Implementing Large-Scale Object-Oriented Components

by

Ioannis Smaragdakis, B.Sc., M.S.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

December 1999

Implementing Large-Scale Object-Oriented Components

Approved by

Dissertation Committee:

_____

_____

_____

_____

_____

Acknowledgments

First, and most importantly, I would like to thank Agapi. She was there during all the hard times, always providing consolation and encouragement. Now that I have to do the same for her, I wish I can be as loving and supportive as she was.

I am also deeply indebted to my advisor, Don Batory. Students usually have something nice to say about their advisors but often only after graduation. This is certainly not the case with me and Don. For years I have been extremely grateful for his constant encouragement and support in so many ways. I cannot say enough about the pleasure of working with Don and the freedom I enjoyed to pursue my research interests both within and outside this dissertation work. Don taught me a lot, and I can now face the future much more confidently because of him. I wish I can be as good an advisor and friend to my future students as he was to me.

Paul Wilson was always extremely supportive. He made me feel much better about my work, offered valuable advice, and taught me a few things about myself. Our research work together was an immensely pleasurable experience and a welcome occasional escape from my dissertation work. Thank you, Paul!

I am also thankful to my other committee members, Professors J.C. Browne, Gordon Shaw Novak Jr., and Dr. Ted Biggerstaff, for their meticulous reading of my dissertation and probing questions.

Many of my fellow graduate students helped me improve both my research and its presentation with their suggestions. All of my friends made the past six years some of the most enjoyable of my life. Thank you! You all know who you are, and I am running out of space.

Finally, I would like to thank my family for their support and encouragement over all these years.

Implementing Large-Scale Object-Oriented Components

Publication No. _____

Ioannis Smaragdakis, Ph.D.
The University of Texas at Austin, 1999

Supervisor: Don S. Batory

The complexity of software has driven both researchers and practitioners toward design methodologies that decompose problems into intellectually manageable pieces and that assemble partial products into complete software artifacts. Modularity in design, however, rarely translates into modularity at the implementation level. Hence, an important problem is to provide implementation (i.e., programming language) support for expressing modular designs concisely.

This dissertation shows that software can be conveniently modularized using large-scale object-oriented software components. Such large-scale components encapsulate multiple classes but can themselves be viewed as classes, as they support the object-oriented mechanisms of encapsulation and inheritance. This conceptual approach has several concrete applications. First, existing language mechanisms, like a pattern of inheritance, class nesting, and parameterization, can be used to simulate large-scale components called *mixin layers*. Mixin layers are

ideal for implementing certain forms of object-oriented designs and result in simpler and more concise implementations than those possible with previous methodologies. Second, we propose new language constructs to provide better support for component-based programming. These constructs express components cleanly (i.e., without unwanted interactions with other language mechanisms) and address the issue of component type-checking.

Contents

Chapter 1

Introduction

1.1 Overview and Contribution

Large software artifacts are arguably among the most complex products of human intellect. The complexity of software has driven both researchers and practitioners toward design methodologies that decompose problems into intellectually manageable pieces and that assemble partial products into complete software artifacts. The principle of separating logically distinct (and largely independent) facets of an application is behind many good software design practices. A key objective in designing reusable software modules is to encapsulate within each module a single, orthogonal aspect of application design. Many design methods in the object-oriented world build on this principle of design modularity (e.g., design patterns [GHJV94] and collaboration-based designs [RAB+92]). The central issue is to provide implementation (i.e., programming language) support for expressing modular designs concisely.

This dissertation shows a way to modularize software statically (i.e., at compile-time) using large-scale object-oriented components. Our large-scale components encapsulate multiple classes and can therefore be viewed as being analogous to conventional *modules*, such as those found in languages like Ada [ISO95] and ML [MTH90]. At the same time, however, these modules can be viewed as

classes, as they support the object-oriented mechanisms of encapsulation and inheritance. Additionally, our components offer a great degree of flexibility because they are generic with respect to the components from which they inherit. This conceptual approach has several concrete applications. Existing language mechanisms, like class nesting, can be used to simulate large-scale components, resulting in more concise implementations than were possible with previous methodologies. Additionally, we propose new language constructs to provide better support (for instance, typing) for component-based programming.

More specifically, the main contributions of this dissertation are as follows:

- We introduce a better way of implementing object-oriented *collaboration-based* (or *role-based*) designs. Such designs decompose an object-oriented application into a set of classes and a set of collaborations. Each application class encapsulates several *roles*, where each role embodies a separate aspect of the class's behavior. A *collaboration* is a cooperating suite of roles. Previous methods for implementing collaboration-based designs include *application frameworks* [JF88] and the technique of VanHilst and Notkin [VN96a-c, Van97]. Our approach involves large-scale components called *mixin layers*. We show that mixin layers preserve the advantages of the VanHilst and Notkin implementation method over application frameworks (i.e., maintain design structure, facilitate reuse, and avoid unnecessary dynamic binding). At the same time, mixin layers correct the scalability problems of the VanHilst and Notkin technique yielding simpler code and shorter compositions. As a further practical validation, we used mixin layers as the primary implementation technique in a medium-size project (the JTS tool suite for implementing domain-specific languages). Our experience shows that the mechanism is versatile and can handle components of substantial size.

It is worth noting that, although mixin layers have a strictly defined form, they can be expressed using a variety of programming language constructs. These include C++ parameterized nested classes, CLOS mixins and reflection, and Java nested mixins (Java mixins have been proposed as an extension to the language—e.g., in [AFM97, FKF98]).

- We show that two significant software construction methodologies, the GenVoca model and object-oriented collaboration-based designs, are closely related. In particular, the GenVoca model has been used to design and develop software for a variety of domains (e.g., [Bat88, BBG+88, OP92, CS93]. In the past, its principles have not been expressed in object-oriented terms. This work shows how GenVoca components can be implemented as mixin layers. Additionally, we discuss how some common GenVoca concepts and mechanisms (for instance, GenVoca *realms* and the validation of a composition of components) can be integrated in the mixin layers framework.
- We address the problem of providing type-system support for large-scale components. We show that a type system needs to support two new properties (termed *deep subtyping* and *deep interface conformance*) in order to express constraints for mixin layer parameters. These ideas have also led to other interesting applications in type systems. Wadler, Odersky and this author [WOS98] have used deep subtyping to demonstrate that parametric types can elegantly emulate *virtual types* (a well-known typing mechanism in the object-oriented world, introduced by the Beta programming language). The emulation employs multiple-class components that are essentially a simpler version of mixin layers.

The intellectual challenge in our work was to identify the principles, concepts, and, eventually, constructs that will allow developing orthogonal aspects of a

software application in isolation and later composing them consistently. The constructs presented in this dissertation rise to the challenge. They represent a clear advancement over previous techniques and succeed in expressing independent application aspects as separate components by employing a scalable approach based firmly on object-oriented principles and techniques.

1.2 Outline

Subsequent chapters explore the problem of modularizing software, explain our proposed solution, and contrast it to other work in the research literature.

In Chapter 2 we describe the collaboration-based design methodology and discuss how such designs can be best implemented. We use an example domain consisting of a few graph algorithms, all based on depth-first traversals of an undirected graph. Ideally, each of the algorithms, as well as the underlying model (i.e., undirected graph) and traversal strategy (depth-first), should be expressible as individual components that would yield the complete application once composed. The C++ technique proposed by VanHilst and Notkin [VN96a-c] attempts to do exactly that but suffers from high complexity of parameterizations. We introduce an alternative in the form of mixin layers. Mixin layers are large-scale components that can be used to directly implement collaboration-based designs. Mixin layers improve upon the VanHilst and Notkin technique by offering more concise and scalable implementations.

In Chapter 3 we discuss several issues related to the correctness of a composition of components. First we show how we can use propositional properties that are propagated in a mixin layer composition using inheritance. This solution is straightforward but not entirely satisfactory as it cannot produce informative error messages due to lack of language support. Then we explore the possibility of add-

ing type-system support for mixin layers. We discuss what the type of a layer is and propose an extension to the Java language that supports types for classes containing nested classes. Finally, we explore another interesting type system issue. Using a fixpoint construction (similar to a common technique used to express formal object-oriented semantics), we can propagate type information from a mixin layer to the layer above it in a composition (its superclass from an inheritance standpoint). Using this technique in conjunction with nested classes of a form similar to mixin layers, we show how parametric types can emulate virtual types—a well-known object-oriented type mechanism.

Chapter 4 demonstrates the scalability of the mixin layers approach. We implemented mixin layers as an extension to the Java language and used them as the main implementation technique in constructing JTS: a set of pre-compiler/ compiler tools for adding syntax extensions to programming languages. As a result, the structure of the system is very simple and a large number of feature combinations can be implemented as different compositions of large layers.

In Chapter 5 we position our approach relative to other research work. We discuss the GenVoca model of software construction, which forms a general software engineering framework that encompasses our research. Additionally, we discuss realizations of components in a dynamic setting (i.e., components that can be put together at application run-time). Finally, we describe other research work in the area of automated software construction and software modularization. The discussion is mostly from a programming language standpoint but we also offer insights into related software engineering applications.

In Chapter 6 we review the central results of our work, summarize the primary contributions of our research, and discuss a few areas of future research.

Chapter 2

Collaboration-Based Designs and Mixins

A relatively recent development in the software arena is the advance of *object-oriented (OO)* design and implementation techniques. A large variety of programming languages, design methodologies, and programming tools are based on object-oriented principles. The goals of object-orientation are remarkably similar to the goals of this dissertation. In particular, object technologies attempt to organize software in a way that makes it easier to understand, reuse, and evolve. Modularity is addressed by splitting a program into encapsulated entities (objects or classes), which can be reused in complex configurations. Unfortunately, objects (i.e., collections of data and operations on those data) are rarely ideal for playing the role of modules. The reason is that objects are not self-sufficient but often have complex interactions with other objects. Thus, a unit of modularity (i.e., a program piece that can be defined in isolation) is not a single object but a collection of interrelated objects. Standard object-oriented language mechanisms, like inheritance and polymorphism, are not sufficient to express such modules in a flexible way.

Nevertheless, object-orientation forms a natural starting point for our research, given the similarities with its philosophy and the engineering maturity of OO programming methods (e.g., programming languages like C++, Java, Smalltalk, and CLOS). Additionally, object-oriented *design* techniques (e.g., design patterns [GHJV94], and collaboration-based designs [RAB+92]) can be used to

express modular large-scale components. Such design components contain multiple objects and capture their interactions. The challenge is to translate object-oriented designs (i.e., artifacts that exist usually only on paper) into implemented programs *that preserve the elegant structure of the design*. This chapter shows how it can be done using novel combinations of mechanisms that can already be found in object-oriented programming languages. The greatest value of expressing components in this manner is that they become smoothly integrated both at the language level and at the conceptual level, with the fundamental mechanism of inheritance playing the main role.

The chapter is organized as follows: Section 2.1 offers a short introduction to object-orientation and defines some basic terms. Section 2.2 discusses object-oriented collaboration-based designs. Collaboration-based designs are modular decompositions of an application. Mapping them into an implementation that preserves the structure of the design is sufficient for obtaining the kind of modularity we seek. To do so, we need to introduce some object-oriented constructs, called *mixin classes* and *mixin layers*. These are discussed in detail in Section 2.3. Mixin layers are one of the main contributions of this dissertation, and we claim that they are ideal for implementing collaboration-based designs. We show how this implementation can be effected and how mixin layers compare to other techniques in Section 2.4.

2.1 Brief Introduction to Object-Orientation

This dissertation assumes that the reader is familiar with object-oriented principles and constructs. Hence, the purpose of this section is not to provide a comprehensive introduction to object-oriented programming, but to briefly lay out some fun-

damental concepts and terms. For a more thorough treatment of object-orientation, the reader should consult the references given throughout this chapter.

Even though no exact definition of object-orientation exists, the main idea can easily be described in informal terms. If all programming is viewed as the definition of operations on data, traditional programming languages (e.g., Pascal, C, Lisp, etc.) promote a *functional* program organization: operations (also called *functions*, or *procedures*) are the central entities and have clearly defined bounds (input parameters and return values). If an operation applies to more than one kind of data, the functionality for all cases is collected under a single heading—the specification of the operation. In contrast, object-oriented programming concentrates on collections of data that can be considered as representing a single entity. Such collections are called *objects* and form the fundamental building blocks of object-oriented programs. In object-oriented programming, the functionality of a single operation is distributed in the objects to which the operation is applicable. (The distributed elements of an operation pertaining to a single object are called the object's *methods*.) In this way, an object is an isolated entity with a clear interface to the outside world. The fundamental goal of *data hiding* is achieved by making the object's implementation invisible and having other parts of the program rely only on the object's interface. This ability to hide the internals of an object is commonly known by the name *encapsulation*.

With objects playing the central role in object-oriented design and implementation, several high-level relationships on objects can be defined. The most important ones, which are often considered essential elements of object-oriented languages, are *inheritance* and *polymorphism*. Inheritance provides a way of obtaining new objects by incrementally refining existing ones. Polymorphism, on the other hand, is the mechanism used to select the appropriate operation for an object in the presence of inheritance-induced refinements. Another important con-

cept in object-oriented programming is that of a *class*. Most widely used object-oriented programming languages (e.g., Java [GJS96], C++ [Str97], CLOS [KRB91]) are *class-based*. That is, new objects are created by instantiating classes—patterns that describe the object's data and operations. Thus, in class-based object systems, methods are associated with classes and not with objects directly. Consequently, all objects belonging in the same class support an identical set of methods. Since classes are the only way to create new objects, inheritance is only applicable to classes and not objects in class-based languages.

2.2 Collaboration-Based Designs[1]

Collaboration-based or *role-based* designs are the topic of several pieces of work in the object-oriented research community [BC89, HHG90, Hol92, RAB+92, VN96a]. These concepts probably originated with Reenskaug, et al. [RAB+92] but have been used in various forms, often without being named and documented. We will not offer an extensive introduction to object-oriented design techniques, as our work examines software development from a programming languages standpoint. A good introduction to collaboration-based design can be found in the presentation of the OORAM approach [RAB+92]. A detailed treatment of collaboration-based designs, together with a discussion of how to derive them from use-case scenarios [Rum94] can be found in VanHilst's Ph.D. dissertation [Van97].

2.2.1 Collaborations and Roles

In an object-oriented design, objects are encapsulated entities but are rarely self-sufficient. Although an object is fully responsible for maintaining the data it encap-

1. Parts of this section and Section 2.4.3 are taken from reference [SB98a] (© 1998 IEEE).

sulates, it needs to cooperate with other objects to complete a task. An interesting way to encode object interdependencies is through collaborations. A *collaboration* is a set of objects and a protocol (i.e., a set of allowed behaviors) that determines how these objects interact. The part of an object enforcing the protocol that a collaboration prescribes is called the object's *role* in the collaboration. Objects of an application generally participate in multiple collaborations simultaneously and, thus, may encode several distinct roles. Each collaboration, in turn, is a collection of roles, and represents relationships across the corresponding objects. Essentially, a role isolates the part of an object that is relevant to a collaboration from the rest of the object. Different objects can participate in a collaboration, as long as they support the required roles.

In collaboration-based design, we try to express an application as a composition of largely independently-definable collaborations. *Viewed in terms of design modularity, collaboration-based design acknowledges that a unit of functionality (module) is neither a whole object nor a part of it, but can cross-cut several different objects*. If a collaboration is reasonably independent of other collaborations (i.e., a good approximation of an ideal module) the benefits are great. First, the collaboration can be reused in a variety of circumstances where the same functionality is needed, by just mapping its roles to the right objects. Second, any changes in the encapsulated functionality will only affect the collaboration and will not propagate throughout the whole application.

