Approaches to Composition and Refinement in Object-Oriented Design

Marcel Weiher
Matrikelnummer 127367

Diplomarbeit
Fachbereich Informatik
Technische Unversität Berlin

Advisor: Professor Bernd Mahr

August 21, 1997

Contents

List of Figures

Chapter 1

Introduction

Since their introduction in the early to mid 1980ies, object-oriented (OO) techniques have gained wide acceptance as a standard approach to software development and system design. At the same time that initial success stories are being corroborated by more widespread reports of improvements in productivity and product quality, there are also increasing reports of problems when trying to apply the supposed reuse benefits of OO systems and design methods in large scale settings, not all of which can be attributed to compromised implementations of object-oriented concepts.

1.1 Object Orientation

Object orientation is a data-centric approach taking much from earlier work in data modelling and abstract data types. It was recognized that clustering procedures around data abstractions instead of the other way around provides both a convenient encapsulation boundary and a useful mode of thinking about programming. An *object* therefore consists of some data associated with a set of operations that provide the only means of accessing and manipulating the data encapsulated by the object.

The grouping of operations around data gives rise to the idea that it is the objects themselves that are smart, and naturally leads to an implementation of *polymorphism*: operation names are dissociated from operation implementations, an object autonomously decides which actual operation to invoke when a given named operation is to be performed on that object, *late binding* of operation names to operations takes place. The term *message passing* emphasizes the conceptual model of a smart object that autonomously decides how to react to requests of service from clients. The same basic idea is well established for operations on built-in data types such as floating point numbers and integers, to which compilers automatically assign the appropriate floating point or integer multiplication routines/instructions when a generic multiplication operator appears in the program text.

Classes provide the grouping and classification mechanism for objects, corresponding very closely to abstract data types. Most concrete implementations of object-oriented programming languages treat classes as both templates for the individual object instances, and as a shared storage for the operations, which are

therefore identical for all objects of a class. The polymorphic aspect of operation invocation is dealt with by storing a mapping d from operation names, *selectors*, to operation implementations, *methods*, in the class:

$$d : \text{selectors} \rightarrow \text{methods} \tag{1.1}$$

With classes and objects comes the notion that classes may actually be related. For example, integer and floating point numbers share many similarities, in addition to having some crucial difference. Capturing these similarities makes sense both from a modelling and from an implementation perspective. Treating different types of numbers alike where their differences don't matter reduces conceptual complexity and also allows code that implements common operations to be potentially shared. Object-oriented languages support this through *inheritance*, allowing a class to define itself as a *subclass* of a *superclass* from which the class inherits both operations and state.

A subclass can add to both the state template and the operations provided by the superclass, but it can also *override* inherited operations. The inheritance operator for operations only can be denoted with the operator $\oplus$ and defined as follows:

$$d_1 \oplus d_0 = d_1 \cup \{(s \rightarrow m) \in d_0 | s \notin domain(d_1)\} \tag{1.2}$$

If the mapping function d_1 of a subclass defines a method for a selector that is also defined by a superclass d_0, the subclass definition takes precedence. Due to the late binding of all message sends, this new definition also applies to messages sent by methods defined in the superclass. Most object-oriented languages also provide a special mechanism for accessing the superclass's definition of an overriden method from within the overriding class, essentially applying early binding to these message sends.

1.2 Composition and Refinement

To the basic concept of abstract data types at least partially implemented in previous procedural languages, object-oriented programming adds the complementary notions of polymorphism using late bound message sends and inheritance hierarchies for capturing conceptual similarities of types and permitting code sharing in implementations.

Both techniques reduce some of the non essential, or accidental [Bro95, page 189] difficulties of software development by removing the cognitive load on developers caused by unnecessary duplication. Arguably more alluring is the fact that these techniques directly support *reuse* of existing components.

Inheritance provides language support for incremental programming, where a new piece of software can be *refined* from an existing one simply by specifying the differences between the two. Differential programming using inheritance mechanisms directly supports iterative development processes, and unlike the extra-linguistic variants frequently practiced when inheritance is not available – "cut-and-paste" programming – fully supports backtracking within such a process because the original code, the modifications and the relationships between the two are all retained.

The polymorphism implied by late-bound message passing also allows flexible *composition* of objects, because an object is not statically coupled to its implementation components the way it would be with the module mechanisms of procedural languages

The remainder of this text will look at techniques for composition and refinement in the context of object-oriented programming and design. An overview of the state of the art in chapter 2 shows how these techniques are combined to support high-level reusable components, but at the same time fail to be applicable to the resulting artifacts. Interfaces that provide better support for the object-oriented model are presented in chapter 3. The asymmetry between the dynamic applicability of object composition and static applicability of class inheritance is addressed in chapter 4. The large scale composition of components into systems raises issues that go beyond object technology into the realm of software architecture, a brief overview of which is given in chapter 5. Finally, chapter 6 looks at taking refinement beyond the restrictions imposed by the linear inheritance mechanism and existing, one-dimensional encapsulation boundaries.

Chapter 2

State of the Art

One reason inheritance has been such a remarkably useful tool is that it can be applied incrementally, unlike parametrization methods it can derive a new piece of software from an existing, fully functional piece of software [Mey86]. The benefits of reuse are much easier to explain when not burdened with large up front costs.

Despite this, one of the most effective reuse mechanisms in object-oriented programming has been the concept of an *abstract-class*. An abstract class leaves some of its methods or constituent components undefined, essentially as formal parameters to be bound to concrete implementations by subclasses. Unlike standard parametrization, which only allows the structures to refer to their parameters, abstract classes also allows the parameters to refer back to the parametrized structure [Bra92].

2.1 Frameworks

Programmers using object-oriented technology often recount how they built a complex application in just a couple of days, a feat that otherwise seems only possible using special purpose tools such as fourth generation database languages, not with a general purpose programming language. Most of these successes can probably be attributed to the existence of object-oriented frameworks, a larger-scale application of the idea of abstract classes using sets of cooperating classes to create a reusable design for a specific class of software[JF88].

A framework predefines all the design parameters common to a class of software (sub-)systems, including the overall architecture, the domain-specific classes populating that architecture, the overall control flow and most of the object interactions. The framework user only has to implement the parts specific to a particular application, usually by subclassing predefined abstract classes of the framework.

The *hot-spots* of the framework, the points that allow variable behavior and therefore act as parameters of the framework, can be implemented either as methods of one or more abstract framework class that have to be implemented by the application programmer, or as object-references that can be set to user-defined objects. The former case is often referred to as an instance of a *white-box* framework, because inheritance makes much more of the internal interface visible to clients, whereas the latter type of frameworks are conversely referred to as *black-box* frameworks because encapsulation behind object interfaces applies.

In either case, the user is effectively refining the framework to make it fit a particular use, though this type of refinement is typically called *framework instantiation* to differentiate it from the case where the framework itself is adapted to fit a different class of problems. Despite the analogy to refining an abstract (or concrete) superclass, it should be obvious from the description above that frameworks are not objects or classes, and therefore the standard object-oriented mechanisms for refinement and composition do not apply to the framework in its entirety.

The similarity in productivity to domain-specific languages such as fourth-generation database language is telling: frameworks can be considered domain specific languages implemented by defining their vocabulary in terms of objects and their grammar in terms of object interactions. Not surprisingly, domain frameworks and domain specific languages are two of the possible artifacts resulting from domain analysis.

The crucial difference between domain specific languages and frameworks is that whereas the former are typically static once defined, limiting expressibility to just the domain and no more, the latter are fully integrated with the general purpose programming language they're defined in. The language analogy also helps clear up the difference between class libraries, which just define language elements, and frameworks, which also define at least some of the grammar.

2.1.1 Framework Construction

One thing about framework construction is commonly accepted: it is very difficult. The overall framework development process is typically iterative, with either experience developing multiple applications, domain analysis or often both providing input to the development of the actual framework. The framwork is then applied to further application development leading to further input to the framework development, closing the circle [Mat96]. The actual task of domain analysis is considered outside the scope of framework-specific development issues.

Design guidance in mapping domain experience and analysis to a framework has mostly taken the form of specific advice on refactoring application classes and libraries into frameworks [JF88], and on turning white-box frameworks relying mostly on inheritance into easier to reuse black-box frameworks relying more on object composition. Much of this advice is cast in pattern form [Rüp96][GHJV95][FO95].

A more general approach [DMNS96] takes the design space of the framework, identified with object-oriented domain analysis techniques, and makes it explicit by reifying various features of the axes of this design space, turning them into black-box components. Some of these features are represented through reified object interactions.

1. Each of the design-space axes constitutes one level of variability in the framework design, and should thus be represented by a class of objects that can vary, through different instances and/or subclasses to represent various points on each of the axes.

2. Distribution requirements, such as concurrency, access and integrity control, logging and caching are orthogonal to the design-space axes, but potentially

have to "wrap" every one of certain types of operations, for example data access or modification, of the principal classes.

In order to make sure all operations are really trapped, the set of operations for each design axis is specified by a contract which is then reified.

3. A final set of objects for each design axes represents possible configurations, assuring that object-instantiations, a typical achilles heel of framework generality (see section 2.1.2) is handled flexibly and independently for each design axis.

Following these guidelines should result in a black-box framework that has appropriate hot-spots for its domain variabilities and orthogonal distribution requirements as well as being extensible because the object-instantiation patterns can be customized without large-scale modifications to the constituent classes.

2.1.2 Reusing Frameworks

Though using a well-designed framework is much simpler than constructing one, much of the published effort in framework development has been on documenting frameworks for reusers, often using patterns as the medium [BJ94][HJE95]. More on patterns in the next section.

Actual reuse of frameworks, creating new frameworks from existing frameworks by either refining a framework or composing one from a serieas of other frameworks, seems to be a much more difficult problem than simply applying a framework to a concrete problem, as just a short list of the problems encountered with refining and composing frameworks demonstrates:

- [MN93] shows that framework refinement (in the sense of adapting a framework to yield another abstract framework) is not possible under either covariant (Eiffel) or contravariant (Modula-3) typing rules.

- Refinement of Frameworks leads to conflicts with the fact that object-instantiation is early bound even in OO languages, so special measures have to be taken to allow existing class and object relationships of a framework to be retained in the face of subclassing [Kic93].

- Composition of independently developed components, which is considered a prerequisite for pervasive reuse, requires integration adaptation to resolve the inevitable small differences that occur even in principally compatible components, with all the problems that framework refinement entails [Höl93].

- The fact that frameworks define an architecture causes problems when two or more frameworks to be composed disagree about the type of architecture [GAO95].

- The inversion of control also leads to problems in framework composition because several frameworks assume that they are in charge of the event loop [Ber90][GAO95].

The problems reported in [GAO95] for example, describing the implementation of the Aesop system described in section 5.1.4, led to a four-fold increase in development time, a semi-automatic, error-prone construction process, difficult to maintain software and unacceptable performance in the generated tools. All this despite the fact that the components – the OBST object-oriented databases, the Interviews GUI framework, the HP Softbench event integration tool and the Mach Interface Generator Mig – are individually without fault and widely regarded as high quality pieces of software.

2.2 Design Patterns

Having started as a technique for documenting the internal workings of the object collaborations making up frameworks, *design patterns* have evolved into an element of object-oriented software in their own right. They use an informal but structured documentation technique gleamed from Christopher Alexander to describe object-oriented micro-architectures that solve a generic software design problem in a specific context [GHJV95].

Documentation structured as a pattern includes a *name* and shorthand *intent* description to quickly identify and classify a pattern; a design problem with illustrated solution based on the pattern serving as a *motivation* for the pattern; brief sections delimiting the *applicability* of the pattern, its graphical *structure* and a textual description description of the *participants* and the nature of their *collaboration*; as well as some of the known *consequences* of using the pattern, *related patterns* and how they are different from this particular one.

2.2.1 Reusing Patterns

Patterns are supposed to be "Elements of Reusable Object-Oriented Software" [GHJV95, subtitle], but how exactly is reuse supposed to happen if the patterns themselves are abstract and object languages have no support for encapsulating patterns?

Recent work has studied the automatic generation of code from Design Patterns [BFVY96]. It includes a hypertext rendition of *Design Patterns* with each individual design pattern rendered as a set of linked HTML pages. Each pattern is augmented with a code-generation page, an HTML form page that presents the user with choices regarding the trade-offs of this particular pattern and communicates these choices to a table-driven code generator when the user chooses to generate code for the pattern. The code generator generates C++ code and is itself implemented in PERL. The authors report that although the integration of documentation and code generation as a single hypertext presentation has worked well, there are the usual problems associated with integrating generated code into existing class hierarchies, which the authors hope to address with techniques from subject-oriented programming (see section 6.2).

Pointing to experience with inference patterns in knowledge engineering, and making the analogy to design patterns plausible by showing the structural and pragmatic similarities between the two types of patterns, Tim Menzies [Men97] argues that the reuse benefit of patterns may be elusive.

