantiation dependencies. Interfaces in languages such as Java and C# are already used to declare object types in places that have no knowledge of how the object was instantiated, and this is evidence for the reasonableness of the IOP approach, i.e., that objects can be correctly used without knowledge of the specific classes involved in their implementation. In fact, the whole idea of subtype polymorphism depends on this assumption. The key difference is that IOP extends this decoupling to all object type declarations, including instantiation dependencies such as inheritance and dynamic object creation.

One effect of this practice is that implementation selection at the point of instantiation dependency is delegated to some agent other than the client of the dependency. This in turn demands that interfaces be sufficiently prescriptive to enable this other agent to make a suitable choice. As a result, IOP demands additional discipline in identifying meaning with interfaces, as well as programming language mechanisms that support and encourage the practice.

In response, we propose the following general approach to a solution:

1. We adopt the convention to identify interface semantics informally with names of abstract interfaces. By doing this we intend to capture everything that users of interfaces should care about in the names of abstractions. We do not address the issue of formally specifying the semantics of an interface, nor do we consider how the implementation of an interface might formally be verified against such a specification. [2] The task of identifying which implementations are acceptable to a user of an interface is thereby reduced by convention to that user's selection of the right abstract interface. An arbitrary implementation of the interface can then be chosen separately, apart from the code containing the dependency.

2. The design of a component implementing some interface should encapsulate a minimal implementation strategy, and this strategy will generally dictate the use of certain subordinate interfaces. The capabilities and semantics required of these interfaces is an essential aspect of a design that should serve the implementation strategy. The choice of what subordinate capabilities to use in an implementation should be made by selecting existing abstract interfaces, or creating a new ones. How such interfaces are implemented is a non-essential detail that should be excluded from the design of the component that is using them. As a rule, an implementation should include only *essential* implementation details that follow from the encapsulated strategy, details that should also be encapsulated if they are not

[2] While we consider formal specification and verification to be desirable in principle, applying them successfully in practice remains an open problem, a problem that is somewhat orthogonal to ours. We believe that interface-oriented programming neither prevents nor requires but could be usefully combined with formal specification and verification, but demonstrating this requires additional research.

also *essential* to the interface being implemented. On the other hand, *non-essential* details should be omitted from both the interface being implemented and its implementation, and should be deferred instead to the implementations of subordinate interfaces.

3. Automatic support should be provided for selecting and evolving the non-essential details that are necessary to generate a concrete, executable implementation of a component and all of its subordinate components.

3.2 Principles of Interface-Oriented Program Design

We now elaborate the strategy into a more detailed set of principles that should govern interface-oriented program design. The fundamental ideas expressed by these principles are the separation of interfaces and implementations, the semantic interpretation of interfaces, the use of abstract interfaces for all program dependencies, and the encapsulation of just the right amount of information in an implementation, leading to the need for partial implementations. Implicit in the principles is the acceptance of multiple inheritance for both interfaces and implementations.

Principle 1 (Interface-Implementation Separation) *Use of interface abstractions should be decoupled from choice of implementations – it should be possible to use an interface abstraction without having to choose an implementation for it.*

Principle 2 (Interface Semantics) *An abstract interface should imply a fixed semantics (an understood, externally visible behavior) for a set of operations that can be expected from any correct implementation, but does not otherwise constrain what implementations are available or possible.*

Principle 3 (Abstract Interface Dependency) *All non-local program dependencies (including instantiation dependencies) should be expressed in the form of abstract interfaces. Implementation names may not be used as type names.*

Principle 4 (Interface-Oriented Subtyping) *Every object is an instance of a complete implementation and has a type determined by the abstract interfaces it implements.*

Principle 5 (Implementation Strategy Relevance) *The implementation of an interface should depend only on the understood interface semantics and an encapsulated implementation strategy.*

Principle 6 (Partial Implementation) *Implementations may exist as partial implementations of abstract interfaces. A (partial) implementation may implement some or all of the methods in the interfaces it implements, and may also defer implementation of methods to other unspecified partial co-implementations.*

Principle 7 (State Encapsulation) *Instance variables used in a partial implementation should not be visible to clients or other partial co-implementations.*

Principle 8 (Method Cohesion) *An implementation should implement only those methods that depend on the local implementation strategy. All other methods should be assumed to be provided by other partial co-implementations.*

Principle 9 (Weakest Abstract Object) *For each abstract interface used for aggregated objects, method parameters, method return values, and local method variables, the weakest abstraction that satisfies the demands of the implementation context should be chosen.*

Principle 10 (Strongest Abstract Base) *For each abstract interface used as an inherited base, the strongest abstraction that satisfies the demands of the implementation context should be chosen.*

Principle 11 (Local Correctness) *The correctness of a partial implementation should be judged relative to the semantics of the inherited base interfaces and the interfaces of other abstract objects that are used in aggregation.*

We accept a class-based, object-oriented system model as a useful but insufficient basis for software development. We also intend that our language will provide static typing, separate compilation, and support for namespace partitioning, features we consider essential for large scale software development. Programming mechanisms such as inheritance, composition, and delegation are distinct, useful mechanisms for incremental programming, and parametric types and virtual types are useful mechanisms of generic programming.

But to create a system that enables our interface-oriented strategy we must make some modifications. If we denote an object-oriented programming system providing the set of basic capabilities just described as `OOP`, and the desired interface-oriented system as `IOP`, the necessary modifications can be thought of abstractly in terms of the following equation:

$$\texttt{IOP} = \phi(\texttt{OOP}) = \iota(\sigma(\texttt{OOP}))$$

IOP adds new forms of separation, denoted here by σ, due to its strict interface abstraction and support for partial implementations of interfaces. This in turn results in new kinds of program fragments that require new integration mechanisms, denoted here by ι, mechanisms that provide automatic composition of implementation fragments and automatic selection of whole implementations. The composition of these modifications, denoted by ϕ, captures the complete set of changes needed to produce an interface-oriented system.

3.3 Design of an Interface-Oriented Programming System

In this section we present some of the key features of an interface-oriented programming system called ARC that is currently under development. The acronym ARC is derived from the three main constructs in the language, abstractions, representations, and contexts. In order to concentrate on our solution to implementation bias we will limit our discussion to those constructs that deal with it: abstractions and representations. Contexts will not be covered in detail in this report.

ARC can be thought of as a reorganization of Java – not only do we replace interfaces with abstractions, classes with representations, and packages with contexts, but we also but also add new forms of incremental programming, and change how inheritance works. At the statement level the syntax is recognizably Java-like with a few minor adjustments.

3.3.1 Separating Interfaces from Implementations

Like Java, ARC provides separate constructs for interfaces and implementations. ARC interfaces are called abstractions and defined using the `abs` construct, and ARC implementations are called representations and defined using the `rep` construct.

Abstractions Although we often use the terms interface and abstraction interchangeably, we use the term abstraction when referring to the ARC language construct. The different name emphasizes that ARC

interfaces are more than just bundles of method signatures, as a group they also have an associated semantics, albeit one that is not defined formally in ARC but is expected to be established informally by convention.

An abstraction may define the signatures of constructors, methods, and roles. Constructors create objects that represent the interface, methods are used to invoke operations on objects, and roles are abstraction-relative interfaces for objects that collaborate with objects of this interface. For example, the following interface:

```
abs Bag<T> {
    Bag<T>();
    boolean empty();
    void add(T x);
    Iterator<T> elements();
    void addAll(Bag<T> b);
}
```

defines a parameterized abstraction `Bag<T>` with one constructor and four methods.

Syntactically, method signatures in ARC are essentially the same as in Java, but with the addition of constructors and roles which have no corresponding construct in Java interfaces (roles are not shown in this example but will be explained later). As in Java interfaces, abstraction members in ARC are all intrinsically public, and ARC provides no access modifiers to say otherwise.

Representations ARC replaces Java classes with representations. A representation is much like a class in that it provides implementations for the interface methods it represents. Again, we use slightly different terminology and keywords to call attention to the differences with their Java counterparts. Like classes, representations may also declare local instance variables, as this example shows by implementing the stack abstraction in terms of an aggregated array variable:

```
rep RB<T> represents Bag<T> {
    Array<T> array;
    Bag<T>() {
        array = new Array<T>();
    }
    boolean empty() {
        return array.empty();
    }
    void add(T x) {
        array.add(x);
    }
    ... // all other methods of Bag<T> implemented
}
```

The language for declaring instance variables and writing executable statements is essentially a parameterized type version of Java, but with the additional constraint that whenever a type declaration is needed, only an abstraction name or a type expression based on abstraction names and type parameters may be used – representation names like `RB<T>` are not allowed in type declarations. In contrast, Java allows both class names and interface names in type declarations.

For example, instance variable `array` is typed above using abstraction `Array<T>`, which we imagine here to be an abstraction for a flexible array, such as:

```
abs Array<T> {
    Array<T>();
    void add(T x);
    T get(int index);
    ...
}
```

but the representation of `Array<T>` used by `RB<T>` is not specifiable.

The declaration of variable types using abstractions is equivalent to the use of interfaces to declare variables in Java. The place where ARC departs most significantly from Java is the use of types at points of instantiation dependency, such as when new objects are dynamically created. This is illustrated in the initialization of the `array` variable in representation `RB<T>`. In the statement:

```
array = new Array<T>();
```

the constructor expression specifies the abstraction to use but does not specify a representation. If we tried to do this in Java with an interface our code would not compile, as Java does not provide any means for defining constructor signatures in interfaces and does not allow interfaces to be directly instantiated. ARC implements this statement by arbitrarily choosing *some* complete representation of `Array<T>` and invoking its `Array<T>()` constructor to create the object, a mechanism called *non-deterministic instantiation*. If no such representation is available, an error occurs. This error occurs sometime after compile-time (i.e., it is not a compile-time error - the statement compiles successfully), but the error may occur as late as run-time.

3.3.2 Interface-Oriented Inheritance

ARC allows abstractions and representations to be defined incrementally using multiple inheritance. While ARC's interface inheritance is similar to Java's interface inheritance, ARC representations inherit other representations indirectly through interfaces, making ARC's mechanism for implementation inheritance quite different from Java's.

Inheritance of Interfaces One abstraction may extend another, thereby inheriting its interface and semantics and establishing a subtype relationship. Subtyping is defined nominally in terms of the extension relation over named abstractions, established with the `extends` keyword:

```
abs C extends A, B { ... }
```

A subtype may add new methods, and the semantics of the subtype is expected to be a behavioral extension or refinement of the semantics of its supertypes, to ensure that instances of the subtype are always safely substitutable as instances of their supertypes. Again, this expectation cannot be formally guaranteed by the ARC language but is assumed to be established by convention.

ARC's nominal subtyping is similar to Java's. However, Java treats methods with the same signature in different interfaces to be one method when the interfaces are in a subtype relation or when these interfaces are implemented by a single class, whereas ARC keeps them separate. When multiple interfaces are implemented in Java, the set of method signatures to implement is the union of signatures from the interfaces, whereas in ARC it is the disjoint union, with abstraction names used to disambiguate coincident method signatures in the disjoint union.

As a result, each method in an ARC abstraction can be said to occupy a unique semantic slot, a slot is not annihilated or combined with another from some other abstraction just because of signature coincidence.

Consider the example of a `Cowboy` abstraction and a `Picture` abstraction, each having a method named `draw()`:

```
abs Cowboy {
    void draw(); // remove gun from holster
    ...
}

abs Picture {
    void draw(); // render on display
    ...
}

abs CowboyPicture extends Cowboy, Picture {
    ...
}

rep RCP represents CowboyPicture {
    void Cowboy::draw() { ... }
    void Picture::draw() { ... }
    ...
}
```

In general these methods have completely different purposes and semantics, and should not be unified. The fact that they have the same name should at most mean that a fully-qualified name must be used to refer to them, not that they have to share one implementation. ARC expects separate implementations of `Cowboy::draw()` and `Picture::draw()`, as shown above in representation `RCP`.

Interface-Oriented Inheritance of Implementation Representations may also inherit from other representations, but in an indirect way – through an abstraction. One representation inherits another by assuming an abstraction, using `assumes`. For example, our bag example above could be changed to one that inherits an `Array<T>` rather than aggregating one:

```
rep RB2<T> represents Bag<T> assumes Array<T> {
    Bag<T>() assumes Array<T>() { }
    boolean empty() { return Array::empty(); }
    void add(T x) { Array::add(x); }
    ... // all other methods of Bag<T> implemented
}
```

In this case, `RB2<T>` inherits *some* implementation of an `Array<T>` but cannot specify which one. Note also that the constructor for `Bag<T>` now assumes a constructor call to `Array<T>`, and methods are explicitly forwarded to methods in the inherited base `Array<T>` (methods which in this case have the same signature and must be disambiguated using the `Array::` qualifier).

The inheritance of `Array<T>` makes `RB2<T>` essentially a subtype of `Array<T>` but only in the context of the representation `RB2<T>`. In other words, the type of `this` in `RB2<T>` is `A` where `A extends Bag<T>, Array<T>`, whereas the type of `this` in `RB<T>` from our previous example is just `Bag<T>`. So, although

methods in RB2<T> will be able to substitute this as an Array<T>, clients of an object implemented by RB2<T> will not see that object as an Array<T> and will not be able to substitute it for one.

This example of inheritance uses explicit forwarding of methods, because Bag<T> has no relationship to Array<T>, and the syntactic coincidence of the different interface methods Bag::empty() and Array::empty(), for example, does not unify their implementations. Different approaches to forwarding will be discussed in the stack example in section 3.5.1.

Summary of Interface-Oriented Inheritance Relationships Thus far we have encountered three kinds of relationships that can exist between abstractions and representations in ARC:

1. the extends relation between abstractions (interface inheritance);
2. the represents relation between representations and abstractions (interface implementation);
3. the assumes relation between representations and abstractions (interface-oriented implementation inheritance).

The extends relation provides interface-oriented subtyping, the represents relation establishes the interface or client-visible type of a representation, and the assumes relation provides interface-oriented inheritance of implementations.

The represents relation can be thought of as indirectly connecting the extends and assumes relations, providing dual forms of interface-oriented inheritance, one for interfaces and one for implementations. The indirect connection keeps the indirect implementation-inheritance relationships hidden from clients of abstract interfaces when necessary. Whether or not assumed implementations are visible to clients depends entirely on whether the assumed abstractions are also part of the represented abstractions (the represented and assumed abstractions need not be disjoint, and will often overlap).

3.3.3 Partial Representations

There are several modes of partial representation supported in ARC, each of which involves making assumptions about the remainder of the representation of an individual object or a collaboration:

1. representing one abstraction by assuming unrelated abstractions;
2. representing a derived abstraction by assuming base abstractions (ancestors of the derived abstraction);
3. representing some methods of an abstraction and assuming the remaining methods of the same abstraction;
4. representing (part of) one collaborator in a multi-object collaboration and assuming other collaborators;

These modes of partial representation are not independent and can be combined in various ways. Assumptions about other partial representations are always expressed in the form of interfaces. Cases [1]-[3] correspond to interface-oriented base assumptions, and are different ways of partially representing an individual object by assuming an inherited base and implementing the rest. Case [4] refers to interface-oriented collaboration assumptions, which can occur whenever (partially) representing one collaborating object while assuming implementations of other collaborating objects.

Interface-Oriented Base (Object) Assumptions A partial representation implements only a portion of the methods in the interfaces it represents, and assumes all others will be provided by other partial representations. In the previous section, representation `RB<T>` is a complete representation, whereas `RB2<T>` is partial because it assumes implementations for the methods of `Array<T>`. In general, most representations in ARC will be partial.