In abstract terms, a collaboration is a view of an object-oriented design from the perspective of a single concern. For instance, a collaboration can be used to express a producer-consumer relationship between two communicating objects. Clearly, this collaboration prescribes roles for (at least) two objects and there is a well-defined "protocol" for their interactions. Interestingly enough, the same collaboration could be instantiated more than once in a single object-oriented design,

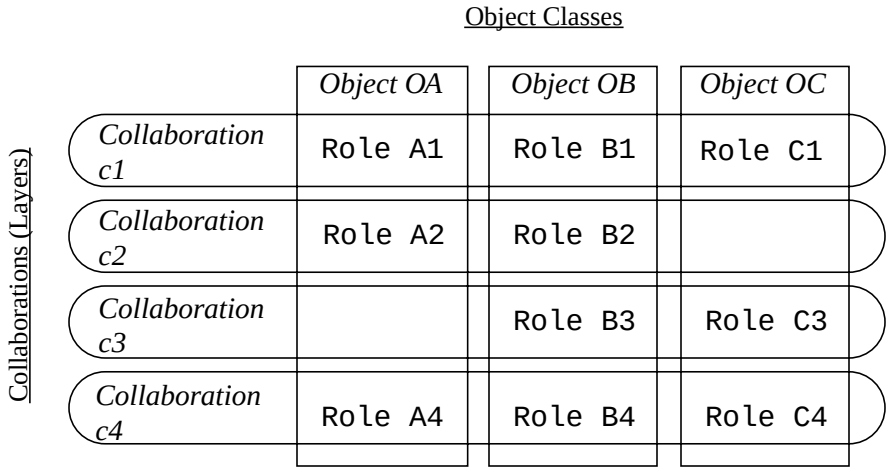

Figure 2.1 : Example collaboration decomposition. Ovals represent collaborations, rectangles represent objects, their intersections represent roles.

with the same objects playing different roles in every instantiation. In the example of the producer-consumer collaboration, a single object could be both a producer (from the perspective of one collaboration) and a consumer (from the perspective of another).

Figure 2.1 depicts the overlay of objects and collaborations in an (abstract) example design. The figure contains three different objects (*OA*, *OB*, *OC*), each supporting multiple roles. Object *OB*, for example, encapsulates four distinct roles: `B1`, `B2`, `B3`, and `B4`. Four different collaborations (*c1*, *c2*, *c3*, *c4*) capture distinct aspects of the application's functionality. To do this, collaborations have to prescribe certain roles for objects. For example, collaboration *c2* contains two distinct roles, `A2` and `B2`, which are assumed by distinct objects (namely *OA* and *OB*). An object does not need to play a role in every collaboration—for instance, *c2* does not affect object *OC*.

It should be noted that the designs we will examine are *static*: the roles played by an object are uniquely determined by its class. For instance, in Figure

2.1, all three objects must belong in different classes (since they all support different sets of roles). In essence, we use the design as a guide to the implementation of an application—not to describe different phases in its dynamic behavior. We will discuss dynamic layered designs in more detail in Chapter 5.

2.2.2 An Example Collaboration-Based Design

As a running example that will help us illustrate important points of our discussion, we will consider the graph traversal application that was examined initially by Holland [Hol92] and subsequently by VanHilst and Notkin [VN96a]. Doing so affords not only a historical perspective on the development of role-based designs, but also a perspective on the contribution of this work. The application defines three different operations (algorithms) on an undirected graph, all based on depth-first traversal: *Vertex Numbering* numbers all nodes in the graph in depth-first order, *Cycle Checking* examines whether the graph is cyclic, and *Connected Regions* classifies graph nodes into connected graph regions. The application itself has three distinct classes: *Graph*, *Vertex*, and *Workspace*. The *Graph* class describes a container of nodes with the usual graph properties. Each node is an instance of the *Vertex* class. Finally, the *Workspace* class includes the application part that is specific to each graph operation. For the *Vertex Numbering* operation, for instance, a *Workspace* object holds the value of the last number assigned to a vertex as well as the methods to update this number.

Recall that in decomposing an application into collaborations, we need to capture distinct aspects as separate collaborations. A decomposition of this kind is relatively straightforward and results in five distinct collaborations.

One collaboration (*Undirected Graph*, above) expresses properties of an undirected graph. This is clearly an independent aspect of the application—the problem could very well be defined for directed graphs, for trees, etc.

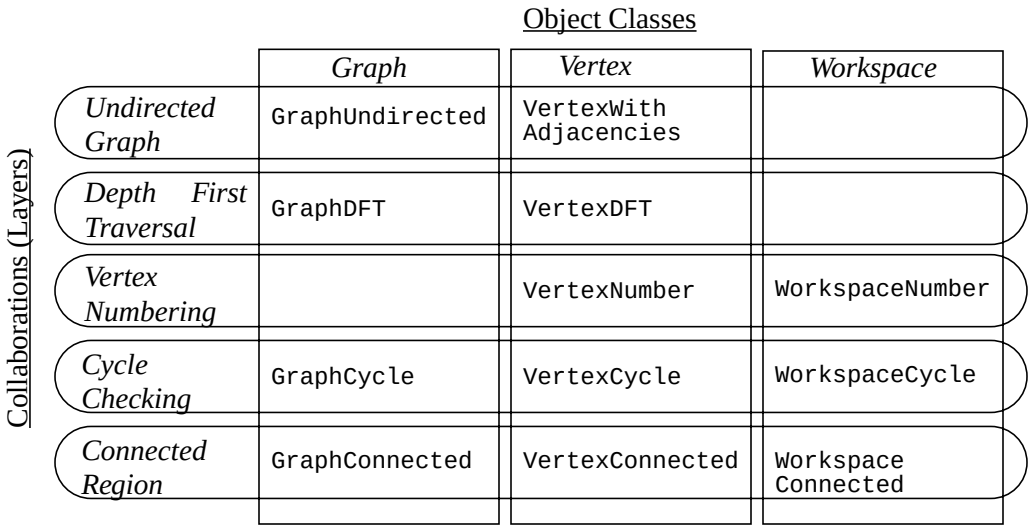

Figure 2.2 : Collaboration decomposition of the example application domain: A depth-first traversal of an undirected graph is specialized to yield three different graph operations. Ovals represent collaborations, rectangles represent classes.

Another collaboration (*Depth First Traversal*, above) encodes the specifics of depth-first traversals and provides a clean interface for extending traversals. That is, at appropriate moments during a traversal (the first time a node is visited, when an edge is followed, and when a subtree rooted at a node is completely processed) control is transferred to specialization methods that can obtain information from the traversal collaboration and supply information to it. For instance, consider the *Vertex Numbering* operation as a simple refinement of a depth-first traversal. This can be effected by specializing the action performed the first time a node is visited during the traversal. The action will assign a number to the node and increase the count of visited nodes.

Using this approach, each of the three graph operations can be seen as a refinement of a depth-first traversal and each can be expressed by a single collaboration. Figure 2.2 is reproduced from [VN96a] and presents the collaborations and

classes of our example application. The intersection of a class and a collaboration in Figure 2.2 represents the role prescribed for that class by the collaboration. A role encodes the part of an object that is specific to a collaboration. For instance, the role of a *Graph* object in the "*Undirected Graph*" collaboration supports storing and retrieving a set of vertices. The role of the same object in the "*Depth First Traversal*" collaboration implements a part of the actual depth-first traversal algorithm. (In particular, it contains a method that initially marks all vertices of a graph *not-visited* and then calls the method for depth-first traversal on each graph vertex object).

Note that the design of Figure 2.2 does *not* define any particular composition of collaborations in an application. It is really just a decomposition of a restricted software domain into its fundamental collaborations. Actual applications may not need all three graph operations. Additionally, a single application may need more than one operation applied to the same graph. The latter is accomplished by having multiple copies of the "*Depth First Traversal*" collaboration in the same design (each traversal will require its own private variables and traversal methods). We will later see examples where composing instances of the collaborations of Figure 2.2 will yield an actual application design.

The goal of a collaboration-based design is to encapsulate within a collaboration all dependencies between classes. In this way, collaborations themselves have no outside dependencies and can be reused in a variety of circumstances. The "*Undirected Graph*" collaboration, for instance, encodes all the properties of an undirected graph (pertaining to the *Graph* and *Vertex* classes, as well as the interactions between objects of the two). Thus, it can be reused in any application that deals with undirected graphs. Ideally, we should also be able to easily replace one collaboration with another that exports the same interface. For instance, it would

be straightforward to replace the "*Undirected Graph*" collaboration with one representing a directed graph.

Of course, simple interface conformance will not guarantee composition correctness—the application writer must ensure that the algorithms used (for example, the depth-first traversal) are still applicable after the change. The algorithms presented by Holland [Hol92] for this example are, in fact, general enough to be applicable to a directed graph. If, however, a more efficient, specialized-for-undirected-graphs algorithm was used (as is, for instance, possible for the *Cycle Checking* operation) the change would yield incorrect results. Chapter 3 discusses in detail the issue of ensuring that components are actually interchangeable.

2.3 Mixin Classes and Mixin Layers

To implement collaboration-based designs directly we build on an existing object-oriented construct called a *mixin*. Mixins are similar to classes but with some added flexibility, as described in the following sections. Unfortunately, mixins alone are not sufficient to express large-scale components—they suffer from only being able to describe a single class at a time and not a collection of cooperating classes. To address this, we introduce *mixin-layers*: a scaled-up form of mixins that can contain multiple smaller mixins.

2.3.1 Introduction to Mixins

The term *mixin class* (or just "mixin") has been overloaded to mean several specific programming techniques and a general mechanism that they all approximate. Mixins were originally explored in the context of the Lisp language with object-systems like Flavors [Moo86] and CLOS [KRB91]. In that context, mixins are classes that allow their superclass to be determined by *linearization* of multiple

inheritance. In C++, the term has been used to describe classes in a particular (multiple) inheritance arrangement: as superclasses of a single class that themselves have a common *virtual base class* (see [Str97], p.402). Both of these mechanisms are approximations of a general concept described by Bracha and Cook [BC90], and here we will use “mixin” in this general sense.

The main idea implemented by mixins is quite simple: in object-oriented languages, a superclass can be defined without specifying its subclasses. This property is not, however, symmetric: when a subclass is defined, it must have a specific superclass. Mixins (also commonly known as *abstract subclasses* [BC90]) represent a mechanism for specifying classes that will eventually inherit from a superclass. This superclass, however, is not specified at the site of the mixin’s definition. Thus a single mixin can be instantiated with different superclasses yielding widely varying classes. This property of mixins makes them appropriate for defining uniform incremental extensions for a multitude of classes. When the mixin is instantiated with one of these classes as a superclass, it produces a class incremented with the additional behavior.

Mixins can be easily implemented using parameterized inheritance. In this case, a mixin is a parameterized class with the parameter becoming its superclass. Using C++ syntax we can write a mixin as:

```
template <class Super> class Mixin : public Super {
  ... /* mixin body */
};
```

Mixins are flexible and can be applied in many circumstances without modification. To give an example, consider a mixin implementing *operation counting* for a graph. Operation counting means keeping track of how many nodes and edges have been visited during the execution of a graph algorithm. (This simple

example is one of the non-algorithmic refinements to algorithm functionality discussed in [WeiWeb]). The mixin could have the form:[2]

```
template <class Graph> class Counting: public Graph {
  int nodes_visited, edges_visited;
public:
  Counting() : Graph() {
    nodes_visited = edges_visited = 0; }

  node succ_node (node v) {
    nodes_visited++;
    return Graph::succ_node(v);
  }

  edge succ_edge (edge e) {
    edges_visited++;
    return Graph::succ_edge(e);
  }

  // example method that displays the cost of an algorithm in
  // terms of nodes visited and edges traversed
  void report_cost () {
    cout << "The algorithm visited " << nodes_visited <<
            " nodes and traversed " << edges_visited <<
            " edges" << endl;
  }
  ... // other methods using this information may exist
};
```

By expressing operation counting as a mixin we ensure that it is applicable to many classes that have the same interface (i.e., many different kinds of graphs). (Clearly, the implicit assumption is that classes, like `Dgraph` and `Ugraph`, have been designed so that they export similar interfaces. By standardizing certain

2. We use C++ syntax for most of the examples of this chapter, in the belief that concrete syntax will clarify, rather than obscure, our ideas. To facilitate readers with limited C++ expertise, we avoid several cryptic idioms or shorthands (for instance, constructor initializer lists are replaced by assignments, we do not use the `struct` keyword to declare classes, etc.). A convention followed in our code fragments is that class declarations and their syntactic delimiters are highlighted. This will enhance readability in later sections, where classes can be nested.

aspects of the design, like the method interfaces for different kinds of graphs, we gain the ability to create mixin classes that can be reused in different occasions.) We can have, for instance, two different compositions:

```
Counting < Ugraph > counted_ugraph;
```

and

```
Counting < Dgraph > counted_dgraph;
```

for undirected and directed graphs. (We omit parameters to the graph classes for simplicity.) Classes produced by the two above compositions have the extra capabilities provided by the `Counting` mixin. Note that the behavior of the composition is exactly what one would expect: any methods not affecting the counting process are exported (inherited from the graph classes). The methods that do need to increase the counts are "wrapped" in the mixin.

2.3.2 Mixin Layers

To implement entire collaborations as implementation components we need to use mixins that encapsulate other mixins. We call the encapsulated mixin classes *inner mixins*, and the mixin that encapsulates them the *outer mixin*. Inner mixins can be inherited, just like any member variables or methods of a class. An outer mixin is called a *mixin layer* when *the parameter (superclass) of the outer mixin encapsulates all parameters (superclasses) of inner mixins*.[3] This is illustrated in Figure 2.3. `ThisMixinLayer` is a mixin that refines (through inheritance) `SuperMixinLayer`. `SuperMixinLayer` encapsulates three classes: `FirstClass`, `SecondClass`, and `ThirdClass`. `ThisMixinLayer` also encapsulates three inner classes that are themselves mixins. Two of them refine the corresponding classes of `SuperMixinLayer`, while the third is an entirely new class.

3. Inner mixins can actually themselves be mixin layers.

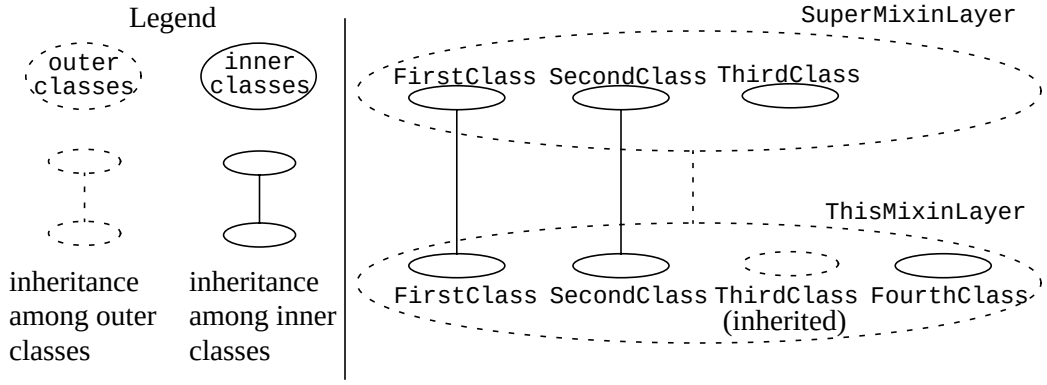

Figure 2.3 : Mixin layers schematically.

Note that inheritance works at two different levels in a mixin layer. First, the layer can inherit inner mixins from the layer above it (for instance, `ThirdClass` in Figure 2.3). Second, the inner mixins inherit member variables, methods, or other classes from their superclass.

2.3.3 Mixin Layers in Various OO Languages

The mixin layer concept is quite general and is not tied to any particular language idiom. Many flavors of the concept, however, can be expressed via specific programming language idioms: as stand-alone language constructs, as a combination of C++ nested classes and parameterized inheritance, as a combination of CLOS class-metaobjects and mixins, etc. We study these different realizations next. The introduction of technical detail is necessary at this point as it will help us demonstrate concretely, in Section 2.4, the advantages of mixin layers for implementing collaboration-based designs.

C++. We would like to support mixin layers in C++ using the same language mechanisms as those used for mixin classes. To do this, we can standardize the names used for inner classes implementations (make them the same for all layers).

This yields an elegant form of mixin layers that can be expressed using common C++ features. For instance, using C++ parameterized inheritance and nested classes, we can express a mixin layer (see again Figure 2.3) as:

```
template <class LayerSuper>
class LayerThis : public LayerSuper {
public:
  class FirstInner : public LayerSuper::FirstInner  { ... };
  class SecondInner: public LayerSuper::SecondInner { ... };
  class ThirdInner : public LayerSuper::ThirdInner  { ... };
  ...
};                                                          (2.1)
```

The code fragment (2.1) represents the form of mixin layers that we will use in the examples of this chapter. Note that specifying a parameter for the outermost mixin automatically determines the parameters of all inner mixins. Composing mixin layers to form concrete classes is now as simple as composing mixin classes. If we have four mixin layers (`Layer1`, `Layer2`, `Layer3`, `Layer4`), we can compose them as:

```
Layer4 < Layer3 < Layer2 < Layer1 > > >
```

where "`<...>`" is the C++ operator for template instantiation. The above composition creates two different class hierarchies: one for the layers themselves and one for the inner classes. Note that `Layer1` has to be a concrete class (i.e., not a mixin class). Alternatively we can have a class with empty inner classes that will be used as the root of all compositions. (A third alternative is to use a *fixpoint* construction and instantiate the topmost layer with the result of the entire composition! This pattern has several desirable properties but to avoid complicating our discussion, we discuss it separately in Chapter 3.)