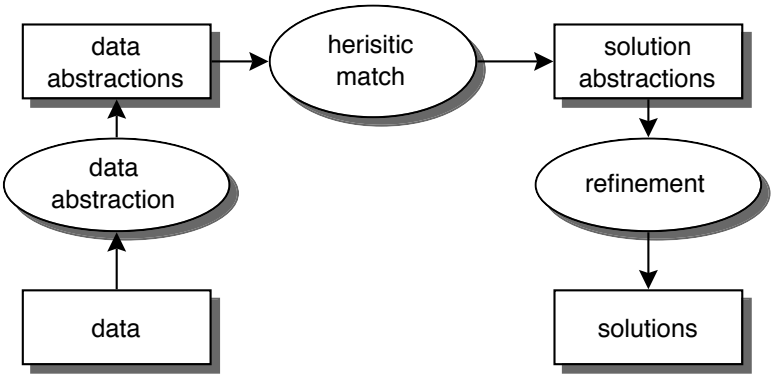

Figure 2.1: Heuristic Classification inference pattern

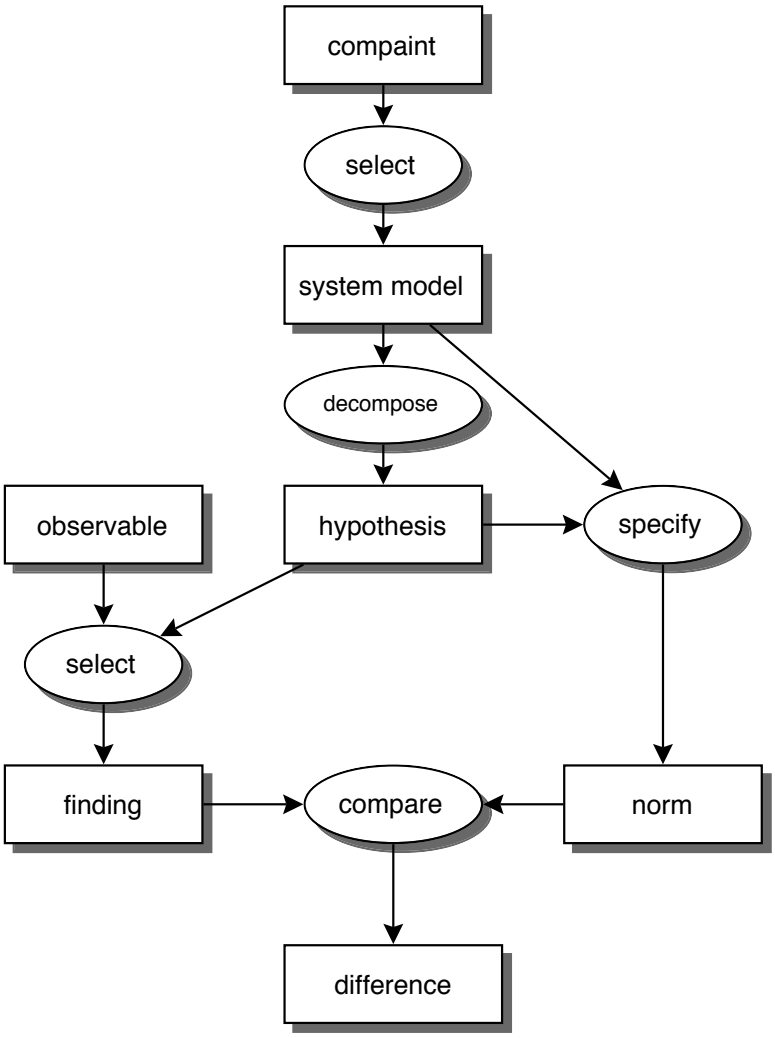

Figure 2.2: KADS *diagnosis* inference pattern

Analysis of early expert systems revealed that all of them used special instances of the same general inference pattern, which was called *heuristic classification* and is shown in schematized way in figure 2.1. Further of these abstract inference patterns were later discovered and also found to consist of a distinct set of inference subroutines, shown as ovals in the diagrams presented here. According to the author, the SPARK/BURN/FIREFIGHTER system based on a library of 13 of these subroutines produced the largest documented gain of productivity in any software development approach, ranging from a factor of 14 to apparently up to a factor of 60.

Further productivity gains were hoped from the reuse of the higher level inference patterns, but despite the fact that mature libraries of such patterns have been widely available for a while and tools exist for automatically assembling such patterns into executable systems, reuse is the exception rather than the rule with researchers building their own preferred patterns instead of reusing existing ones, with sometimes radically different approaches to the same problem.

Furthermore, a study where subjects had to extract knowledge from an expert dialogue, a doctor interviewing a patient, from which 20 respiratory disorders and a total of 304 "knowledge fragments" could be identified, showed no performance gains, in terms of identified knowledge, for subjects using an inference pattern, In fact, the subjects who were given a given the well established diagnosis pattern (figure 2.2) in order to help them with the classification task did significantly *worse* than subjects not given any model (only 55% of disorders and 34% of the knowledge fragments identified, versus 75% and 41% for the control group).

The author goes on to suggest that patterns may actually confuse the modelling process instead of clarifying it, that lower level representations provide more practical insights, at least in knowledge engineering.

2.2.2 Communicating Patterns

Although it is not clear whether the micro-architectures usually associated with patterns are actually reusable either as code or as design, the value of patterns as a structured documentation tool for communicating solutions to complex, context dependent problems that require careful balancing of the various forces influencing the decision. Patterns capture not only the actual solution, but also the starting and resulting contexts and the forces to be balanced by the solution.

James Coplien [Cop96a],[Cop96b] states that the connection between patterns and object-orientation is at best accidental, with objects just being the current paradigm in place when patterns happened on the stage, and the decidedly non-object-oriented nature of patterns a perfect fit for picking up the pieces when object-oriented technique start to fail, for example in the specification of the interaction patterns in object collaborations.

Techniques from the field of software architecture (see section 5), though initially focused on higher level structures, can also be successfully applied to the micro-architectures described by many of the current object-oriented design patterns. On the other end of the spectrum, more advanced object languages have no need for many of C++ language idioms that provide workarounds to shortcomings in the language and a further reduction can be expected if and when languages start

to support class composition and object refinement.

2.3 Commentary

The problem with framework seems to be that the elements of the object-oriented model, classes and objects, are not closed under the respective operations refinement and composition. With that diagnosis in mind, a solution might lie in considering framework-like units as basic elements and making both classes and objects special cases of this elementary construct, which would entail considering run-time and compile-time as lying more on a continuum than being completely distinct.

More immediate, and possibly more fruitful solution-proposals range from making object type systems more flexible (section 3), making software architecture explicit (section 5) and extending class hierarchies in place (section 4.3). Aspect-oriented programming (section 6.3) promises to help with the fact that many of these problems are not well localized but spread throughout the code.

Patterns are a great method for documenting and communicating complex (design) solutions applicable to problems and domains not well suited to completely formal descriptions. At present, though, it looks like the object-oriented micro-architectures are amenable to completely formal description, which may mean that focus will shift back to the potential of patterns for documenting process and rationale, rather than just structure.

Chapter 3

Enhanced Interfaces

Interfaces of components are supposed to specify the set of of services available from that component, as well as the preconditions necessary for provision of the services. Ideally, interfaces would include full logical specifications of the preconditions as well as the state after a specific service has been provided. Practically, interfaces have to be limited to automatically verifiable subsets of a full specification, which itself isn't generally verifiable automatically.

The static operation signatures inherited from their procedural predecessors by object-oriented languages for static type-checking purposes don't fill their requirements because they provide too little information for safely inheriting from a class or interacting with objects of that class, while at the same time requiring too stringent preconditions for operation invocation.

3.1 Interface Typing

Tpying has traditionally been closely linked with implementation considerations rather than safety concerns. For the compiler of a procedural language to apply the proper operation to the contents of a variable, it has to know the type of that variable and only allow values of that type to be assigned to that variable. Parameters to procedures also have to be typed in order for the type information to be retained across procedure calls.

In an object-oriented language with late-bound message passing, the task of selecting the appropriate operation for a given operation name has been delegated to the object itself, so the exact type (or class) of the object does not have to be known by the compiler in order for appropriate operations to be invoked at run-time, which is why some languages have completely eliminated static type-systems in favor of the dynamic typing already implied by the object-oriented model. This abolition of typing information at the interfaces, however, leads to pre-conditions that are too weak to ensure run-time type-safety, because objects can only select an appropriate implementation for a message if they actually have an implementation for that particular message, something that isn't and cannot be guaranteed.

Static typing ensures that all invoked operations are actually available and also has positive documentation and learning-curve effects [BG93]. However, the type systems of current statically typed object-oriented languages such as Eiffel and C++ insist on the unnecessary check whether the receiver of a message is a member of

a particular class when all that is relevant in object-oriented systems is whether an object responds to a particular message, messages being the only valid means of interacting with an object[ISO95a]. This undue restriction eliminates many useful constructs from the vocabulary of a language, such as plug-compatible objects not sharing a common ancestor, and make refining frameworks[MN93] and composing independently developed components[Höl93] unduly difficult or downright impossible. For these reasons, static typing is frowned upon by users of dynamic languages and frequently subverted by users of static ones, for example using type-casting in C++ or unsound parts of the type system in Eiffel.

Introducing protocols, sets of message signatures, as independent entities allows the definition of a type-system that respects the object-oriented model's assertion that an object's only visible interface is through its messages. This *interface typing*, which separates the type-system, defined via protocols, from the implementation hierarchy, is supported in many newer object oriented languages such as Java, Objective-C or Emerald, and is being actively retrofitted to both statically types languages such as C++ [BR97] and dynamically typed ones such as Smalltalk [BG93]. In addition, object-oriented integration mechanisms such as CORBA [OMG95] and SOM [CCD+92] feature first-class interface definitions without actually providing a native implementation mechanism.

Although first-class interfaces are a good start at defining safe interactions, they still don't deal with the preconditions of two important facets of object-oriented development: class refinement via inheritance and the specification of object collaborations.

3.2 The Refinement Interface

Object-oriented systems, or adaptable systems in general, have two types of users: in addition to clients that just wish to use the system as is, they also have to accommodate *specializers*, who wish to adapt the system to their own special needs, to refine all or parts of the system. This class of users needs more information about the internal structure of a class than the client interface can and should provide. Some language, for example C++, make it possible to selectively restrict visibility of operations and state to just subclasses or the defining class, but still don;t provide any additional structural infomation necessary for user code to run safely inside the encapsulation boundaries of the class [KL92].

3.2.1 Protocol Dependencies

Lamping [Lam93] suggests that much of the structural information needed by specializers is how methods in a class depend on one another. This type of information can be captured by a simple technical measure: extending type information for a class's operations to include not only the argument and return types of the operation, as defined by the message protocol, but also the facilities of the receiving class used by that operation.

The basic assumption behind this proposal is that well designed classes have an internal organization to how their methods rely on each other and on the hidden

```
@interface AbstractSet : Object
{
}
...
-(void)add:newObject;
-(void)addAll:(id <set>)otherSet;
@end

@interface HashSet : AbstractSet
{
   id linkedList;
}

-(void)add:newObject;
@end

@interface PersistenSet : AbstractSet
{
   id hashTable;
}

-(void)addAll:(id <set>)otherSet;
@end
```

Figure 3.1: Abstract Set and concrete subclasses

state of the class. A typical structure is that of core methods manipulating and accessing state information, with other methods implementing the rest of the exported interface in terms of these core methods. It is this type of structuring that turns method overriding into a reuse tool with consistent and predictable consequences.

Consider the partial interfaces of an an abstract set class and two concrete subclasses in figure 3.1. The abstract class defines two methods for adding elements, one adds elements singly, the other entire sets of them at a time. A well designed abstract class will leave one of the two methods as an abstract operation to be implemented by a concrete subclass and define the other in terms of that abstract operation. However, the interface does not make it clear to the inheritor which is the abstract operation to override and which the already implemented one relying on the former.

It may be considered "obvious" that the the simpler **add** operation is abstract and **addAll** implemented in terms of **add**, an assumption made by the designer of the concrete **HashSet** subclass. However, a designer with a database background may well have fixed transaction overhead in mind when making **addAll** the abstract operation that **add** relies on. Whichever case is correct, one of the two subclasses is making an incorrect assumption, and only a look at the implementation of

```
@implementation AbstractSet

-(void)addAll:(id <set>)otherSet
{
   for ( each element in otherSet )
   {
      [self add:element];
   }
}

-(void)add:element
{
   [self subclassResponsibility];
}

@end
```

Figure 3.2: AbstractSet implementation

AbstracSet in figure 3.2 will reveal which is correct.

Making dependency information explicit in the interface so that subclasses know which methods they can and must override without source code availability can be accomplished by two simple extensions of an object oriented programming language's type system. First, a class's interface is partitioned into the separate interfaces implied by the layering above. Second, the implicit **self** parameter is typed, using interface typing, just like any other parameter. This treatment of self as a parameter can be justified by observing that in the context of a superclass method and with subclasses present, a method may encounter a certain amount of variability in its **self** parameter. With these additions in place, a method can restrict its self-type to be a one of the partial interfaces defined for the class instead of the full class interface that is the default.

The example in figure 3.3 shows how self-typing could be expressed in an extension to the Objective-C language. The required type of the self parameter is expressed in square brackets between the return type specification and the start of the selector name. In Objective-C, **Set*** is an object reference typed to be an object of class Set or one of its subclasses, **id** is a generic object pointer similar to Smalltalk object references and **id <set>** is an object reference that guarantees conformance to the set protocol.[1] The specification of the actual class in the dependency declaration indicates that a method relies not only on the full protocol of the class, but also on its internal representation. For backward compatibility a missing dependency specification should be equivalent to this full dependency specified by the class name. An empty dependency declaration, on the other hand,

[1] Objective-C syntax is used instead of the modified Eiffel syntax used in [Lam93] because it has notation for protocols as well as typed and untyped objects

```
@protocol set_core
-(void)add:newObject;
...
@end

@protocol set <set_core>
-(void)addAll:(id <set>)otherSet;
...
@end

@interface AbstractSet <set> : Object
{
}

// — specialization interface follows

-(void)[Set*]add:newObject;
-(void)[<set_core>]addAll:(id <set>)otherSet;
...

@end
```

Figure 3.3: AbstractSet specialization interface

indicates that a method follows a functional protocol, depending only on the explicit arguments, indicating to a caller that the result of a call to such a method could be cached without adverse effects.