To illustrate how different kinds of assumptions occur in partial representations we return to the bag example with a slightly different abstraction hierarchy, where `Bag<T>` and `Array<T>` are now siblings with a common ancestor abstraction, `Collection<T>`:

```
abs Collection<T> {
    Collection<T>();
    boolean empty();
    Iterator<T> elements();
}

abs Array<T> extends Collection<T> {
    Array<T>();
    void set(int index,T x);
    T get(int index);
    int size();
}

abs Bag<T> extends Collection<T> {
    Bag<T>();
    void add(T x);
    void addAll(Bag<T> b);
}
```

In our previous bag examples, `RB<T>` had no assumed bases but merely aggregated a reference to an array, and `RB2<T>` assumed the unrelated abstraction `Array<T>`. In our next example, we split the bag representation into two partial representations. The first part, `RB3<T>`, implements the basic bag operations in terms of an assumed `Array<T>` base and assumes the inherited `Collection<T>` operations (such as `empty()`) and the remaining bag operations (such as `addAll(Bag<T>)`). The second part, `RB3all<T>`, implements the bulk operation `addAll(Bag<T>)` in terms of `add(T)` and assumes the rest.

```
rep RB3<T> represents Bag<T> assumes Collection<T>, Bag<T>, Array<T> {
    ...
    void add(T x) {
        set(size(),x); // No need for Array:: qualifier
    }
    ...
}
rep RB3all<T> represents Bag<T> assumes Collection<T>, Bag<T> {
    ...
    void addAll(Bag<T> b) {
        for (T x in b.elements()) {
            add(x);
```

```
            }
        }
        ...
    }
```

The assumption of `Bag<T>` while representing `Bag<T>` is a common IOP idiom corresponding to case [3] above, representing some methods of an interface while assuming the rest. The assumption of `Collection<T>` (which is actually implied by the assumption of `Bag<T>` but is listed explicitly for clarity) is another common idiom in which the ancestor of a represented abstraction is assumed in the implementation of that abstraction. Finally, note that the assumption of `Array<T>` is needed only in `RB3<T>`, because `RB3all<T>` does not use the array assumption, and this allows `RB3all<T>` to be combined with other partial representations that implement the basic bag operations in different ways.

Interface-Oriented Collaborator Assumptions Often there exist collections of objects that are separately instantiated yet need to be implemented in some mutually consistent way. Such collections of mutually dependent objects are called collaborations. The challenge is to keep collaboration implementation details secret from clients but allow the proper amount of secret sharing among members of the collaboration.

Consider the simple example of a list of elements, as modelled by the abstractions `Element<T>` and `List<T>`:

```
abs Element<T> {
    field T value;
}

abs List<T> {
    role Element<T>;
    ...
    void add(role Element<T> e);
    void insertAfter(role Element<T> p, role Element<T> e);
    ...
}
```

(Note the use of an abstraction `field` for the `value` member of `Element<T>`. Abstraction fields are just syntactic sugar in interfaces for set/get method pairs, with the addition of a presumed default implementation (the obvious one) that can be automatically provided in partial representations, or manually overridden.)

It is common for a list implementation to make assumptions about the elements, e.g., singly-linked vs. doubly-linked, etc. The `List<T>` abstraction above does not imply any particular implementation of an `Element<T>`, but by introducing an abstract role via `role Element<T>` it indicates that implementations of `List<T>` may depend on abstract aspects of particular implementations of `Element<T>`.

The static type of a list element, or `role Element<T>`, is actually relative to a particular `List<T>` *object*, and we use instance-anchored types based on Ernst's family polymorphism concept [Ern01] to ensure *statically* that only the proper kinds of elements and lists may be combined, even though we do not know statically what their implementations are:

```
final List<T> list1 = new List<T>();
final List<T> list2 = new List<T>(); // might be different representation than list1
```

```
Element<T> e1 = new list1.Element<T>(); // consistent with list1's representation
Element<T> e2 = new list2.Element<T>(); // consistent with list2's representation

list1.add(e1); // OK
list2.add(e2); // OK

list1.add(e2); // Compile-time Error
list2.add(e1); // Compile-time Error
```

The `final` declaration is necessary to enable checking that two types have the same instance-anchor without having to do dataflow analysis on the value of the instance-anchor.

Armed with the declaration of the abstract role for `Element<T>`, representations of `List<T>` may now assert dependencies on particular kinds of list elements, and are assured by the type system and the representation inference mechanism that their assumptions will always be satisfied. For example, we might introduce element types for singly-linked and doubly-linked lists as follows:

```
abs SingleLinked<T> extends Element<T> {
    field SingleLinked<T> next;
}

abs DoubleLinked<T> extends Element<T> {
    field DoubleLinked<T> next;
    field DoubleLinked<T> prev;
}
```

A singly-linked list representation (`SL<T>`) will assert that elements are represented as `SingleLinked<T>` using a `with role ... as ...` clause:

```
rep SL<T> represents List<T> with role Element<T> as SingleLinked<T> {
    ...
    void insertAfter( role Element<T> p, role Element<T> e ) {
        e.next = p.next;
        p.next = e;
    }
}
```

and a doubly-linked list representation (`DL<T>`) will depend on elements being `DoubleLinked<T>`:

```
rep DL<T> represents List<T> with role Element<T> as DoubleLinked<T> {
    ...
    void insertAfter( role Element<T> p, role Element<T> e ) {
        e.next = p.next;
        e.prev = p;
        p.next = e;
        e.next.prev = e;
    }
}
```

3.3.4 Other Details

In this section we briefly mention some of the other details of ARC, some of which are necessary for IOP and others which are convenient to have but not strictly necessary, including:

- constructors and constructor assumptions,
- lowering constraints on assumed bases,
- lowering constraints on roles,
- method constraints,
- abstraction properties,
- exceptions,
- control abstraction,
- contexts, and
- static members.

Some of these details (such as those dealing with constructors and various forms of representation constraints) will be explored further in examples to be presented later, but a detailed explanation of most will be deferred to future reports.

Lowering constraints on assumed bases and roles, as well as constraints on methods, are all expressed in representations. They do not affect abstractions per se, but they do affect how representations may be combined to implement them.

Lowering constraints on base abstractions and collaborating roles are necessary to ensure proper sharing and sufficiently specialized implementation of common interfaces across parts of an object or across the members of a collaboration. Representation inference, the mechanism responsible for combining partial representations into whole ones, is responsible for adhering to the constraints expressed each each partial representation, making sure that partial representations that are incompatible are not combined. Similarly, method constraints are used to declare additional abstract assumptions that a partial representation makes about the implementation of an assumed method, and representation inferences ensures that only consistent method implementations are combined.

Abstractions may have abstract properties with associated possible sets of values, to provide a more expressive way to represent interface semantics than just abstraction names alone. Derived abstractions may add new properties and refine or restrict the values of existing ones, and representations may declare those property values they are consistent with.

Exceptions thrown by implementations of an interface can become a source of implementation bias, and demand a different approach than the strict approach taken by Java. The implementation of iterators and other control abstractions also demands better support in the language. Both of these aspects of ARC are still being designed.

Contexts are structures similar to Java packages that provide a hierarchy of enclosing namespaces for abstractions and representations, and can be used to restrict the visibility and access permissions of abstractions and other contexts. Static members are not provided in abstractions or representations, but are allowed in contexts in the form of context objects.

3.4 Mechanisms of Interface-Oriented Program Development

Interface-oriented programming enforces a separation of concerns that demands corresponding support for automatic integration of concerns. In this section we briefly outline the required capabilities of the enabling mechanisms of interface-oriented programming: representation inference, non-deterministic instantiation, and the segmented object model. The design and implementation of these mechanisms is ongoing, therefore a detailed explanation is deferred to future reports.

3.4.1 Representation Inference

Representation inference can be thought of as a new form of program linking, a step that occurs after compilation and before run-time, generating for each interface a set of alternative complete representations based on available partial representations. Partial representations must be unified in a way that provides a complete implementation of all interface methods and also respects each partial representation's local constraints on assumed base types, unspecified methods, and role implementations.

3.4.2 Non-Deterministic Instantiation

Whereas representation inference generates a set of alternative implementations for an abstraction, representation selection is simply the means to select one of those alternatives. This mechanism amounts to a built-in factory mechanism and can be as simple or as sophisticated as desired. We envision a number of approaches that are configurable and controllable at the meta-level, from simple arbitrary selection of one of the available alternatives for each interface considered separately, to context-dependent, evolutionary optimization of the overall set of representation choices for a whole application.

3.4.3 Segmented Object-Model

ARC's interface-oriented support for multiple inheritance and the incremental representation style it encourages demand an underlying object model which supports this flexibility in an efficient way. Our goal has been to enable abstractions and representations to be separately compiled while still supporting dynamic dispatch in constant-time.

Dynamic Dispatch in Object-Oriented Languages In languages such as C++ or Java, an object can be thought of as the concatenation of instance variables taken from all of the classes participating in the implementation. The participating implementation classes as well as the total number and size of their instance variables are generally known at compile-time, at least to the implementation classes if not to client code. The manner in which objects are structured and methods are dispatched varies depending on whether single or multiple inheritance is supported, and whether first-class interfaces are provided.

In a system with only single inheritance and no interfaces (such as C++ before multiple inheritance was added), the layout of an object includes a pointer to a shared method dispatch table (which C++ calls a virtual function table) followed by a vector of instance variables. Method dispatch is accomplished by indexing the method dispatch table to retrieve a pointer to the method, using a statically-determined index corresponding to the method selector. Conversion of an object reference to a base type requires no action because the relative offset of the dispatch table and the instance variables for all base types of the object are the same under single inheritance - zero. Calling the method is accomplished by dereferencing the method pointer and passing the object reference along with any other arguments. The actual method then uses the object reference to access instance variables.

Multiple inheritance adds an extra level of complication. With multiple inheritance, the offset of a base type's instance variables in a linearization of all instance variables is no longer zero, so calls to a base class method may require offsetting the object reference pointer such that base class methods see their instance variables at the expected relative offsets. This also requires that multiple pointers to dispatch tables be strewn across the object such that when the address of an object reference is adjusted to the start of a base class's instance variables, a dispatch table pointer also exists at a known relative offset (typically zero) from the adjusted base address. In one implementation of C++ multiple inheritance [Str87a], objects have multiple virtual function table pointers, and method dispatch involves two steps, method lookup and object reference offsetting. This reference offsetting also complicates the code generation required for reference comparisons, which can no longer be handled as raw pointer comparisons.

Interface dispatch in Java is further complicated by several factors. One issue is that a method call through an interface reference may not be aware of the actual class implementing the object. The other issue is that an object may implement multiple interfaces and different interfaces may override or redefine the same method selector. Not only can this lead to semantic conflicts, e.g., the artistic cowboy problem discussed earlier, but there is also no way to statically linearize the method selectors for the purpose of indexing a compact dispatch table in a way that applies to all implementations of an interface. Thus, although it is statically known that an interface reference implements the methods of the interface, the methods themselves are not quickly locatable, and run-time search is required to find the method to invoke. This is actually reminiscent of early object oriented systems that implemented method dispatch as search, starting with the lowest level class and search through its parent classes until a method with a matching method selector is found. More efficient methods of interface dispatch involving selector coloring and large dispatch tables, caching of search results, or hashing have been suggested (e.g., [ACF+01]), but an approach involving a constant-offset lookup of a compact table is simply not possible with Java's design.

Segmented Instances and Multi-Faceted Dynamic Dispatch IOP in general and ARC in particular add even more challenges that must be overcome by the dispatch mechanism. The first is that assumed bases of a representation are abstractions and the size of their implementations is unknown at compile-time. The second is that the set of abstractions that comprise a base is itself not fully known to each partial representation at compile time. ARC's solution to these challenges is based on the use of segmented instances and multi-faceted dispatch tables.

Each object instance in ARC has an implementation type that corresponds to the abstractions it represents and assumes. The actual type exposed to a particular client or within a partial representation will be a subset of this implementation type. An object instance in ARC consists of an instance-level dispatch table pointer and a vector of pointers to segments, one segment per partial representation. The particular combination of partial representations is determined by the representation inference mechanism sometime after compile-time and is fixed by the time an object is instantiated. Each segment provides the set of instance variables and methods defined by its corresponding partial representation. The number of segments comprising the instance of an abstraction is not known at compile-time, and the actual representations used for each segment are also not known at compile-time. This leads to a multi-faceted dispatch architecture involving three forms of dynamic dispatch: segment dispatch, method dispatch, and base type dispatch.

When a method is called, both the target method and the target segment are dynamically bound. The dispatch table of the instance contains two kinds of subtables, segment tables and method tables. Both are indexed by a compile-time constant domain based on a linearization of the methods in the abstraction. The segment table returns an index of the segment to use, and the method table returns the pointer to the method to call. When a method is called, both the instance pointer and the segment pointer are passed as

```
class Instance {
    InstanceTable table;
    Segment[] seg;
}
class Segment {
    SegmentTable table;
    Type1 Var1;
    ...
}
class SegmentTable {
    int[] segment;
    Method[] method;
}
class InstanceTable extends SegmentTable {
    int[] base;
    InstanceTable[] basetable;
}
class Method {
    void call0( Instance inst, Segment seg );
}
```

```
// client call
Instance inst = ...;
int iseg = inst.table.segment[WHICHMETH_CLIENT];
Segment seg = inst.seg[iseg];
Method meth = inst.table.method[WHICHMETH_CLIENT];
meth.call0(inst,seg);

// self call
int iseg = seg.table.segment[WHICHMETH_SELF];
Segment newSeg = inst.seg[iseg];
Method meth = seg.table.method[WHICHMETH_SELF];
meth.call0(inst,newSeg);

// client type conversion: newInst = inst
Instance inst = ...;
int ibase = inst.table.base[WHICHBASE_CLIENT];
InstanceTable t2 = inst.table.basetable[ibase];
Instance newInst = new Instance(t2,inst.seg);
```

Figure 1: Java Implementation Model of Segmented Dispatch

arguments to provide access to the object's state within the method. Each method only has access to the local variables of its partial representation, which are all contained in its segment.

When a representation calls itself (makes a "self-call"), the dispatch table stored in the segment is used instead. Note that the indexes used for methods in one segment do not necessarily correspond to the indexes used in other segments, or at the instance level. This is because a client of an abstraction has a different view of what methods are available than the internal representations, and similarly each representation may have its own unique view. Thus, the set of base abstractions being linearized is different in each case, and this is a key to achieving constant-time, interface-oriented dispatch. Furthermore, inside a partial representation we need only assign indexes to the methods that are actually called by that partial representation, because that partial representation is the only user of its segment level tables. In contrast, the instance-level tables used by clients must assign indexes for all methods in the abstraction.

The different viewpoint issue actually occurs again at the external client level. Due to multiple inheritance, a base type's linearization of its members may not be an initial segment of a derived type's linearization of its members. This leads to the need for multiple instance-level tables and for base type dispatch. Base type dispatch refers to the replacement of the instance-level dispatch table when an instance reference is converted from one type to another. To accommodate this change, instance references in IOP must be widened to refer both to the instance as well as the dispatch table to use to access the instance based on the type of the reference.

A Java model of the implementation of segmented dispatch is given in figure 1, showing the Java classes that would implement instances, segments, and dispatch tables on the left, and illustrating on the right how those data structures would be used to implement client dispatch, self dispatch, and type conversion (base type dispatch). This model is not intended to represent a production implementation, its purpose is merely to illustrate the dispatch architecture.

A client call of a method given an instance reference `inst` is accomplished by segment dispatch and method dispatch via the instance-level tables (ignoring the handling of additional method arguments and method return values), where `WHICHMETH_CLIENT` and `WHICHMETH_SELF` are statically determined values derived from the method being invoked and its position in the view-specific method linearization of the static type of `inst`. A self call is handled slightly differently (this example assumes `inst` and `seg` were passed into the

```
abs Collection<T> {
  boolean contains(T);
  boolean empty(T);
  control Iterator<T> elements();
}

abs List<T> extends Collection<T> {
  List<T>();
  void addFront(T);
  void addRear(T);
  T removeFront();
  T removeRear();
  T front();
  T rear();
}

abs Stack<T> extends Collection<T> {
  Stack<T>();
  void push(T);
  void pushAll(Collection<T> c);
  T pop();
  T top();
}
```

```
rep RSc represents Stack<T> assumes Stack<T> {
  List<T> list;
  Stack<T>() assumes Stack<>() { list = new List<T>(); }
  boolean empty() { return list.empty(); }
  boolean contains(T x) { return list.contains(x); }
  control Iterator<T> elements() { return list.elements(); }
  void push(T x) { list.addRear(x); }
  T top() { return list.rear(); }
  T pop() { return list.removeRear(); }
  void pushAll(Collection<T> c) uses push(T);
}

rep RSi represents Stack<T> assumes Stack<T>, List<T> {
  Stack<T>() assumes Stack<T>(), List<T>() { }
  void push(T x) { addRear(x); }
  T top() { return rear(); }
  T pop() { return removeRear(); }
  void pushAll(Collection<T> c) uses push(T);
}

rep RSall represents Stack<T> assumes Stack<T> {
  void pushAll(Collection<T> c) uses push(T) {
    for (T x in c.elements()) push(x);
  }
}
```

Figure 2: ARC Abstractions and Representations for a Stack

method that is invoking some other method of self), using the segment level tables for segment dispatch and method dispatch.