In code fragment (2.1) we mapped the main elements of the mixin layer definition to specific implementation techniques. We used nested classes to implement class encapsulation. We also used parameterized inheritance to implement

mixins. *The mixin layer definition is completely independent of these implementation choices*. There are very different ways of encoding the same design in other languages.

CLOS (and other reflective languages). The *Common Lisp Object System* (CLOS) [KRB91] is a high-level object-oriented language with very powerful *reflective* capabilities, through the well-known CLOS *meta-object protocol*. Reflective mechanisms allow programs to modify fundamental aspects of the system under which they are operating. Here we are interested in the case of object-oriented programming languages and reflection provided by meta-object protocols. Meta-object protocols allow the policies of an object system to be changed, affecting what happens when an object is created, when methods are dispatched, when a class inherits from other classes, etc. A common capability in OO reflective systems is that of handling classes as *first-class* entities (i.e., classes can be assigned to variables and checked for identity). Even the most fundamental forms of reflection, like simple *introspection* protocols (for instance, Java Reflection [Jav97a]), allow manipulating classes as first-class entities.

We can encode mixin layers in CLOS (and many other reflective systems) by simulating their main elements using reflection (classes as first-class entities). The main elements of mixin layers are class encapsulation (classes containing other classes) and mixins. Since CLOS has native mixin support, we only need to implement class encapsulation by defining a class with member methods that return CLOS class-metaobjects (in essence, other classes). Unlike our C++ example, no lexical nesting of any kind is necessary. This combines nicely with the method-based character of CLOS mixins and the reflective capabilities of the language. The reader should keep in mind, however, that the semantics of every incarnation of mixin layers depends on the semantics of the host language. Thus, CLOS

mixin layers are not semantically equivalent to C++ mixin layers (for instance, there is no default class data hiding: class members are by default accessible from other code in CLOS). Nevertheless, the two versions of mixin layers are just different flavors of the same idea.

The main mixin layer template, analogous to code fragment (2.1), is written in CLOS as:

```
(defclass first-dummy() (...))
  ; Definition of 1st inner mixin
(defclass second-dummy () (...))
  ; Definition of 2nd inner mixin
(defclass third-dummy () (...))
  ; Definition of 3rd inner mixin
...
(defclass layer-this () ())

; Encapsulate classes as methods. Each method returns a
; linked list of class-metaobjects (one per inner mixin)
(defmethod first-inner ((self layer-this))
  (cons (find-class 'first-dummy) (call-next-method)))

(defmethod second-inner ((self layer-this))
  (cons (find-class 'second-dummy) (call-next-method)))

(defmethod third-inner ((self layer-this))
  (cons (find-class 'third-dummy) (call-next-method)))
                                                    (2.2)
```

Note that, just like in the C++ example, the root of the outer inheritance hierarchy must be concrete (i.e., not parameterized). In the above, this means that its methods defining inner classes should not use `call-next-method`. Composition of mixin layers is a simple matter of using CLOS multiple inheritance (same as with regular mixins). For instance, if we have mixin layers `first-layer`, `second-layer`, `third-layer`, their composition is defined as:

```
(defclass composition
  (first-layer second-layer third-layer) (...))

(setq composite-obj (make-instance 'composition))
```

In (2.2), methods defining inner classes (like `first-inner`, `second-inner`, etc.) return a list of all inner mixins. Constructing the inner classes is then a simple matter of creating classes programmatically using this list. This is a standard CLOS technique (e.g., see function `find-programmatic-class` in [KRB91], p.68). For instance, creating the first of the inner classes could be expressed as:

```
(setq first-inner-class (find-programmatic-class
                          (first-inner composite-obj)))
```

The above idiom should be taken as a proof-of-concept, rather than an optimal implementation of mixin layers in CLOS. The powerful syntactic extension (macros) capabilities of Common Lisp can be used to add syntactic sugar to the mechanism.

The ideas used to express mixin layers in CLOS are also applicable to other reflective languages. For instance, although we have not experimented with the Smalltalk language, we expect that mixin layers are expressible in Smalltalk. Smalltalk has been a traditional test bed for mixins, both for researchers (e.g., [BG96, Mez97, SCD+93]) and for practitioners [Mon96]. Like CLOS, the language has powerful reflective capabilities. These can be used to emulate encapsulated classes by methods that return classes. We believe that this technique can be used in conjunction with existing mixin mechanisms to implement mixin layers. It should be noted that a straightforward (but awkward) way to implement mixins in Smalltalk is as *class-functors*; that is, mixins can be functions that take a superclass as a parameter and return a new subclass.

Java. The Java language is an obvious next candidate for mixin layers. Java has no support for mixins and it is unlikely that the core language will include mixins in the near future. As will be described in Chapter 4, we have implemented our own language extensions to Java that capture mixins and mixin layers explicitly. In this effort we used our JTS set of tools [BLS98] for creating pre-compilers for domain-specific languages. The system supports mixins and mixin layers through parameterized inheritance and class nesting, in much the same way as in C++.[4] Additionally, the fundamental building blocks of the JTS system itself were expressed as mixin layers, resulting in an elegant bootstrapped implementation.

Adding mixins to Java is also the topic of other active research [AFM97, FKF98]. The work of [FKF98] presented a semantics for mixins in Java. This is particularly interesting from a theoretical standpoint as it addresses issues of mixin integration in a type-safe framework. As we saw, mixins can be expressed in C++ using parameterized inheritance. There have been several recent proposals for adding parameterization/genericity to Java [AFM97, OW97, BOSW98, MBL97, Tho97], but only the first [AFM97] supports parameterized inheritance and, hence, can express mixin layers.

It is interesting to examine the technical issues involved in supporting mixins in Java genericity mechanisms. Three of these mechanisms [OW97, BOSW98, Tho97] are based on a *homogeneous* model of transformation: the same code is used for different instantiations of generics. This is not applicable in the case of parameterized inheritance—different instantiations of mixins are not subclasses of the same class (see [AFM97] for more details). Additionally, there may be conceptual difficulties in adding parameterized inheritance capabilities: The genericity

4. The Java 1.1 additions to the language [Jav97b] support nested classes and interfaces (actually both “nested” classes as in C++ and *member* classes—where nesting has access control implications). Nested classes can be inherited just like any other members of a class.

approach of [Tho97] is based on virtual types. Parameterized inheritance can be approximated with virtual types by employing *virtual superclasses* [MM89], but this is not part of the design of [Tho97].

The approaches of Myers et al. [MBL97] and Agesen et al. [AFM97] are conceptually similar from a language design standpoint. Even though parameterized implementations do not directly correspond to types in the language (in the terminology of [CW85] they correspond to *type operators*), parameters can be explicitly constrained. This approach, combined with a *heterogeneous* model of transformation (i.e., one where different instantiations of generics yield separate entities) is easily amenable to adding parameterized inheritance capabilities, as was demonstrated in [AFM97].

2.4 Implementing Collaboration-Based Designs

Armed with our knowledge of powerful object-oriented programming language constructs we can now attempt to express collaboration-based designs directly at the implementation level. We will show how our mixin layers technique can be used to perform the task and examine how it compares to two previous approaches. One is the straightforward implementation technique of application frameworks [JF88] using just objects and inheritance. The other is the technique of VanHilst and Notkin that employs C++ mixins to express individual roles.

2.4.1 Using Mixin Layers

Mixin layers are ideally suited for implementing collaboration-based designs. A single mixin layer can capture an entire collaboration. The roles played by different objects are then expressed as nested classes inside the mixin layer. The general pattern is:

```
template <class CollabSuper>
class CollabThis : public CollabSuper {
public:
  class FirstRole  : public CollabSuper::FirstRole  { ... };
  class SecondRole : public CollabSuper::SecondRole { ... };
  class ThirdRole  : public CollabSuper::ThirdRole  { ... };
  ...         // more roles
};                                                          (2.3)
```

Again, mixin layers are composed by instantiating a layer with another as its parameter. This produces two classes that are linked as a parent-child pair in the inheritance hierarchy. For four mixin layers, `Collab1`, `Collab2`, `Collab3`, `FinalCollab` of the above form, we can define a class `T` that expresses the final product of the composition as:

```
typedef Collab1 < Collab2 < Collab3 < FinalCollab > > >  T ;
```

or (alternatively):

```
class T :
  public Collab1 < Collab2 < Collab3 < FinalCollab > > >
{ /* empty body */ };
```

For now we will consider the two forms to be equivalent. Their differences are an artifact of C++ policies and are not important for the discussion of this section (they will be examined together with other C++ specific issues in Chapter 3).

The individual classes that the original design describes are members (nested classes) of the above components. Thus, `T::FirstRole` defines the application class `FirstRole`, etc. Note that classes that do not participate in a certain collaboration can be inherited from collaborations above (we will subsequently use the term “collaboration” for the mixin layer representing a collaboration when no confusion can result). Thus, class `T::FirstRole` will be defined even if

`Collab1` (the bottom-most mixin layer in the inheritance hierarchy) prescribes no role for it.

Example. For a concrete example, consider the graph traversal application of Section 2.2.2. Each collaboration will be represented using a mixin layer. *Vertex Numbering*, for example, prescribes roles for objects of two different classes: *Vertex* and *Workspace*. Its implementation has the form:

```
template <class NextCollab> class NUMBER : public NextCollab
{
public:
  class Workspace : public NextCollab::Workspace {
  ... // Workspace role methods
  };

  class Vertex : public NextCollab::Vertex {
  ... // Vertex role methods
  };
};                                                          (2.4)
```

Note how the actual application classes are nested inside the mixin layer. For instance, the roles for the *Vertex* and *Workspace* classes of Figure 2.1 correspond to `NUMBER::Vertex` and `NUMBER::Workspace`, respectively. Since roles are encapsulated, there is no possibility of name conflict. Moreover, we rely on the standardization of role names. In this example the names `Workspace`, `Vertex`, and `Graph` are used for roles in all collaborations. Note how this is used in code fragment (2.4): Any class generated by this template defines roles that inherit from classes `Workspace` and `Vertex` in its superclass (`NextCollab`).

Other collaborations of our Section 2.2.2 design are similarly represented as mixin layers. Thus, we have a `DFT` and a `UGRAPH` component that capture the *Depth-First Traversal* and *Undirected Graph* collaborations respectively. For instance, methods in the `Vertex` class of the `DFT` mixin layer include `visit-`

DepthFirst and isVisited (with implementations as suggested by their names). Similarly, methods in the Vertex class of UGRAPH include addNeighbor, firstNeighbor, and nextNeighbor, essentially implementing a graph as an adjacency list.

To implement default work methods for the depth-first traversal, we introduced an extra mixin layer, called DEFAULTW. The DEFAULTW mixin layer provides the methods for the Graph and Vertex classes that can be overridden by any graph algorithm (e.g., *Vertex Numbering*) used in a composition.

```
template <class NextCollab> class DEFAULTW : public
NextCollab
{
public:
  class Vertex : public NextCollab::Vertex {
  protected:
    bool workIsDone( NextCollab::Workspace* )    {return 0;}
    void preWork( NextCollab::Workspace* )          {}
    void postWork( NextCollab::Workspace* )         {}
    void edgeWork( Vertex*, NextCollab::Workspace* ) {}
  };

  class Graph : public NextCollab::Graph {
  protected:
    void regionWork( Vertex*, NextCollab::Workspace* ) {}
    void initWork( NextCollab::Workspace* )           {}
    bool finishWork( NextCollab::Workspace* )    {return 0;}
  };
};
```

The introduction of DEFAULTW (as a component separate from DFT) is an implementation detail, borrowed from the VanHilst and Notkin implementation of this example [VN96a]. Its purpose is to avoid dynamic binding and enable multiple algorithms to be composed as separate refinements of more than one DFT component. (In Chapter 3 we will discuss how more powerful parameterization mechanisms than C++ templates can eliminate the need for components like

```
typedef DFT < NUMBER < DEFAULTW < UGRAPH > > > NumberC;
```

Figure 2.4(a) : A composition implementing the vertex numbering operation

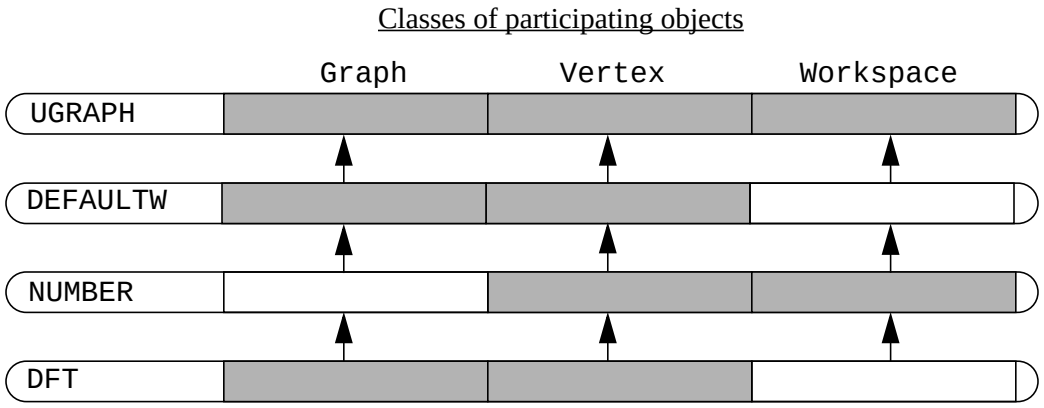

Figure 2.4(b) : Mixin-layers (ovals) and role-members (rectangles inside ovals) in the composition. Every component inherits from the one above it. Shaded role-members are those contained in the collaboration, unshaded are inherited. Arrows show inheritance relationships drawn from subclass to superclass.

`DEFAULTW` by allowing fixpoint constructs to propagate more complete type information upwards in the inheritance hierarchy.)

Consider now a simple collection of collaborations—for instance, one describing the vertex numbering graph operation. The resulting application is obtained from the composition of Figure 2.4(a). We will soon explain what this composition means but first let us see how the different classes are related. The final implementation classes are members of the product of the composition, `NumberC` (e.g., `NumberC::Graph` is the concrete graph class). Figure 2.4 shows the mixin layers and their member classes, which represent roles, as they are actually composed. Each component inherits from the one above it. That is, `DFT` inherits role-members from `NUMBER`, which inherits from `DEFAULTW`, which inherits from `UGRAPH`. At the same time, `DFT::Graph` inherits methods and variables from `NUMBER::Graph`, which inherits from `DEFAULTW::Graph`, which inherits from

UGRAPH::Graph. This double level of inheritance is what makes the mixin-layer approach so powerful. Note, for instance, that, even though NUMBER does not specify a Graph member, it inherits one from DEFAULTW. The simplicity that this design affords will be made apparent in the following sections, when we compare it with alternatives.

The interpretation of the composition in Figure 2.4 is straightforward: Every component is implemented in terms of the ones above it. For instance, the DFT component is implemented in terms of methods supplied by NUMBER, DEFAULTW, and UGRAPH. An actual code fragment from the visitDepthFirst method implementation in DFT::Vertex is the following:

```
for ( v = (Vertex*)firstNeighbor();
      v != NULL;
      v = (Vertex*)nextNeighbor() )
 {  edgeWork(v, workspace);
    v->visitDepthFirst(workspace);  }                    (2.5)
```

The firstNeighbor, nextNeighbor, and edgeWork methods are not implemented by the DFT component. Instead they are inherited from components above it in the composition. firstNeighbor and nextNeighbor are implemented in the UGRAPH component (as they encode the iteration over nodes of a graph). edgeWork is a traversal refinement and (in this case) is implemented by the NUMBER component.