3.2.2 Reuse Contracts

The need for predictable yet flexible composition and refinement interfaces, *grey box* interfaces, has also been a motivation for the development of *reuse contracts*[LSM97], a formal documentation technique for structural dependency information that builds on the concepts presented in the preceding section. Reuse contracts extend the notion of structural dependencies to cooperations between classes and specifically try to address the problems of separately evolving core and client classes, a situation that frequently arises when frameworks are used[SLMD96].

The method dependencies present in the **Set** example of the previous section would be encoded using the following reuse contracts:

Reuse Contract abstract Set

abstract	
add(Element)	{}
concrete	
addAll(Set)	{add}

end

Reuse Contract HashSet concretizes **abstract**Set
  concrete
    add(Element) {}
end

Reuse Contract PersistentSet concretizes **abstract**Set
  concrete
    addAll(Set) {}
end

Each contract specifies the abstract and concrete methods defined in a class, and each method specification indicates (in curly braces) what other methods are invoked during execution of the method.

The fact that **addAll** is implemented in terms of **add** in **AbstractSet** is immediately visible, and inspection of the combined reuse contract for PersistentSet – combination follows the usually subclass override semantics – shows that the resulting definition still has an abstract class, and should therefore not be instantiated, whereas all HashSet methods are concrete.

The explicit dependency information also means that a **Bag** class implemented in terms of set:

Reuse Contract Bag concretizes AbstractSet
    concrete
      incrementCount {}
      add(Element) { incrementCount }
end

can be sure that **incrementCount** will actually be called by **addAll** by simply following the dependency clauses, a task that can easily be automated.

A corrected definition of the PersistentSet, redefining **addAll** as the primary method used by **add**

Reuse Contract PersistentSet concretizes AbstractSet
    concrete
      add(Element) { addAll }
      addAll(Element) { - add }
end

removes the dependency from **addAll** to **add**, so applying the counting extension above to this method can be easily shown not to count the elements added with **addAll** just by noting that there is no longer a dependency path between **addAll** and **incrementCount**. Not knowing the internal structure of the **PersistentSet** implementation, an obvious "fix" might involves having the **Bag** extension reintroduce the dependency between **addAll** and **add** as follows:

Reuse Contract PersistentBag concretizes PersistenSet

concrete
incrementCount {}
add(Element) { incrementCount, super add }
addAll(Element) { add }
end

However, this leads to an unintended recursion, with **addAll** calling **add** and **add** invoking its superclass's definition of add, leading to another call of **addAll**. This problem can also be (automatically) detected by examining just the dependency information in the reuse contracts.

A more comprehensive treatment of reuse contracts, that extends the concept to interactions between different objects can be found in [Luc97]. It defines a *basic reuse contract* as a set of participant objects with standard signature-based object interfaces, extended by per-participant *acquaintance clauses* listing relationships with other participants and per-operation *specialization clauses* listing use-relationships between operations of the participants. Figure 3.4 shows a basic reuse contract in the graphical notation also introduced in [Luc97].

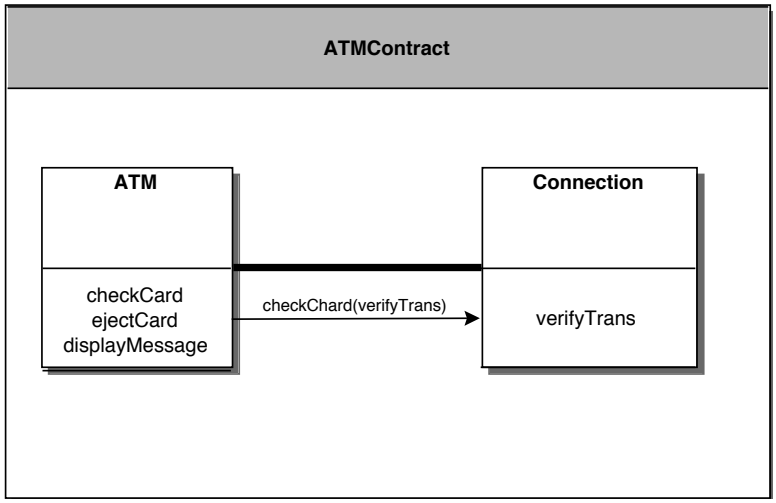

Figure 3.4: A basic reuse contract with two participants

Reuse contracts must be well-formed; this is the case if

1. every name listed in an acquaintance clause of one of the participants is itself a participant of the reuse contract and

2. for each operation-invocation $a.b$ in a specialization clause of an operation of participant p, a is listed in p's acquaintance clause and b is listed as an operation in a's interface.

Reuse Operators are used to model the evolution of software systems by acting on the basic reuse contracts defined above. The basic functions of the reuse operators are as follows:

- Extension and Cancellation add or remove elements of the reuse contract. They are self-contained: extensions may not refer to any elements in the reuse contract (just within the extension) and all elements that are referred to by a cancelled element must also be removed.

- Refinement and Coarsening add or remove dependencies between elements, operation invocations at the context level and acquaintance relationships at the participant level.

The formal definition gives conditions under which the well-formedness of reuse contracts is preserved. With the operators modelling changes to the class library, it becomes possible to detect *reuse conflicts*, including the example problems presented above, through a purely formal analysis. Some conflicts can be detected by just examining the reuse operators, others require application of the operators to the base reuse contract(s), and computing the resulting reuse contract.

These primitive reuse operators can also be combined to form more semantically rich operators and extensive operator chains. Combined application of all the operators in a chain can be shown to cause at most the same, but often fewer conflicts than the operators applied individually due the fact that conflicts may cancel each other out, a phenomenon also examined more closely in [Luc97].

3.3 Interaction Interfaces

Having extended the refinement interface to include structural dependency information, we now examine the client interface more closely, and find that just specifying the operations available at an interface is often not sufficient to fully characterize the interface[LVM95]. Instead, interfaces need to specify what interactions between collaborating objects, naming the kinds of potentially bidirectional collaboration activity possible at a given interface.

3.3.1 Interaction Protocols

The Interaction Protocol Definition Language [Bok96b] adds the ability to describe multi-role interactions to the OMG's CORBA Interface Definition Language.

IDL *interface* definitions are replaced by *interaction* definitions in IPDL, which include roles for the possible participants as well as actions, potential invocations of operations of one role by another role. Standard CORBA-IDL interfaces can be expressed as interactions with only two roles (client and server) and all actions unidirectional from client to server.

As in normal interfaces, the order of action declarations in an interaction definition does not indicate a run-time ordering of the actions (operations), but such an ordering can be defined in a protocol section allowing repetition, sequential composition, and alternatives. The expressiveness of the protocol language is restricted to that of regular expressions to allow static checking of interaction definitions.

Actions in interaction definitions can themselves be interaction definitions, allowing arbitrary hierarchical nesting. More recently, interaction specifications have

been applied to the specification of object interactions in frameworks [Bok96a], with the aim of extending OO language type systems with protocol information.

```
interface Manager
{
   void lock( in MCLient c);
   void unlock();
}

interface MClient
{
   short getID();
}
```

Figure 3.5: CORBA IDL Interfaces for a locking protocol [Bok96b]

```
interaction Mutex<inner=nop>
{
   roles MClient,Manager;
   Oneway( MClient, Manager ) void lock();
   Invocation( Manager, MClient ) short getID();
   Oneway( Manager, MClient ) void lock_granted();
   Invocation( MClient, Manager ) void unlock();
   protocol:
   loop
   {
      lock <
         getID;
      >;
      inner;
      unlock;
   }
}
```

Figure 3.6: Parametrized IPDL mutex interaction [Bok96b]

3.3.2 Protocol Specification and Matching

A more direct application of interaction protocols to conventional interfaces is given in [YS97]. Instead of replacing the participating interfaces with a single multi-role interaction specification, object interfaces are extended with fixed client/server roles (the current interface is always in the server role), the messages made available

by client and server in this collaboration and a protocol describing the sequencing constraints at this interface. The sequencing constraints are expressed using a finite state grammar with named states and one transition for each message that can be sent or received from a particular state:

```
<state> : <direction> <message>  -> state
```

where `state` is the symbolic name of a state; `<direction>` indicates either outgoing ("-") or incoming ("+") messages; and `message` is the name of an appropriate incoming or outgoing message defined in the interface.

The question of interface compatibility is extended from normal type constraints to include protocol matching, protocols are said to be compatible if there are no unspecified message receives and the protocol is deadlock free. Compatibility is weaker than equivalence in that the receiver may offer to receive more messages than the sender needs.

A sub-protocol relationship is introduced that allows protocol-compatibility preserving sub-protocols to be defined in an analog fashion to sub-types that preserver type-compatibility. Adapters can be generated that will automatically bridge some discrepancies between fundamentally compatible protocols.

3.4 Commentary

Procedural type systems obviosuly need to be extended to make static typing viable for object-oriented programs. With interfaces becoming separate, shareable entities, the idea of enriching interfaces with even complex additional information becomes much more tenable than it currently is.

Chapter 4

Orthogonalization

One surprising aspect of composition and refinement in current object-oriented languages is that they are not orthogonal to the concept of class and object, but tightly bound to one or the other. While classes can be refined at compile-time, objects can be composed at run time. Neither class-composition nor dynamic object-refinement are directly expressible, forcing developers into implementation mechanism that don't reflect the actual design. Examples include uses of object-composition in order to support static but parametrized functionality, and subclassing (with delegation) to support dynamically refined object behavior.

Instead of investing great deals of time and energy in documenting these workarounds and packaging them for reusability, the following approaches try to directly address the problem by removing the deficiencies that make the workarounds necessary. While most of these approaches focus on just one of the two missing features: refinable objects or composable classes, there is some work in making code composition techniques applicable at run-time.

4.1 Delegation

Delegation takes the theme of replacing subclassing with object-composition to its logical conclusion by generally expressing variable behavior with object-composition. Instead of implementing some concrete behavior itself, an object delegates all messages regarding this behavior to another object, the delegate. Among the chief advantages of delegation is that it allows fine-grained variability in object-behavior at run-time as well as in many cases eliminating the need for additional subclassing if the need for a varying behavior can be met by an already existing class that's used as a delegate.

General support for delegation is fairly easy and widespread in dynamic languages such as Smalltalk and Objective-C, due to the fact that the part of the delegate object can be specified fairly loosely and dynamic runtime systems make it easy to implement generic forwarders, proxy objects that delegate all their behavior to another object and can be subclassed create specific forwarder instances. More rigid type systems require the creation of completely separate classes with no ability to share the generic forwarding behavior, turning the notion of delegation from a single class of objects to a series of design patterns [GHJV95].

However, even dynamic languages suffer from the "SELF problem": when a delegate object refers to itself using the **self** pseudo variable, it is not clear whether this should actually refer to the original object, which may not be available to the delegate[Lie86]. In other words, is the delegate loosely coupled to its parent, in which case self messages should probably go to the delegate object itself, or is it really considered to be a tightly coupled part of the parent object, in which case it *should* probably send self messages to the parent, even though it may not be able to do this because it has no reference to the parent and the particular message did not provide one as a parameter

The SELF problem can be ameliorated through the use of conventions like always including a pointer to the parent in calls to the delegate [Vil95], but a general solution requires specific language support for delegation.

4.1.1 Prototypes Languages

Representing generalizable knowledge about objects using prototypes is an alternative to the abstract-set approach inherent in class-based object-oriented programming. Instead of defining the class of all objects that share some properties and behavior before being able to create instance of that class, prototypes represent an exemplar, an actual instance of a concept and then define other instances of the concept relative to that prototype [Lie86].

In object-oriented programming, the idea of prototypical objects can be used to abolish classes as stores of common behavior, relying instead on inter-object delegation of responsibilities to model shared behavior.

The SELF language is a prototype language that incorporates a generic delegation mechanism and derives its name from the special treatment it gives to the pseudo-variable self in order to appropriately deal with the SELF problem, for which it chooses the tight-coupling interpretation of delegates[CUCH91].

A SELF object consist of a number of named slots holding objects, references to other SELF objects. All access to the slots of an object is via messaging, if the slot contains a method, it executes itself, a plain object just returns itself. Changing the contents of a slot can be allowed by associating a slot with a special assignment slot that will set the value of its associated slot when messaged.

Inheritance is modeled by marking slots with a parent attribute; if no matching slots are found for a message to an object, the parent slots are searched in a customizable order, allowing sharing of methods as well as data. This approach to inheritance eliminate the distinction between compile-time inheritance and run-time object-composition found in class-based object-oriented languages, as all sharing, whether using inheritance or object-composition is based on run-time instances.

4.1.2 Composition Filters

Instead of replacing class-based inheritance with delegation, composition filters [Ber94] add delegation to class based languages. as part of an approach that adds limited reflection on messages to a base language in order to support a wide range of domain-specific techniques in a single framework. The composition filters object model shown in figure 4.1 extends a *kernel object*, which could be defined

using almost any current object model and language, with an *interface part* that is specifically designed to mediate access to that object's operations.