Since implicit type conversions involve change of view, implementing them requires table translations to accommodate the potentially different static offsets used by different types. Each type essentially represents a view of an object, with its own index domain for method selectors and base types. The `basetable` field refers to a collection of instance-level dispatch tables, one per base type implemented by a complete representation. The new type-converted reference has a possibly modified dispatch table that is found using dynamic base type dispatch using a statically determined index for the base type (`WHICHBASE`), but shares the segment vector. The dynamic object creation (`new Instance(...)`) shown in this example is necessary in our simplified Java model but can be optimized away in a production implementation of ARC by widening object references and copying fields.

Finally, two references are considered to be equal (refer to the same object) if their segment vectors are identical, so reference comparison must be overloaded to ignore the table pointer and just check the segment vector pointer.

3.5 More Examples in ARC

This section further illustrates the concepts of IOP via more examples in the ARC language.

3.5.1 A Stack Example

We will now describe the design of a stack in ARC. Figure 2 shows the abstractions `Stack<T>` and `List<T>`, defined as subtypes of `Collection<T>`, along with three partial representations of `Stack<T>`, `RSc`, `RSi`, and `RSall`.

Representation `RSc` implements the basic methods of a stack in terms of an aggregated list object, and

```
rep RSf represents Stack<T> assumes Stack<T>, List<T> {
  Stack<T>() assumes Stack<T>(), List<T>() { }
  void push(T x) forward addRear(T);
  T top() forward rear();
  T pop() forward removeRear();
  void pushAll(Collection<T> c) uses push(T);
}
```

Figure 3: Stack representation with forwarding optimization

assumes that the bulk operation `pushAll` is provided by some other partial representation constrained to use `push`, as indicated by the `uses push(T)` method constraint. Note that the method constraint is local to `RSc` and does not affect the `Stack<T>` abstraction. Since we use composition in `RSc`, we must explicitly forward method calls (at run-time) to the appropriate methods of the list object. The list object is appropriately encapsulated in `RSc`, preventing clients from misusing the stack object in a way that violates its abstraction.

In contrast, representation `RSi` uses interface-oriented implementation inheritance of `List<T>` to accomplish the same effect more efficiently, both in terms of run-time overhead and notationally. Despite the fact that the `List<T>` base is invisible to clients, no forwarding or adaptation is needed for the `empty`, `contains`, and `elements` methods – the versions provided by `List<T>` are unified by assumption with those required by `Stack<T>` because they come from a common assumed base abstraction, `Collection<T>`. Forwarding is needed to adapt the other methods of `Stack<T>` to `List<T>`, however. This forwarding is needed due to the translation of one interface to another, but it is still done in terms of self calls. Thus, this example illustrates use of inheritance for both specialization and adaptation.

The forwarding in this example happens to be trivial in the sense that we are calling a different method with the exact same argument list and return type. Although not all forwarding falls into this category, when it does, a more efficient notation using a more direct forwarding mechanism is possible, a mechanism that is applied statically, not at run-time. For example, the representation of `push(T)` in `RS1i` can be changed to `void push(T x) forward addRear(T);` which means that the dispatch table entry for `Stack<T>::push(T)` will directly invoke whatever the representation is for `List<T>::addRear(T)`, even though the client of the stack object is not even aware that a list is being used. Converting `RSi` to use this forwarding optimization whenever possible results in `RSf`, shown in figure 3. Note that this optimization is *not* possible in `RS1c`, because an identity change is required – a dispatch table change alone is insufficient because we have to enter the `push` method and reference the `list` variable to invoke the appropriate list method.

Finally, representation `RSall` implements the bulk operation `pushAll` in terms of `push`. Since each representation only partially implements the `Stack<T>` interface, each must also assume `Stack<T>` as a basis. Multiple representation fragments must therefore be combined to yield a whole `Stack<T>`, and these in turn must be combined with some suitable representation of `List<T>` (not shown). We can now represent `Stack<T>` in three general ways:

- `RSc` $\oplus$ `RSall` $\oplus$ `rep(List<T>)`
- `RSi` $\oplus$ `RSall` $\oplus$ `rep(List<T>)`
- `RSf` $\oplus$ `RSall` $\oplus$ `rep(List<T>)`

where $\oplus$ indicates representation unification and the term `rep(List<T>)` requires further expansion, generating even more possibilities when alternative ways of representing `List<T>` are substituted for it. `RSall` is combinable with `RSc`, `RSi`, or `RSf` because its method constraints and base assumptions are unifiable with

```
abs Countable {
    Countable(int);
    int getCount();
}

abs Counter extends Countable {
    Counter(int);
    void incr(int);
}

abs CountableStack<T> extends Countable, Stack<T> {
    CountableStack<T>();
}
```

Figure 4: Abstractions for Countable Stacks.

theirs. If `RSall` had implemented `pushAll` in terms of something other than `push` it would not be unifiable with `RSc`, `RSi`, or `RSf`

Our representations of `Stack<T>` do not and should not care about which representations of `List<T>` are chosen, and similarly, the client of `Stack<T>` should not care which representation combination is used to represent `Stack<T>`. Thus, when a client invokes:

```
Stack<T> s = new Stack<T>();
```

the instantiation is interpreted non-deterministically – all the client expects is that some valid representation of `Stack<T>` will be used. Not only is the client freed from the burden of selecting a representation of `Stack<T>`, the client can also ignore the selection of representations for other subordinate abstractions. This is in stark contrast to existing approaches (such as mixins or parameterized components) that burden the client with the need to select and compose a mutually consistent set of implementation fragments. The closest approximation to our design would involve composition of a stack implementation in terms of a list, and the client would have to select specific implementations. The client invocation above would be transformed into:

```
Stack<T> s = new Stack_ListImpl<T>( new ArrayList<T>() );
```

A mixin-oriented approach would contain the same information, but the syntax would be different and composition would be replaced with inheritance.

To continue our example, suppose it is desired to provide a stack that keeps track of the number of elements it contains, a countable stack. We assume for the sake of argument that our previously mentioned abstractions for collections and lists do not already provide methods returning the size of the collection (which they don't, but in actual practice they might). In terms of abstractions, a countable stack might be defined by multiple interface inheritance in terms of `Countable` and `Stack<T>`, as shown in Figure 4.

First we notice that there is no syntactic connection between `Countable` and `Stack` other than their coexistence as bases of `CountableStack`. Some semantics must be established for the interface `CountableStack`, albeit by informal convention, not formally. For example, the combination of `Countable` and `Stack<T>` could mean just about anything – it could mean that calls to pop are counted, or that the total number of elements in the stack is counted, or something else entirely. Whatever it is, it is fixed by informal convention, perhaps captured in documentation, and all implementations of `CountableStack` are expected to follow this convention. The problem of formally specifying interfaces and verifying implementations is a problem that is desirable to solve, but orthogonal to our work.

Assuming we have decided that a `CountableStack` keeps track of the number of elements it contains, an incremental representation for it might look like representation `RCS`, shown in figure 5. Now suppose we

```
rep RCS represents CountableStack assumes Counter, Stack<T> {
    CountableStack<T>() assumes Stack<T>(), Counter(0) { }
    void pushAll(T) uses push(T);
    void push(T x) {
        super.push(x);
        incr(1);
    }
}
```

```
rep RCSX represents Stack<T> assumes List<T> {
    ...
    void pushAll(Collection<T> c) {
        for (T x in c) {
            addRear(c);
        }
    }
}
```

Figure 5: Incremental Representations of a Countable Stack

had defined another representation of the basic elements of `Stack<T>` that did not define `pushAll` in terms of `push`, such as `RSCX`, also shown in figure 5. This is a perfectly reasonable representation, but not one that can be combined with representations that depend on the method constraint `pushAll(Collection<T>) uses push(T)`.

The client requests a countable stack as follows:

```
CountableStack<T> cs = new CountableStack<>();
```

and the system instantiates a representation that has the following general form:

- `RCS` $\oplus$ `rep(Stack<T>)`$_{\texttt{pushAll(T) uses push(T)}}$ $\oplus$ `rep(Counter<T>)`

indicating that `RCS` may be combined with any representation of `Stack<T>` that also satisfies the constraint `pushAll(T) uses push(T)` and with any (unconstrained) representation of `Counter`.

3.5.2 A Graph Collaboration Example

Our next example further illustrates the use of partial representations, exploring not only how independent portions of a single object interface can be separately represented, but also how distinct participants in a multi-object collaboration can be decoupled. We sketch a set of abstractions and representations for graph collaborations, inspired by an example presented by Mezini and Ostermann [MO02], who proposed a technique for defining, extending, and composing collaborations in a way that enables on-demand remodularization. Their approach allows for definition and extension of nested interfaces and the corresponding nested classes that implement them. Our approach inverts the dependencies between collaboration and collaborators, thus avoiding nesting constructs and the need for dynamic lifting and lowering translations.

Mezini and Ostermann's example shows how two specialized forms of graph collaboration, a colored graph and a matched graph, could be defined as extensions to a base graph collaboration. Their implementation strategy for a colored graph demands that vertices in a graph be colored, and similarly their strategy for a matched graph demands that the edges of the graph be marked for matching. However, their language for expressing these strategies forces client interfaces to express these demands explicitly. Furthermore, to construct a graph instance, the client is responsible for explicitly selecting and combining implementation components that satisfy all the demands of the collaboration interface.

The problem with this approach is that it exposes details about a particular way of implementing the colored and matched graph collaborations to the client of the collaboration, and the client is forced to compose separate pieces and resolve interdependencies manually. In ARC, abstractions focus on what clients care about, and the underlying demands of an implementation are hidden expressed in an interface-oriented way, and resolved automatically.

We begin our version of the example with Figure 6, which shows the abstractions `Graph`, `Vertex`, and `Edge` comprising the foundation of our graph collaborations. Depending on how we choose to interpret the

```
abs Graph {
    Graph( Graph g );
    ...
    role Vertex;
    role Edge;
    ...
    int getNumVertices();
    int getNumEdges();
    ...
    control Iterator<role Edge> edges();
    control Iterator<role Vertex> vertices();
    control Iterator<role Vertex>
        adjacentVertices( role Vertex v );
    ...
}
```

```
abs Vertex within Graph {
    Vertex(int v);
    ...
    int getValue();
    control Iterator<role Edge> edges();
    ...
}

abs Edge within Graph {
    Edge(role Vertex v1, role Vertex v2);
    ...
    role Vertex getVertex1();
    role Vertex getVertex2();
    ...
}
```

Figure 6: Abstractions for a graph collaboration.

`Edge` abstraction, this collaboration could be used for either directed or undirected graphs. To keep things simple, we assume graphs contain integer values at the nodes. A more realistic approach would be to treat the value at each node as a type parameter, but this complication is unnecessary for our purposes.

A `Graph` defines two roles, `Vertex` and `Edge`. The role name is the same as the base abstraction name in this example, but in general a role can have its own name apart from the base name (by using the form `role BaseName [RoleName]`). The abstractions `Vertex` and `Edge` are declared to be `within Graph` to indicate their participation in a collaboration and provide a scope for occurrences of role types such as `role Vertex` and `role Edge` that are embedded within the collaborators themselves. This enables the types of `Vertex` and `Edge` instances to be anchored with instances of `Graph`, following the family polymorphism approach of Ernst [Ern01].

But it is not always necessary for collaborators to be coupled to their collaboration abstractions, as `Vertex` and `Edge` are coupled to `Graph`. It is only when collaborators need to refer to other collaborators that the need for an enclosing collaboration type appears. For example, we could have chosen to omit the `edges()` control method from `Vertex`, and could have instead placed an equivalent operation in `Graph`. Doing so would have eliminated the need to refer to `role Edge` within the `Vertex` abstraction, allowing the `within Graph` declaration to also be omitted from `Vertex`.

There is little in our base graph collaboration that constrains how a graph or its vertices and edges are represented. In particular, the existence of constructors for `Vertex` and `Edge` does not imply direct storage of identifiable vertices or edges, it only implies use of vertex and edge objects as a vehicle for communicating with the client. Also, this particular collaboration defines an immutable graph, as it provides no operations for adding or removing vertices or edges. Another significant and potentially complicated aspect of a real implementation is the method for initializing such a graph (without simply creating an empty graph and modifying it, which requires a mutable graph!). We assume there may be various ways of specifying the initial state of the graph via parameters to the graph constructor, and will rely here only on a simple constructor that accepts another graph to be copied.

Choosing a representation for a graph forces us to consider a number of questions, including:

- Is the graph mutable? If so, how are edges and vertices be added and removed, and is there any limit to the size of the graph? Are mutable graphs expected to be relatively stable, or highly dynamic;

```
abs IndexedVertex extends Vertex within Graph {
    IndexedVertex(int v);
    void setIndex(int index);
    int getIndex();
}

abs IndexedVertexGraph extends Graph {
    IndexedVertexGraph(Graph g);
    int getMaxVertices();
    role Vertex getVertex(int index);
    int getIndex(role Vertex v);
}
```

Figure 7: Abstractions for indexed vertices.

sparsely connected, or dense?

- How do we refer to edges and vertices in the graph? Are identifiable edges or vertices stored explicitly, or is their existence implied by some other data structure? Who controls the lifetime of separately identifiable edges or vertices, the client or the graph representation?
- How is connectivity maintained?
- What, if any, additional information is associated with edges and vertices?

Some of these questions are relevant to the client and some are not. The right answers will surely depend on the context in which the graph is to be used. Our goal is to attack these issues one piece at a time, enabling clients to select the weakest appropriate abstractions, and delegating the task of selecting and composing the right combination of pieces to some other agent in our system.

As for the question of how to refer to vertices, one approach that we might take is to assume that the vertices are indexed from $0..n$ such that internal data structures can be associated with vertices via that index, either by explicitly storing the index or via parallel arrays of information with the same index domain. We can express this dependency in an interface-oriented way with the `IndexedGraph` abstraction, as shown in Figure 7.

The representation `RIVG` shown in Figure 8 is a partial representation of `IndexedVertexGraph` that defines only those methods related to vertex indexing. Everything else, including the representation of connectivity, is assumed to be provided by some other partial implementation of `Graph`. It is also assumed that vertices themselves are represented as instances of `IndexedVertex`. (Note, this assumption is needed in `RIVG` but not in `IndexedVertexGraph`.) By separating the connectivity from the vertex indexing in this manner, we enable both to change independently.

To represent connectivity we could use an adjacency matrix, as shown in representation `RMatrixG` in Figure 9. The adjacency matrix approach depends on the assumption that vertices are indexed, so the representation must assume an implementation of `IndexedVertexGraph` for the graph itself, but the assumption that vertices are represented by `IndexedVertex` is not necessary. Alternatively, changing the call `getIndex(v)` to `v.getIndex()` would require assuming `IndexedVertex`.