We can now more easily see how mixin layers are in fact both reusable and interchangeable. The DFT component of Figure 2.4 is oblivious to the *implementations* of methods in components above it. Instead, DFT only knows the *interface* of the methods it expects from its parent. Thus, the code of (2.5) represents a skeleton expressed in terms of abstract operations firstNeighbor, nextNeighbor, and edgeWork. Changing the implementation of these operations merely requires the

swapping of mixin layers. For instance, we can create an application (`CycleC`) that checks for cycles in a graph by replacing the `NUMBER` component with `CYCLE`:

```
typedef DFT < CYCLE < DEFAULTW < UGRAPH > > > CycleC;
```

The results of compositions (`CycleC` above and `NumberC` in Figure 2.4(a)) can be used by a client program as follows: First, an instance of the nested `Graph` class (`NumberC::Graph` or `CycleC::Graph`) needs to be created. Then, `Vertex` objects are added and connected in the graph (the `Graph` role in mixin-layer `UGRAPH` defines methods `addVertex` and `addEdge` for this purpose). After the creation of the graph is complete, calling method `depthFirst` on it will execute the appropriate graph algorithm.

Note that no direct editing of the component is necessary and multiple copies of the same component can co-exist in the same composition. For instance, we could combine two graph algorithms by using two instances of the `DFT` mixin layer (in the same inheritance hierarchy), refined to perform a different operation each time:

```
class NumberC :
  public DFT < NUMBER < DEFAULTW < UGRAPH > > > {};
class CycleC  :
  public DFT < CYCLE < NumberC > > {};                    (2.6)
```

As another example, the design may change to accommodate a different underlying model. For instance, operations could now be performed on directed graphs. The corresponding update (`DGRAPH` replaces `UGRAPH`) to the composition is straightforward (assuming that the algorithms are still valid for directed graphs as is the case with Holland's original implementation of this example [Hol92]):

```
typedef DFT < NUMBER < DEFAULTW < DGRAPH > > > NumberC;
```

Again, note that the interchangeability property is a result of the independence of collaborations. A single UGRAPH collaboration completely incorporates all parts of an application that relate to maintaining an undirected graph (although these parts span several different classes). The collaboration communicates with the rest of the application through a well-defined and usually narrow interface.

For this and other similar examples, the reusability and interchangeability of mixin layers helps solve the classical *library scalability problem* [BSST93, Big94]: there are *n* features and often more than *n*! valid combinations (because composition order matters and feature replication is possible [BO92]). Hard-coding all different combinations leads to library implementations that do not scale: the addition of a single feature doubles the size of the library. Instead, we would like to have a collection of building blocks and compose them appropriately to derive the desired combination. In this way, the size of the library grows linearly in the number of features it can express (instead of exponentially, or super-exponentially).

Multiple Collaborations in a Single Design. An interesting question is whether mixin layers can be used to express collaboration-based designs where a single collaboration is instantiated more than once with the same class playing different roles in each instantiation. The answer is positive, and the desired result can be effected using *adaptor* mixin layers. Adaptor layers add no implementation but adapt a class so that it can play a pre-defined role. That is, adaptor layers contain classes with empty bodies that are used to "redirect" the inheritance chain so that predefined classes can play the required roles.

Consider the case of a producer-consumer collaboration, which was briefly discussed in Section 2.2.1. Our example is from the domain of compilers. A parser in a compiler can be viewed as a consumer of tokens produced by a lexical ana-

lyzer. At the same time, however, a parser is a producer of abstract syntax trees (consumed, for instance, by an optimizer). We can reuse the same producer-consumer collaboration to express both of these relationships. The reason for wanting to provide a reusable implementation of the producer-consumer functionality is that this functionality could be quite complex. For instance, the buffer for produced-consumed items may be guarded by a semaphore, multiple consumers could exist, etc. The mixin layer implementing this collaboration takes `Item` as a parameter, describing the type of elements produced or consumed:

```
template <class NextCollab, class Item>
class PRODCONS : public NextCollab
{
public:
  class Producer : public NextCollab::Producer {
    void produce(Item item) { ... }
    // The functionality of producing Items is defined here
  ... // other Producer role methods
  };

  class Consumer : public NextCollab::Consumer {
    Item consume() { ... }
    // The functionality of consuming Items is defined here
  ... // other Consumer role methods
  };
};
```

Now we can use two simple adaptors to make a single class (`Parser`) be both a producer and a consumer (in two different collaborations). The first adaptor (`PRODADAPT`) expresses the facts that a producer is also going to be a consumer (the actual consumer functionality is to be added later) and that the `Optimizer` class inherits the existing consumer functionality. This adaptor is shown below:

```
template <class NextCollab> class PRODADAPT :
  public NextCollab
{
```

```
public:
  class Consumer : public NextCollab::Producer { };
  class Optimizer : public NextCollab::Consumer { };
  class Producer { };
};
```

The second adaptor (`CONSADAPT`) is similar:

```
template <class NextCollab> class CONSADAPT :
  public NextCollab
{
public:
  class Parser : public NextCollab::Consumer { };
  class Lexer : public NextCollab::Producer { };
};
```

Now a single composition can contain two copies of the `PRODCONS` mixin layer, appropriately adapted. For instance:

```
typedef COMPILER < CONSADAPT < PRODCONS <
        PRODADAPT < PRODCONS < ..., Tree> >, Token > > >
  CompilerApp;                                          (2.7)
```

In the above, the `COMPILER` mixin layer is assumed to contain the functionality of a compiler that defines three classes, `Lexer`, `Parser`, and `Optimizer`. These classes use the functionality supplied by the producer-consumer mixin layer. For instance, there may be a `parse` method in `COMPILER::Parser` that repeatedly calls the `consume` and `produce` methods. To better illustrate the role of adaptors, we present in Figure 2.5 the desired inheritance hierarchy for this example, as well as the way that adaptors are used to enable emulating this hierarchy using only predefined mixin layers. Note that each of the layers participating in composition (2.7), above, appears as a rectangle in Figure 2.5(b).

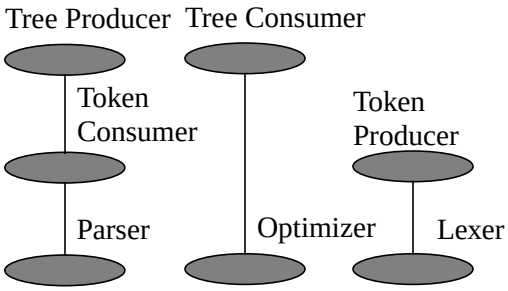

Figure 2.5(a) : The desired inheritance hierarchy has a Parser inheriting functionality both from a consumer class (a Parser is a consumer of tokens) and a producer class (a Parser is a producer of trees).

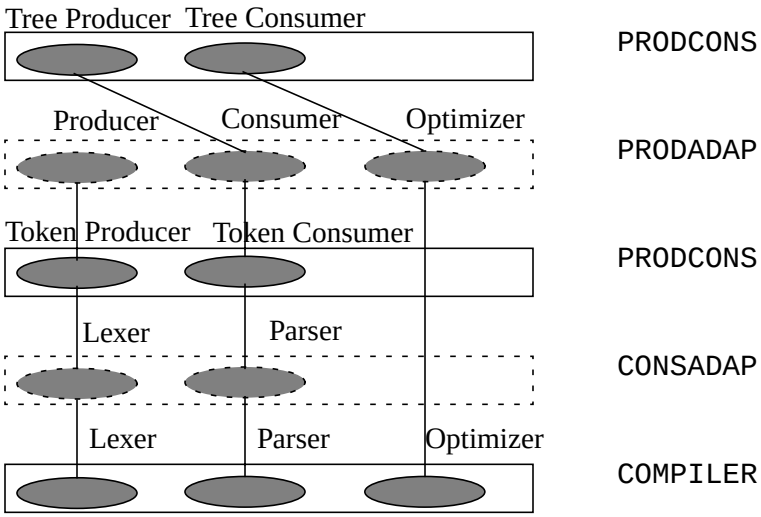

Figure 2.5(b) : By using adaptor layers (dotted rectangles), one can emulate the inheritance hierarchy of Figure 2.5(a), using only pre-defined mixin layers (solid rectangles). Since a single mixin layer (`PRODCONS`) is instantiated twice, adaptors help determine which class will play which role every time.

2.4.2 Comparison to Application Frameworks

In object-oriented programming, an *abstract* class is one that cannot be instantiated (i.e., cannot be used to create objects) but is only used to capture the common-

alities of other classes. These classes inherit the common interface and functionality of the abstract class. An *object-oriented application framework* (or just *framework*) consists of a suite of interrelated abstract classes that embodies an abstract design for software in a family of related systems [JF88]. Each major component of the system is represented by an abstract class. These classes contain dynamically bound methods (`virtual` in C++), so that the framework user can add functionality by creating subclasses and supplying definitions for the appropriate methods. Thus, frameworks have the advantage of allowing reuse at a granularity larger than a single abstract class. But frameworks have the disadvantage that users may have to manually specify system-specific functionality.

In a *white-box framework*, users specify system-specific functionality by adding *methods* to the framework's classes. Each method must adhere to the *internal* conventions of the classes. Thus, using white-box frameworks is difficult, because it requires knowledge of their implementation details. In a *black-box framework*, the system-specific functionality is provided by a set of classes. These classes need adhere only to the proper *external* interface. Thus, using black-box frameworks is easier, because it does not require knowledge of their implementation details. Using black-box frameworks is further simplified when they include a library of pre-written functionality that can be used as-is with the framework.

Frameworks can be used to implement collaboration-based designs, but the amount of flexibility and modularity they can afford is far from optimal. The reason is that frameworks allow the reuse of abstract classes but have no way of specifying collections of concrete classes that can be used at will (i.e., either included or not and in any order) to build an application. Intuitively, frameworks allow reusing the skeleton of an implementation but not the individual pieces that are built on top of the skeleton. This can be seen through a simple combinatorics argument. Consider a set of four features, *A*, *B*, *C*, and *D* that can be combined arbitrarily to

yield complete applications. For simplicity, assume that feature *A* will always be first, and that no feature repetition is allowed. Then a framework may encode feature combination *AB*, thus allowing the user to program combinations *ABCD* and *ABDC*. Nevertheless, these combinations will have to be coded separately (i.e., they cannot use any common code other than their common prefix, *AB*). The reason is that each instantiation of the framework creates a separate inheritance hierarchy and reusing a combination is possible only if one can inherit from one of its (intermediate or final) classes. That is, only common prefixes are reusable. In our four-feature example, combinations that have no common prefix with the framework (for instance, *ACD*) simply cannot take advantage of it and have to be coded separately. This amounts to exponential redundancy for complex domains.

In the general case, assume a simple cost model that assigns one cost unit to each re-implementation of a feature. If feature order matters but no repetitions are possible, the cost of implementing all possible combinations using frameworks is equal to the number of combinations (each combination of length *k* differs by one feature from its prefix of length *k-1*). Thus, for *n* features, the total cost for implementing all combinations using frameworks is $\sum_{k=0}^{n} \frac{n!}{(n-k)!}$. (This number is derived by considering the sum of the feature combinations of length *k*, for each *k* from 0 to *n*.) In contrast, the cost of using mixin layers for the same implementation is equal to *n*—each component is implemented once and can be combined in arbitrarily many ways. With mixin layers, even compositions with no common prefixes share component implementations.

Even though our combinatorics argument represents an extreme case, it is reflective of the inflexibility of frameworks. Optional features are quite common in practice and frameworks cannot accommodate them, unless all combinations are

explicitly coded by the user. This is true even for domains where feature composition order does not matter or features have a specific order in which they must be used.

Another disadvantage of using frameworks to implement collaboration-based designs comes from the use of dynamically bound methods in frameworks. Even though the dynamic dispatch cost is sometimes negligible or can be optimized away, often it can impose a run-time overhead, especially for fine-grained classes and methods. With mixin layers, this overhead is avoided, as there is little need for dynamic dispatch. The reason is that mixin layers can be ordered in a composition so that most of the method calls are to their parent layers.

This reveals a general and important difference between mixin-based programming and standard object-oriented programming. When a code fragment in a conventional OO class needs to be generic, it is implemented in terms of dynamically bound methods. These methods are later (re-)defined in a subclass of the original class, thus refining it for specific purposes. With mixin classes, the situation is different. A method in a mixin class can define generic functionality by calling methods in the class's (yet undefined) *superclass*. That is, generic calls for mixins can be both up-calls and down-calls in the inheritance hierarchy. Generic up-calls are specialized statically, when the mixin class's superclass is set. Generic down-calls provide the standard OO run-time binding capabilities. In this way, mixin classes can be used with greater freedom regarding their position in an inheritance hierarchy. Refinement of existing functionality is not just a top-down process but involves composing mixins arbitrarily, often with many different orders being meaningful.

Example. We can illustrate the above points with our usual graph algorithm example of Section 2.2.2. The original implementation of this application [Hol92] used

a black-box application framework on which the three graph algorithms were implemented. The framework consists of the implementations of the `Graph`, `Vertex`, and `Workspace` classes for the *Undirected Graph* and *Depth First Traversal* collaborations. The classes implementing the depth-first traversal have methods like `preWork`, `postWork`, `edgeWork`, etc., *which are declared to be dynamically bound* (`virtual` in C++). In this way, any classes inheriting from the framework classes can refine the traversal functionality by redefining the operation to be performed the first time a node is visited, when an edge is traversed, etc.

VanHilst and Notkin discussed the framework implementation of this example in detail [VN96a]. Our presentation here merely adapts their observations to our above discussion of using frameworks to implement collaboration-based designs. A first observation is that, in the framework implementation, the base classes are fixed and changing them requires hand-editing (usually copying and editing, which results in redundant code). For instance, consider applying the same algorithms to a directed, as opposed to an undirected graph. If both combinations need to be used in the same application, code replication is necessary. The reason is that the classes implementing the graph algorithms (e.g., *Vertex Numbering*) must have a fixed superclass. Hence, two different sets of classes must be introduced, both implementing the same graph algorithm functionality but having different superclasses.

A second important observation pertains to our earlier discussion of optional features in an application. In particular, a framework implementation does not allow more than one refinement to co-exist in the same inheritance hierarchy. Thus, unlike the mixin layer version of code fragment (2.6) in Section 2.4.1, we cannot have a single graph that implements both the *Vertex Numbering* and the *Cycle Checking* operations. The reason is that the dynamic binding of methods in the classes implementing the depth-first traversal causes the most refined version

of a method to be executed on every invocation. Thus, multiple refinements cannot co-exist in the same inheritance hierarchy since the bottom-most one in the inheritance chain always supersedes any others. In contrast, the flexibility of mixin layers allows us to break the depth-first traversal interface in two (the `DEFAULTW` and the `DFT` component, discussed earlier) so that `DFT` calls the refined methods *in its superclass* (i.e., without needing dynamic binding). In this way, multiple copies of the `DFT` component can co-exist and be refined separately. At the same time, obviating dynamic binding results into a more efficient implementation—dynamic dispatch incurs higher overhead than calling methods of known classes (although sometimes it can be optimized by an aggressive compiler).

2.4.3 Comparison to the VanHilst and Notkin Method

The VanHilst and Notkin approach [VN96a-c, Van97] is another technique that can be used to map collaboration-based designs into programs. The method employs C++ mixin classes, which offer the same flexibility advantages over a framework implementation as the mixin layers approach. Nevertheless, the components represented by VanHilst and Notkin are small-scale, resulting in complicated specifications of their interdependencies.