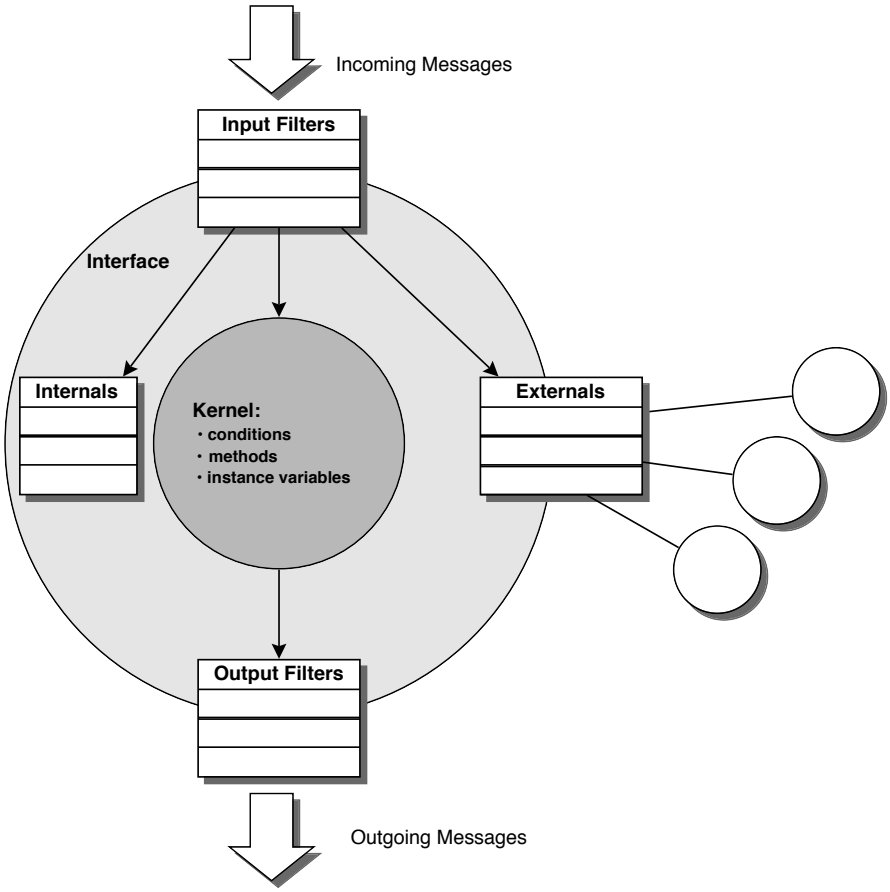

Figure 4.1: Composition Filters Object Model

The mediation is accomplished via *composition filters* contained as two ordered sets in the interface part of an object, one set for filtering incoming and one for outgoing messages. A message sent to the object is first passed through all the input filters until it is either dispatched or discarded. Each filter consists of three parts: a *condition* that has to be true for the filter to activate, a *matching part* specifying a pattern for the target and selector of the messages this filter acts upon, and a *substitution part* specifying optional replacements for bot the selector and the target. An incoming message can therefore be ignored, substituted with another message and/or dispatched to a different object, all without requiring changes to the kernel object.

The difference between tightly coupled and loosely coupled components of an object is made explicit by making objects either *internal*, in which case they are a fully encapsulated part of the enclosing object, or *external*, which refers to to external objects with their own lifetimes. The SELF problem is also addressed by making distinctions explicit in the language, with pseudo-variables identifying all the participants of a message send: the *sender* refers to the object that originally sent the message, the *server* refers to the interface part the original receiver of the

message, before any filtering that may have altered the receiver, *self* is the interface part of the currently active object and *inner* refers to the kernel part of the currently active object.

The purposely limited approach to reflection taken by composition filters has been used to provide solutions to at least examples of several problems in object-oriented development, including the inheritance anomaly of concurrent objects, implementations of dynamic behavior such as states or dynamic supertypes and support for abstract communication types that abstract patterns of communications and synchronization between objects [AB92].

4.2 Mixins

In Smalltalk and most other object-oriented languages, refinement takes the form of implementation inheritance, a mechanism for deriving the code of a new class from an existing class by implementing the differences, the *delta* from the existing class in the new class.

This process can be described formally by modeling classes, or rather class descriptions, as functions mapping selectors to classes[OH92]. Instance variables can be handled in an analogous fashion. Having defined a class as the function

$$d : \text{selectors} \rightarrow \text{methods} \tag{4.1}$$

and the class combination operator $\oplus$ as

$$d_1 \oplus d_0 = d_1 \cup \{(s \rightarrow m) \in d_0 | s \notin domain(d_1)\} \tag{4.2}$$

Smalltalk-style inheritance with the usual semantics of newly defined methods in a refinement $\square$ overriding the ones inherited from the base class d_0 can now be defined as follows:

$$d_1 = \square \oplus d_0 \tag{4.3}$$

A subclass d_1 is derived from a base d_0 by writing a refinement $\square$ which is combined with the base class using the class combination operator $\oplus$. Even though equation 4.3 refers to the refinement with $\square$, Smalltalk-style inheritance hides this refinement in the definition of the new subclass, whereas both the base class and the derived class are named and can later be reused,

Bracha and Cook [BC90] turn refinements, *mixins*, into the primary definitional construct and model inheritance as the composition of mixins. Mixins are abstract subclasses that can be applied to different base-classes to define new concrete subclasses.[1]

Mixins can be composed $m_2 = m_1 \oplus m_0$ to yield new mixins or applied to classes $d_1 = m \oplus d_0$ to achieve Smalltalk-style inheritance. Behavior identical to Smalltalk-style inheritance can be achieved by allowing anonymous mixins to be applied inline in a subclass definition. Reversing the parameters of the composition $d_1 = d_0 \oplus m$ yields Beta-style inheritance, where base class definitions always

[1]The treatment given here differs from [BC90] in that the class-description from [OH92] is used, **super/inner** is not modelled explicitly and mixins are not treated as a generalization of classes

take precedence over subclass definitions. Subclasses can add methods, or provide extension to existing methods, which are invoked by the superclass through use of the **inner** pseudo variable, which is similar to **super** in Smalltalk except that it allows calls from the base class to the subclass instead of the reverse.

4.2.1 Mixins and Multiple Inheritance

Mixins can be implemented by an idiomatic use of multiple inheritance in languages supporting MI; the first implementations of mixins using this technique in CLOS – Common Lisp Object System – actually predate the presentation of the mixin formalism. The idiom is simple: mixins are declared either as base-classes or as direct subclasses of the base-class if user-defined base-classes are not permitted by the language or environment; they can then be mixed-in using normal multiple inheritance. Due to the linearization of the inheritance graph performed by CLOS, the mixin is provided with a parent.

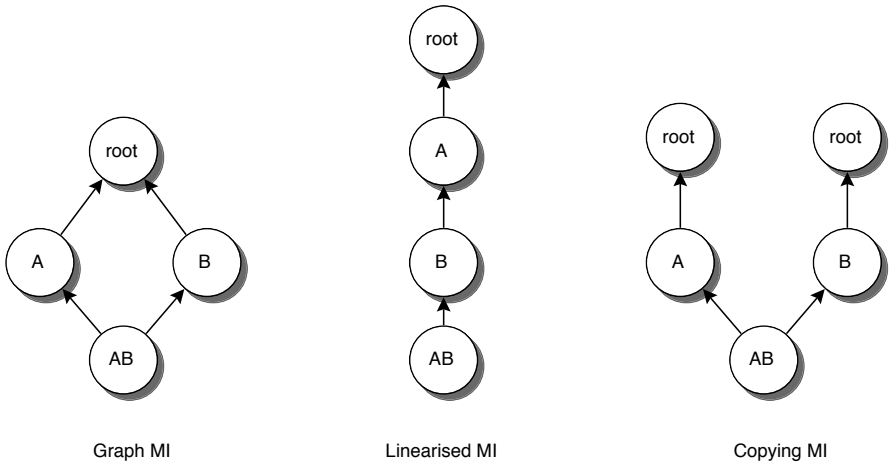

Figure 4.2: Multiple Inheritance (MI)

Mixins can thus be seen as a subset of multiple inheritance that avoids the technical problems associated with the conflicts and ambiguities inherent in the multiple inheritance model's inheritance graph and absent from the inheritance tree found in single inheritance systems.

4.2.2 Nested Dynamic Mixins

Instead of mixins being a special case of a general multiple inheritance mechanism, the language Agora [SCD+93] implements its inheritance mechanism solely through the use of mixins. A class defines all the mixins that are applicable to it in *mixin methods* that can be invoked just like ordinary methods by sending messages to objects of that class. The mixin method defines additional state and methods, including mixin methods, to be added to a base class. When a mixin method of an object is invoked, an object that has its class set to the class resulting from the application of that particular mixin to the object's current class is returned, the object's class therefore extended dynamically at run-time.

The mixins of a class can only be applied to that class or one of its future subclasses, so the class normal hierarchy also defines a hierarchical namespace for mixins, though global availability can be effected by defining all mixins in the root class. Polymorphism also applies, so subclasses can redefine a mixin attribute in order to provide a more appropriate extension than the one provided by the base class.

4.3 Extension Hierarchies

Whereas mixins allow the extension of single classes, extension hierarchies are applicable to entire inheritance hierarchies or arbitrary subsets thereof. Extension hierarchies were introduced in [OH92] with the stated goal of allowing extensions to existing classes, for example to avoid invalidating already instantiated persistent objects or object creation code in the face of software updates, but also to solve intergroup communication bottlenecks that can arise from the centralized class ownership implied by the normal object-oriented encapsulation boundaries [Bat96].

The extension mechanism proposed clearly separates extensions of all kinds from the base hierarchy, as extensions are made in separate, usually sparse inheritance hierarchies. Successive extensions can be combined using an *extension* operator, parallel extensions using a *merge* operator. Even changes to the base hierarchy such as bugfixes can be encoded this way and thus benefit from the conflict catching mechanisms encoded in the operators. Figure 4.3 shows how a simplified shape hierarchy can be extended with a color attribute without the need for destructively editing the base class.

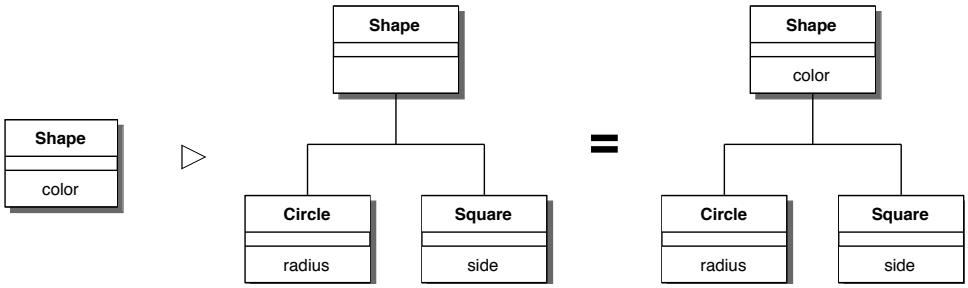

Figure 4.3: Extending a shape hierarchy with a color attribute

4.3.1 Extension Operators

The definition of the two extension operators $\triangleright$ (*extend*) and $\diamond$ (*merge*) uses the modeling of class-records as functions mapping names to selectors introduced in section 4.2. An inheritance hierarchy $H$is defined as a triple $H = (N, D, S)$, with N a set of class names, a function $D : N \rightarrow \text{class} - \text{descriptions}$ mapping class names to actual class descriptions and a *superclass function* $S : N \rightarrow seq(N)$ specifying the sequence of immediate superclasses of each class.

Both the class description function and the superclass function can be partial, leading to incomplete hierarchies. Only complete hierarchies, with total superclass

function and total class description function are actually executable, incomplete hierarchies can only be combined to form complete hierarchies.

The class extension operator $\triangleright$ is defined relative to the class-record combination operator $\oplus$ of the base language to which hierarchy combination is applied. The extension of a class hierarchy $H_0 = (N_0, D_0, S_0)$ by an extension hierarchy $H_1 = (N_1, D_1, S_1)$, denoted $H_1 \triangleright H_0$, is defined as follows:

$$H_1 \triangleright H_0 = (N_1 \cup N_0, D_1 \triangleright D_0, S_1 \triangleright S_0)$$

with the subdefinitions of $\triangleright$ being

$$\begin{aligned} D_1 \triangleright D_0 \quad = \quad & \{(n \to d_1 \oplus d_0) | (n \to d_1) \in D_1 \wedge (n \to d_0) \in D_0\} \cup \\ & \{(n \to d_1) \in D_1 | n \notin \text{domain}(D_0)\} \cup \\ & \{(n \to d_0) \in D_0 | n \notin \text{domain}(D_1)\} \cup \end{aligned} \tag{4.4}$$

for the class-description function and

$$S_1 \triangleright S_0 = S_1 \cup \{(n \to q) \in S_0 | n \notin \text{domain}(S_1)\}$$

for the superclass function, respectively. The use of a set union to combine the class names, together with the use of overriding for duplicate methods means that classes can be extended *in-place*, that is functionality can be added to a class without creating a subclass. The other terms state that both the base hierarchy and the extension can define new classes, and that the extension can arbitrarily redefine subclassing relationships, though this is not a recommended operation.

The intended application of the hierarchy merge operator $\diamond$ is the integration of independently developed extensions before applying them to a base hierarchy. The independence of the extensions is enforced by requiring that neither extension overrides the other, which is the case if the $\triangleright$ operator becomes commutative:

$$H_1 \triangleright H_0 = H_0 \triangleright H_1$$

If this equation doesn't hold, both extensions have made changes to overlapping parts of the base hierarchy, making them *conflicting*. Such conflicts have to be resolved manually, so merge is defined as a partial operation that is defined only when the two hierarchies to be combined are *non-conflicting*:

$$H_1 \diamond H_0 = \begin{cases} H_1 \triangleright H_0 & \text{if} H_1 \triangleright H_0 = H_0 \triangleright H_1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Applying the merge operator to independently developed extensions ensures that these extensions don't override each other unwittingly.