Another approach to connectivity that may depend on vertex indexing would be to maintain an array of adjacency lists, where each list is implicitly associated with a particular vertex by its index in the array.

```
rep RIVG represents IndexedVertexGraph
         assumes Graph
         with role Vertex as IndexedVertex {

    int maxVertices;
    Array<role Vertex> va;

    IndexedVertexGraph( Graph g ) {
        maxVertices = g.getNumVertices();
        va = new Array<role Vertex>(maxVertices);
        for (int i = 0; i < maxVertices; i++) {
            va[i] = null;
        }
    }
    control Iterator<role Vertex> vertices() {
        for (int i = 0; i < maxVertices; i++) {
            if (va[i] != null) {
                yield next( va[i] );
            }
        }
    }
    int getMaxVertices() {
        return maxVertices;
    }
    role Vertex getVertex( int index ) {
        return va[index];
    }

    int getIndex( role Vertex v ) {
        return v.getIndex();
    }
}
```

Figure 8: Partial representation of a graph using indexed vertices.

```
rep RMatrixG represents Graph
        assumes IndexedVertexGraph
        // with role Vertex as IndexedVertex - not necessary
   {

    Matrix<boolean> matrix;

    Graph( Graph g ) assumes IndexedVertexGraph(g) {
        int max = getMaxVertices();
        matrix = new Matrix<boolean>(max,max);
        // initialize matrix ....
    }
    control Iterator<role Edge> edges() {
        int max = getMaxVertices();
        for (int i = 0; i < max; i++) {
          for (int j = i; j < max; j++) {
            if (matrix.get(i,j)) {
              yield next(new role Edge(getVertex(i),getVertex(j));
            }
          }
        }
    }
    control Iterator<role Vertex>
    adjacentVertices( role Vertex v ) {
        int index = getIndex(v); // instead of v.getIndex();
        int max = getMaxVertices();
        for (int j = 0; j < max; j++) {
          if (matrix.get(index,j)) {
            yield next(getVertex(j));
          }
        }
    }
}
```

Figure 9: Partial graph representation based on an adjacency matrix.

```
rep RLG represents Graph
        assumes IndexedVertexGraph
        // with role Vertex as IndexedVertex - not necessary
    {

    Array<MutableSeq<role Vertex>> lists;

    Graph( Graph g ) assumes IndexedVertexGraph(g) {
        int max = getMaxVertices();
        lists = new Array<MutableSeq<role Vertex>>(max);
        // initialize lists ....
    }
    control Iterator<role Edge> edges() {
        int max = getMaxVertices();
        for (int i = 0; i < max; i++) {
          Seq<role Vertex> adjacencyList = lists.get(i);
          if (adjacencyList != null) {
            for (role Vertex v in adjacencyList) {
              yield next(new role Edge(getVertex(i),getVertex(j));
            }
          }
        }
    }
    control Iterator<role Vertex>
    adjacentVertices( role Vertex v ) {
        int index = getIndex(v);
        int max = getMaxVertices();
        for (int j = 0; j < max; j++) {
          if (matrix.get(index,j)) {
            yield next(getVertex(j));
          }
        }
    }
}
```

Figure 10: Partial graph representation based on adjacency lists.

This approach is illustrated by representation `RLG` in Figure 10.

Note that both `RMatrixG` and `RLG` represent just the connectivity portion of a basic `Graph`, not an `IndexedVertexGraph`, although internally they both assume an `IndexedVertexGraph` as a foundation. The reason is that both `RMatrixG` and `RLG` represent methods that occur in `Graph`, but the implementation of these methods depends on methods that occur in `IndexedVertexGraph`.

Other approaches to vertex identification and connectivity are possible – this discussion is not meant to exhaust the possibilities. Our intent is to show how independent implementation strategies affect different portions of the interface being implemented, and to show how they can be captured and expressed with minimal interdependence. The advantages in expressiveness are similar to those of abstract base classes or mixins, but without the tight coupling and feature combinatorics problems they create.

The next challenge we will face in this section is to extend the graph collaboration to be able to handle graph coloring and matching. We show how to create independent extensions for colored vertices and matched edges, which are then combined to yield a graph that supports coloring and matching.

Our interfaces for the colored graph collaboration, shown in Figure 11, differ from Mezini and Ostermann's example by addressing only the needs of the client. In our design, for example, a client of `ColoredGraph`

```
abs ColoredGraph extends Graph {
    ColoredGraph(Graph g);
    void computeMinColoring();
    int getColor(role Vertex v);
}

abs ColoredVertex extends Vertex within Graph {
    ColoredVertex(int v,int c);
    int getColor();
    void setColor(int c);
}
```

Figure 11: Colored graph abstractions.

only needs to request that the graph be colored and to retrieve the color of each vertex. To enable a client to modify the color of a vertex by exposing a `setColor(c)` operation would violate this abstraction. Also, the `ColoredVertex` abstraction is declared to be `within Graph`, not `within ColoredGraph`, reflecting that `ColoredVertex` does not depend on `ColoredGraph`.

In fact, analogous to the situation above with `IndexedVertex`, the need for a `ColoredVertex` abstraction does not appear until we implement the `ColoredGraph` abstraction, as illustrated by representation `RCG` shown in Figure 12. `RCG` makes no assumptions about connectivity (it just assumes `Graph`) but does assume that vertices are colorable via the `ColoredVertex` abstraction. This dependence on `ColoredVertex` is not visible to clients of `ColoredGraph`. Hiding this decision is appropriate, primarily because clients should not alter colors, but also because there are other reasonable ways to represent color, for example, by storing colors in a parallel structure that is indexed by vertex index. (Such an approach, of course, would depend on the indexed vertex assumption.) Note that the constructor interface to `ColoredVertex` is not identical to the constructor signature of `Vertex`, but somehow a `ColoredVertex` must be constructed whenever an instance of `role Vertex` is created in this collaboration. To accomplish this, a representation of `ColoredVertex`, because it is implementing a role within a collaboration, must implement the constructor defined in the role's base abstraction. Calls of the form:

```
role Vertex v = new role Vertex(3);
```

effectively represent type-nondeterministic, virtual constructor calls. Representation `RCV` does this, defining a constructor for `Vertex(int)` that assumes a call to `ColoredVertex(int,int)`. Note also that the implementation of the `ColoredVertex(int,int)` constructor must explicitly assume the `super` version of `Vertex(int)` to avoid referring back to its own locally defined version.

Figure 13 shows our matched graph collaboration, `MatchedGraph`, which extends the base graph collaboration with the interface needed by graph matching clients. Similar to graph coloring, our design for graph matching only allows a client to invoke the matching algorithm and then be able to retrieve whether or not a given edge is a matched edge.

The representation `RMatchedG` shown in Figure 14 makes no special assumptions about the connectivity of the graph but does assume that edges provide the `MatchedEdge` interface, an assumption that is again hidden from the clients of `MatchedGraph`. As before, the `RME` representation must provide a constructor for the `Edge` role in order to satisfy type-nondeterministic constructor calls of the form `new role Edge(v1,v2)`.

```
rep RCG represents ColoredGraph
        assumes Graph
        with role Vertex as ColoredVertex {

    ...

    ColoredGraph( Graph g ) assumes Graph(g) {
        ...
    }
    void computeMinColoring() {
        ... v.setColor(c); ...
    }
    int getColor( role Vertex v ) {
        return v.getColor();
    }
}

rep RCV represents ColoredVertex
                   within Graph
                   assumes Vertex {

    int color;

    Vertex(int v) assumes ColoredVertex(v,0) { }

    ColoredVertex(int v, int c) assumes super.Vertex(v) {
        color = c;
    }
    void setColor( int c ) {
        color = c;
    }
    int getColor() {
        return color;
    }
}
```

Figure 12: Partial representation for a colored graph.

```
abs MatchedGraph extends Graph {
    MatchedGraph(Graph g);
    void computeMinMatching();
    boolean isMatched(role Edge e);
}

abs MatchedEdge extends Edge within Graph {
    MatchedEdge(role Vertex v1, role Vertex v2,boolean matched);
    void setMatched(boolean m);
    boolean isMatched();
}
```

Figure 13: Matched graph abstractions.

```
rep RMatchedG represents MatchedGraph
        assumes Graph
        with role Edge as MatchedEdge {
    ...
    MatchedGraph( Graph g ) assumes Graph(g) {
       ...
    }
    void computeMinMatching() {
        ... e.setMatched(bool); ...
    }
    boolean isMatched( role Edge e ) {
        return e.isMatched();
    }
}

rep RME represents MatchedEdge
        within Graph
        assumes Edge {

    boolean matched;

    Edge(role Vertex v1, role Vertex v2) assumes MatchedEdge(v1,v2,false) { }

    MatchedEdge(role Vertex v1, role Vertex v2, boolean m) assumes super.Edge(v1,v2) {
        matched = m;
    }
    void setMatched( boolean b ) {
        matched = b;
    }
    boolean isMatched() {
        return b;
    }
}
```

Figure 14: Partial representations for matched graph abstractions.

```
abs ColoredMatchedGraph extends ColoredGraph, MatchedGraph {
    ColoredMatchedGraph(Graph g);
}

rep RCMG represents ColoredMatchedGraph assumes ColoredGraph, MatchedGraph {
    ColoredMatchedGraph(Graph g) assumes ColoredGraph(g), MatchedGraph(g) {
        ;
    }
}
```

Figure 15: Composition of multiple graph collaboration abstractions.

But there is another problem. Earlier we indicated that the existence of the `Vertex` and `Edge` abstractions did not imply direct storage, so the identity of a vertex object received through the interface did not necessarily correspond to the identity of an actual vertex in the graph, and similarly for edges. However, the dependencies between `ColoredGraph` and `ColoredVertex`, and between `MatchedGraph` and `MatchedEdge`, appear to imply such assumptions. Both the coloring and matching operations invoke operations on vertices and edges, the effects of which are expected to be shared by the graph and persist. Not capturing these assumptions explicitly is a design error, an error that must be repaired by capturing these assumptions in a way that guarantees vertices and edges provided to the client are in some sense identifiable with those stored in the graph, so that any changes made to or observed through a client's instance of a vertex or edge is consistent with the graph's notion.

For the case of vertices, one way to accomplish this would be to introduce `IdentifiableVertexGraph`, a new abstraction that extends `Graph` and acts as a carrier for this concept.[3] Realizing that the `RIVG` actually provides identifiable vertices as a side effect of indexing, `RIVG` could be modified to represent `IdentifiableVertexGraph` in addition to representing `IndexedVertexGraph` (there are other ways to provide identifiability, this is just one). Finally, `RCG` should also assume `IdentifiableVertexGraph` so that its modifications to vertices persist.

For edges we could similarly introduce an `IdentifiableEdgeGraph` abstraction. Unfortunately, in this case we have a little more work to do, because although `RMatchedG` comes close, no representation we have defined so far provides identifiable edges. We would have to modify `RMG` or create something new to solve this problem, and then modify `RMatchedG` to assume `IdenifiableEdgeGraph`.

Finally, assuming we have resolved the identifiability problems and have a sufficiently large pool of partial representations to produce at least one complete representation, we now would like to create and use a graph that supports coloring and matching. As clients of the `ColoredGraph` and `MatchedGraph` abstractions we have two alternatives for doing this. We could define a new named abstraction `ColoredMatchedGraph`, whose purpose is merely to compose two abstractions into one, along with an associated representation `RCG` that just composes the constructors of the two assumed bases, as shown in figure 15). Alternatively, a more direct approach inspired by compound types of Büchi and Weck [BW98] is to simply compose them on the fly. Figure 16 illustrates this latter technique, also showing how a client might use the abstraction after an instance is created.

[3] An abstract property would be a better way to do this.

```
Graph g = ... ;
final [ColoredGraph,MatchedGraph] cmg =
    new [ColoredGraph(g),MatchedGraph(g)];
...
cmg.computeColoring();
cmg.computeMinMatching();
...
for (cmg.Vertex v in cmg.vertices()) {
    int color = v.getColor();
    ...
}
...
for (cmg.Edge e in cmg.edges()) {
    if (e.isMatched()) {
        ...
    }
    ...
}
```

Figure 16: Composition and use of multiple graph collaboration abstractions.

3.5.3 Avoiding Implementation Bias

In this section we show how ARC eliminates each form of implementation bias that was introduced in section 2.1. The most basic form of bias is the need to select an implementation when dynamically creating an object of some abstraction, a choice no longer necessary due to ARC's support for abstraction constructors and non-deterministic instantiation:

```
void foo() {
    A a = new A();
}
```

Explicitly programmed factories are not needed – in essence, ARC provides a built-in factory mechanism for all abstractions.

The need to use composition, as well as the pollution of client interfaces with implementation-specific details are both eliminated. Representation `C_impl`, for example, implements `A`, `B`, and `C` in terms of `AX` and `BX`, but uses inheritance that is not visible to the client (in particular, the implementation types `AX` and `BX` do not appear as client-visible arguments in the constructor as they were in section 2.1):

```
rep C_impl represents A, B, C assumes AX, BX {
    C() assumes AX(), BX() {
       ...
    }
    void cMethod() {
        ... // local implementation, may self-call AX, BX
    }
}
...
C c = new C();
```

The use of inheritance, even though private, eliminates the need to forward calls at run-time to `aMethod` and `bMethod`, because the versions inherited via `AX` (which is derived from `A`) and `BX` (which is derived from `B`) are used automatically. The assumed bases and automatic composition essentially provide mixin-inheritance but with automatic mixin-composition, and mixin assumptions that have nothing to do with the interface being represented are invisible to clients.

ARC also supports incremental implementation of interfaces - it is not necessary for a representation to implement all of the methods of a represented abstraction, hence the term "partial representation." We could, for example, implement the methods of abstraction `W` from section 2.1 individually.

```
rep W_impl1 represents W assumes W {
    void w1() { ... }
}
rep W_impl2 represents W assumes W {
    void w2() { ... }
}
...
W w = new W();
```

Again, the structure of the implementation of `W` is invisible to the client, and the system (ARC's representation inference and non-deterministic instantiation mechanisms) will assemble the partial representations into complete representations and then select one for us.

4 Related Work

Although introduced decades ago, the principles of stepwise refinement [Wir71], structured programming [Dij72], separation of concerns [Par72], and data abstraction [Gut77] are still fundamentals of software engineering practice today. Much of the evolution of programming languages, such as object-orientation, component-orientation, and aspect-orientation can be explained as attempts to equip programming languages with better support for these fundamental ideas, and interface-oriented programming aims to continue the trend. In this section we explore how interface-orientation in general and the ARC programming language in particular relate to work in the following areas:

- Programming language mechanisms for interface abstraction and inheritance;
- Language support for collaborations, roles, and layered design;
- Programming techniques which support interface abstraction;
- Advanced separation of concerns;
- Component generators and generative programming; and
- Distributed component technology and middleware.

This report is part of the author's dissertation research and is an extension and refinement of ideas originally presented in [Var03].

4.1 Interfaces, Implementations, and Inheritance

Two important but controversial concepts in program development are interfaces and inheritance. Burstall and Lampson [BL84] suggested long ago that "the most important recent development in programming languages is the introduction of an explicit notion of *interface* to stand between the implementation of an abstraction and its clients." Object-orientation and one of its defining mechanisms, inheritance, were once viewed as a keys to incremental development of reusable software. However, interfaces have only recently received first class status in mainstream programming languages, and inheritance as an incremental programming technique remains implementation-oriented and controversial.

ARC provides a new level of interface abstraction that decouples the inheritance mechanisms from the implementation binding mechanisms, providing a clean solution to some long-standing language design issues. In this section we explore how our design relates to previous work in several areas, including separation of interfaces from implementations, subtyping vs. subclassing, multiple inheritance, and mixin inheritance.

Separation of Interfaces and Implementations A fundamental principle of IOP is that interfaces and implementations should be treated as separate concerns. It is generally accepted that this separation should exist, and that interface inheritance or subtyping is distinct from implementation inheritance or code reuse ([Sny86], [AvdL90], [CCHO89], [CP89], [CHC90],[Coo92]). It is also agreed that implementation specializers and implementation clients demand distinct interfaces ([KL92], [Lam93]). Despite these agreements in principle, interfaces and implementations are not strongly separated in programming languages of today.

One form of weak separation occurs in languages without first class interfaces, such as C++ [Str87a]. In C++, interfaces are implicit in the design of classes, not separately identifiable. The other form of weak separation occurs when separately identifiable classes and interfaces are nevertheless merged into one type

lattice, as they are in Java [AGH00] and C# [ADM01]. The separate inheritance constructs (`implements` vs. `extends`) for interfaces and implementations in these languages present an illusion of separation, but classes and interfaces are intertwined in one type hierarchy, and implementation inheritance implies subtyping for clients and specializers alike. Identifying subtyping with subclassing leads either to logical problems (attributing to an object a type that it should not expose to its clients), code replication (copying implementations instead of inheriting them to avoid incorrect typing), or the inefficiency of method forwarding (in order to mediate access to an encapsulated implementation object). It should be noted that C++ [Str87a] supports private inheritance which can be used to inherit implementations without creating a subtype relationship, but this relationship is private only to the programmer, not the compiler.