VanHilst and Notkin use mixin classes in C++ to represent roles. More specifically, each individual role is mapped to a different mixin and is also parameterized by any other classes that interact with the given role in its collaboration. For an example, consider role `B4` in Figure 2.6 (which replicates Figure 2.1 for easy reference). This role participates in a collaboration together with two other roles, `A4` and `C4`. Hence, it needs to be aware of the classes playing the two roles (so that, for instance, it can call appropriate methods). With the VanHilst and Notkin tech-

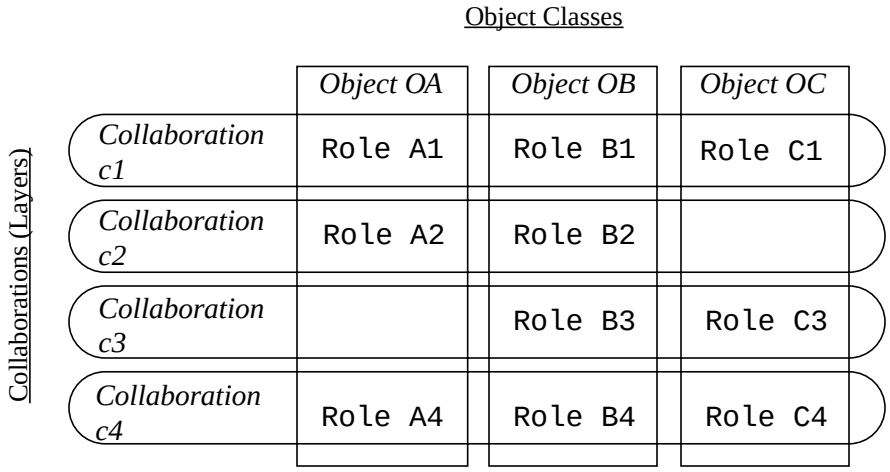

Figure 2.6 : Example collaboration decomposition. Ovals represent collaborations, rectangles represent objects, their intersections represent roles.

nique, the role implementation would be a mixin, also parameterized by the two extra classes:

```
template <class RoleSuper, class OA, class OC>
class B4 : public RoleSuper {
   ... /* role implementation, using OA, OC */
};                                                    (2.8)
```

Consider that the actual values for parameters `OA`, `OC` would themselves be the result of template instantiations, and their parameters also, and so on (up to a depth equal to the number of collaborations). This makes the VanHilst and Notkin method complicated even for relatively small examples. In the case of a composition of *n* collaborations, each with *m* roles, the VanHilst and Notkin method can yield a parameterization expression of length m^n. Additionally, the programmer has to explicitly keep track of the mapping between roles and classes, as well as the collaborations in which a class participates. For instance, the mixin for role `A4` in Figure 2.1 has to be parameterized with the mixin for role `A2`—the programmer cannot ignore the fact that collaboration *c3* does not specify a role for object *OA*.

From a software evolution standpoint, local design changes cannot easily be isolated, since collaborations are not explicitly represented as components. These limitations make the approach *unscalable*: various metrics of programmer effort (e.g., length of composition expressions, parameter bindings that need to be maintained, etc.) grow exponentially in the number of features supported. (This is the same notion of scalability as in our earlier discussion of the library scalability problem.)

Conceptually, the scalability problems of the VanHilst and Notkin approach are due to the small granularity of the entities they represent. In their methodology, each mixin class represents a role. Roles, however, have many external dependencies (for instance, they often depend on many other roles in the same collaboration). To avoid hard-coding such dependencies, we have to express them as extra parameters to the mixin class, as in code fragment (2.8). Actual reusable components need to have few external dependencies, as made possible by using mixin layers to model collaborations.

Example. Our above discussion can be best illustrated with a simple example from our graph algorithms application of Section 2.2.2. Consider a composition implementing both the *Cycle Checking* and the *Vertex Numbering* operation on the same graph. We select which of the two is to be performed on a certain graph object by qualifying method names directly, e.g., `g->NumberC::Graph::Traverse()`. (An alternative would be to cast an object pointer to the appropriate type and use it to call the depth-first traversal method.) Recall that the ability to compose more than one refinement (or multiple copies of the same refinement) is an advantage of the mixin-based approach (both ours and the VanHilst and Notkin method) over frameworks implementations.

```
class NumberC: public DFT <NUMBER <DEFAULTW <UGRAPH> > > {};
class CycleC : public DFT < CYCLE < NumberC > >          {};
```

Figure 2.7(a) : Our mixin layer implementation of a multiple-collaboration composition. The individual classes are members of `NumberC`, `CycleC` (e.g., `NumberC::Vertex`, `CycleC::Graph`, etc.).

```
class Empty {};
class WS         : public WorkspaceNumber                {};
class WS2        : public WorkspaceCycle                 {};
class VGraph     : public VertexAdj<Empty>               {};
class VWork      : public VertexDefaultWork<WS,VGraph>   {};
class VNumber    : public VertexNumber<WS,VWork>         {};
class V          : public VertexDFT<WS,VNumber>          {};
class VWork2     : public VertexDefaultWork<WS2,V>       {};
class VCycle     : public VertexCycle<WS2,VWork2>        {};
class V2         : public VertexDFT<WS2,VCycle>          {};
class GGraph     : public GraphUndirected<V2>            {};
class GWork      : public GraphDefaultWork<V,WS,GGraph>  {};
class Graph      : public GraphDFT<V,WS,GWork>           {};
class GWork2     : public GraphDefaultWork<V2,WS2,Graph> {};
class GCycle     : public GraphCycle<WS2,GWork2>         {};
class Graph2     : public GraphDFT<V2,WS2,GCycle>        {};
```

Figure 2.7(b) : Same implementation using the VanHilst/Notkin approach. `V` corresponds to our `NumberC::Vertex`, `Graph` to `NumberC::Graph`, `WS` to `NumberC::Workspace`, etc.

The components (mixins) used by VanHilst and Notkin for this example are similar to the inner classes in our mixin layers, with extra parameters needed to express their dependencies to other roles in the same collaboration. This complicates the source code needed to compose components, making the VanHilst and Notkin composition code much longer than the corresponding mixin layers source code.[5] Our specification is shown in Figure 2.7(a) (reproducing code fragment (2.6)). A compact representation of a VanHilst and Notkin specification is shown

5. The object code is, as expected, of almost identical size.

in Figure 2.7(b). (A more readable version of the same code included in [VN96a] is even lengthier).

Figure 2.7(b) makes apparent the complications of the VanHilst/Notkin approach. Each mixin representing a role can have an arbitrary number of parameters and can instantiate a parameter of other mixins. In this way, parameterization expressions of exponential (to the number of collaborations) length can result. To alleviate this problem, the programmer has to introduce explicitly intermediate types that encode common sub-expressions. For instance, `V` is an intermediate type in Figure 2.7(b). Its only purpose is to avoid introducing the sub-expression `VertexDFT<WS,VNumber>` three different times (wherever `V` is used). Of course, `VNumber` itself is also just a shorthand for `VertexNumber<WS,VWork>`. `VWork`, in turn, stands for `VertexDefaultWork<WS,VGraph>`, and so on.[6] Additional complications arise when specifying a composition: users must know the number and position of each parameter of a role-component. Both of the above requirements significantly complicate the implementation and make it error-prone.

Using mixin layers, the exponential blowup of parameterization expressions is avoided. Every mixin layer only has a single parameter (the layer above it). By parameterizing a mixin layer *A* by *B*, *A* becomes implicitly parameterized by all the roles of *B*. Furthermore, if *B* does not contain a role for an object that *A* expects, it will inherit one from above it. This is the benefit of expressing the collaborations themselves as classes: they can extend their interface using inheritance.

Another practical advantage of the mixin layer approach is that it encourages consistent naming for roles. No name conflicts are possible among different

6. Some compilers (e.g., MS VC++, g++) internally expand template expressions, even though the user has explicitly introduced intermediate types. This caused page-long error messages for incorrect compositions when we experimented with the VanHilst and Notkin method, rendering debugging impossible.

mixin layers, since role representations are encapsulated in the outer class. Hence, instead of explicitly giving unique names to role-members, we have standard names and only distinguish instances by their enclosing mixin layer. In this way, `VertexDFT`, `GraphDFT`, and `VertexNumber` become `DFT::Vertex`, `DFT::Graph` and `NUMBER::Vertex`, respectively.

In [VN96a], VanHilst and Notkin questioned the scalability of their method. One of their concerns was that the composition of large numbers of roles "can be confusing even in small examples..." The observations above (length of parameterization expressions, number of components, consistent naming) show that the mixin layer approach addresses this problem and does scale gracefully.

Chapter 3

Programming Language Issues

As we showed in the previous chapter, mixin layers are capable of expressing elegant, modularized software components. These components exist at the language level and their composition is performed statically, i.e., at language translation (compilation) time. Several programming language issues arise in connection with mixin layers and their compositions. Most of these issues pertain to the interactions of mixin layers with type systems. Type information can be used to detect errors in a composition of mixin layers. At the same time, layers are defined in isolation and the problem of propagating type information between layers is especially interesting.

This chapter is an amalgam of several such topics related to mixin layers. Section 3.1 examines the general problem of verifying the correctness of a layer composition. We present an approach based on propositional properties that are propagated using inheritance and can be checked by other layers. Propositional properties can be disguised as classes so that standard language mechanisms (like class inheritance and access control) can be used to validate the properties.

Section 3.2 discusses type system support for mixin layers. We propose an extension to a type mechanism based on explicit types (Java interfaces) so that constraints on mixin layers can be expressed.

Section 3.3 examines the problem of propagating type information across mixin layers. The proposed solution is similar to the one offered by Wadler, Odersky, and this author [WOS98] in a different context. We discuss how this solution applies to unconstrained parameterization mechanisms (e.g, C++) as well as constrained parameterization mechanisms (e.g., the Java generics proposal of [AFM97]).

Finally, Section 3.4 describes the interaction of C++ mixins with various idiosyncrasies of the language. This information is important from a practical standpoint and offers the opportunity to discuss some C++-specific issues pertaining to parameterization and inheritance.

3.1 Verifying Composition Correctness

Given a set of mixin layers, not all compositions of layers in the set may be meaningful. Often layers need to be used in a specific order, or cannot be used more than once. Other times a layer expects some core functionality from the layers above it, thus requiring that at least a layer with the desired functionality be present. This is true of the layers we encountered in the previous chapter. For instance, in our example graph application, `DFT` always has to precede `DEFAULTW` in the composition read from left to right (i.e., `DEFAULTW` has to occur higher than `DFT` in the inheritance hierarchy). In case this constraint is not satisfied, a compile-time error will occur (or worse, under C++ the error may occur in some contexts but not others, as we explain in Section 3.4). The reason is that `DFT` attempts to call methods that only the `DEFAULTW` layer defines. In other words, the compiler can tell that the composition is invalid because of an interface mismatch: the interface exported by the superclass of the `DFT` layer does not support some expected methods.

The issue of explicitly specifying the expected interface of a layer parameter is an important one, both for verifying a composition, and for enabling separate compilation of layers. Interfaces for mixin layers are independently interesting and will be discussed separately in Section 3.2. In general, however, compositions may fail for reasons other than interface mismatch. In this section we will address the general problem of detecting when component compositions are invalid. Often layers are perfectly compatible from an interface standpoint (i.e., they contain the expected methods and variables) but their composition does not produce correct results. Incorrect compositions will either fail with a run-time error or not perform as expected. The designer of mixin layers is probably aware of which compositions are actually meaningful and which are not. We would like to develop techniques for enabling the expression of this information so that compilers can validate compositions automatically. This is the purpose of the method discussed in Section 3.1.2, but first we will introduce a set of example layers from the domain of data structures to help illustrate the problem.

3.1.1 An Example Application

Our example data structure design was used in both the P2 lightweight DBMS generator [BT97, Tho98], and in the DiSTiL generator for data structures [SB97]. In this example we add functionality to a data structure by assigning more roles to the classes that participate in the design. There are two such classes: a *node* class, of which all data nodes are instances, and a *container* class, which has one instance per data structure. A third class for data structure cursors (iterators) is generally needed but to keep the example simple we will equate cursors with pointers to node objects. This model for data structure construction is, in fact, quite general. Composite data structures, run-time bound checks, garbage collection, a lock and transaction manager, etc., can all be specified as new roles for the node and con-

tainer classes (see [BT97]). This can be achieved using mixin layers, as we will show with extensions to a binary tree data structure.

Our target data structure consists of four different collaborations: *bintree*, *alloc*, *timestamp*, and *sizeof*. *Bintree* captures the functionality of a binary tree. *Alloc* captures the functionality of memory allocation. *Timestamp* is responsible for maintaining timestamps for data structure and element updates. *Sizeof* simply keeps track of the data structure size. The design is simple and we will not concern ourselves with its schematic representation (in the form of Figure 2.1) or the way we obtained it. We remind the reader that a good reference on how to obtain collaboration-based designs from use-case scenarios [Rum94] is VanHilst's Ph.D. dissertation [Van97].

A mixin layer implementing a binary tree collaboration has the form:[1]

```
template <class Super> class BINTREE : public Super {
public:
  class Node : public Super::Node {
    Node* parent_link,
          left_link, right_link ;   // Node data members
  public:
    ...                             // Node interface
  };

  class Container : public Super::Container {
    Node* header;                 // Container data members
  public:
    void insert ( EleType el ) { ... }
                         // Definition of EleType inherited
    void erase  ( Node* node ) { ... }
    bool find  ( EleType* el ) { ... }
```

1. We will present simplified code fragments, ignoring implementation details that are not directly relevant to our discussion. We will highlight class definitions for readability and use ellipses (. . .) for omitted code.

```
    ...                          // Other methods
  };
};
```

Note that the `Container` class is aware of the `Node` class (e.g., it declares a member variable of type `Node*`). The two classes must be designed together and, hence, it makes sense to encapsulate both in a single unit.

Now consider the implementation of the *timestamp* collaboration: the data structure maintains the time of its last update, as well as the creation and update time of each node. The set of exported operations on the data structure can be enriched (e.g., by defining an operation that returns the data structure update time, as well as a variant of `find`: `find_newer`). This enrichment can be viewed as a collaboration prescribing roles for both the `Node` and the `Container` class. Its implementation using mixin layers has the form:

```
template <class Super> class TIMESTAMP : public Super {
public:
  class Node : public Super::Node {
    time_t creation_time, update_time; // Node data members
  public:
    bool more_recent (time_t t) { ... }
    ...                        // Other time-related methods
  };

  class Container : public Super::Container {
    time_t update_time;          // Container data members
  public:
    bool find_newer ( EleType* el, time_t t ) { ... }
    void insert ( EleType el ) { ... }
    ...                        // Other time-related methods
  };
};
```

Recall that not all collaborations need to specify roles for all classes in a design. The *sizeof* collaboration, for instance, only needs to maintain a counter of elements associated with a container and only prescribes a role for the `Container`

class. It can be implemented as a mixin layer that is a trivial wrapper around a mixin class:

```
template <class Super> class SIZEOF : public Super {
public:
  class Container : public Super::Container {
    int count;                  // Container data members
  public:
    Container() : Super::Container() {
       count = 0; }             // Constructor
    void insert ( EleType el ) {
       Super::Container::insert(el); count++; }
    void erase ( Node* node ) {
       Super::Container::erase(el); count--; }
    int size () { return count; }
  };
};
```

Again, classes generated by instantiating the `SIZEOF` mixin layer do have a `Node` nested class—this class is inherited from mixin layers above `SIZEOF` in the inheritance chain.

To put everything together we need a concrete (i.e., non-mixin) class to be the root of our inheritance hierarchy. This could be a “dummy” class, containing only empty roles. In most applications, however, it is easy to identify a collaboration, which has to be the basis upon all other functionality is built. In this particular example, the *alloc* collaboration serves this purpose. *Alloc* is responsible for the actual memory allocation for the data structure. Note that the implementation of this collaboration (as well as any of the other mixin layers) can have parameters other than the one we used to designate the superclass. These extra parameters can be used to specify polymorphic behavior. In our example, it makes sense to parameterize the layer representing *alloc* by the type of the elements stored in the data structure. Then we have:

```
template <class Element> class ALLOC {
public:
  class Node {
    Element element;  // The actual stored data
  public:
    ...             // Any methods pertaining to stored data
  };

  class Container {
  protected:
    typedef Element EleType;
                      // The actual type of stored data
    void* node_alloc();
    ...               // Other allocation methods
  };
};
```

With our layers defined, we form data structures by composing layers. A binary tree storing integers and maintaining time information and size is defined as:

```
typedef SIZEOF < TIMESTAMP < BINTREE < ALLOC < int > > > >
        Tree1;                                          (3.1)
```

The `Node` and `Container` classes are accessible[2] as `Tree1::Node` and `Tree1::Container`. An outline of the composition of (3.1) is shown in Figure 3.1. We have annotated the design with some of the inherited member variables and methods. Note how both the `SIZEOF` and the `TIMESTAMP` mixin layers depend on layers above them to insert and erase elements from the data structure. We will return to this later.

2. There is no reason why the `Node` class should be user accessible. What really needs to be user accessible is an iterator class, which for this example is the same as a pointer to a `Node` object.