4.3.2 Conflicts: Detection, Resolution, Avoidance

Although it is possible to detect conflicts as defined above using the provided mechanism, freedom from conflict does not assure semantic compatibility, nor does presence of conflict necessarily mean that extensions actually interfere. Although work has been done on automatically integrating non-interfering extensions using

program slicing techniques as well as control and data dependency graphs [HPR89], these techniques are currently not applicable to real-world programming languages with inheritance and polymorphism, forcing hand-integration of the changes in the presence of conflict. For ways in which non-conflicting changes could interfere as well as methods to automatically detect this type of interference see the discussion of reuse-contracts in section 3.2.2.

Conflict avoidance can be aided by the concept of *partitioned extensions* which follow the general rule of *modification by addition*, meaning that modifications to classes should be additive instead of overriding whenever possible. Extension using only subclassing automatically leads to partitioned extensions if class-name resolution mechanisms for extensions are implemented, because each extensions has its own namespace and therefore cannot conflict with other extensions. Other techniques have been developed to extend the kinds of additions that can be made with assured conflict-freedom:

1. method subdivision: Method dispatching can be extended to include criteria other than the class of the receiver, for example the classes of one or more of the additional arguments (if any), implementing multiple-dispatching, or even on the values of some of the arguments. Composition filters (see section 4.1.2 allow exactly this type of user-defined dispatching, with an even more general dispatch-mechanism.

2. "structure-bound" messaging: dispatching messages to several receivers who may or may not be interested in them. This is similar both to Adaptive Programming's Propagation Patterns and Implicit Invocation.

3. instance variable access methods developed by the authors that is independent of the exact layout of objects in memory.

The extensions attainable by these means will not interfere with each other, but neither will they cooperate much, complementing each other instead.

4.4 Canonic Components

4.4.1 Components

Components in the □-Language [SG94] follow the object-oriented model by allowing components to only access data types (classes) and their associated operations, but also extend this model by allowing each component to define more than one data type at a time. Each component consists of four parts, the import, body, export and common parameters sections.

The export section defines the exported types and services, the body section contains the implementation and the import section specifies the features required by the component. Binding to components that export these features is done in a separate step, the component itself is self-contained, it does not contain references to other components. The common parameters section allows imported features to be exported unchanged, a feature that allows even strictly layered systems to provide low-level features in their higher-level interfaces.

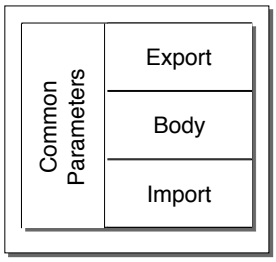

Figure 4.4: The structure of a Component

Components can be composed into larger components by matching the import requirements of one components with the exported features of another. The resulting component composition graph is restricted to being acyclic in order to facilitate verification, maintenance and understandability of the composed system.

4.4.2 Views

A component's sections are specified in views, each view being a partial specification of some aspect of the current section, The interface sections (export,import), contain three views:

- the **type view** specifies the the data types using algebraic specification techniques[EM85] by defining the exported operation signatures and defining their properties with equations;
- the **imperative view** specifies side effects by providing the imperative signatures of the defined operations (enriching the signatures with the classifications **in**, **out** and **return** for the parameters;
- the **concurrency view** defines possible orderings of execution using a variant of path expressions[CH74].

The body section of a component defining a single data type contains a type view specification that defines the construction of the exported types from the imported types using the same algebraic techniques as the export interface's type view. It also contains an imperative-view specification implementing the algorithms and data structures of the module in a conventional procedural programming language.

Instead of implementing a single data type, a component can also specify a composition of other components, in which case the body, the configuration specification, consists of the following parts:

- the list of constituent components,
- the definition of the component interconnections,
- the mapping of exported data types of the constituent, interfaces to the component's export interface as well as the mapping of constituent import interfaces that haven't been resolved in the composition to the component's import interface.

In addition to allowing generic parametrized modules such as a generic list, □'s component model also allows partial compositions that only satisfy some of a component's import requirements, resulting in a further parametrizable module.

4.5 Commentary

Extension hierarchies provide a general mechanism for in-place class extension, a mechanism that is sorely missing from most of current object-oriented methods[2] despite the fact that it is not only useful for independent development of shared classes, but also as an integration mechanism that does not require either source code access, retyping or wrapping of changed classes. However, the mechanisms presented are just that, pure mechanism without an underlying model. That deficiency is later rectified by the authors with the introduction of subject-oriented programming (see section 6.2), though extending the mixin concept to support in-place extension and simultaneous mixing-in of class hierarchies seems like a less radical addition to current object-oriented notations.

□-Components also extend the granularity of the basic units of software constructions to entities larger than single classes, and add the idea of multiple views on different properties of software, but fall short of even present-day object-orientation by insisting on a pure hierarchical structure without overriding, reducing opportunitie for incremental programming and client-parametrizable library code.

[2] A notable exception being Objective-C categories

Chapter 5

Software Architecture

With object-oriented technology facilitating the development of complex software systems, there is a growing need to consider the overall structure of these systems and the ways that this structure provides conceptual integrity for the system, or fails to do so. This level of design, dealing with the composition and interconnection of the top level, typically module-sized, interacting components of a system is referred to as *software architecture*.

In addition to the coarse grain structure of a system, an architecture also typically is the link between requirements and design, showing how the required features are to be reflected in the final software system. Current practice typically describes architectures with box-and-arrow drawings as well as informal, idiomatic descriptions of the overall *style* to be used, for example a "*client-server* architecture", a "*layered* system" or a "*dataflow* driven" design. Being informal, these notations don't help much in bridging the semantic gap between requirements and implementation languages.

Current research in software architecture is aimed at narrowing this gap by making explicit the notion of system composition from subsystems, an activity substantially different from programming the underlying algorithms and data structures. To this end, notations in the form of architecture description languages (ADLs) are being developed and existing architectures classified into styles Due to the youth and breadth of this field, no general unifying picture or underlying formalism has emerged. Instead, there is a variety of sometimes contradictory formalisms, notations and approaches focusing on different aspects of the problem space.

One area of agreement though is that architecture is generally concerned with the *components* of a system, the *connectors* that bind components together and the overall structure of how connectors and components are combined to form a system.

5.1 Architectural Styles

Software developers recognize a number of distinct architectural styles as indicated by the use of idiomatic phrases in describing architectures. A style is a set of design rules that identify the kinds of components and connectors that may be

used in composing a system and constraints how they may be used [SC96]. Examples of constraints are interconnection topologies, synchronization requirements, directionality of data flow and control issues.

Common architectural styles include the following broad categories [SG96]:

- **Dataflow** systems such as pipes-and-filters or batch-sequential processes;
- **Call-and-return** styles including procedural and object-oriented variants; layered systems imposing additional topological constraints are also often implemented as call-and-return systems;
- **Independent components** including communicating processes and event systems;
- **Virtual machines**, interpreters and rule-based-systems;
- **Data centered systems** such as databases, hypertext systems and blackboards.

These styles neither represent an exhaustive list, nor are they always mutually exclusive, a fully orthogonal classification of architectural styles not having been found yet. Three of the substyles loosely corresponding to the operation, action and dataflow interactions specified by the RM-ODP [ISO95b] are presented below, with the Key Words in Context problem proposed by Parnas serving as a common illustrative example expressed in different styles.

5.1.1 Pipes and Filters

Probably the most widely known examples of the pipe-and-filter dataflow oriented style are UNIX shell scripts, with independent commands transforming data coming from the *standard input* stream to the *standard output* stream and a controlling program such as a shell setting up the proper communication paths between the individual components according to some description of the dataflow between the components. A UNIX shell command for implementing the Keywords in Context problem might look as follows, given appropriate individual filters:

```
<infile input | circular-shift | alphabetize | output >outfile
```

UNIX filters are actually a specific instance of the pipe-and-filter style. More generally, components in a pipe-and-filter style take data from one or more inputs and output them to one or more output, whereas connectors transport data from one of a filter's outputs to one of another filter's inputs. The component filters are completely independent of one another, not sharing any state with, passing control flow or even knowing the identity of any other filter in the system.

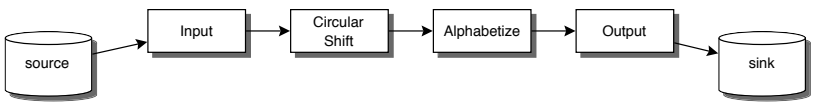

Figure 5.1: KWIC in a Pipes and Filters style

Among Advantages of pipe and filter systems are the syntactic interchangeability of the components, the fact that components need not know about their

collaborators and the ease with which the combined behavior can be treated as a simple composition of the individual behaviors. On the downside, it is difficult to use filters in situations where complex behaviors are required. The typical implementation of filters as separate tasks also leads to increased modularity and reusability, but also has the potential of significantly reducing performance where high data exchange bandwidths are required.

5.1.2 Call and Return

The call-and-return architectural style includes the widely used procedural, abstract data-type and object-oriented programming models, which won't be discussed in detail here. It also overlaps with styles based on independent overlapping processes such as client-server paradigms and, due to its availability in programming languages, is frequently used to implement other styles.

One significant drawback of this style is that the called entity must be known to the caller, enforcing a tight coupling between components. Although object-oriented programming loosens the coupling somewhat using late binding of method invocations, the receiver still has to be specified explicitly and object creation is still largely early bound.

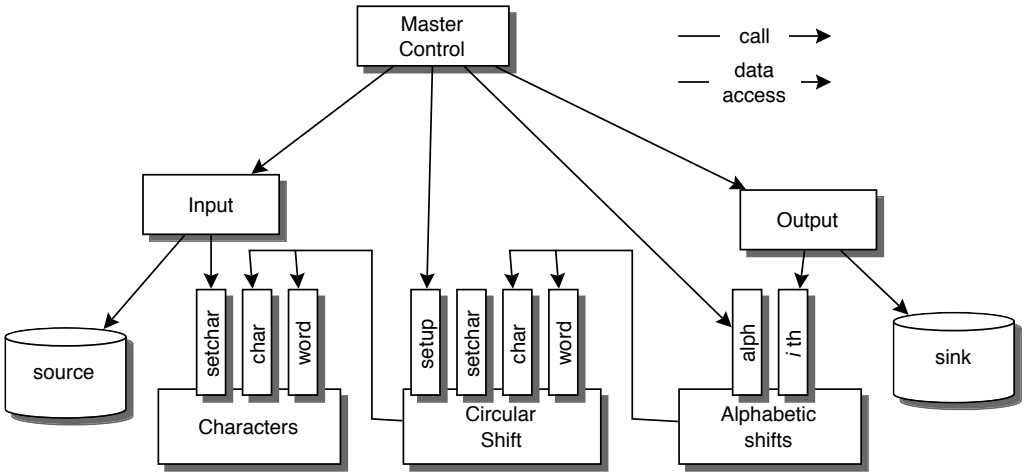

Figure 5.2: KWIC in an Abstract Data Type style

5.1.3 Implicit Invocation

Event based systems use implicit invocation, also known as selective broadcast or reactive integration, to decrease the coupling further than possible with pure object-oriented techniques. Instead of selecting a particular recipient for a message, a component simply *broadcasts* the message. A centralized dispatcher then forwards the message to all other components that previously registered an interest in receiving that particular message.

Implicit invocation systems provide strong support for reuse, because a new component can be introduced into a system simply by registering it with the dispatcher for the events that it requires, the other components need not be modified,

or even made aware of the change. Because components cannot know what, if any, processing takes place in response to an event they broadcast, implicit invocation is usually complemented with explicit invocation (procedure calls or OO messaging) in practical systems. Another issue is the provision of data with events, which must also support the broadcast nature of the event system.

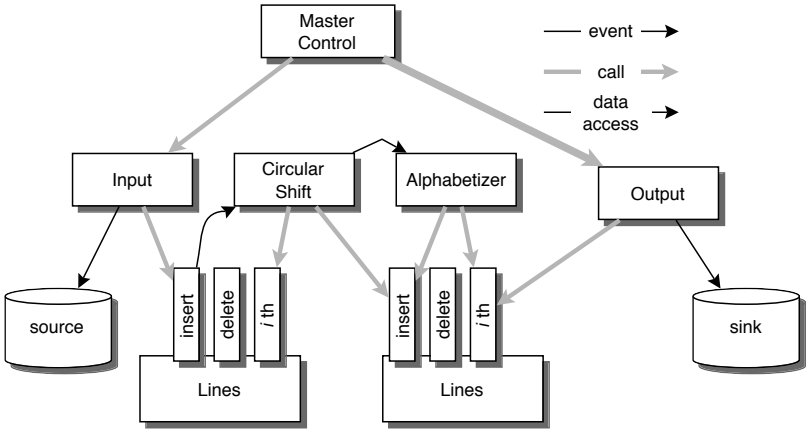

Figure 5.3: KWIC in an Implicit Invocation style

5.1.4 Using Styles

An explicit notion of architectural styles and their properties helps designer develop a clear understanding of the trade-offs involved in choosing one style over another, aiding in the selection of an architecture appropriate for a particular application and set of requirements. This type of design guidance can be encoded in design patterns such as [Sha95] and [Edw95], or using techniques that try to make the entire design space with its associated dimensions and trade-offs visible [Lan90][ASBD92].

Although tools for supporting developers in the use of specific architectural styles, such as the HP Softbench Encapsulator for composing particular event based systems, general support for style-based development and analysis is still in its infancy. The Aesop system (see figure 5.4) is a style-based meta tool, it facilitates the development of design tools suited for a particular architecturally style [Gar96a]. Styles are defined using a subtyping hierarchy that defines objects for the style-specific design vocabulary (connectors, components, roles and ports) and user-written methods that encode the stylistic constraints.