ARC strengthens the separation of interfaces from implementations by providing two distinct channels of inheritance, one for abstractions (interface inheritance, or subtyping) and one for representations (implementation inheritance, or subclassing). Furthermore, ARC uses *interface-oriented* inheritance in each channel and clients of an abstraction see only the interfaces intended for clients. Subtyping in ARC is a combination of nominal and structural approaches - it is nominal at the level of abstractions which define semantic extensions to their base, and structural above that level, where the structural elements are abstraction names, not method signatures. This means that if an abstraction extends its base abstraction in any way, another abstraction must extend that abstraction by name in order for it to be considered a subtype. There is no structural conformance at the level of method signatures in ARC.

Even when interfaces are separated from implementations, the question of whether or not subclassing should preserve subtyping can still be asked. Although some have argued that implementation inheritance should be allowed to violate subtyping [CHC90], it is not necessary for it to do so, and we find it useful for subtyping relative to private base assumptions to be preserved privately in the context where those assumptions are visible. Interface-oriented inheritance of implementation is encapsulated in ARC – the type of `this` is a subtype of both the represented interfaces and the assumed interfaces in the context of the representation of `this`, but outside of its representation an object is only a subtype of the represented abstraction in use by the object's client. It is an open issue of this research whether it would be beneficial to allow subtyping to be violated within representations, and if so, how it could be handled in ARC.

Emerald [BHJL86, RTL$^+$91] strictly separates types from implementations, and subtyping is structural. There is no connection between types and implementations other than structural conformance of their signatures. Jade is a programming language environment based on Emerald with extensions for code sharing. Jade provides a way to define partial implementations which may assume the presence of other implementation fragments. The signature of the assumed fragments, called the habitat, provides a low-level form of interface-oriented reuse. However, the programmer is responsible for composing implementation fragments together with the use construct, and instantiation of objects requires explicit mention of the implementation (or composition) to be instantiated. ARC's interface-oriented composition together with representation inference relieve the programmer of the burden of selecting a particular combination of consistent implementations.

POOL-I [AvdL90] also provides strict separation of types from implementations. Subtyping in POOL-I is a combination of signature conformance (structural subtyping) and property conformance. POOL-I's property identifiers act as abbreviations for formal behavioral specifications, much like type names in a nominally typed system such as ARC. POOL-I's inheritance for code reuse is implementation oriented - to reuse code, the specific implementation must be identified. Also, when objects are created, the programmer must name the implementation to instantiate, resulting in implementation bias.

ARC's inheritance approach yields three interface-oriented relations – interface extension, interface representation, and interface assumption. Interface extension relates interfaces to interfaces in a subtype lattice, interface representation relates interfaces to partial representations which implement them, and interface assumption relates representations to the interfaces they depend on for implementation reuse. Thus, as

suggested by Snyder [Sny86], inheritance for implementation purposes is a private decision in ARC.

Multiple Inheritance Multiple inheritance has somehow acquired a bad reputation in programming language design. Many early object-oriented languages provided it, but more recent languages have avoided it. Java and C# allow a restricted form of multiple inheritance for interfaces but only support single inheritance for implementations. C++, the predecessor to Java and C#, allowed multiple inheritance [Str87a]. The current status of multiple inheritance is perhaps due to difficulty in using and implementing multiple inheritance constructs. In our view, the root cause of this difficulty has been the lack of appropriate separation of implementation and interface.

ARC supports multiple inheritance for both abstractions and representations. In both cases, inheritance is interface-oriented. The semantics of multiple inheritance in ARC essentially takes the union of abstractions, and implementation fragments are not assigned to interface fragments until after this union is taken, so there is no diamond inheritance problem. Viewed at the level of method signatures, the semantics of multiple inheritance can be thought of as the disjoint union of the method signatures, with abstraction names acting as the entities that keep identical method signatures apart in the union.

Although the subtype relation in Java is nominal, the names of interface members may still conflict across two differently named interfaces. This means that an implementation may only provide one implementation for that name, and it will be used in both interface contexts. This problem does not occur in ARC because the same method signature in two different abstraction refers to two different methods. The `extends` clause of Java and C# only supports single inheritance of other classes, so name conflicts are vertical and automatically imply overriding.

Mixin Inheritance ARC's use of interface inheritance for implementation reuse is similar to mixin-inheritance as described in [BC90]. In a class-based model of mixin-inheritance, a mixin can be thought of as a class-to-class function – composing a mixin with a class argument yields another class. Mixins are defined in terms of abstract base class parameters, and composing a mixin with a specific base class argument creates a new class with the additional features of the mixin. Two proposals for adding mixin support to Java include MIXEDJAVA [FKF98] and Jam [ALZ00]. MIXEDJAVA is a theoretical language and Jam is an upward-compatible extension to Java. A number of other mixin-like approaches have been defined but will be discussed in a subsequent section.

In MIXEDJAVA, mixins inherit one or more base interfaces by name. Base interfaces are instantiated by composing the mixin with classes that implement the interfaces. A mixin must be explicitly composed together with specific classes that implement its inherited interfaces. If a mixin composition has any base interfaces with unspecified implementations, it cannot be used to create a new object; it must be composed further. Mixins in MIXEDJAVA may also be composed with other mixins to create new mixins. Constructors are not supported in MIXEDJAVA, so mixins are forced to rely on a simplistic form of default initialization.

In contrast, mixins in Jam inherit individual features of unspecified classes, they do not inherit named interfaces. Jam mixins must also be explicitly composed such that all features are resolved in order to be instantiable. Mixins in Jam may not be composed with other mixins, and classes do not need to implement any particular interfaces, they only need to provide the individual features inherited by the mixin. Inheritable features of classes include instance variables and methods, and mixins are allowed to access all features of their base classes. Constructors cannot be declared in Jam mixins, but are possible in mixin composition expressions.

The mixins of MIXEDJAVA and Jam are similar to ARC's representations in the sense that they all assume implementations indirectly (in MIXEDJAVA and ARC via inherited interfaces, and in Jam via

inherited features). Furthermore, although mixins assume interfaces, they typically do not implement them, so mixins themselves are implementations like classes, and must be explicitly named to be used. With mixins, as with classes and interfaces in Java and C#, the type and implementation hierarchies are mixed together. This forces the logical type hierarchy for clients to coincide with the implementation hierarchy used by specializers. Mixins must also be explicitly composed with specific base implementations to bind inheritance dependencies. If a mixin has any base interfaces or inherited features with unspecified implementations, it cannot be used to create a new object, and must be composed further. Constructor support is also limited with mixins, and the structural inheritance of MIXEDJAVA and Jam can lead to unexpected overriding problems or multiple inheritance ambiguities.

While the mixin approach demands explicit composition of implementation fragments in terms of named mixins and classes, ARC's representation inference and selection approach allows partial representations to be selected and composed automatically based on abstract interface references. ARC representations also support constructors, and do not suffer from unexpected overriding problems or ambiguous multiple inheritance.

4.2 Programming Techniques

Various programming techniques and forms of run-time support have also been proposed to decouple the client of an abstract interface from the implementation of that interface, or to decouple collaborating implementations from each other. In this section we discuss a few of these techniques, including factories, service facilities, exemplars, encapsulated class hierarchies, and mediators.

Factories and Service Facilities The Factory Method and Abstract Factory patterns [GHJV95] are simple techniques for encapsulating the implementation name behind some other interface, but as shown previously this merely transfers the implementation dependency to the factory, and the factory itself must still be instantiated somehow. Service facilities [SWB02] aim to improve on abstract factories by encapsulating access to an object even after it is created, but this approach is statically unsound and complicates inheritance. ARC provides a sound form of indirect instantiation which is integrated with its inheritance mechanisms.

Exemplars and Encapsulated Class Hierarchies More complicated and powerful forms of the factory approach include Coplien's exemplar idiom [Cop92], Lortz and Shin's exemplar idiom [LS94], and Riehle's encapsulated class hierarchies [Rie95]. Each of these approaches selects an implementation at run-time by searching a list of candidate implementations until one is found that matches the abstract properties specified by the client. These approaches enable retrieval of complete classes based on matching of simple properties, insulating the client from dependence on specific implementations, but are limited to selecting from a fixed set of complete representations that have been manually predetermined. There is no support for using abstract properties in the derivation of new classes, nor is there any way for these factories to automatically compose compound implementations out of partial implementation fragments.

In contrast, ARC automatically infers complete representations by composing separately defined, partial representations. Also, despite allowing representations to be selected as late as run-time, ARC provides a static guarantee through its role construct and assumed implementation constraints that incompatible partial representations will not be combined.

Mediators Sullivan and Notkin [SN92] use mediators to directly represent integration relationships between other components. Integration relationships are captured in mediators, separate from the components that are integrated. The binding between tool components and mediators is accomplished by implicit invocation, while still encapsulating design decisions within components. They show how this approach improves the evolvability of components using a graph collaboration example.

An important contribution of their work is the realization that the events produced by a component for consumption by the components users can be considered part of the component's type. They use the term *abstract behavioral types* (ABT) to distinguish their approach from abstract data types (ADTs). An ABT is an ADT which also includes interface declarations for exported events.

4.3 Collaborations, Roles, and Layers

Collaborations are groups of cooperating objects, with each object playing a defined role in the collaboration. Most object oriented languages today support some form of class nesting but do not provide adequate first class support for collaborations or collaboration types, where multiple object instances need to share implementation details. This section discusses some of the programming techniques and language extensions which have been proposed to support collaborations and layered design, including role components, feature-oriented programming, mixin layers, adaptive plug-and-play components, pluggable composite adapters, Java layers, family polymorphism, on-demand remodularization, traits, dynamic component adaptation, and object teams.

Role Components Van Hilst and Notkin's role components [VN96a, VN96b, VN96c] are mixins implemented as C++ class templates. Template parameters for mixin base classes are untyped so the implementation of a role component cannot be type-checked against its base class until the component is composed with another role component. This can lead to obscure compile-time or run-time errors which are difficult to track down. Role components also force the programmer to compose a specific set of implementations.

Feature-Oriented Programming Feature-oriented programming [Pre97] is a mixin-like approach that separates the feature composition logic (the lifters) from a feature's core functionality. Features are like classes that implement interfaces, but the implementation is split into two or more parts, one for the core functionality, and one for each of the different pairwise feature interactions. Features may also assume the presence of other features in an interface-oriented way, which is similar to the interface-oriented implementation inheritance of ARC, except that the use of features for implementation purposes is exposed in the public interface of the composed feature, as are instance variables. Features are composed in terms of abstract feature names, but each feature name is bound to one implementation of core functionality, and each ordered feature interaction pair is bound to one lifter implementation.

In contrast, ARC uses interfaces for implementation inheritance but strictly separates what is visible in the client's interface from what is used internally for implementation purposes, and supports multiple, separately-defined, partial implementations for each interface. Also, in general we do not want to expose all elements of all features in the composition, because doing so would violate encapsulation. Feature-oriented programming exposes the full interface to all composed features in the composition even though some features are used only for implementation purposes and are not intended for direct use by the client. The feature-oriented programming approach also does not handle interactions across more than two features at a time, which ARC can handle with multiple assumed base abstractions.

Mixin Layers Smaragdakis and Batory [SB98] proposed mixin layers as a technique for implementing layered collaboration-based designs, using a parameterized set of nested classes (in C++) to describe a collaboration between a set of classes. This technique improves on the technique of VanHilst and Notkin by replacing multiple role-based superclass parameters with a single superclass parameter for the base collaboration. Mixin layers therefore represent mixins (collaborations) which encapsulate other mixins (roles), and this has the effect of simplifying the composition of collaborations. Nevertheless, mixin layers must still be composed manually and in implementation-oriented terms.

Adaptive Plug-and-Play Components Adaptive Plug-and-Play Components (APPCs) [ML98] are implementations of collaborative behavior that can be reused with a range of concrete participant behaviors. The design of an APPC depends only on the interfaces to the participants. APPCs use traversal strategy graph technology [LPS97] to achieve structural genericity, and use variational programming to interfaces [Mez97] to decouple the implementation of the collaboration from that of the participants. An APPC consists of a definition of an interface class graph (ICG) and a behavior definition. The ICG includes a structural interface part that defines the participant roles and any patterns of relationships between them, and a behavioral interface part that provides the signature of operations required of each participant. The behavior definition defines the behavior for all of the participants, with each participant's behavior definition resembling a nested class definition with attribute and method definitions. Behaviors may be defined in terms of traversal strategies defined on interfaces in the ICG.

Instantiating an APPC requires that the roles and operation names be explicitly mapped to concrete implementations in the instantiation. Relations between participants have to be resolved by explicit mapping of edges in the ICG to strategies over the concrete class graph (CCG) of actual participants in the instantiation. The result of the instantiation is generated code, but there is no object-oriented organization to this code. APPCs are also like mixins in that they may extend abstract base APPCs and be composed with other concrete APPCs.

Pluggable Composite Adapters Pluggable Composite Adapters (PCAs) [MSL01] separate customization code from component implementation to provide non-invasive, encapsulated, and dynamic customization. Mezini's adapters are implemented via a class-like `adapter` extension to Java that implements one interface in terms of another. They are pluggable because they have the mixin-like property, the ability to be combined with different base class implementations. Adapters are composite when an `adapter` construct includes nested `adapter` constructs. Composite adapters support collaborations by allowing a related set of interface classes to be adapted to a related set of implementation classes. Finally, the adapters are dynamic because they can be plugged into a base object dynamically and the adaptation mechanism dynamically provides type-lifting and type-lowering to prevent wrapper identity problems [H93] – base objects are lifted to adapter objects, and adapter objects are lowered to base objects automatically when needed. Mezini's implementation of PCAs does not support overriding, although it is suggested that this could be added.

Java Layers Cardone et al. ([CL01],[CBML02]) describe Java Layers (JL), an extension to Java which provides the ability to encapsulate design features in layers and compose layers to form components. Like other mixin-based approaches, JL requires that specific mixin implementations be manually composed. However, unlike mixin layers which are implemented in terms of templates, JL provides language constructs and can deliver better error messages and optimizations, and also supports constructor propagation.

Family Polymorhphism Family polymorphism [Ern01] is a method for defining collaborations while providing a statically safe form of covariance. Nested classes are defined and extended in a way that allows the compiler to ensure that objects from different families are not combined. Types that play roles in the collaborations are treated as virtual types relative to some enclosing object, but safe use of these types generally requires that references to the enclosing object be declared `final`. This general approach is also adopted in ARC's collaborations, but family membership in ARC is more loosely defined in a way that avoids the feature combinatorics problem. Family members (abstractions which play roles in other abstractions) may be defined separately from the collaborations themselves, and multiple implementations which assume different relationships with other family members are possible. Also, Ernst's approach does not separate interface from implementation.

Component-Oriented Mixins Sreedhar [Sre02] proposes a component extension language with support for mixin classes, called ACOEL (A Component-Oriented Extension Language). The external interface to ACOEL components consists of typed ports which must be explicitly bound with object instances, and the internal implementation consists of classes, methods, and fields. ACOEL is intended as an independent extension of other object-oriented languages such as C++, Java, or C#.

A component in ACOEL is similar to a nested class, and ports can be thought of as special instance variables of the top-level component (class). However, ACOEL treats classes as "second-class citizens," as classes defined within a component do not exist beyond the component, and cannot be extended by classes in other components. A subtype relation is defined between components based on the types of the ports, and a component may explicitly extend one other component. An extension component inherits all of the ports of the base component but may not directly access any of the other implementation classes or methods of that base. Two components that do not directly extend each other but have the appropriate relationship in terms of port types may also be explicitly declared to be subtypes.