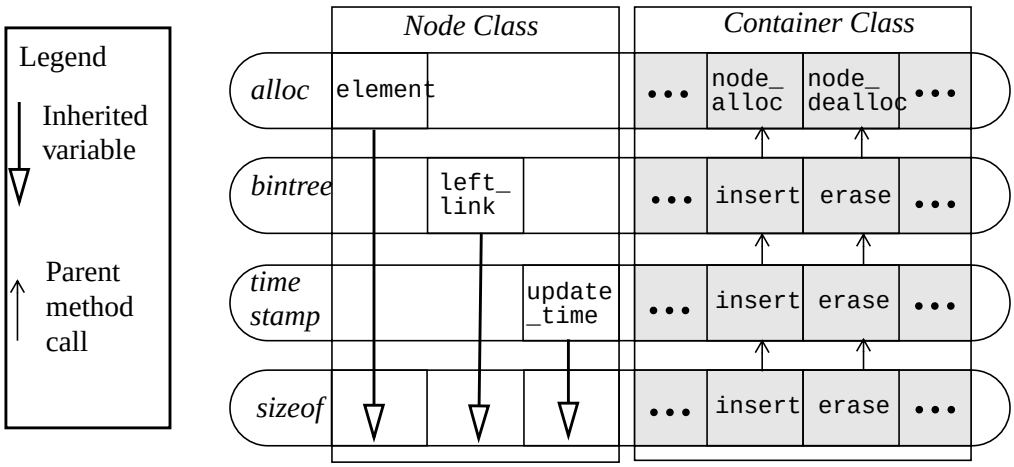

Figure 3.1: A composite data structure. The intersections of rectangles and ovals represent the roles played by each class in each collaboration.

3.1.2 Verifying Consistency with Propositional Properties

The most important issue arising in mixin layer composition is ensuring composition correctness. Some mixin layers depend on the existence or the right ordering of others. Many problems can be detected immediately. As shown in Figure 3.1, the `BINTREE` layer calls the allocator directly for every element insertion (i.e., it does not propagate the `insert` and `erase` operations). Omitting the `BINTREE` layer in (3.1) should cause a compilation error: operations like `insert` that are propagated by `SIZEOF` and `TIMESTAMP` will be undefined. [3]

3. In fact, such mistakes may not actually cause a compilation error, even for statically typed languages. In C++, for instance, if `insert` is never called in user code, no error will be signalled even though `SIZEOF::Container` has an explicit call to the insert method of its superclass and no such method is defined. This has to do with the treatment of methods in parameterized classes as function templates, as we will discuss in Section 3.4. In essence, the `insert` method for `SIZEOF::Container` is never compiled since it is not needed, thus the error is never discovered.

Other problems, however, are more subtle. Consider reordering the `BINTREE` and `SIZEOF` layers in a composition:

```
typedef BINTREE < SIZEOF < ALLOC < int > > >  Tree3;
```

This will cause the `insert` and `erase` methods of `SIZEOF` to be shadowed (overridden) by those of `BINTREE`. Hence, the implementation is wrong: the count of elements in the data structure will never be updated (since this is only done in the `insert` and `erase` methods of `SIZEOF` and these methods are not called by `BINTREE`). The `size` operation will be visible, however, and will always return 0, although the data structure may not be empty.

In general, mixin layers may have subtle semantic dependencies that are not reflected in their interfaces. In large libraries there may be a variety of layers supporting identical interfaces but implementing different semantics. Many combinations of layers could be illegal but there may not be a way to detect this from the interfaces alone.

This problem has been studied before in the context of layered systems. The *design rule checking* approach of [BG97] offers a solution using propositional properties and requirements that are propagated both up and down a layer hierarchy. The *nested mixin-methods* of [SCD+93] resulted in a powerful constraint system. Nesting of mixins was used as a way to restrict their scope. A mixin class of [SCD+93] can define other mixins that can be composed with it, inherit some mixins when composed, and cancel inherited mixins. The *feature-oriented* programming approach of [Pre97] uses the `assumes` keyword to express the property that the correctness of one feature (layered component) assumes the existence of another.

Interestingly enough there is a simple way to express basic dependencies within the mixin layers framework. Every mixin layer can export propositional

properties describing its behavior (essentially encoding semantic knowledge in its interface). Recall that when mixin layers are composed, they are linked in an inheritance chain. Properties are propagated in the same direction as inherited methods and variables: from superclasses to subclasses. Layers can explicitly make inherited properties unavailable to their subclasses. Finally, a layer can check (require) whether it has inherited a property or not. A composition is correct if none of these requirements fail. This technique is similar to the `assumes` functionality of [Pre97] and the design rule checking of [BG97]. Consider the example of Section 3.1.1. There are four requirements that we need to express:

- A `BINTREE` mixin layer cannot have a `SIZEOF` layer as an ancestor in its inheritance chain (because otherwise the `insert` method of `SIZEOF` will be shadowed).
- A `BINTREE` mixin layer cannot have a `TIMESTAMP` layer as an ancestor (same reason as above).
- A `SIZEOF` mixin layer needs to ensure that some sort of a data structure is present in the composition. In our example the only data structure is a binary tree but we can easily imagine the same mixin layer being composed, for instance, with a doubly linked list layer.
- A `TIMESTAMP` mixin layer also needs to ensure that a data structure is present.

These can be specified as requirements on the existence of three properties (inherited from ancestors in the inheritance chain):

- No `SIZEOF` layer is present (call this property `P_NoSizeof`).
- No `TIMESTAMP` layer is present (call this property `P_NoTimestamp`).
- A data structure layer is present (call this property `P_DataStructure`).

An approximate implementation of this scheme is straightforward. All properties can be expressed as empty classes encapsulated in a mixin layer. Proper-

ties are inherited but can be negated by using access control (that is, “hiding” of class members—e.g., by making them “private” members in C++). If the class representing the property is made visible to subclasses (either by declaration or by inheritance without “hiding”), then the property is asserted. Otherwise the property is negated. The requirement that a certain property be satisfied is then enforced by declaring an instance of this class. (This technique is really an approximation of the desired functionality: We “hijack” the nested class mechanism and use it to express propositional properties. As we will see, this method has some limitations.)

In our example, `BINTREE` exports property `P_DataStructure` and requires properties `P_NoSizeof` and `P_NoTimestamp`.

```
template <class Super> class BINTREE : public Super {
protected:
  class P_DataStructure { };
     // Assert this property for subclasses
private:
  P_NoSizeof dummy1;
  P_NoTimestamp dummy2;
     // Require P_NoSizeof and P_NoTimestamp from ancestors
public:
  ...   // nested mixins (same as before)
};
```

The other three mixin layers are modified accordingly:

```
template <class Super> class SIZEOF : public Super {
private:
  class P_NoSizeof { };    // Negate property for subclasses
  P_DataStructure dummy1;  // Require P_DataStructure
public:
  ...                      // nested mixins (same as before)
};

template <class Super> class TIMESTAMP : public Super {
private:
```

```
  class P_NoTimestamp { }; // Negate property for subclasses
  P_DataStructure dummy1;  // Require P_DataStructure
public:
  ...                      // nested mixins (same as before)
};

template <class EleType> class ALLOC {
protected:
  class P_NoSizeof { };   // Assert property for subclasses
  class P_NoTimestamp { };// Assert property for subclasses
public:
  ...                     // nested classes (same as before)
};
```

Note how the constraint is enforced: the `ALLOC` mixin layer asserts properties `P_NoSizeof` and `P_NoTimestamp`. The `BINTREE` layer requires that they not be negated by some layer between `BINTREE` and `ALLOC` in the inheritance hierarchy. `SIZEOF` and `TIMESTAMP` negate `P_NoSizeof` and `P_NoTimestamp`, respectively. Also they require that they have some ancestor asserting property `P_Datastructure`. This accurately describes the constraints we want to impose on the compositions of these four mixin layers: a `BINTREE` has to be present and if a `TIMESTAMP` or `SIZEOF` are present they must be descendants of `BINTREE` in the inheritance chain.

The method described above only makes use of access control (such as commonly found in C++ or Java and easily emulated in CLOS) and the same general language mechanisms used for mixin layers. The method's clarity could be improved using some form of syntactic sugar. In the absence of static typing (e.g., if we were to implement this technique in CLOS) the checking would have to be performed at run-time by calling an appropriate method. We have developed other constraint techniques for C++ but they are language-specific (or even compiler-specific as is the case with many compile-time techniques that rely on constant-folding).

There are more important restrictions of the technique we presented, however. First, even though it is easy to have a layer express requirements for other layers *above* it in the inheritance hierarchy, it is quite hard to do the same for layers *below* it. Such requirements can only be expressed by having a "catch-all" layer at the bottom of all compositions, to check for unsatisfied requirements. A second problem of the above technique is that even when an erroneous composition is detected, the error message may be far from informative. In essence, we express relatively deep errors (e.g., semantic incompatibilities among large scale components) through the absence of an inherited class. The compiler will still complain about an undefined type, but the cause of the error (not to mention a possible fix) is not immediately apparent. The problem is intensified in the case of mixin layers developed and used independently by different programmers. A casual user will expect much more expressive error reporting from a black-box component than our technique can offer. Reference [BG97] presents a general technique for automatically detecting (and suggesting repairs to) errors in layered implementations.

3.2 Interfaces for Mixin Layers

The parameterization examples that we have considered this far are all in the C++ language. C++ templates are an example of an *unconstrained* parameterization mechanism. That is, templates offer no way to constrain the possible values that a template parameter may assume. For instance, consider a simple mixin:

```
template <class Super> class Mixin1 : public Super {
  void foo() { Super::bar(); }
  ... /* other methods, vars */
};
```

`Super` is a parameter to the template and some restrictions on its structure are apparent from the code. For example, `Super` has to export a method `bar` in its

interface (“public” or “protected” in C++). Furthermore, `Super::bar` should have no arguments. Nevertheless, these restrictions are never made explicit by the programmer and the compiler has no knowledge of them. This approach has two main disadvantages:

- The compiler cannot easily verify that all restrictions are satisfied. Errors are discovered belatedly, hindering accurate error reporting. For instance, if the programmer parameterizes `Mixin1` with a class defining no `bar` method, the resulting parameterization error will only be detected as a call to an undefined method. In C++, in particular, the error will only be discovered when (and if) method `foo` is actually used (this is discussed in detail in Section 3.4).
- The modularity of templatized code is not preserved. Ideally we would like the compiler to process each parameterized unit of code independently. This way parameterized modules (like mixin layers) will only need to be linked together at composition time. With unconstrained parameterization this is not possible. Type-checking (e.g., checking for undefined methods as in the above example) has to occur after the composition is specified. The same is true for code generation, meaning that a piece of parameterized code has to be processed once for each composition it participates in. Thus, the lack of modularity of templatized code means that such code offers few opportunities for separate, incremental compilation. Furthermore, having incomplete type information at compile-time does not allow certain combinations of parameterized components (like a useful fixpoint construction that we discuss in Section 3.3).

To avoid the problems of unconstrained parameterization, many *constrained* parameterization mechanisms have been proposed (e.g., [AFM97, OW97, BOSW98, MBL97]). In this section we examine constrained parameterization

from the standpoint of mixin layers. The ideas presented here are general, but, for illustration purposes, we will focus on the Java language, and, in particular, on the constrained parameterization mechanism for Java proposed by Agesen et al. [AFM97]. We describe this mechanism in Section 3.2.1. In Section 3.2.2 we argue that, with regards to mixin layers, even though the principles upon which the mechanism is based are correct, it fails in practice. This failure is due to the way nested classes and interfaces are handled in Java. By demonstrating the problem, we essentially identify the properties of an object-oriented type system, powerful enough to support constraints for mixin layers. We propose extensions to the Java language to correct the problem without affecting existing Java programs. Finally, we discuss some closely related work in Section 3.2.3.

3.2.1 Constrained Parameterization

Before we introduce the parameterization mechanism of Agesen et al. [AFM97] we discuss briefly the Java "interface" mechanism.

Background: Java Interfaces. *Interfaces in Java are used to specify explicit type signatures for classes*. Consider the following example of a Java `interface` declaration:

```
interface Foo1 {
  Foo1     meth1 ();
  boolean  meth2 (Foo1 foo);
}
```

Any (concrete) Java class that conforms to this interface has to define two methods `meth1` and `meth2` with the exact type signatures (return types and argument types) specified in the interface. Conformance is declared using the `implements` keyword. For instance:

```
class Baz implements Foo1 {
  public Foo1 meth1() { return new Baz(); }
  public boolean meth2(Foo1 foo) { return true; }
}
```

The definition of `Baz` is legal because it supports both methods prescribed by the interface (i.e., it defines both methods with *identical* signatures to the interface prototypes).[4] It should be clear from this example that interface specifications are simply constraints on class definitions.

It is worth noting that the Java type system (of which interfaces are a part) forms an incomplete constraint language. Even though some properties are easy to express (for instance, "class `A` should support a method `foo` that takes no argument and returns an object of class `B`") others are not expressible (for instance, "class `A` should support a method named `foo`"). That is, interfaces only support complete type signatures for methods. This is only one of the restrictions of the mechanism. Such restrictions are usually imposed because of technical limitations (e.g., simplified parsing) and lack of significant need in everyday programming.

A Constrained Parameterization Mechanism. The parameterization mechanism of Agesen et al. [AFM97] is superficially similar to C++ class templates but allows parameterizations to be constrained using interface specifications or subtype relations. That is, parameterized classes have the general form:

```
class SomeClass< parameters > { ... }
```

where parameters is a list of type variables (representing classes, interfaces, or primitive Java types) which may be constrained in either of two forms:

4. Java is *non-variant* with respect to method signatures in superclasses and interfaces (i.e., method signatures have to be identical). The language is *co-variant* with respect to arrays: an array of subclass instances can be used in place of an array of superclass instances [Tho97].

- `P implements I`: Parameter `P` has to be either a class which implements interface `I` or an interface which has `I` as a super-interface.
- `P extends B`: Parameter `P` has to be a subclass of `B`.

(Interestingly, `I` and `B`, above, can be expressions containing the parameter `P`, thus allowing for powerful fixpoint constructions, like "`P implements Countable<P>`". This is useful because concrete interfaces are not sufficient for describing generic behavior. Thus, interface templates are needed and these can be specialized with actual types. For a good example of this usage, see [AFM97].) Mixins with constrained arguments can easily be specified using this mechanism. As an example, consider the following interface and mixin definitions:

```
interface Foo2 {
  Foo2     meth1 ();
}

class Mix<Super implements Foo2> extends Super {
  Foo2 get_foo() { return super.meth1(); }
  ...
}
```

The `implements` clause in the mixin definition specifies that the mixin parameter (i.e., the superclass of the produced class) should conform to interface `Foo2`. The need for conformance is evident in the body of method `get_foo`: the code calls a method `meth1` in the mixin's superclass.[5]

5. In this case, the dependency could be inferred from the code. That is, by analogy to many other forms of polymorphism in programming languages, the mixin could be considered a polymorphic entity that can be parameterized by any class specifying a method `meth1` with a compatible type signature. We will discuss polymorphism and type inference in more detail in Section 3.2.3.

3.2.2 Interfaces for Nested Classes

Nested classes are a powerful mechanism for integrating some of the benefits of block-structured programming in object-oriented programming languages. Nested classes in Java [Jav97b] behave in many respects like other class members (methods and member variables): they are inherited by subclasses, they have the same access control specifiers (e.g., `public`, `private`), and the outer class acts as a namespace for scoping purposes.

Using a nesting pattern, similar to that of C++ mixin layers, combined with the parameterization mechanism of Section 3.2.1, we can express the general form of mixin layers as:

```
class LayerThis<LayerSuper> extends LayerSuper {
  public class FirstInner extends LayerSuper.FirstInner
  { ... }
  public class SecondInner extends LayerSuper.SecondInner
  { ... }
  public class ThirdInner  extends LayerSuper.ThirdInner
  { ... }
}
```

Ideally, we should be able to constrain the parameterization so that the superclass (`LayerSuper`) always contain three nested classes `FirstInner`, `SecondInner`, and `ThirdInner`. Unfortunately this constraint (as well as many others that have to do with class nesting) is not expressible using Java interfaces, as we discuss below.