Another hope for explicit architectural styles is that they may at least be an aid in the detection and avoidance of architectural mismatch (compare section 2.1.2 and [GAO95]). Furthermore, *style based refinement* [Gar96b] could actually help in in bridging the gap by providing assistance for the common task of implementing one architectural style, used in the design for example, in terms of another that may be the only one available in a given implementation language.

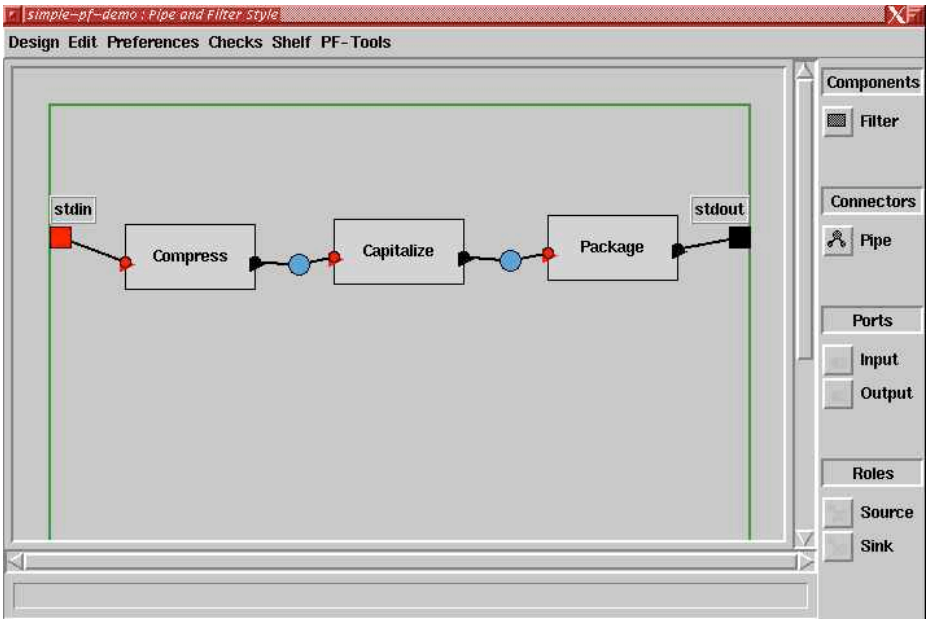

Figure 5.4: Aesop screendump showing pipe-and-filter style

5.2 Architecture Description Languages

Notations for architecture can server a variety of purposes, from simply capturing the architecture of specific systems to performing sophisticated analysis on entire families of architectures. All of the architecture description languages presented below share the notion of systems as configurations of components that are connected somehow, but beyond there is little agreement. UniCon is a constructive tool for generating systems from architectural description, Wright focuses static analysis of formally defined styles and Rapide on dynamic simulation of concrete systems. ACME represents a second generation effort aimed at providing a common notation for the common features of ADLs to aid interchange and serve as a basis for integrated extensions.

5.2.1 UniCon

The **Uni**versal **Con**nector language [SDZ95][Zel92] supports a variety of built-in typed connectors and has linguistic and tool support for automatically generating running systems from architectural descriptions of configurations.

Elements of the UniCon language include components defined by their interfaces and connectors defined by their protocols, with connectors specifying the rules of interaction for components. Interfaces and protocols are both typed, with type-specific properties to further specify specializations. *Players* respective *roles*, both also typed and with special properties further specifying the type, describe the possible points of interaction for components and connectors.

Configurations or systems are represented as components with a *composite* implementation consisting of *uses* statements declaring the parts to be composed, *connect* statements showing how the players of the provided components satisfy

the roles of the specified connectors and *bind* statements mapping the internal configuration to the declared external interface. The implementations of non-composite, *primitive* components are outside of UniCon's domain; all connector implementations are *built-in*.

Standard connector types defined in UniCon include local and remote procedure calls, pipes, file and data access, a real-time scheduler and a bundle type for managing groups of items. Each connector type is handled by an *expert*, a set of plug-ins to the basic UniCon language that manage connection rules and generate the code and build-rules necessary for turning connected components into running systems.

5.2.2 RAPIDE

The Rapide architecture definition language [Rap97], or rather family of languages (see figure 5.5), provides support for the definition and early life cycle prototyping of architectures using an executable model based on Partially Ordered Set of Events (POSETs).

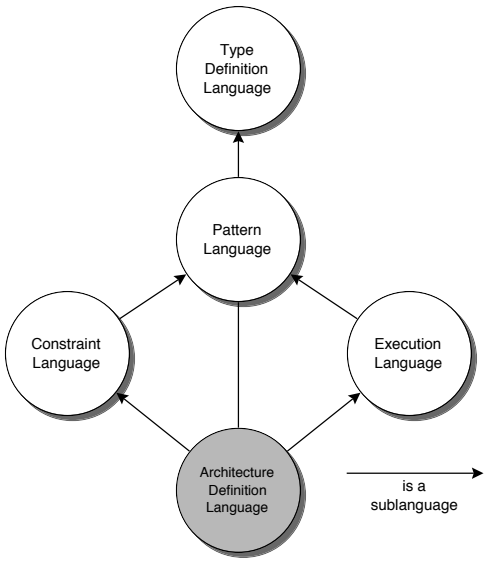

Figure 5.5: Hierarchy of RAPIDE languages

The Types language, which all other languages are based on, is a general language for defining interfaces of modules as sets of required and provided features, which can be functions or objects. All types are specified as interfaces and can form subtyping hierarchies; even pre-defined types are treated as special kinds of interfaces and can be defined by the user. The Pattern Language provides the means to define patterns of events with causality, independence and timing relationship, which are used by other sublanguages to model dynamic connections, reactive programming constructs and formal constraints. These pattern based formal constraints on the behavior of components and architectures are defined in a separate Pattern Language.

The Architecture sublanguage provides additional constructs for the interfaces provided by the Types languages as well as for specifying connections between

the components defined by interfaces. Interfaces can optionally include behaviors consisting of states and reactive transition rules allowing the interface of being both define the type of module and provide a simple module of that type to be used in simulations. Rapide does not provide first-class connectors, instead relying on simple matching of required with provided features. Scalability issues with this approach are handled by allowing sets of provided and required features to be bundled into services. Dual services, pairs of services that match provided for required features in both directions, can be wired using a single connection, implementing a plug-and-socket architecture [LVM95].

Rapide specifications can be executed for verification purposes, resulting in event traces that can be analized using specialized tools developed by the Rapide team. One unique feature of the traces generated by a Rapide simulation is that the events contain causality information, captured by the partially ordered sets of events.

```
class Parser is
   provides:
      C_P : service CodeGenerator_Parser;
      P_S : dual service Parser_Semanticizer;
   specification ...
end parser
class Semanticizer is
   provides:
      C_S : service CodeGenerator_Semanticizer;
      P_S : service Parser_Semanticizer;
   specification ...
end Semanticizer
class Code_Generator is
   provides:
      C_P : dual service CodeGenerator_Parser;
      C_S : dual service CodeGenerator_Semanticizer;
   specification ...
end Code_Generator
P : Parser; S: Semanticizer; G: Code_Generator
connect
   P.P_S to S.P_S
   G.C_P to P.C_P
   G.C_S to S.C_S
```

Figure 5.6: Sample partial Rapide architecture specification of a compiler

5.2.3 Wright

Wright [All97] is based on the notion that a blending of formal methods and purely structural architectural description languages is needed for defining architectural abstractions that are both amenable to formal and tool based analysis and practically useful. It is based on a subset of Hoare's CSP formalism, an algebraic model of processes with operators including sequencing, alternative and iteration, partly in

connector Pipe =
  role Writer = write!x → Writer ⊓ close → √
  role Reader =
    let ExitOnly = close → √
    in let DoRead = (read?x → -> Reader □ read-eof → ExitOnly)
    in DoRead ⊓ ExitOnly
  glue =
    let ReadOnly = Reader.read!y → ReadOnly
                □ Reader.read.read-eof → Reader.close → √
      in let WriteOnly = Writer.write?x → WriteOnly □ Writer.close → √
      in Writer.write?x → **glue** □ Reader.read!y → **glue**
                □ Writer.close → ReadOnly
                □ Reader.close → WriteOnly
      spec
        ∀ Reader.read$_i$!y • ∃ Writer.write$_j$?x •$i = j \wedge x = y$
        ∧ Reader.read-eof ⇒ (Writer.close ∧ #Reader.read = #Writer.write)

Figure 5.7: Pipe Connector defined in Wright[SG96]

order to benefit from the off-the shelf availability of tools for automated CSP model analysis.

Structural information is specified in Wright using *components*, *connectors* and *configurations*. Components represent computations with *interfaces* specifying *ports* at which interaction with the environment can occur. Interactions between components are governed by *connectors* that define *roles* for the ports of interfaces to fill. *Glue* further specifies the exact interaction patterns allowed at a connector, just as *computation* is specified in components.

All elements are specified as CSP processes as seen in the Wright connector specification in figure 5.2.3. The connector specified is a Pipe with two roles, a Reader and a Writer. The Writer role specifies that any participant in this role has two options: either writing a data item x and continue the connection or signal a close action and end the communication. Similarly, a participant taking the Reader role may terminate the communication before all data is read, but must also account for the situation that no more data is available. The permissible behaviors of potential participants having been defined, the glue section defines the actual behavior of the Pipe: data is either read by from the Writer or written to the Reader; if the Reader closes the connection the Pipe will continue to just read data from the Writer and if the Writer closes the connection the Pipe will continue to send remaining buffered data to the Reader until **eof** is reached. The specification part ensures that the data items written to the Reader are actually the same data items received from the Writer, in the same order, and that all data must be delivered before signaling **eof** to the Reader.

```
System KWIC
   ComponentInput
      port Out
      comp spec ...
   Component CircularShift
      port Unshifted
      port Shifted
      comp spec ...
   Component Alphabetizer
      port Unsorted
      port Sorted
      comp spec ...
   Component Output
      port In
      comp spec ...
   Connector Pipe
      ...
Instances
   in:Inport; shift:CircularShift; alphasort:Alphabetizer;out:Output;
   p1,p2,p3:Pipe;
Attachments
   in.Out as p1.Writer;
   shift.Unshifted as p1.Reader;
   shift.Shifted as p2.Writer;
   alphasort.Unsorted as p2.Reader;
   alphasort.Sorted as p3.Writer;
End KWIC
```

Figure 5.8: Key Words in Context configuration in Wright

5.2.4 ACME

ACME [DG97] is a more recent language intended to serve both as an architecture exchange language and as a base for future architecture research. In order to perform its function as a useful tool for architectural interchange, ACME directly supports the purely structural aspect of architecture as a hierarchical structure of interconnected components that most if not all ADLs share, while at the same time offering an annotation mechanism that allows non-structural information to be stored using externally defined sublanguages. A typed, parametrized macro mechanism supports abstractions such as architectural styles.

ACME *components* represent the computation elements and data stores of a system, interfacing via named *ports*, whereas *connectors* represent interactions having *roles* as their interfaces. *Systems* represent configuration of components hooked up via connectors, or more specifically with components' ports *attached* to the roles provided by connectors.

Hierarchical decomposition of architectures is supported by binding a higher level entity, component or connector, to one or more lower level *representations*, each representing a view of the decomposition, via *rep-maps*, representation-maps. In addition, every basic element can be annotated with attributes, structured/typed entities that are not interpreted by ACME.

5.3 Commentary

The architectural notations are a good fit for the need to specify the large and small-scale structural relationships in object-oriented systems, a lightweight approach like ACME could make a useful addition to both the designer's and the developer's toolset.

```
Family PipeFilterFam = {
    // Declare component types
    Component Type FilterT = {
        Ports { stdin; stdout; };
        Property throughput : int;
    }
    Component Type UnixFilterT extends FilterT with {
        Port stderr;
        Property implementationFile : String;
    }
    // Declare the pipe connector type.
    Connector Type PipeT = {
        Roles { source; sink; };
        Property bufferSize : int;
    }
    Template pipe(source_port, sink_port : Port)
            defining(conn:Connector) =
    { Connector conn = {
        Roles { source; sink; }
        ; Property bufferSize : int = 2048; }
    Attachments: { source_port to conn.source;
          sink_port to conn.sink; }
    }
    Property Type StringMsgFormatT = Record [ size:int; msg:String; ];
    Property Type TasksT = enum {sort, transform, split, merge};
} // end PipeFilterFam Family definition

System KWIC : PipeFilterFam = {
    Component input : FilterT = new FilterT extended with {
        // Implementation info for input filter
    };
    Component shift : FilterT = new FilterT extended with {
        // Implementation info for shift filter
    };
    Component alphabetize : FilterT = new FilterT extended with {
        // Implementation info for alphabetize filter
    };
    pipe(input.stdout, shift.stdin);
    pipe(shift.stdout, alphabetize.stdin);
    pipe(alphabetize.stdout, output.stdin);
};
```

Figure 5.9: Filter Style and Key Words in Context in ACME

Chapter 6

Non-linear Refinement

One of the goals of object-oriented programming, or functional decomposition techniques and commonality/variability analysis in general, is to cleanly state and encapsulate all the design decisions that went into a (sub-)system so they can be implemented, reasoned about and changed independently. In short, the program text should be a clean expression of how the program is supposed to work, self-documenting as demanded in [Bro95, page 169].

Current programming paradigms and languages fall short of this goal, as they only allow the decomposition of systems into functional modules, objects, subsystems or procedures. Many of the issues of concern that programmers have don't fit these abstractions, don't fall easily inside modular boundaries, but the non-linear relationships required for their variable implementation can only be achieved by connecting components at module boundaries. Object-oriented programming has helped by making these boundaries more flexible and the use of these boundaries as hot-spots for parametrization more pervasive, but there are limits to the granularity and global applicability of these methods due to overheads not only in run-time efficiency, but also in notational bulk, as each hot-spot requires an explicit message send or procedure call to implement. The result of this failure of abstraction is that the code for those concerns that are not easily modularized is spread throughout the system and becomes *tangled* with the code implementing other concerns.