Mixins in ACOEL are treated as type decorators rather than first class types, and implement exactly one interface. They can be passed as input parameters to components which may compose them with other classes to create mixin classes, and mixins themselves may be defined in terms of component parameters that will be instantiated with the containing component. Mixins cannot be directly instantiated, and may not contain constructors.

On-Demand Remodularization and Caesar Many component models involve the definition of provided and required interfaces where the provided interface is what the component provides to its clients, and the required interface is what the component requires from its run-time environment or context (e.g., [CL01], [MO02]). We refer to this model as the *dual-ported component* model. On-demand remodularization [TOHS99] refers to the dynamic mapping of the external requirements of a component (what a component itself needs to rely on from outside) to the environment in which the component is to be used. Mezini and Ostermann [MO02] describe a method for on-demand remodularization based on collaboration interfaces. Their method is a dual-ported component approach which incorporates family polymorphism. Interfaces and implementations are separated in a manner equivalent to Java, but in Mezini and Ostermann's method both interfaces and classes may be nested, and components expose collaboration interfaces which define provided and expected interfaces. Mezini and Ostermann's latest proposal called Caesar [MO03] incorporates aspect collaboration interfaces, an extension of collaboration interfaces with better support for aspect-orientation.

One difference between ARC and all of the dual-ported component approaches including on-demand remodularization is that dual-ported components expose an implementation dependency in their public interface. The exposure of the binding interface is an implementation bias that is not logically required by

all implementations of the component interface – if it were, its methods could be located in the component interface. The approach to defining multiple nested classes with priviledged access both encourages violation of encapsulation across classes participating in a collaboration, and makes it difficult to reuse implementation fragments in different collaborations. Finally, the approach does not allow constructors for the component implementation (only the binding implementation may use them), it requires manual composition of implementations before a client can use the component interface, and it suffers from the need for wrapper recycling.

In ARC, implementation dependencies such as those expressed by Mezini and Ostermann's binding interface are hidden via assumed interfaces. It may also be the case that there are several logically distinct internal interfaces that need to be satisfied and should not be collected in one interface. Furthermore, implementation dependencies are captured at the representation fragment level via interface assumptions, not by priviledged access across representations. Interfaces are bound to implementations automatically in a context-dependent manner, not via manual composition and selection. All abstractions in ARC may define constructors, regardless of whether they are used in a component implementation or binding implementation role. Indeed, the same abstraction might be used in both roles in different cases, and in general the abstraction designer should not have to be aware of this distinction. Furthermore, constructors are defined incrementally, support arguments, and are automatically blended together when multiple representation fragments are composed. There is no need for automatically generated wrappers or wrapper recycling in ARC as there is in Mezini and Ostermann's method.

Traits Traits [SDNB03] represent another dual-ported technique for implementation reuse. Traits are fragments of code with provided and required methods but no state, and are composed together with instance variables, glue code, and other traits to form classes.

The diamond inheritance problem is handled in traits in a way similar to ARC's handling. Two occurrences of the same entity (method or abstraction) inherited via multiple paths are treated as one occurrence. Schärle et al. claim that the diamond problem doesn't occur because traits do not define state, but in our view the issue is the semantic interpretation of inheritance, and *when* state is assigned to elements of the inheritance lattice. Representations in ARC define state, yet the diamond problem does not exist because state (representations) are assigned once to each element of the merged inheritance lattice. The diamond inheritance problem occurs in other approaches to multiple inheritance when the same interface is assigned multiple representations that must then be merged after state is assigned, which creates conflicts with two or more implementations of the same interface fragment.

Traits do not themselves separate interface from implementation, they are a means for reuse in the implementation domain only. For example, traits do not implement named interfaces, trait composition is implementation-oriented as opposed to interface-oriented, and composition of traits can lead to structural conflicts (two method implementations with the same name but inherited from different traits). The closest situation in ARC would be the assumption of multiple interfaces. But in this case, two occurrences of the same method name in different abstractions are treated as different entities, both present in the composition, and each with its own (unspecified) implementation. ARC requires fully qualified names to be used to invoke such methods, because semantically they denote distinct behaviors.

As with other dual-ported approaches, traits expose implementation bias via their required methods. Changes to required methods can propagate across multiple levels of composition, but it is possible to encapsulate these changes after they are made by addressing them in direct users of the trait that changed.

ARC's approach is to encourage collaborators to minimize their interdependencies and to capture them via interfaces. The representations that participate in a collaboration may still have more detailed knowl-

edge about each other than a client of the collaboration has about any one of them, but this knowledge is expressed in ARC in the form of hidden interface assumptions between the representations of the collaborators, and collaborators are grouped automatically based on consistency of assumptions. This allows different alternatives to be used for each collaborator, provided that all the assumptions are satisfied, and avoids the feature combinatorics problem that occurs whenever multiple collaborators must be specified in one place.

Dynamic Composition and Adaptation In this section we briefly mention additional proposals for dynamic composition and adaptation. These methods typically intercept and filter methods as a way to adapt behavior dynamically, or modify dispatch tables to achieve the same effect. Examples include type adaptation [HÖ93], composition filters [AWB+93], context relations [SPL96], adaptive components [BMN+00], ionic types [DM00], generic wrappers [BW00], implicit context and contextual dispatch [WM00], Lasagne's dynamic composition of extensions [TVJ+01], unanticipated dynamic adaptation [RC02], delegation layers [Ost02], and prototype-based component evolution [Zen02].

Many of these approaches require anticipation and explicit programming of the adaptive behavior, impose significant run-time costs, suffer from wrapper identity problems, or may violate encapsulation. Although ARC does not currently provide dynamic modification of inferred representations, it is conceivable that it could do so in an interface-oriented way. A more detailed assessment of ARC with respect to these dynamic methods is left to future work.

Object Teams Veit and Herrmann [Her03][VH03] propose object teams as a means for designing object collaborations using instance-anchored types as in Ernst's family polymorphism and automatic lifting and lowering as in Mezini and Ostermann's on-demand remodularization. Object teams distinguish type-based visibility from instance-based scoping.

An object team groups a set of role objects, much like the nesting of inner classes in Java, but an object team combines properties of a class with properties of a package. Inheritance between teams establishes inheritance between roles. A limited form of deferred creation is supported in which an abstract role class may be instantiated. The team containing such an instantiation must be marked abstract, which means that a derived team must supply the concrete implementation of the abstract role class. The `playedBy` keyword establishes a relationship between abstract roles and base implementation classes and enables the automatic handling of call-in and call-out bindings that implement lowering and lifting translations. Call-out bindings forward calls from role to base objects, and call-in bindings forward calls from base objects to roles. These bindings may also include method renaming and parameter mappings.

The lifting translation performed on call-out creates or retrieves a role object given a triple consisting of a base instance, a team object, and an expected role type. Each role object stores a reference to its base object, so the lowering translation performed on call-in simply retrieves this base reference stored in the role object. Each data flow that crosses a team boundary is automatically instrumented to perform the lifting and lowering translations. However, when needed, lifting and lowering translations may also be declared explicitly. For example, a method declaration of the form `m(Base as Role)` requires an instance of `Base` from the caller but provides an instance of `Role` to the callee.

Object teams suffer from the same feature combinatorics problems that other collaboration-oriented design techniques suffer from – feature combinations must be explicitly composed together, team members must be specified together and not incrementally, and team interfaces are not separated from team implementations. ARC separates team interfaces from implementations and allows team members to be specified independently of their teams. The set of teams a partial representation can participate in is implicit in its local type constraints. ARC's role constraints are similar in purpose to lifting and lowering, but ARC's

automatic combination of representations before run-time obviates the need for dynamic lifting and lowering. Thus, ARC accomplishes statically what object teams handle dynamically. Whether or not ARC partial representations could also be merged dynamically is a question for future research.

4.4 Advanced Separation of Concerns

Classes and objects provide good support for functional decomposition, but often there exist design concerns that resist encapsulation in individual classes. In this section we explore techniques for dealing with this problem, including multi-dimensional separation of concerns and aspect-oriented programming.

Multi-Dimensional Separation of Concerns Tarr and Ossher [TOHS99, TO00, OT00] point out that programming languages today typically require that systems be decomposed along a single, fixed dimension of concern (such as data or classes). They refer to this problem as the *tyranny of the dominant decomposition*. They claim that what is needed is multi-dimensional separation of concerns (MDSOC), the facility to separate and integrate overlapping concerns along multiple dimensions.

Their proposed solution, called hyperspaces, is similar to earlier work on subject-oriented programming [HO93, OKH+95]. In the hyperspace approach, hyperslices are defined and composed together to form hypermodules. A hyperslice is a set of conventional modules written in existing formalisms, and encapsulate concerns other than the dominant one. A hypermodule is a set of hyperslices together with a composition rule which determines how the slices will be combined. The hyperspace approach is a general technique for organizing elements of a design, and is not tied to any one programming language (or to programming languages at all, for that matter). Arbitrary forms of composition and merging are possible, and the details vary greatly depending on the kinds of artifacts being composed, and the languages in which they are written. The process in general involves three steps, matching, reconciliation of differences, and integration.

Hyper/J [OT00, TO00] is an implementation of multi-dimensional separation of concerns and hyperspaces for Java. It provides a means for organizing concerns along concern dimensions that reference fragments of Java code. Developers must compose concerns together, and the system generates Java code. As such, Hyper/J is not an extension to the Java language, it is a system for for managing and integrating fragments of Java text, and supports merging of program fragments in arbitrary ways.

Aspect-Oriented Programming Aspect-oriented programming (AOP) [KLM+97] is a programming technique for isolating non-functional design decisions that would otherwise be spread out across the implementation. AOP is based on the observation that certain design decisions *cross-cut* the system and are difficult to isolate in one place using traditional object-oriented languages. These decisions are called *aspects*. AOP encapsulates the expression of aspects.

An AOP-based implementation of an application consists of a set of components written in one or more component languages, a set of aspects written in one or more aspect languages, and an aspect weaver which combines the aspects with the components. Aspect weaving can be done at compile-time or run-time. The component and aspect languages are independent but must have some common concepts to enable the aspect weaver to combine them. One such example is the concept of *join points*, which are elements of the component language that aspect programs coordinate with.

AspectJ [KHH+01] is a compatible extension to Java for AOP, designed to facilitate adoption by existing Java programmers. AspectJ's extensions to Java include join points, pointcuts, advice, and aspects. Join points identify points in the execution of a program such as method calls, instance variable accesses, etc. A pointcut is a collection of join points that may expose values from the execution context, and a pointcut can

be thought of as a pattern that matches certain join points at run-time. Advice is a mechanism for defining code to execute near the join points in a pointcut. AspectJ supports various forms of advice distinguished by when they are invoked relative to the join point such as `before`, `after`, `around`, `after returning` and `after throwing`. Since pointcuts may have parameters, it is possible for advice to access values that are involved in the associated join points. Multiple pieces of advice for one join point can be composed together in an order based on specificity. Finally, aspects are like class declarations but also contain pointcuts and advice. Aspect inheritance is provided, and abstract aspects may be defined with abstract pointcuts that must be overridden in derived aspects.

The implementation described in [KHH+01] does most of the aspect weaving at compile time, with some special cases deferred to run-time. Each advice body is compiled into a method that is called at appropriate points in the code. AOP also presents new challenges for the programming environment, and demands additional support to visualize the crosscutting structure of aspect-oriented programs.

Aspect-orientation has also been characterized as requiring quantification and obliviousness [FF00], meaning that aspect invocation points are defined by quantification over program structures that are oblivious to the aspects being invoked. Unfortunately, this violates the encapsulation of components. Program structures that aspects depend on may change, altering when aspects are invoked and impairing software evolution [TBG03]. The absence of an explict contract or point of reference between aspects and components makes it difficult for aspect writers to know what to depend on, and for component writers to know what sorts of aspects might be invoked and at what places in the code.

In our opinion, aspect-orientation really needs *symmetric* quantification and *bounded* obliviousness. Symmetric quantification means not only that aspects can quantify over components, but also that components may quantify over aspects. Bounded obliviousness, which is really a consequence of symmetry in quantification, means that a component may limit the aspects that are enabled in its code, just as aspects limit the components to which they apply. Thus, neither components nor aspects can be entirely oblivious of the other.

However, aspects should be oblivious to program structure, and they should be defined in terms of abstract events which occur, not statement level syntax patterns. The events of a component that are observable by aspects should also be controlled by the component. It may be useful to combine the concepts of implicit invocation and abstract behavioral types of Sullivan and Notkin [SN92] together with aspect-orientation, and do this in an interface-oriented way whereby all aspect/component interdependencies are expressed in the form of abstract interfaces. This requires further research.

4.5 Component Generators and Generative Programming

Thus far we have focused primarily on direct programming language support for expressing a design. Code generation based on concise specifications of component compositions is yet another technique. We now discuss two examples based on code generation, GenVoca component generators and Generative Programming.

GenVoca Component Generators Batory and O'Malley [BO92] present a model of software system design based on reuse of interchangeable components. Their model, called GenVoca, merges ideas from two other projects, the Genesis system of database system components and the Avoca system of network protocol components.

A component in GenVoca is a closely knit cluster of classes. Every component has an abstract interface and a concrete implementation. The realm of an abstract interface T is the set of all components that realize the interface. Such components are plug-compatible and interchangeable. A component in GenVoca is a member of precisely one realm. A library is a particular set of components for an interface T, thus a library

is a subset of a realm (a realm refers to all possible implementations of an interface, most of which are never implemented, and a library refers to a particular set of implementations for an interface T). Every component implements an abstract to concrete mapping by translating operations in the abstract interface to operations on lower-level objects in terms of their abstract interfaces. These lower level objects are provided to the component at instantiation time via parameters, and their implementations are unknown to the component. A key feature of the GenVoca model is the idea of symmetric components, components whose abstract interface is parameterized by the abstract interface itself. A component of realm T is symmetric if it has at least one parameter of type T.

A software system is a type expression that indicates a composition of components. The set of all type expressions that provide interface T is called the domain of T (realm T includes components, domain T includes type expressions that denote component compositions). A software tool that constructs type expressions according to some rules of composition is a layout editor. Layout editors provide a language for expressing type expressions, and the subset of domain T that is expressible by the layout editor is called the family of T.

These concepts can be modeled as a grammar, with realms representing non-terminals and parameterized components representing the right-hand sides of productions. The right-hand side of a production includes a terminal for the component name and non-terminals for the types of each of the component's parameters. The language generated by such rules corresponds to the set of type expressions for the realms involved.

Batory and O'Malley show how this approach can model the implementation of Genesis database components and Avoca network protocol components. Limitations of the approach are that component parameter binding is done statically, components must be hand-crafted, and the GenVoca model does not support inheritance (components belong to one and only one realm). Design rules are used to restrict how components may be combined, and design rule checking can go beyond mere type checking. The Genesis system has a layout editor that ensures that design rules are not violated and thus avoids illegal component compositions. Generated software is untuned, however, because tuning constants that are part of every component are assigned default values which can be altered separately.

In later work, Batory and Geraci ([BG96], [BG97]) describe in more detail how the GenVoca system validates compositions of parameterized components with associated design rules. Their approach relies on attribute grammars, uses explanation-based error reporting, and adapts the consistency checking approach of the Inscape Environment (see Perry [Per89b], [Per89a]).

More recently the GenVoca model has been applied to the generation of development tool suites [BLHM02] and has been reformulated as a new component model called AHEAD (Algebraic Hierarchical Equations for Application Design) [BLS03][BSR03]. AHEAD extends the components-as-equations technique to other software artifacts, not just code, much like the hyperspace approach.

Interface-oriented programming differs from component generators such as GenVoca and AHEAD in a number of ways. In IOP we retain separate compilation, compose pieces automatically without code mangling, express design rules simply as type constraints in the programming language, and we do this in a way that preserves encapsulation and separates client concerns from implementation details. The work to add a new partial representation is proportional to the size of that representation, not the number of representations with which it can be combined. Most importantly, constraints in IOP are used to derive compositions automatically, not to check manually created ones.