Problems with Interfaces and Nested Classes. We mentioned previously that the Java type system has some restrictions with respect to the properties expressible in it. One of the restrictions has to do with expressing properties for nested classes. More specifically, there is no way to constrain a class with respect to the nested

classes that it must contain. One may think that nesting interfaces will achieve the desired result. For instance, consider the interface declaration:

```
interface ThreeInners {
  interface FirstInner  { ... }
  interface SecondInner { ... }
  interface ThirdInner  { ... }
}
```

It may seem that a class that implements interface `ThreeInners` has to contain nested classes implementing `ThreeInners.FirstInner`, `ThreeInners.SecondInner`, and `ThreeInners.ThirdInner`.[6] This is *not*, however, the case in Java. Interface nesting only has namespace significance and does not imply any constraints for the class implementing the outer interface! We will first define precisely the general form of constraints that the type system needs to be able to express to support nested classes. Then we will present an extension to Java that supports these constraints without changing the semantics of existing programs.

General Form of Constraints for Nested Classes. We are trying to ensure that the `extends` and `implements` clauses have straightforward extensions for the case of nested classes. This would be valuable, for instance, in type-checking mixin layers compositions (but also in other occasions as we will see in Section 3.2.3). There are two general forms of constraints that we would like to be able to express:

6. For instance, Bruce, Odersky, and Wadler [BOW98] presented an example with nested interfaces which required functionality similar to what we suggest. They write: “The intention is that any implementation of the ‘outer’ interface [...] must provide implementations of the ‘inner’ interfaces.” They did not, however, recognize that this intention is not supported by Java.

- (*deep subclassing*) the constrained class `C` is a *deep subclass* of another class `B`. That is, `C` is a subclass of `B` and for every publicly accessible nested class `B.N`, there is a publicly accessible class `C.N` that is a deep subclass of `B.N`.
- (*deep interface conformance*) the constrained class `C` *conforms deeply* to interface `I`. That is, `C` conforms to `I` and for each publicly accessible nested interface `I.N`, there is a publicly accessible class `C.N` that conforms deeply to `I.N`.

Note that both definitions are recursive and can be applied to class nesting of arbitrary depth. (For instance, we could specify the property "class `A` should contain nested class `B` which contains nested classes `C` and `D` conforming to interfaces `I` and `J`, respectively".)

Expressing Constraints. Obviously, a programming language can use deep subclassing and deep interface conformance as the only kinds of subclassing and interface conformance. That is, a language may enforce that every subclass is a deep subclass, and every class conforming to an interface conforms deeply to it. Although this is a reasonable design choice, in the case of Java it necessitates changing the meaning of existing programs.

A second alternative is to maintain both regular subclassing and deep subclassing (and similarly for interface conformance). In Java, this could be supported by adding new syntax to the language so that the two cases are differentiated (i.e., it becomes clear which of the nested classes or interfaces are intended for use under deep subtyping or deep interface conformance). Next, we will describe informally a small set of changes to the Java syntax (as well as the corresponding extensions to the semantics) to support deep interface conformance without changing the semantics of programs that use standard Java interface conformance. The same ideas apply to integrating deep subtyping in Java.

Backwards-Compatible Deep Interface Conformance for Java. The constraints that we are interested in expressing could be addressed by allowing *class prototypes* to be nested inside interfaces. By "class prototype" we mean a class declaration with no class body.[7] This is analogous to the current Java syntax for function prototypes in interfaces. The semantics for this extension is straightforward: we specify that *a publicly accessible class prototype nested inside an interface declaration means that classes conforming to the interface should have a publicly available nested class conforming to the prototype*. Consider the following example:

```
interface DS {
  interface IfElement {
    void      set (IfElement element);
    IfElement get ();
  }
  interface IfContainer {
    void      insert(IfElement element);
    boolean   find(IfElement element);
  }

  class Element   implements IfElement;  // Syntax extension
  class Container implements IfContainer;// Syntax extension
}
```

This example describes a simplified (partial) interface for a component encapsulating classes that provide basic data structure functionality. For a class to implement the `DS` interface (under our extension) it has to contain two publicly visible nested classes called `Element`, and `Container` with each of them conform-

7. The current syntax for class declarations is `ClassDeclaration: ClassModifiers(opt) class Identifier Super(opt) Interfaces(opt) ClassBody`. Our proposed syntax for class prototypes is `PrototypeDeclaration: ClassModifiers(opt) class Identifier Interfaces(opt)` with the restriction that prototypes can only appear nested inside an interface.

ing to the above interfaces (`IfElement`, `IfContainer`). For instance, consider a class `BinaryTree` that implements this interface:

```
class BinaryTree implements DS {
  public class Element implements DS.IfElement {
    public void set (DS.IfElement element)
       { ... } // implementations omitted
    public DS.IfElement get ()
       { ... }
  }
  public class Container implements DS.IfContainer {
    public void insert(DS.IfElement element)
       { ... }
    public boolean find(DS.IfElement element)
       { ... }
  }
}
```

The "`implements DS`" clause in the class declaration makes the class conform to the `DS` interface. This entails the presence of two nested classes implementing the corresponding interfaces. It is worth noting that a class prototype may be declared to implement more than one interface but there is no notion of inheritance among prototypes (i.e., a prototype declaration has no `extends` clause).

Note that the proposed scheme does not change the meaning of existing interface nesting (thus, no existing Java programs are affected by the changes). Also, no new keywords are required in the language.

Applications of Deep Interface Conformance. Java classes are second-class entities: they cannot be assigned to variables or passed as arguments to functions but there are language mechanisms that manipulate classes (most notably, inheritance). Type systems exhibit their benefits mainly in the presence of variability (for instance, arguments of functions are unknown but have a specific type). Hence, one would expect that a more powerful type system (i.e., one supporting deep

interface conformance) will be useful in the case of *class functors*: functions with class arguments and/or producing new classes. Indeed deep interface conformance is invaluable in the case of mixin layers—mixins are the most common kind of class functors in object-oriented languages.

For a demonstration, we will re-use our example of Section 3.1, expressed in Java. In this example, four data structure layers are defined, containing refinements for the `Container` and `Element` classes. We would like to define generalized interfaces for allocators and data structures (e.g., binary trees, hash tables, lists). Interface conformance will serve as a static check of the interchangeability of the corresponding mixin layers. This could be effected with the following interface declarations (note that the first is reproduced from our previous example):

```
interface DS {
  interface IfElement {
    void      set (IfElement element);
    IfElement get ();
  }
  interface IfContainer {
    void      insert(IfElement element);
    boolean   find(IfElement element);
  }

  class Element   implements IfElement;
  class Container implements IfContainer;
}


interface ALLOC {
  interface IfElement { }
  interface IfContainer {
    IfElement    alloc_node();
  }
  class Element   implements IfElement;
  class Container implements IfContainer;
}
```

Consider now an example mixin layer defining a binary tree. We would like to constrain the mixin parameter so that its valid values are classes supporting the `ALLOC` interface.

```
class BinaryTree<Alloc implements ALLOC>
implements DS extends Alloc
{
  class Element implements DS.IfElement extends
Alloc.Element {
    public void set (DS.IfElement element)
     { ... } // Implementation omitted
    public DS.IfElement get () { ... }
  }
  class Container implements DS.IfContainer
  extends Alloc.IfContainer
  {
    public void insert(DS.IfElement element)
     { ... }
    public boolean find(DS.IfElement element)
     { ... }
  }
}                                                    (3.2)
```

The constraint on the mixin parameter ("`Alloc implements ALLOC`") prevents the `BinaryTree` layer from being instantiated with classes that will result in invalid compositions.

3.2.3 Discussion/Related Work

The work presented in this section has a few direct connections to other work that are worth mentioning:

Deep subtyping was introduced by Wadler, Odersky, and this author [WOS98] in a slightly different context (that of the GJ language). The mechanism is of general utility in Java, however, and complements the proposal of deep inter-

face conformance to form a type system that fully supports constraints for nested classes.

A class containing nested classes can be considered a large scale component. Extending the interface functionality to include such classes is a significant step in increasing the granularity of reusable software entities. Type signatures for multiple-class components have been studied before. The GenVoca model of software construction (see Chapter 5) defines the idea of a component's *realm*. A realm identifies the set of all components that are interchangeable in a parameterization, exactly like our extended interfaces. In fact, the concept of a realm is reified as a programming language construct in the P++ language [Sin96].

As we mentioned before, interfaces can be viewed as explicit types for classes. From a programming language standpoint it makes sense to ask whether the type of a class can be inferred from its definition. This is (to an extent) true in all the examples we discussed. Consider, for instance, the code fragment (3.2) presented previously. Both requirements on the mixin parameter can be inferred from the mixin layer definition:

- The `Element` nested class has a superclass called `Element`, nested inside the mixin parameter (`Alloc.Element`)
- The `Container` nested class has a superclass called `Container`, nested inside the mixin parameter (`Alloc.Container`). The additional requirement that this superclass provide a method `alloc_node` is expected to be deducible from the definition of the `insert` method (omitted in (3.2)).

In other words, instead of using explicit constraints on classes, we could consider them polymorphic: they assume the most general type permitted by their definitions. Nested classes are no different from other members of the class when it comes to type inference. Even though we have not explored the possibilities, this

approach could result in a type inference technique that can handle the typing requirements of nested classes and mixins.

3.3 Communicating Static Information in a Composition

Mixin layers are source code components that are defined in isolation but used in conjunction with one another. Often a mixin layer needs to acquire static information (e.g., types) about other layers or the entire composition it participates in. This may be hard when the information needs to move upwards in the inheritance hierarchy (i.e., from a more refined to a more general class). This section presents techniques for propagating such information and discusses some language/type system issues that arise.

3.3.1 Introduction: Virtual Types

An interesting issue arises in various layered implementations that use inheritance together with static typing. This is essentially a symmetric problem to the one that originally motivated mixins. Recall that mixins were introduced to remove the restriction that the definition of a subclass in an inheritance relation needs to reference its superclass. This restriction, however, means that superclasses are generally known when a subclass is defined (and references to them may exist in subclass code) while the converse is not true. This is not a problem when a superclass only needs to transfer control to a subclass (i.e., when a superclass needs to call a subclass method). The usual dynamic binding (or *late binding*) of methods—the hallmark of object-oriented programming—deals with exactly this. When, however, superclass code depends on type information that is specific to the current sub-

class, the problem is harder—type sub-languages usually do not have late binding capabilities.

Recall the ALLOC layer from our data structure example (in C++). ALLOC is the root of the inheritance hierarchy for all compositions of mixin layers in Section 3.1.1. One of the compositions we examined is replicated here:

```
typedef SIZEOF < TIMESTAMP < BINTREE < ALLOC < int > > > >
  Tree1;                                                    (3.3)
```

The node_alloc method in the Container nested class of ALLOC is responsible for allocating storage for a data structure element. One would think that the implementation of this method would be as simple as:

```
{ return new Node; }
```

Unfortunately, this is not true. The actual allocated object should not be of type Node, as defined in the ALLOC layer (that is, ALLOC<int>::Node in (3.3)). Instead it should be of class Node as defined in the *most refined* layer (i.e., the final subclass in the hierarchy—Tree1::Node in (3.3)). In this way, the allocated node will have enough room for the stored data as well as fields added by every one of the mixin layers of composition Tree1 (e.g., the parent_link, left_link, and right_link pointers added by BINTREE). We can circumvent this problem by weakening our type constraints and obtaining the necessary information at run-time through dynamic binding. In this particular example we need to set the return value of the node_alloc method to a universal pointer type (void*) and get the size of the allocated node through a C++ virtual call (not shown). This solution is general but inconvenient, error-prone (type information is lost), and possibly inefficient (depending on the overhead of dynamic binding).

A complete and elegant solution to the problem is offered by *virtual types* language mechanisms. Virtual types can be refined by subclasses in an inheritance

chain and the most refined version is the one used by superclass code. In our data structure example, by declaring `Node` as a virtual type we express precisely our intention. Any references to `Node` (for instance, in "`new Node`") are taken relative to the most refined class in the inheritance chain (`Tree1::Node` in (3.3)).

Virtual types first appeared as *virtual class patterns* in the Beta programming language (see [MMN93], ch.9). Recently they have been employed in a variety of programming language mechanisms implementing parameterization and layered frameworks similar to mixin layers. The work of [Tho97], proposes an approach for genericity in Java using virtual types. We recognize the "`assumes inner`" primitive of feature-oriented programming [Pre97] as a virtual type declaration specifier. The `forward` construct in the P++ language [Sin96] serves exactly the same purpose, declaring that a certain type will be refined by subsequent layers in a composition. Our language extensions to Java that add support for mixin layers [BLS98] include virtual types.

3.3.2 Emulating Virtual Types through Parameterization

Virtual typing is often viewed as an alternative to explicit parameterization of generic code templates. Thus, virtual types have often been compared to explicit parameterization mechanisms in terms of expressibility. Bruce, Odersky, and Wadler [BOW98] offered a discussion of the relative advantages of the two approaches. Later, Wadler, Odersky, and this author [WOS98] gave a more elegant method for emulating virtual types through explicit parametric types. That solution was presented in the context of Generic Java (GJ [BOSW98]), an extension of Java with parametric polymorphism, based on a homogeneous model of transformation (and, thus, not supporting mixins). Here we will discuss the same idea from the perspective of mixins. This technique is far from specific to mixins, however (e.g.,

it has been used by Czarnecki and Eisenecker [CE99a-b] in their C++ meta-programming methodology).

Initially we present a way to propagate type information through parameterization in C++. The main idea enabling the propagation of type information from subclasses to superclasses is to parameterize the superclass with the entire class hierarchy. For instance, consider a composition of three mixins, `Mixin1` to `Mixin3`. If `Mixin1` expects a type parameter describing the entire composition, then the composition could be expressed as:

```
class Total: public Mixin3 < Mixin2 < Mixin1 < Total > > >
{ /* empty body */ };                                        (3.4)
```

Note how the result of the composition (`Total`) is used as the parameter for the inner-most mixin layer. This recursive declaration of class `Total` corresponds to a fixpoint construction and allows the superclass to obtain static knowledge of the type of the subclass. For instance, `Mixin1` could have the form:

```
template < class Param > class Mixin1 {
public:
  Param *allocate() { return new Param; }
  ...
};
```

This way the right kind of object gets allocated, and all three mixins may have contributed data members to this object.

The same technique can be used to statically dispatch to methods defined in subclasses. Note that this is *different* from dynamic binding: even though the method invoked is defined in a subclass, the method is uniquely determined at compile-time. For example, consider the composition of code fragment (3.4), above, with `Mixin1` containing code that invokes a method defined in one of its

subclasses (i.e., the method will be part either of `Mixin2` or `Mixin3`). This could be effected by defining `Mixin1` through an idiom like:

```
template < class Param > class Mixin1 {
public:
  void invoke_below() { ((Param *)this)->Param::method(); }
  // "method" is defined in Mixin2 or Mixin3
  ...
};
```

3.3.3 Limitations and the Value of Constraints

The fixpoint technique presented above offers an interesting way to pass type information from a subclass to a superclass. Even though the same idea works in multiple environments (e.g., in C++ as well as GJ) it has a few limitations, especially in the context of unconstrained parameterization. These limitations become apparent when we try to apply this idea to C++ mixin layers.

Consider again the binary tree data structure defined in Section 3.1.1 through the composition of the `SIZEOF`, `TIMESTAMP`, `BINTREE`, and `ALLOC` layers. As we pointed out in Section 3.3.1, the `ALLOC` layer needs to have static knowledge of the type of node that is to be allocated. One might think that it is sufficient to parameterize `ALLOC` by the result of the entire composition, just like in our previous examples with simple mixins:

```
class Tree:
  public SIZEOF <TIMESTAMP <BINTREE <ALLOC <int, Tree> > > >
{ /* empty body */ };                                        (3.5)
```

Nevertheless, this composition is not valid in C++ (in contrast to the simpler compositions presented in Section 3.3.2, which are perfectly valid). To see the problem, consider how the `ALLOC` layer might be defined:

```
template < class Element, class Param > class ALLOC {
public:
  ...
  class Container {
  protected:
    Param::Node *allocate() { return new Param::Node; }
    // error! “Param::Node” is not a legal type
  ...
  };
};
```

The reason for the problem is that the type parameter `Param` represents an incomplete type and its member types (e.g., `Param::Node`) cannot be accessed.[8] The problem occurs when the compiler attempts to instantiate the `ALLOC` template with a recursive reference to the entire composition, as in code fragment (3.5).