6.1 Domain Specific Software Generators

Component libraries have been a successful application of object technology. However, current techniques face an inherent problem when trying to scale these libraries in the size and richness of the components offered while retaining the generality of components necessary for wide applicability. With largely orthogonal features to support, each new feature has the potential of at least doubling library size as each existing component must be subclassed or otherwise modified to add components that have this feature, a phenomenon known as *feature combinatorics* that lies at the heart of the "library scaling problem" [Big94].

The example most frequently cited to illustrate this problem is the Booch component library, a class library that provides primitive data structures. The library provides 17 abstractions such as stacks, queues, strings and sets, and 4

global feature dimensions that apply to all abstractions (in addition to some local feature only applicable to some abstractions):

- Concurrency control with 4 variations
- Garbage Collection with 3 variations
- Static or dynamic memory allocation
- Versions that provide an iterator and ones that do not

This small set of abstractions and features becomes tangled into 501 concrete components implemented in just under 150,000 lines of Ada code, or about 30,000 lines of C++ code, whereas the lower limit for implementing just the sum of the features is estimated at approximately a thousand lines of code[Big94].

One of the ways around the library scaling problem has been the creation of domain specific software libraries, fueled by the success of domain analysis, which follows the advice of Parnas[Par76] to regard software as families of related systems, instead of single systems taken from individual requirements to individual solutions. These libraries or frameworks capture the domain specific commonalities found in domain analysis, leaving the variabilities as parameters or more generally, as replaceable parts in the terminology of [KL92].

If a domain is stable enough, it may be worthwhile to produce a software system generator that can automatically build parts of any of a series of related products. The structural similarities found in two otherwise completely unrelated generators, Genesis for DBMS construction and Avoca for generating network protocol software, prompted the development of a domain independent model for constructing domain specific software generators[BO92].

6.1.1 A Domain Independent Model

The *GenVoca* model [BDG+94] deals with modules called *components*, parametrized clusters of cooperating classes. Every component performs a mapping of its abstract interface to its concrete implementation, a description that is complementary to the typically accepted view of constructing the interface from the implementation, though both describe the exact same process.

Every component belongs to a *realm*, with all members of a realm implementing the same interface. The interface of a realm is a the set of classes, including their objects, operations and interrelationships, exported by each member. All members of a realm are plug-compatible and therefore interchangeable, though components may also enrich the realm interface. A *component library* is the set of *concrete* components that implement a realm.

Components reference each other via parametrization, with the formal parameters specified as realms and concrete parameters being components. The concrete interface of a component is defined by the union of the realms of its parameters' interfaces.

One important special case are *symmetric components* that export the same realm interface they expect as a parameter, making them arbitrarily stackable, similar to UNIX pipes.

6.1.2 Implementation

The GenVoca model has been implemented in the P1 and P2 ANSI-C extensions [BGT94], and more recently in the P++ system[Sin96], which adds the linguistic elements to support realms and components to standard C++. The integration of the component language into an existing programming language is intended to avoid the language boundary discontinuity found in traditional MILs that complicates the inter-component code shuffling necessary for optimizing performance of highly layered systems.

The P++ language extensions support the declaration of realms, the declaration and implementation of parametrized components and the instantiation of subsystems via type expressions. Given the definitions in figure 6.1 and assuming implementations for the declared components, the following type expressions yield concrete subsystems:

1. `typedef memory_linked_list<int> int_list;`
2. `typedef array<int><5> int_array;`
3. `typedef linked_list<T><array<5> list_of_T_in_array;`

The first two expression yield concrete types, lists and arrays of integers respectively out of parametrized types that provide concrete storage mechanisms, an array and a pointer-based linked list, for arbitrary element types. The third, on the other hand, uses a linked list abstraction that is parametrized not just by the element type, but also by the concrete storage method for the linked-list elements, which in the `memory_linked_list` is hard-coded to in-memory storage. The type equation specifies this storage method, a 5 element array, but leaves the element type open, resulting in a further parametrized type. Type expressions are also checked for validity using predicates on attributes exported by individual components [BG97], the details of the checking are beyond the scope of this paper.

The P2 implementation of a data structure generator has been tested against the libg++ and Booch component libraries, both of which use objects and parametrized types to implemenent a wide variety of data structure, and found to have comparable, usually somewhat faster performance and smaller source-code sizes [BST92]. Re-engineering the LEAPS compiler for OPS5 production systems as a LEAPS $\rightarrow$ P2 $\rightarrow$ C compiler reportedly produced about 10% faster code than the original, hand-crafted implementation of the LEAPS compiler, was developed much quicker while also being much easier to extend. Adding support for persisten stores was accomplished in a small fraction of the time it had taken for the hand-crafted compiler[BDG+94]. The published implementation of the P2 system contains about 10 KLOC for the implementation of data structures.

6.2 Subject Oriented Programming

Subject Oriented Programming [HO93] goes beyond the extension hierarchies in section 4.3 by removing the special 'base' hierarchy. Instead, multiple independent hierarchies, called *subjects* are defined, each modeling the problem area according

```
realm collection<class T>
{
    class container
    {
       container ();                 // constructor
    }
    class cursor
    {
       cursor (container *);          // constructor
       void   first();                // traversal
       void   next();                 // traversal
       int    eoc();                  // end of container
       void   insert(T);              // add item
       void   remove();               // remove item
       T&     get_value();            // get item
    };
};
component memory_linked_list : collection<class T>
{
   class element
   {
      element(T);
      T data;
      element *next,*prev;
   };
   class container
   {
      element *head;
   }
   class cursor
   {
      element *current_pos;
      container *cont;
   };
};
component array <int_size> : collection<class T>
{
   class container
   {
      T items[size];
      int free_slot;
   };
   class cursor
   {
      int index;
      container *cont;
   }
}
component linked_list<collection<element> next_layer> : collection<class T>
{
   class element   { ... };
   class container { ... };
   class cursor    { ... };
}
```

Figure 6.1: P++ parametrized realm and component declarations

to its own special needs. These subjects are later composed into a final system, either at compile, link or run-time.

Among the goals of subject-oriented programming is the ability to develop application independently and later compose them into cooperating suites, without the applications being explicitly dependent on one another, without dictating the coupling and without requiring modifications, thus supporting inclusion of *unanticipated* new applications. Since the individual applications are developed independently, each should enjoy the full benefits of object orientation: encapsulation, polymorphism and inheritance.

6.2.1 Subjects

In contrast to views, which represent subsets of a shared underlying representation, or mixins that allow adding state and behavior to a base hierarchy, each subject *independently* defines the state and behavior needed to model "the world" from a particular perspective. The only intrinsic property of an object is its identity, all state and behavior is deemed subjective.

Within a subject an object has an implementation class that defines its state and behavior for that subject. The classes of an object in different subjects can be completely different, including different inheritance relationships. Separation of interface definition from implementation (see section 3) is even more important for objects in a subject-oriented environment, because it allows matching of operations across subjects without any need for corresponding implementation hierarchies.

Though written in an unmodified object-oriented language, subjects need to be compiled by a special subject-compiler that emits composable binary code and a *label* declaring the subject's composable entities[OKH+95]. Labels also describe the results of compositions and consist of declaration clauses listing entities such as operations, classes, instance variables and methods (called *realizations* in this treatments):[1]

$$
\begin{aligned}
s &= \textbf{subject} \\
s.o &= \textbf{operation with signature } g \\
s.c &= \textbf{class} \\
s.c.v &= \textbf{instance variable of type } t \\
s.o.c &= \textbf{realization set returning } u \\
s.o.c.r &= \textbf{realization}
\end{aligned}
$$

The left hand side of the declarations specifies the fully qualified name in dot notation, with s being the subject name and o an operation and c a class name within that subject, respectively. Instance variable names v are specified relative to the subject and class, and realization (method) names r relative to subject, operation and class. In order to avoid cycles and allow composition of subjects compiled with languages having different notions of inheritance, all inherited information is made explicit in each individual class, leading to a flattening of the hierarchy for labels.

[1] The treatment here is simplified slightly from the one given in [OKH+95] with the hope of preserving the essential information

Apart from the additional information deposited in labels, each subject is defined using the classic object-oriented model, and as long as there is no interaction between subjects, subject-oriented programming is identical to object-oriented programming.

6.2.2 Subject Composition

Although subjects individually see only classical object-oriented programming, the intention of subject-orientation is to let these independent subjects interact in a meaningful way to achieve an overall goal. The goal of subject-composition is to coordinate activity and state in different subjects for a single object in such a way that a consistent overall model emerges.

A subject can interact with others by sharing state, notifying others of relevant changes or using other subjects to perform one of its own operations, but before they can interact successfully, there must be some agreement on how classes and operations correspond between subjects. The most simple strategy for agreement between subjects would be for all to specify identical class names and interfaces for operations, which would still allow classes to be composed out of individually developed pieces but is too restrictive to achieve the goal of composing pre-existing applications.

Composition clauses are the semantic foundation for subject composition. They have the same form as the subject labels introduced in the previous section with *combinators* specifying how different clause elements are to be combined in the resulting clause. (The structural similarity is important because compositions can themselves be composed).

$$
\begin{aligned}
s &= \textbf{subject} \\
s.o &= \textbf{operation with signature } C_g(G) \\
s.c &= \textbf{class} \\
s.c.v &= \textbf{instance variable of type } C_t(T) \\
s.o.c &= \textbf{realization set returning } C_u(U) \\
s.o.c.r &= \textbf{realization}
\end{aligned}
$$

C_g is a *signature combinator* for determining a result signature from the list of input signatures G. C_t and C_u are type and return value combinators that operate on type lists T and return values U, respectively. Basic combinators equally applicable to signatures types and results are:

1. **identity**: the list must be of length 1
2. **equivalent**: all values in the list must be equivalent
3. **first/last**: the first or last element is selected.

Though too primitive to be used directly, these basic combinators allow the definition of composition rules that can model various combination semantics, including no permitted overlap, non-conflicting *merge* ($\diamond$) and *override*[2] ($\triangleright$/$\oplus$), as well as method combination found in CLOS.

[2] see section 4.3.1

6.3 Aspect Oriented Programming

Aspect-oriented programming tries to avoid the tangling problem altogether by letting programmers define non-localized features separately from the main module structure of a program and letting an automated processor produce the tangled procedural code needed to implement the composition of the main, localized functionality and the non-local aspects.

6.3.1 Aspects

Similar to views (section 4.4.2), mixins (section 4.2), and subjects (section 6.2), aspects allow a single entity to be considered from the different perspectives independently, with the ability to integrate those perspectives into a single, coherent representation. Aspects differ in that they do not have to align with the module, class or method boundaries imposed by modular decomposition.

A property of a system implemented using generalized procedural programming can therefore be of one of two types:

- **component**: A property of a system that is easily encapsulated in a module, an object, a procedures or some other functional component of the system.
- **aspect**: A property of a system that cannot be easily encapsulated as a functional component of a system. Aspects tend to be properties of the system or of functional components of the system,

With these terms, the goal of aspect-oriented programming can be restated as allowing the decomposition of a system into both components and aspects, and the later re-composition into a coordinated overall system.

6.3.2 Join Points and Weaving

After cleanly separating the concerns into separate aspect programs and languages, obtaining an executable system entails recombining these aspects with the base program in order to produce the tangled program that would otherwise have been constructed by hand.

An aspect weaver does this by first analyzing the component programs and turning them into a *a join-point representation*, transforming this representation according to the rules of the appropriate aspect language and finally emitting source code for a standard procedural or object-oriented language.

The notion of join-points is similar to hot-spots in object-oriented frameworks, they are the points of variability. Frameworks need to explicitly define hot-spots using object-connections or abstract methods; reflective approaches such as meta-object protocols or composition filters (see section 4.1.2) allow any method invocation to be transformed into a hot spot post-facto, but at the cost of run-time penalties.

The join-point representation of aspect-oriented programming can be considered an extension of the hot spot idea to representations not directly visible in the program structure but available through analysis, such as the data-flow graph of a program, though join points can of course also be method invocations, data structure usages or any other syntactic entity of the source program(s).

6.3.3 Aspect Languages

Aspect-oriented programming is still in its infancy, with no more than a position paper and a couple of case studies publically available, so little of general applicability can be said about aspect languages. However, the existing case studies do provide a glimpse at example aspect languages.

COOL is a fairly simple aspect language that allows synchronization constraints on method access in Jave language to be stated declaratively instead of being interspereed in the code [LK97]. The following code example, an aspect program for coordinating instance of the Java class BookLocator

```
coordinator BLCoord : BookLocator {
   selfexclusive { register, unregister }
   mutexclusive { register, unregister, locate }
};
```

replaces 35 lines of Jave code sprinkled throughout the implementation of the BookLocator class. The join-points in this case are simply the messages sends and receives, so the aspect-weaver is fairly simple.

Aspect-oriented programming was also used in creating high-performance linear algebra routines from high-level algorithmic description by separating aspects for data representation, numerical stability and loop-fusion control [ILG+97]. The base algorithm consists of 14 lines of code written in MatLab a high-level mathematics language. This compares favorably with 300 lines of FORTRAN code in a standard numerics library, but the performance of the MatLab happens to be 100 times worse than the optimized FORTRAN code. Adding another 14 lines of aspect code, split between languages controlling the data representation, partial pivoting and row-permutation control is enough to boost the MatLab code to the same performance level as the tangled FORTRAN code, while keeping all the aspects cleanly separated.