In contrast, GenVoca and AHEAD essentially provide high-level equations that drive code generators for parameterized components without support for inheritance. These code generators operate at the level of syntactic code fragments, fragments that do not distinguish between client-relevant details and implementation choices. All changeable details must be exposed at the client interface as parameters and eventually resolved in component expressions. Actual values for component parameters are type checked, but addi-

tional design rules must be specified and used to further constrain the set of legal component expressions. In the end, the design of a set of composable components is treated as a product-line design problem, a problem requiring awareness and anticipation of all possible components in order to describe how they may be composed. Adding a new component requires describing how that component may be combined with all other components. Instantiation of a product in the product-line requires explicit selection of all features to be combined in the product, including low-level implementation details.

In some respects the GenVoca and AHEAD models can be said to assume the solution to the problem they attempt to solve. That is, they solve the feature combinatorics problem but only when all features to be combined are known ahead of time. ARC defines each feature locally and incrementally - everything we can say about how a feature (a partial representaton) may be combined with others is specified locally, the size of this specification depends only on what this feature depends on and not on the size of the total feature space, and this specification is monotonic with the addition of new features to the feature space (we don't have to revise our constraints with the addition of new features). The space of possible feature combinations in ARC is implicit in these constraints, whereas in GenVoca and AHEAD it must be explicitly managed as a whole.

In Batory's product line paper [BLHM02] a contrast was also drawn between component generators and what Batory called *interface-based programming*, where clients program to standardized interfaces and components implement these interfaces. GenVoca itself has been presented as an example of interface-based programming, but Batory claims that interface-based designs are brittle. The problem is that it is difficult to determine what methods belong in an interface, that changing the methods in a standardized interface requires updating all of the components which implement that interface, and that eventually it becomes too expensive to update outstanding dependencies on the interface.

The problem of wanting to change the definition of something that exists, and not having access to it or not having the resources to change all of the things that depend on the definition is fundamental to systems development and is not addressed by Batory's Origami matrices or anything in ARC. One thing we can do is make it easier to change things that we have access to, and Batory's solution essentially assumes that everything can be regenerated and provides a convenient means for doing so. ARC can recombine partial representations whenever new partial representations make new combinations possible, they do not have to be specifically requested. Another approach is to design software up front to depend on less and therefore be less affected by such changes. In this respect ARC fares better by localizing the impact of changes, whereas GenVoca and AHEAD require thinking about a change across many code fragments at once. Finally, we we can try to avoid the problem by not changing things which already exist but instead add new components with new names, requiring users who want the new functionality to change their dependencies.

ARC actually has a unique advantage when it comes to adding methods to an existing interface. Suppose for example we wish to extend some interface I. The partial representation style promoted by ARC means that many representations of I will implement some methods and assume the rest by assuming I. If we can implement the new methods of I in terms of some private state or in terms of existing methods of I, then all we have to do is extend I and add at least one such representation of the new methods in I (again, assuming the rest of I). This extension can then be automatically recombined with other partial representations of I. Although we still have to recompile users of I, we did not have to edit any code for representations of I other than the methods we added.

Generative Programming Czarnecki and Eisenecker [CE99][CE00] present a component generation technique based on feature modeling [KCH$^+$90] in which highly parameterized components are assembled into concrete configurations by a configuration generator, and configuration knowledge is stored apart from the

components themselves. In this technique, implementation fragments are associated with abstract features, and the algorithms for combining features are expressed as template metaprograms which accept feature identifiers as arguments and compose the corresponding implementation fragments. The implementation of a particular feature combination can then be instantiated by invoking the template metaprogram with the corresponding feature identifiers, causing the template metaprogram to be specialized at compile-time to produce the desired code.

In developing this method Czarnecki and Eisenecker recognized that it is useful to model feature variation explicitly and abstractly, independently of the particular programming language mechanisms for variation such as inheritance, dynamic parameterization, and static parameterization. A common mistake made in object-oriented analysis and design is to model concepts directly in these implementation terms. As an alternative, they propose feature models as a means to describe the configurability of a concept in more abstract terms. Feature models are then implemented in a separate step by mapping feature model elements to implementation components. Implementation components are themselves organized into layers, with the bottom layer as the configuration repository and each layer above it taking the layer beneath it as a parameter, similar to the GenVoca model.

The building block components and layers of generative programming are implemented in terms of C++ templates in a manner similar to other layered approaches discussed previously. Parameterized templates are composed together with *traits classes* [Mye95] to form complete classes for particular feature configurations. However, since doing this manually for a large number of combinations is tedious and error prone, a method for generating collections of implemented feature combinations automatically from abstract specifications is suggested. In this method, the configuration knowledge or recipe for generating particular feature combinations is implemented as a C++ template metaprogram with feature parameters, and the abstract specification is the actual set of feature arguments used to invoke the metaprogram.

In template metaprogramming, template code is written with conditional branches that are evaluated at compile-time to produce the desired configuration classes. Instead of directly writing classes that combine features, a generator template is written that accepts a list of feature arguments and yields, at compile-time, a class or set of classes that that implement the feature combination. This is not unlike the use of preprocessor directives but is more tightly integrated with the underlying language.

While this reduces the tedium associated with manual composition, it is still necessary to compose specific combinations of features, so the benefit is primarily notational convenience in how feature combinations are expressed. Also, each space of possible combinations requires its own template metafunction that must anticipate all of the different features which will be usable as arguments. And, as with abstract feature models, template metafunctions do not distinguish between features that clients need to understand and features that are implementation details they should ignore – with template metaprogramming, the client has to be aware of all of them. Generative programming also lacks a clean separation of interface and implementation. A feature is modeled as an implementaton with some interface, but multiple implementations of the same interface are not considered.

Finally, whereas generative programming separates configuration knowledge as explicit procedural recipes governing a specific set of features, our approach embeds essential configuration knowledge locally in type and method constraints. It is not necessary to group components into related sets and then define how the members of that set may be combined; each component dictates how it may be combined with an unbounded set of other components that match its local constraints.

4.6 Distributed Component Technology and Middleware

Distributed systems development is complicated by the fact that distributed computing occurs over distinct address spaces on remotely located, heterogeneous nodes. This leads to an array of complexities that must be anticipated and handled by successful distributed programs, such as remote communication, heterogeneity, latency, concurrency, nonuniform memory access, and partial failures. To ignore these fundamental issues and treat distributed systems development as if it were just like local development is a mistake which Deutsch claims "essentially everyone" makes with their first distributed application [Deu]. Waldo et al. [WWWK94] also argue that hiding implementation choices associated with these complexitities behind an abstract interface, thus "papering over" the distinction between local and remote objects, is a mistake. We agree that distribution issues cannot always be hidden behind an interface that makes remote objects look like local ones, but often there are distinctions that can and should be hidden, and a language that separates interface from implementation is essential to express them. In cases where a distribution issue needs to be seen and handled by a client, the hooks for dealing with that issue must be visible in the interface, but by rendering the those hooks in an abstract interface we have still decoupled the client from the underlying implementation.

Hiding the mechanics of distribution whenever possible is believed to simplify system development and is sometimes referred to as distribution transparency. The Reference Model for Open Distributed Processing (RM-ODP) [Put01] defines eight separate kinds of distribution transparency, including access, failure, location, migration, relocation, replication, persistence, and transaction transparency. The predominant approach to achieving distribution transparency in commercial system development employs distributed object containers [Emm02], otherwise known as application servers or middleware.

Middleware [Emm00] is a software layer between the network operating system and application components that supports applications distributed over a network of potentially heterogeneous nodes. The goal of middleware is to hide the complexity associated with distribution issues such as concurrency control, network communication, transaction management, fault tolerance, and security. By hiding these complexities, middleware not only focuses developers on the logic of their applications, but also allows applications to transparently scale to new workloads and dynamically adapt to platform changes. For example, remote procedure calls (RPC) [BN84] can be thought of as a simple form of middleware that makes calls crossing system boundaries to appear as if they were local, hiding the complexity of data marshalling, translation, and communication across machine boundaries. Middleware has undergone significant evolution since RPC, and today middleware platforms are used widely in industrial application development [Cha99] as a fundamental building block of e-commerce systems.

Emmerich [Emm00] divides middleware platforms into four categories, transactional, message-oriented, procedural, and object or component-oriented middleware. These categories are distinguished by the primitives they provide – distributed transactions, message passing, remote procedure calls, and remote object requests. We focus here on object-oriented and component-oriented middleware, which evolved from remote procedure calls and also tends to incorporate features of the other middleware styles. Examples of object-oriented and component-oriented middleware platforms include Microsoft's COM [Box98] and DCOM [EE98], CORBA [COR02], Java RMI [Jav], J2EE [J2E], Jini [Jin], Jxta [Gon01], Microsoft's .NET Framework [Pla], and web services [BHM+]. An overview and comparison of these platforms can be found in [Szy98].

For our purposes, middleware platforms can be described by the general model of service-oriented middleware in which three kinds of parties interact: service clients, service brokers, and service providers. Service clients request the identity of services from service brokers. Service brokers accept advertisements from service providers and maintain a registry to map service types or names to service identities or addresses, and provide lookup services to clients. Service providers advertise their services to service brokers, and accept

requests from clients to perform services. The three key issues most relevant to our work are (1) the manner in which service interfaces and implementations are separated and bound, (2) support for incremental service development, and (3) how services are composed to form higher level services.

Service Lookup and Binding Implementations are necessarily separated from interfaces in middleware systems due to platform heterogeneity and distribution between clients and servers. The way in which clients identify services, and the manner in which a service instance is bound with a client varies from one system to another. The simplest middleware platforms typically require clients to name their services. With naming, the client is bound to the identity of the service although its physical location is determined by dynamically looking up the name in a registry. Trading [Bea93] has been proposed as a more flexible alternative to naming in which the client identifies the type of the service together with other properties it expects of that service (indicators of quality-of-service, or *QoS*), and a trader looks up the identity of a service that matches the requested type and properties. The OMG has adopted this approach [OMG], and Jini's discovery and lookup protocols provide a similar facility [Jin]. In effect, naming binds a client to *the* implementation provider that has the specified name, whereas trading binds to *some* implementation provider that has the specified type and properties. Each is a distributed version of the abstract factory pattern.

Incremental Service Development Middleware does not provide any additional support for incremental service development beyond that which is available in the underlying programming language, and in some cases it provides less. Programming techniques such as abstract factories yield the same unsatisfactory results when applied at the middleware level, and some mechanisms of incremental development that are available in the underlying programming language – such as inheritance – are not available at the middleware level since composition occurs in terms of explicit forwarding across remotely located service instances.

The factory-like mechanism that middleware uses for dynamic binding results in the same disadvantages of local abstract factories. Dynamic binding must be done in terms of whole services – a service provider must implement the entire interface of the service it provides. Also, when a service provider internally uses other underlying services in its implementation, the binding to these services must be explicitly programmed using the same implementation binding protocol that other service clients go through. It is not clear whether current middleware platforms provide adequate support for these indirect bindings, bindings that are needed by the service provider but not relevant to the service client. As a result, services must either hard-code their dependence on particular underlying services or program their bindings based on context parameters extracted from higher level binding requests. Unfortunately, information about the binding context is not propagated downward, leaving explicit programming of service dependencies as the only viable approach. This in turn exacerbates the feature combinatorics problem, as deployment of new feature combinations demands whole new service implementations.

Service extension and overriding is complicated by lack of support for service inheritance or identity-preserving method delegation across collaborating service instances. For example, if derived service D wants to extend base service B by adding methods, D must intercept and forward all of the methods of B in addition to the new methods it adds. That methods must be forwarded between remotely located services is obvious, the problem here is that the programmer of D must wrap B and forward them explicitly. In addition to the inconvenience, this adds a hop between the client and B for methods which D is just forwarding (this problem is analogous to the problem that existed in early object-oriented systems in which inheritance was implemented as forwarding of methods to a parent class, creating forwarding chains proportional to the depth of the inheritance hierarchy). Virtual methods are also complicated by the opposite problem – B cannot forward methods back to D. If D wants to override a method that is virtual in B and called by B, it

must override not only the virtual method but all methods that call it in B, because there is no way with middleware service composition mechanisms to call down to an overridden method.

It is conceivable that the ARC segmented object-model and representation inference algorithm can be extended to the distributed case to solve some or all of these problems in middleware systems. Services could be implemented as partial representations that may assume the presence of base abstractions that will ultimately be bound to other services. Base abstractions are resolved to representations by a distributed variant of the representation inference algorithm, and segment references for these bases refer either to local object segments or remote object stubs. Similarly, the method tables for the base objects, whether locally or remotely located, would have to be updated to reflect any overriding for downcalls to service extensions, and these tables may have to be carried in session state to allow a single instance of a service to handle multiple clients with different bindings for aggregated or inherited service instances. In this model a complete representation of a service for a client then corresponds to a collection of possibly distributed service fragments, with method tables that automatically forward methods to other service fragments. Working out the details of this idea requires further research.

Service Composition Sometimes what the client needs for a service does not correspond to any existing service, and needs to be created by composing other more primitive services together. Composition of web services, for example, is an active area of current research. Srivastava and Koehler [SK03] review two basic approaches to web service composition that have been proposed, explicit flow composition and AI planning approaches based on semantic web annotations. In the explicit flow composition approach a program is written in a web services composition language to orchestrate the behavior of the more primitive services that must be identified or treated as parameters. In the AI planning approach (e.g., [MS02]), an AI planner takes a specification of the desired service combined with semantic constraints and derives a program which coordinates the primitive services. In addition to constraints, a domain model based on semantic web annotations [BLHL01] is also required. Sirin et al. [SHP03] describe a hybrid approach where a composite service refers to supporting services not by name but by the service capabilities required, as in trading. This information is combined with semantic web annotations to present choices interactively to a user who filters recommended services and decides which ones to use.

ARC's approach is most similar to Sirin's [SHP03], although in our case the binding between the composition and the supporting services is abstract and not resolved until representation inference time, and non-deterministic choice is used instead of user interaction to resolve multiple alternatives when the alternatives are deemed equivalent relative to the client request. New abstractions in ARC must be explicitly designed and implemented by at least one representation that explicitly composes underlying abstractions. When a client uses the new abstraction, partial representations for the new abstraction along with any supporting abstractions are selected and combined automatically.

Middleware's value is that it modularizes code that deals with the unique problems of distribution. It does not make components more reusable, although it may make components easier to access and integrate. Web services, for example, are often hyped as a solution that will dramatically improve reusability. But web services are just another form of middleware, a set of standard implementation technologies that provide RPC over the Internet. These technologies do not improve our ability to design component functionality for reusability in any way, what they do is leverage a commodity infrastructure to improve access to services that must be designed for reusability using other means. Wrapping a legacy application in a web-service interface does not improve the intrinsic reusability of that application's functionality, it just makes it more accessible.

To be fair, however, we should look past the hype and realize that middleware platforms such as web services are really only intended as support for distribution and integration, intended for use with components of large granularity developed using other methods, not as a general purpose programming facility that applies at all levels of software development. But since middleware-based services have to be used as-is or completely wrapped, the library scaling problem already present in object-oriented programming languages is even more difficult to overcome in middleware-based libraries.

In summary, service-oriented middleware provides modular support for distributed systems development and can make it easier to access and integrate software components, but it does not facilitate separation of concerns other than distribution issues any more than the underlying programming language, and in some cases makes matters worse. There are several significant differences between interface-oriented programming and service-oriented middleware in their support for implementation binding, incremental programming, and composition. In service-oriented middleware, implementation binding has the forwarding and overriding drawbacks of factories, inheritance across services is not supported, and composition requires explicit selection of composed services. Perhaps the most important distinction to remember is that middleware is intended to facilitate distribution of relatively large-grained components developed by other means, and not as a general purpose mechanism for separation of concerns that can be applied "all the way down." IOP, in contrast, is intended as a general purpose mechanism that can be applied "all the way down." Whether IOP can be used as a basis for more powerful middleware systems is a question for future research.

5 Status and Future Directions

The concepts of IOP in general and the constructs of ARC in particular are still under development. A compiler for a subset of ARC that compiles ARC programs into Java has been written in Java. In the near future we plan to both formalize and stabilize the definition of enough of the language to allow us to evaluate techniques for representation inference as well as explore different approaches to representation selection, including evolutionary optimization (where the space of representation choices is used as a program genotype). Constructs and areas of language design that need more work include support for exceptions, control abstraction, contexts (namespaces and modular packaging), and properties. We also expect that there may be some interactions with parametric polymorphism that have yet to be explored.