A procedural image library using operators on entire images allows the easy specification of new image-operators, because the combination of operators corresponds dirctly to combining the results of two procedure calls in a third. Unfortunetely, this clean design has unacceptable performance because each operator has to loop over and buffer an entire image. Loop fusion and buffer sharing techniques can only be applied because the total structure is too complicated for compiler to handle, resulting in a system of originally 768 lines of code to be expanded to 32513 lines after tangling all the aspects for optimum performance. The tangled code is also very difficult to maintain or upgrade because the code has be mentally and manually untangled, modified and then retangled.

6.4 Commentary

Subject-oriented programming probably represents both the most conservative and radical apprach presented here. It is radical because it tries to deal with the conceptual difficulty of pressing real-world classification into a hierarchies, and static ones at that, by abandoning the objectivist view of modelling activity. Instead, the

presence of multiple interpretations of a situation is not just allowed, but a fundamental part of the approach. It is conservative in that the split into subjects allows present object-oriented techniques to be applied to the basic programming tasks unchanged, though it is still unclear wether composition of subjects is practically viable.

The GenVoca model requires a fundamentally different style of programming that is probably only applicable to well understood domains where at least the dimensions of the parameter spaces are known. Within those parameters the concepts of *realms* as module interfaces and modules as mappings from realm to realm, combined with the technique of symmetric components that export the same realm they import make for a refinement mechanism that can deal with module (framework) sized components and be applied repeatedly.

Like the GenVoca generators, aspect-oriented programming focuses on source-code transformations to allow highly modular descriptions to be used in performance sensitive application domains, but it provides a much more general model for both decomposition and recombination. On the other hand, it is completely implementation oriented, there are no typing mechanisms, so there is no static method for checking if and how a certain aspect can be applied.

Chapter 7

Perspective

Composition and refinement of software components are the key to achieving pervasive reuse, the long sought-after software-IC. Instead of focusing on the requirements and looking for language elements that optimally support these mechanisms, much effort is expended on trying to make composition and refinement work with existing techniques, despite the fact that a simple analysis can demonstrate that this is likely futile.

In this particular instance, it has been shown that interfaces in current object-oriented languages don't even support the current object-oriented model, and that the model itself suffers from an asymmetry that seems directly related to the failures of scalability with the basic composition and refinement techniques.

Language support both for enriched first-class interfaces and generalized mixins allowing in-place extension looks like an obvious step in the right direction, with neither step requiring too great a departure from current object-oriented thinking, notations and tools, and both promising immediate benefits.

Support for declarative configuration specifications taken from software architecture also look like a necessary addition, one that has been foreshadowed by special-purpose object-composition tools like NeXT's InterfaceBuilder. The non-local aspects of an architectural style, on the other hand, such as the packaging needed by each component cooperating in a style, look like perfect candidates for implementation via aspect-oriented programming.

The GenVoca generators also look like they could benefit from integration into a general aspectual framework, which in turn could benefit from the fairly mature ideas about large-scale component and aspect interfaces inherent in the GenVoca approach. Both could benefit from a generic, language independent representation of program code in order to make aspect languages and weavers independent of the vagarities of a particular base language.

Bibliography

[AB92] Mehmet Aksit and Lodewijk Bergmans. Obstacles in object-oriented software development. Technical report, 1992.

[All97] Robert Allen. *A Formal Approach To Software Architecture*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA 15213, May 1997.

[ASBD92] Toru Asada, Roy Swonger, Nadine Bounds, and Paul Duerig. The quantified design space: A tool for the quantitative analysis of design. Technical Report CMU-CS-92-213, Carnegie Mellon University, November 1992.

[Bat96] Don Batory. Subjectivity and software system generators. Technical Report 32, The University of Texas, Austin, Texas 7812, 1996.

[BC90] Gilad Bracha and William Cook. Mixin-based inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, number 7, pages 303–311. ACM, October 1990.

[BDG+94] Don Batory, Sankar Dasari, Bart Geraci, Vivek Singhal, Marty Sirkin, and Jeff Thomas. Archieving reuse with software system generators. *IEEE Software*, 1994.

[Ber90] Lucy Berlin. When objects collide: Experience with reusing multiple class hierarchies. In *ECOOP '90 Proceedings*, number 7, 1990.

[Ber94] Lodewijk Bergmans. *Composing Concurrent Objects*. PhD thesis, University of Twente, June 1994.

[BFVY96] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[BG93] Gilad Brach and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *OOPSLA 93 Proceedings*, number 7, 1993.

[BG97] Don Batory and Bart Geraci. Archieving reuse with software system generators. *IEEE Transactions on Software Engineering*, 1997.

[BGT94] Don Batory, Bart J. Geraci, and Jeff Thomas. *Introductory P2 System Manual*, 1994.

[Big94] Ted Biggerstaff. The library scaling problem and the limits of concrete component reuse. In *Proceedings of the Third International Conference on Software Reuse*, November 1994.

[BJ94] Kent Beck and Ralph Johnson. Patterns generate architecture. In *Proceedings of ECOOP '94*, 1994.

[BO92] Don Batory and Sean O'Mally. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, October 1992.

[Bok96a] Boris Bokowski. Interaction protocols: Typing of object interactions in frameworks. Technical Report B 96-10, FU Berlin, Institut für Informatik, November 1996.

[Bok96b] Boris Bokowski. Ipdl - interaction protocols for distributed objects. In *Notes of the KI'96 Workshop on Agent-Oriented Programming and Distributed Systems*. Deutsches Forschungszentrum für Künstliche Intelligenz, 1996.

[BR97] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for c++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[Bra92] Gilad Brach. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, The University of Utah, March 1992.

[Bro95] Frederick P. Brooks. *The Mythical Man Month: Essays on Software Engineering, 20th anniversary edition*. Addison Wesley, 1995.

[BST92] Don Batory, Vivek Singhal, and Jeff Thomas. Scalable software libraries. In *ACM SIGSOFT '93*, December 1992.

[CCD+92] Mike Conner, Nurcan Coskun, Scott Danforth, Larry Loucks, Andy Martin, Larry Raper, and Roger Sessions. Developing language neutral class libraries with the system object model (som). In *OOPSLA '92 Addendum*, 1992.

[CH74] R. H. Campbell and A.N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Operating Systems. Proceedings of an International Symposium*, pages 89–102. Springer-Verlag, 1974.

[Cop96a] James A. Coplien. Broadening beyond objects to patterns and to other paradigms. *IEEE Software*, November 1996.

[Cop96b] James A. Coplien. Broadening beyond objects to patterns and to other paradigms. *ACM Computing Surveys*, December 1996.

[CUCH91] Crag Chambers, David Ungar, Bay-Wei Chang, and Urs Hoelzle. Parents are shared parts of objects: Inheritance and encapsulation in self. *LISP AND SYMBOLIC COMPUTATION*, 4(3), 1991.

[DG97] David Wilkie David Garlan, Robert Monroe. Acme: An architecture description interchange language. Technical report, Carnegie Mellon University, January 1997.

[DMNS96] Serge Demeyer, Theo Dir Meijler, Oscar Nierstraß, and Patrick Steyaert. Guidelines for tailorable frameworks. Submitted to the Communications of the ACM October '97 Issue on Object Oriented Frameworks, 1996.

[Edw95] Stephen H. Edwards. Streams: A pattern for pull driven processing. In James A. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 6, pages 417–426. Addison-Wesley, 1995.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification 1, Equations and Initial Semantics*. Springer-Verlag, 1985.

[FO95] Brian Foote and William F. Opdyke. Lifecycle and refactoring patterns that support evolution and reuse. In James A. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 4, pages 239–258. Addison-Wesley, 1995.

[GAO95] David Garlan, Robert Allen, and John Ockerbloom. Architectural mismatch: Why reuse is so hard. *IEEE Software*, pages 17–26, November 1995.

[Gar96a] David Garlan. An introduction to the Aesop system, 1996.

[Gar96b] David Garlan. Style-based refinement for software architecture, 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[HJE95] Herrmann Hüni, Ralph Johnson, and Robert Engel. A framework for network protocol software. In *Proceedings OOPSLA '95*, pages 358–369. ACM Press, 1995.

[HO93] William Harrison and Harold Osher. Subject-oriented programming (a critique of pure objects). In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, number 7, pages 411–428. ACM, September 1993.

[Höl93] Urs Hölzle. Integrating independently-developed components in object-oriented languages. In *ECOOP 93 Proceedings*, number 7, 1993.

[HPR89] Susan Horwitz, Jan Prins, and Thomas Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 11(3):3450387, July 1989.

[ILG+97] John Irwin, Jean-Marc Loingtier, John Gilbert, Gregor Kiczales, John Lamping, Anurag Mendhekar, and Tatiana Shpeisman. Aspect-oriented programming of sparse matrix code. Technical Report SPL97-007 P9710045, Xerox PARC, February 1997.

[ISO95a] ISO/IEC. Itu-t x.901 | iso/iec 10746-1 odp reference model part 1. overview, May 1995.

[ISO95b] ISO/IEC. Itu-t x.901 | iso/iec 10746-3 odp reference model part 2. architecture, 1995.

[JF88] R.E Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, June 1988.

[Kic93] Gregor Kiczales. Traces (a cut at the "make isn't generic" problem). In *Proceedings of ISOTAS '93*, 1993.

[KL92] Gregor Kiczales and John Lamping. Issues in the design and specification of class libraries. In *OOPSLA '92 Proceedings*, number 7, 1992.

[Lam93] John Lamping. Typing the specialization interface. In *OOPSLA '93 Proceedings*, number 7, 1993.

[Lan90] Thomas G. Lane. Studying software architecture through design spaces and rules. Technical Report CMU-CS-90-175, Carnegie Mellon University, November 1990.

[Lie86] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, number 7, pages 214–223. ACM, September 1986.

[LK97] Cristina Videira Lopes and Gregor Kiczales. D: A language framework for distributed programming. Technical Report SPL97-010 P9710047, Xerox PARC, February 1997.

[LSM97] Carine Lucas, Patrick Steyaert, and Kim Mens. Research topics in composability. Technical report, Vrije Universiteit Brussels, October 1997.

[Luc97] Carine Lucas. *Documenting Reuse and Evolution with Reuse Contracts*. PhD thesis, Vrije Universiteit Brussel, 1997.

[LVM95] David C. Luckham, James Vera, and Sigurd Meldal. Three concepts of system architecture. Technical report, Stanford University, 1995.

[Mat96] Michael Mattson. *Object-Oriented Frameworks: A survey of methodological issues*. PhD thesis, University College of Karlskrona/Ronneby, S-372 25 Ronneby, Sweden, February 1996.

[Men97] Tim Menzies. Object-oriented patterns: Lessons from expert systems. *Software Practice and Experience*, To appear, 1997.

[Mey86] Bertrand Meyer. Genericity versus inheritance. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, number 7, pages 391–405. ACM, September 1986.

[MN93] Gail C. Murphy and David Notkin. The ineteraction between static typing and frameworks. Technical Report 93-09-02, University of Washington, Seattle WA, USA 98195, October 1993.

[OH92] Harold Ossher and William Harrison. Combination of inheritance hierarchies. In *OOPSLA '92 Proceedings*, number 7, 1992.

[OKH+95] Harold Osher, Matthew Kaplan, William Harrison, Alexander Katz, and Vincent Kruskal. Subject-oriented composition rules. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications*, number 7, pages 230–250. ACM, September 1995.

[OMG95] OMG. The common object request broker: Architecture and specification. http://www.omg.org/, 1995.

[Par76] David Parnas. On the design an development of program families. *IEEE Transactions on Software Engineering*, 10(2), May 1976.

[Rap97] Rapide Design Team. Guide to the rapide 1.0 language reference manuals. Technical report, Stanford University, July 1997.

[Rüp96] Andreas Rüping. Framework patterns. Technical report, Forschungszentrum Informatik (FZI) Bereich Programmstrukturen, Karlsruhe, 1996.

[SC96] Mary Shaw and Paul Clements. A field guide to boxology: Preliminary classification of architectural styles for software systems. Technical report, CMU, 1996.

[SCD+93] Patrick Steyaert, Wim Codenie, Theo D'Hondt, Koen De Hondt, Carine Lucas, and Marc Van Limberghen. Nested mixin-methods in agora. In *ECOOP 93*, Lecture Notes in Computer Science. Springer Verlag, 1993.

[SDZ95] Mary Shaw, Robert DeLine, and Gregory Zelesnik. Abstractions and implementations for architectural connections. Technical report, Carnegie Mellon University, 1995.

[SG94] Harald Schumann and Michael Goedicke. Component-oriented software development with pi. Course Material, 1994.

[SG96] Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[Sha95] Mary Shaw. Patterns for software architecture. In James A. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 6, pages 453–462. Addison-Wesley, 1995.

[Sin96] Vivek P. Singhal. *A Programming Language for Writing Domain-Specific Software System Generators*. PhD thesis, The University of Text at Austin, August 1996.

[SLMD96] Patrick Steyaert, Carine Lucas, Kim Mens, and Theo D'Hondt. Reuse contracts: Managing the evolution of reusable assets. In *OOPSLA 1996 Conference Proceedings*, ACM Sigplan Notices, pages 268–285. ACM Press, 1996.

[Vil95] Panu Viljamaa. Client-specified self. In James A. Coplien and Douglas C. Schmidt, editors, *Pattern Languages of Program Design*, chapter 7, pages 495–504. Addison-Wesley, 1995.

[YS97] Daniel M. Yellin and Robert E. Strom. Protocol specfifications and component adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, March 1997.

[Zel92] Gregory Zelesnik. The UniCon language reference manual. Technical report, Carnegie Mellon University, May 1992.