IOP also seems potentially useful in other research domains such as aspect-oriented programming, configurable self-managing systems, and middleware/service-oriented systems.

It may be possible, for example, to extend IOP to create interface-oriented aspects, and to extend the representation inference approach to subsume aspect weaving. This could provide an interesting new way to think about AOP, possibly providing a statically safe form of AOP.

Self-managing systems aim to automatically reconfigure themselves in response to changing environments. It may be possible to extend IOP's representation inference to the dynamic case, allowing representations to be partially re-generated based on dynamically evaluated constraints. A similar issue exists in the pervasive/ubiquitous computing domain, and dynamic IOP may be able to provide a useful mechanism for implementing context-awareness.

Finally, middleware systems are hampered by their lack of support for incremental development, in particular, by the lack of support for implementation inheritance across remotely deployed components or services. A distributed form of IOP and representation inference may be able to help solve this problem, providing an easier way to incrementally develop services as well as compose separately developed services on the fly.

6 Conclusion

In this paper we have characterized the problem of implementation bias at points of instantiation dependency as a weakness of modern object-oriented languages, a problem that hinders software reuse and incremental development. As a solution to this problem we have introduced the concept of interface-oriented programming (IOP), a new style of program organization where all type dependencies are expressed in the form of abstract interfaces, and implementation names are never used in type declarations, neither when creating whole objects nor when inheriting implementation. We have proposed a new programming language called ARC that provides the strict separation of interface and implementation needed to effectively support IOP. ARC builds on established and emerging forms of object-oriented and component-oriented program expression and adds the extensions needed for IOP. The primary difference between ARC and traditional object-oriented languages is that ARC replaces complete classes with partial representations, implementations that partially implement their interfaces and inherit implementation privately and indirectly via assumed base interfaces. Partial representations can be used not only to incrementally define the implementations of individual objects, but also to incrementally define the implementations of objects participating in multi-object collaborations.

IOP also demands new capabilities from the program construction and run-time system. Partial representations must be composed automatically to form complete representations in a manner that respects the interface being represented and all internal assumptions made by each partial representation. Since the complete representation for an abstract interface is not necessarily unique, a means to select a particular representation is also required. We have proposed representation inference and non-deterministic instantiation with configurable representation selection policies as the mechanisms that provide these capabilities, and have defined a segmented object model that enables them while providing constant-time dynamic dispatch.

References

[ACF+01] B. Alpern, A. Cocchi, S. Fink, D. Grove, and D. Lieber. Efficient implementation of java interfaces: Invokeinterface considered harmless. In *Proceedings of OOPSLA*, 2001.

[ADM01] B. Albahari, P. Drayton, and B. Merrill. *C# Essentials*. O'Reilly & Associates, Inc., 2001.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language, Third Edition*. Addison Wesley, 2000.

[Ald04] Jonathan Aldrich. Open modules: Reconciling extensibility and information hiding. In *Proceedings of Workshop on Software Engineering Properties for Languages for Aspect Technologies (SPLAT 04)*, 2004.

[ALZ00] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam: A smooth extension of java with mixins. In *Proceedings of ECOOP*, pages 154–178, 2000.

[AvdL90] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *Proceedings of ECOOP/OOPSLA*, pages 161–168, 1990.

[AWB+93] Mehmet Aksit, Ken Wakita, Jan Bosch, Lodewijk Bergmans, and Akinori Yonezawa. Abstracting object interactions using composition filters. In *Proceedings of ECOOP '93 Workshop on Object-Based Distributed Programming*, 1993.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of ECOOP/OOPSLA*, 1990.

[Bea93] M. Y. Bearman. ODP - trader. In *Proceedings of International Conference on Open Distributed Processing*, 1993.

[BG96] D. Batory and B. J. Geraci. Validating component compositions in software system generators. In *Proceedings of International Conference on Software Reuse*, pages 72–81, 1996.

[BG97] D. Batory and B. J. Geraci. Component validation and subjectivity in genvoca generators. *IEEE Transactions on Software Engineering*, pages 67–82, 1997.

[BHJL86] Andrew Black, Norman Hutchinson, Eric Jul, and Henry Levy. Object structure in the emerald system. In *Proceedings of OOPSLA*, pages 78–86, 1986.

[BHM+] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. Technical report, World Wide Web Consortium.

[Big94] T. J. Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proceedings of the International Conference on Software Reuse*, pages 102–109, 1994.

[BL84] R. Burstall and B. Lampson. *A kernel language for abstract data types and modules*. Springer Verlag, 1984.

[BLHL01] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientific American*, May 2001.

[BLHM02] D. Batory, R. E. Lopez-Herrejon, and J.-P. Martin. Generating product-lines of product families. In *Proceedings of the 17th International Conference on Automated Software Engineering*, 2002.

[BLS03] D. Batory, J. Liu, and J. N. Sarvela. Refinements and multi-dimensional separation of concerns. In *Proceedings of SIGSOFT ESEC/FSE*, pages 48–57, 2003.

[BMN+00] P. Boinot, R. Marlet, J. Noyé, G. Muller, and C. Consel. A declarative approach for designing and developing adaptive components. In *Proceedings of the IEEE International Conference on Automated Software Engineering*, 2000.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[BO92] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM TOSEM*, 1(4):355–398, 1992.

[Box98] D. Box. *Essential COM.* Addison Wesley Longman, 1998.

[BSR03] D. Batory, J. Sarvela, and A. Rauschmayer. Scaling stepwise refinement. In *Proceedings of International Conference on Software Engineering*, pages 187–197, 2003.

[BSST93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas. Scalable software libraries. In *Proceedings of ACM SIGSOFT*, pages 191–199, 1993.

[BW98] Martin Büchi and Wolfgang Weck. Compound types for java. In *Proceedings of OOPSLA*, pages 362–373, 1998.

[BW00] Martin Büchi and Wolfgang Weck. Generic wrappers. In *Proceedings of ECOOP*, pages 201–225, 2000.

[CBML02] R. Cardone, A. Brown, S. McDirmid, and C. Lin. Using mixins to build flexibile widgets. In *Proceedings of AOSD*, 2002.

[CCHO89] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proceedings of OOPSLA*, pages 457–467, 1989.

[CE99] K. Czarnecki and U. W. Eisenecker. Synthesizing objects. In *Proceedings of ECOOP*, 1999.

[CE00] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, 2000.

[Cha99] J. Charles. Middleware moves to the forefront. *IEEE Computer*, pages 17–19, May 1999.

[CHC90] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 125–135, 1990.

[CL01] R. Cardone and C. Lin. Comparing frameworks and layered refinement. In *Proceedings of ICSE*, May 2001.

[Coo92] William R. Cook. Interfaces and specifications for the smalltalk-80 collection classes. In *Proceedings of OOPSLA*, pages 1–15, 1992.

[Cop92] James Coplien. *Advanced C++ Programming Styles and Idioms*. Addison Wesley, 1992.

[COR02] CORBA components. Technical report, Object Management Group, 2002. `http://www.omg.org`.

[CP89] William Cook and Jens Palsberg. A denotational semantics of inheritance and its correctness. In *Proceedings of OOPSLA*, pages 433–443, 1989.

[Deu] Peter Deutsch. The eight fallacies of distributed computing. Technical report. James A. Gosling's web page.

[Dij72] E. W. Dijkstra. *Notes on structured programming*, pages 1–81. Academic Press, New York, 1972.

[DM00] S. Dobson and B. Matthews. Ionic types. In *Proceedings of ECOOP*, pages 296–312, 2000.

[EE98] G. Eddon and H. Eddon. *Inside Distributed COM*. Microsoft Press, 1998.

[Emm00] W. Emmerich. Software engineering and middleware: A roadmap. In *Proceedings of Conference on Future of Software Engineering*, 2000.

[Emm02] W. Emmerich. Distributed component technologies and their software engineering implications. In *Proceedings of International Conference on Software Engineering*, pages 537–546, 2002.

[Ern01] Erik Ernst. Family polymorhpism. In *Proceedings of ECOOP*, pages 303–326, 2001.

[FF00] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *OOPSLA Workshop on Advanced Separation of Concerns*, 2000.

[FKF98] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. Classes and mixins. In *Proceedings of ACM Conference on Principles of Programming Languages (POPL)*, pages 171–183, 1998.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.

[Gon01] Li Gong. Jxta: A network programming environment. *IEEE Internet Computing*, pages 89–95, May/June 2001.

[Gut77] John Guttag. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404, 1977.

[HÖ93] U. Hölzle. Integrating independently developed components in object-oriented languages. In *Proceedings of ECOOP*, 1993.

[Her03] S. Herrmann. Object confinement in object teams – reconciling encapsulation and flexible integration. In *Proceedings of 3rd German Workshop on Aspect-Oriented Software Development*, 2003.

[HO93] W. Harrison and H. Ossher. Subject-oriented programming (a critique of pure objects). In *Proceedings of OOPSLA*, pages 411–428, 1993.

[J2E] Java 2 platform, enterprise edition (J2EE). Technical report, Sun Microsystems, Inc. `http://java.sun.comi/j2ee`.

[Jav] Java rmi specification. Technical report, Sun Microsystems, Inc.

[Jin] Jini(tm) architecture specification. Technical report, Sun Microsystems, Inc.

[KCH+90] K. Kang, S. Cohen, J. Hess, W. Nowak, and S. Peterson. Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, November 1990.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of ECOOP*, pages 327–355, 2001.

[KL92] Gregor Kiczales and John Lamping. Issues in th design and specification of class libraries. In *Proceedings of OOPSLA*, pages 435–451, 1992.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP*, pages 220–242, 1997.

[Lam93] John Lamping. Typing the specialization interface. In *Proceedings of OOPSLA*, pages 201–214, 1993.

[LPS97] K. J. Lieberherr and B. Patt-Shamir. Traversals of object structures: Specification and efficient implementation. Technical Report TR-NU-CCS-97-15, College of Computer Science, Northeastern University, 1997.

[LS94] Victor B. Lortz and Kang G. Shin. Combining contracts and exemplar-based programming for class hiding and customization. In *Proceedings of OOPSLA*, pages 453–467, 1994.

[Mez97] Mira Mezini. *Variation-Oriented Programming: Beyond Classes and Inheritance.* PhD thesis, 1997.

[ML98] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings of OOPSLA*, 1998.

[MO02] Mira Mezini and Klaus Ostermann. Integrating independent components with on-demand remodularization. In *Proceedings of OOPSLA*, pages 52–67, 2002.

[MO03] Mira Mezini and Klaus Ostermann. Conquering aspects with caesar. In *Proceedings of AOSD*, 2003.

[MS02] S. McIlraith and T. C. Son. Adapting golog for composition of semantic web services. In *Proceedings of Conference on Knowledge Representation and Reasoning*, 2002.

[MSL01] Mira Mezini, L. Seiter, and Karl Lieberherr. *Component integration with pluggable composite adapters.* 2001.

[Mye95] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[OKH+95] Harold Ossher, Mathew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. In *Proceedings of OOPSLA*, pages 235–250, 1995.

[OMG] Trading object service specification. Technical report, Object Management Group.

[Ost02] Klaus Ostermann. Dynamically composable collaborations with delegation layers. In *Proceedings of ECOOP*, pages 89–110, 2002.

[OT00] Harold Ossher and Peri Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*. Kluwer, 2000.

[Par72] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[Per89a] D. E. Perry. The inscape environment. In *Proceedings of the International Conference on Software Engineering*, 1989.

[Per89b] D. E. Perry. The logic of propogation in the inscape environment. In *Proceedings of ACM SIGSOFT*, 1989.

[Pla]

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *Proceedings of ECOOP*, pages 419–443, 1997.

[Put01] J. Putnam. *Architecting with RM-ODP*. Prentice Hall PTR, 2001.

[Ran99] Alexander Ran. Software isn't built from lego blocks. In *Proceedings of the Symposium on Software Reusability*, 1999.

[RC02] Barry Redmond and Vinny Cahill. Supporting unanticipated dynamic adaptation of application behaviour. In *Proceedings of ECOOP*, pages 205–230, 2002.

[Rie95] Dirk Riehle. How and why to encapsulate class trees. In *Proceedings of OOPSLA*, pages 251–264, 1995.

[RTL+91] Rajendra K. Raj, Ewan Tempero, Henry M. Levy, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, and Eric Jul. Emerald: A general-purpose programming language. *Software - Practice and Experience*, 21(1):91–118, 1991.

[SB98] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *Proceedings of ECOOP*, pages 550–570, 1998.

[SDNB03] N. Schärle, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings of ECOOP (to appear)*, 2003.

[SHP03] E. Sirin, J. Hendler, and B. Parsia. Semi-automatic composition of web services using semantic descriptions. In *Proceedings of Workshop on Web Services: Modeling, Architecture, and Infrastructure*, 2003.

[SK03] B. Srivastava and J. Koehler. Web service composition - current solutions and open problems. In *Proceedings of Workshop on Planning for Web Services*, 2003.

[SN92] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–268, July 1992.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA*, pages 38–45, 1986.

[SPL96] L. Seiter, J. Palsberg, and K. Lieberherr. Evolution of object behavior using context relations. In *Proceedings of ACM SIGSOFT*, 1996.

[Sre02] V. C. Sreedhar. Mixin' up components. In *Proceedings of International Conference on Software Engineering*, pages 198–207, 2002.

[Str87a] Bjarne Stroustrup. Multiple inheritance for c++. In *Proceedings of the Spring 1987 European Unix Users Group Conference*, May 1987.

[Str87b] Bjarne Stroustrup. What is "object-oriented programming"? In *Proceedings of ECOOP*, Lecture Notes in Computer Science 276, 1987.

[SWB02] N. Sridhar, B. W. Weide, and P. Bucci. Service facilities: Extending abstract factories to decouple advanced dependencies. In *Proceedings of the International Conference on Software Reuse*, pages 309–326, 2002.

[Szy98] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* Addison Wesley, 1998.

[TBG03] Tom Tourwé, Johan Brichau, and Kris Gybels. On the existence of the aosd-evolution paradox. In *Proceedings of SPLAT*, 2003.

[TO00] Peri Tarr and Harold Ossher. Hyper/j^{TM} user and installation manual. Technical report, IBM Research, 2000. `http://www.research.ibm.com/hyperspace`.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering*, 1999.

[TVJ+01] E. Truyen, B. VanHaute, W. Joosen, P. Verbaeten, and B. N. Jorgensen. Dynamic and selective combination of extensions in component-based applications. In *Proceedings of ICSE*, May 2001.

[Var03] L. Robert Varney. Interface-oriented programming in arc. Technical report, UCLA Department of Computer Science, 2003. Ph.D. Dissertation Proposal.

[VH03] M. Veit and S. Herrmann. Model-view-controller and object teams: A perfect match of paradigms. In *Proceedings of Aspect-Oriented Software Development*, 2003.

[VN96a] M. VanHilst and D. Notkin. Decoupling change from design. In *Proceedings of SIGSOFT Foundations of Software Engineering*, 1996.

[VN96b] M. VanHilst and D. Notkin. Using c++ templates to implement role-based designs. In *Proceedings of 2nd JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37, 1996.

[VN96c] M. VanHilst and David Notkin. Using role components to implement collaboration-based design. In *Proceedings of OOPSLA*, pages 359–369, 1996.

[Wal98] Jim Waldo. Code reuse, distributed systems, and language-centric design. In *Proceedings of the 5th International Conference on Software Reuse*, 1998.

[Wir71] N. Wirth. Program development by stepwise refinement. *Communications of the ACM*, 14(4):221–227, 1971.

[WM00] Robert J. Walker and Gail C. Murphy. Implicit context: Easing software evolution and reuse. In *Proceedings of SIGSOFT (FSE-8)*, pages 69–78, 2000.

[WWWK94] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report SMLI-TR-94-29, Sun Microsystems Laboratories, 1994.

[Zen02] Matthias Zenger. Type-safe prototype-based component evolution. In *Proceedings of ECOOP*, pages 470–497, 2002.