Fully supporting the above idiom is much easier if a constrained parameterization mechanism (e.g., see Section 3.2.1) is adopted. In the previous example, if the type signature of the expected type parameter `Param` was known, the technique would work correctly, since `Param` would be guaranteed to have a member class called `Node`. This is another instance of the separate compilation capabilities for parameterized code afforded by constrained parameterization. With a constrained parameterization mechanism, knowledge regarding type parameters becomes explicit and the compiler can handle advanced uses of type parameters independently of their values (i.e., regardless of the actual parameter instantiations).

3.4 Mixins and C++ Idiosyncrasies

Up to this point, most of our mixin layers examples have been in C++. This is hardly surprising since C++ is the most widespread object-oriented language and

8. It may be possible to inform the compiler that the member `Param::Node` is a type, by using the `typename` keyword (e.g., see the example in [CE99b], p.27). Nevertheless, few C++ compilers support this idiom for nested classes and it is not clear if it is required by the C++ standard.

mixin layers can be expressed directly in it. This section discusses some pragmatic issues pertaining to the use of mixins (mixin classes and mixin layers alike) in C++. Most of the points raised below concern fine interactions between the mixin approach and C++ idiosyncrasies. Others are implementation suggestions. They are all useful knowledge before one embarks on a development effort using C++ mixins. Additionally, our observations could serve to guide design choices for future parameterization mechanisms in programming languages.

Lack of template type-checking. Templates do not correspond to types in the C++ language. Thus, they are not type-checked until instantiation time (that is, composition time for mixins). Furthermore, methods of templatized classes are themselves considered function templates (see [Str97], p.330). Function templates in C++ are instantiated automatically and only when needed. Thus, even after mixins are composed, not all their methods will be type-checked (code will only be produced for methods actually referenced in the object code). This means that certain errors (including type mismatches and references to undeclared methods) can only be detected with the right template instantiations and method calls. Consider the following example:

```
template <class Super> class ErrorMixin : public Super {
public:
  ...
  void sort(FOO foo) {
    Super::srot(foo);   // misspelled
  }
};
```

If client code never calls method `sort`, the compiler will *not* catch the misspelled identifier above. This is true even if the `ErrorMixin` template is used to create classes, and methods other than `sort` are invoked on objects of those classes. It is, therefore, a good idea to develop a library with mixin components

simultaneously with a large set of regression tests that will exercise most of the library functionality. This is anyway a good engineering practice for detecting run-time errors.

When "subtype of" does not mean "substitutable for". There are two instances where inheritance may not behave the way one would expect in C++. First, constructor methods are not inherited. Ellis and Stroustrup ([ES90], p.264) present valid reasons for this design choice: the constructor of a superclass does not suffice for initializing data members added by a subclass. Often, however, a mixin class may be used only to enrich or adapt the method interface of its superclasses *without* adding data members (e.g., consider our adaptor layers in Section 2.4.1). In this case it would be quite reasonable to inherit a constructor, which, unfortunately, is not possible. The practical consequence of this policy is that the only constructors that are visible in the result of a mixin composition are the ones present in the outer-most mixin (bottom-most class in the resulting inheritance hierarchy). To make matters worse, constructor initialization lists (e.g.,

```
constr() : init1(1,2), init2(3) {}  )
```

can only be used to initialize direct parent classes. In other words, all classes need to know the interface for the constructor of their direct superclass (if they are to use constructor initialization lists). This is a problem with mixins since a single mixin class can be used with several distinct superclasses. In this case, one can use a standardized construction interface. A way to do this is by creating a construction class encoding the union of all possible arguments to constructors in a hierarchy. Destructors for base classes, on the other hand, are called automatically so they should not be replicated.

The second instance where subtypes are not substitutable in C++ occurs with top-level function templates. Assume a function template of the form:

```
template <class Next> void weird_function ( Mixin<Next> arg)
  { ... }
```

This function template will be instantiated correctly when called with an argument of type `Mixin<Base>`, but not when called with an argument of type `NewMixin<Mixin<Base> >`. Even though the latter type is a subtype of the former, subtyping is not involved in the function template instantiation policy of C++. The problem is solved only by ensuring that the template gets instantiated with an argument of type `Mixin<Base>` (e.g., there is an explicit call to `weird_function` with an argument of this type). Once this is done, the function generated by the template can be invoked with actual arguments that are subtypes of the corresponding formal argument types.

Synonyms for compositions. In the past sections we have used two different idioms to introduce synonyms for complicated mixin compositions. The first was based on `typedef` declarations—e.g.,

```
typedef A < B < C > > Synonym;
```

The second idiom introduces an empty subclass:

```
class Synonym : public A < B < C > > { };
```

The first form has the advantage of preserving constructors of component `A` in the synonym. The second idiom is cleanly integrated into the language (e.g., it can be templatized, compilers create short link names for the synonym, it can support the fixpoint construction of Section 3.3.2, etc.).

Designating virtual methods. Sometimes C++ policies have pleasant side-effects when used in conjunction with mixins. An interesting case is that of a mixin used to create classes where a certain method can be virtual (i.e., dynamically bound) or not, depending on the concrete class used to instantiate the mixin. This is due to

the C++ policy of letting a superclass declare whether a method is virtual, while the subclass does not need to specify this explicitly. Consider a regular mixin and two concrete classes instantiating it:

```
template <class Super> class MixinA : public Super {
public:
  void virtual_or_not(FOO foo) { ... }
};

class Base1 {
public:
  virtual void virtual_or_not(FOO foo) {...}
  ... // methods using “virtual_or_not”
};

class Base2 {
public:
  void virtual_or_not(FOO foo) {...}
};
```

The composition `MixinA<Base1>` designates a class in which the method `virtual_or_not` is virtual. Conversely, the same method is not virtual in the composition `MixinA<Base2>`. Hence, calls to `virtual_or_not` in `Base1` will call the method supplied by the mixin in the former case but not in the latter.

In the general case, this phenomenon allows for interesting mixin configurations. Classes at an intermediate layer may specify methods and let the innermost layer decide whether they are virtual or not.

Single mixin for multiple uses. The lack of template type-checking in C++ can actually be beneficial in some cases. Consider two classes `Base1` and `Base2` with very similar interfaces (except for a few methods):

```
class Base1 {
public:
  void regular() {...}
```

```
  ...
};
class Base2 {
public:
  void weird() {...}
  ... // otherwise same interface as Base1
};
```

Because of the similarities between `Base1` and `Base2`, it makes sense to use a single mixin to adapt both. Such a mixin may need to have methods calling either of the methods specific to one of the two base classes. This is perfectly feasible. A mixin can be specified so that it calls either `regular` or `weird`:

```
template <class Super> class Mixin : public Super {
  ...
public:
  void meth1() { Super::regular(); }
  void meth2() { Super::weird(); }
};
```

This is a correct definition and it will do the right thing for both composition `Mixin<Base1>` and `Mixin<Base2>`! What is remarkable is that part of `Mixin` seems invalid (calls an undefined method), no matter which composition we decide to perform. But, since methods of class templates are treated as function templates, no error will be signalled unless the program actually uses the wrong method (which may be `meth1` or `meth2` depending on the composition). That is, an error will be signalled only if the program is indeed wrong. We have used this technique to provide uniform extensions to data structures supporting slightly different interfaces (in particular, the red-black tree and hash table of the SGI implementation of the Standard Template Library [SGIWeb]).

Hygienic templates in the C++ standard. The (newly adopted) C++ standard imposes several rules for name resolution of identifiers that occur inside templates.

Even though we are not aware of any compiler that implements these rules, it is useful to have them in mind for future compatibility reasons. Realistically, we do not expect that the template name resolution strategy described in the language standard will be commonplace in actual compilers for a few years. (Changes to the entire model of handling templates seem to be required.)

According to the C++ standard, templates have a *hygienic* character: they cannot contain code that refers to "nonlocal" variables or methods. Intuitively, "nonlocal" denotes variables or methods that do not depend on a template parameter and are not in scope at the global point closest to the template definition (the actual rules are quite complicated—e.g., see [Str97], C.13.8). This rule prevents template instantiations from capturing arbitrary names from their instantiation context, which could lead to behavior not predicted by the template author.[9]

To see how such rules impact mixin-based programming, consider the example of a mixin, calling a method defined in its parameter (i.e., the superclass of the class it will create when instantiated):

```
class Base {
public:
  void meth1() { ... }
};

template <class Super> class Mixin : public Super {
public:
  void wrong()   { meth1(); }
  void correct() { Super::meth1(); }
};

void client() {
  Mixin < Base > test;
```

9. This is a well-known problem in programming language research, first identified by work in hygienic macros [KFFD86].

```
  test.wrong(); // currently works but shouldn't, according to the C++ standard
  test.correct();
}
```

Note what is happening in this example: class `Mixin<Base>` inherits method `meth1` from its superclass, `Base`. When template `Mixin` is compiled, the declaration of method `meth1` is nonlocal, hence it cannot be accessed from code in the template body. None of the several compilers we tried was able to detect this error. Nevertheless, qualifying names explicitly (as shown in method `correct`) is a good practice for future compatibility. Note that the naming resolution rules for templates found in the C++ standard have implications on the way template-based programs should be developed. In particular, changing a *correct* class definition into a class template definition (by turning one of the types used into a template parameter) is *not* guaranteed to work any more. As can be seen from the previous example, errors may be quite insidious in the case of mixins: there is no way to quickly tell that an unqualified method name depends on a template parameter and, thus, should be qualified when a regular class is turned into a class template.

As a side observation extending beyond C++, note how this problem does not occur in the case of constrained parameterization (i.e., a language mechanism like that described in Section 3.2). The interface of the superclass is then statically known and the hygienic approach is enforced by default. In this case, turning a correct concrete class into a correct parameterized class is guaranteed to work with no changes to the code for its methods.

Compiler support. Not all compilers have good support for parameterized inheritance (the technique we used for mixins) and nested classes. Although many compilers were virtually trouble-free in our experiments, we have encountered others that will either not accept these language constructs or will require re-coding to get

around some of their peculiarities. Because of the transient nature of this information, we do not include it here, but the interested reader can find it in [SB98c].

There are several important compiler dependencies that are not particular to mixin-based programming but concern all template-based C++ programs. These include limitations on the debugging support, error checking, etc. We will not discuss such issues as they have been presented before (e.g., [CE99a]). Note, however, that mixin-based programming is not more complex than regular template instantiation. The compiler support issues involved in mixin-based programming are about the same as those arising in implementing the C++ Standard Template Library (STL) [SL95].

Chapter 4

An Application: the Jakarta Tool Suite

In the previous chapters, we proposed mixin layers as a modularization technique for object-oriented programs, showed the advantages of mixin layers for implementing collaboration-based designs, and discussed pragmatic issues related to language support for mixin layers. In this chapter, we discuss an application of mixin layers to an actual medium-size software project. The project is the *Jakarta Tool Suite (JTS)* [BLS98]—a set of language extensibility tools, aimed mainly at the Java language. We use mixin layers as the building blocks that form different versions of the *Jak* tool of JTS. Jak is the actual modular compiler in JTS. Different versions of Jak can be created using different combinations of layers. Layers may be responsible for type-checking, compiling, and/or creating code for a different set of language constructs. Additionally, layers may be used to add new functionality across a large group of existing classes. In this way, the user can design a language by putting together conceptual language "modules" (i.e., consistent sets of language constructs) and implement a compiler for this language as a version of Jak composed of the mixin layers corresponding to each language module. Currently available layers support the base Java language, meta-programming extensions, general purpose extensions (e.g., syntax macros for Java), a domain-specific language for data structure programming (P3), etc.

The choice of the compiler domain as a large-scale test case for mixin layers is not arbitrary. Compilers are well-understood, with modern compiler construction benefitting from years of formal development and stylized design patterns. The domain of compilers has been used several times in the past in order to demonstrate modularization mechanisms. Selectively, we mention the *visitor* design pattern [GHJV94], which is commonly described using the example of a compiler with a class corresponding to each syntactic type that its parser can recognize (e.g., there is a class for if-statements, a class for declarations, etc.). In this case, the visitor pattern can be used to add new functionality to all classes, without distributing this functionality across the classes. Our application of mixin layers to the compilers domain has very much the same modularization flavor. We use mixin layers to isolate aspects of the compiler implementation, which can be added and removed at will. Compared to the visitor pattern, mixin layers offer greater capabilities—for instance, allowing the addition of state (i.e., member variables) to existing classes.

Overall, the outcome of applying mixin layers to JTS was very successful. The flexibility afforded by the layered design is essential in forming compilers for different languages. Additionally, mixin layers helped with the internal organization of the code, so that changes were easily localized. Additions that could be conceptually grouped together (like those reflecting the language changes from Java 1.0 to Java 1.1) were introduced as new mixin layers, without disrupting the existing design. Due to mixin layers, JTS was easier to implement and has become easier to maintain.

This chapter discusses JTS and the use of mixin layers in its implementation. Section 4.1 offers some essential background in JTS by describing the way parsers are generated and initial class hierarchies are established based on language syntax. Section 4.2 discusses the actual application of mixin layers in JTS.

4.1 JTS Background: Bali as a Parser Generator[1]

Bali is the JTS tool responsible for putting together compilers. Although Bali is a component-based tool, in this section we will limit our attention to the more conventional grammar-specification aspects of Bali.

With respect to grammar specifications, Bali looks similar to other tools: the syntax of a language is specified using an annotated BNF grammar, extended with regular-expression repetitions. Bali transforms a Bali grammar into a lexical analyzer and parser. For example, two Bali productions are shown below: one defines `StatementList` as a sequence of one or more `Statements`, and the other defines `ArgumentList` as a sequence of one or more `Arguments` separated by commas.

```
StatementList : ( Statement )+ ;
ArgumentList : Argument ( ‘,’ Argument )*;
```

Repetitions have been used before in the literature [Wil93, Rea90]. They simplify grammar specifications and allow an efficient internal representation as a list of trees.

Bali productions are annotated by the class of objects that is to be instantiated when the production is recognized. For example, consider the Bali specification of the Jak `SelectStmt` rule:

```
SelectStmt
  : IF ‘(’ Expression ‘)’ Statement   ::IfStm
  | SWITCH ‘(’ Expression ‘)’ Block   ::SwStm
  ;
```

1. Parts of this section and Section 4.2 are taken from reference [BLS98] (© 1998 IEEE).

```
          // Lexeme definitions
"print"   PRINT
"+"       PLUS
"-"       MINUS
"("       LPAREN
")"       RPAREN
"[0-9]*"  INTEGER

%%        // production definitions
          // start rule is Action

Action : PRINT Expr          :: Print
       ;

Expr   : Expr PLUS Expr      :: Plus
       | Expr MINUS Expr     :: Minus
       | MINUS Expr          :: UnaryMinus
       | LPAREN Expr RPAREN  :: Paren
       | INTEGER             :: Integer
       ;
```

Figure 4.1: A Bali Grammar for an Integer Calculator

When a parser recognizes an "if" statement (i.e., an `IF` token, followed by '`(`', `Expression`, '`)`', and `Statement`), an object of class `IfStm` is created. Similarly, when the pattern defining a "switch" statement (a `SWITCH` token followed by '`(`', `Expression`, '`)`', and `Block`) is recognized, an object of class `SwStm` is created. As a program is parsed, the parser instantiates the classes that annotate productions, and links these objects together to produce the syntax tree of that program.

A Bali grammar specification is a streamlined document. It is a list of the lexical patterns that define the tokens of the grammar followed by a list of annotated productions that define the grammar itself. A Bali grammar for an elementary integer calculator is shown in Figure 4.1. From the grammar specification, Bali

(a)
```
Rule1 : pattern1   :: C1
      | Rule2
      ;

Rule2 : pattern2   :: C2
      | pattern3   :: C3
      ;
```

(b)

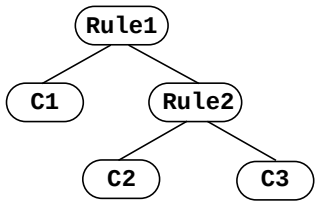

Figure 4.2: Inferring inheritance hierarchies from grammar rules

will generate a lexical analyzer and a parser (we use the `JavaCC` lexer/parser generator as a backend).

Associating grammar rules with classes allows Bali to do more than generate a parser. In particular, Bali can deduce an inheritance hierarchy of classes representing different pieces of syntax. Consider Figure 4.2(a), which shows rules `Rule1` and `Rule2`. When an instance of `Rule1` is parsed, it may be an instance of `pattern1` (an object of class `C1`), or an instance of `Rule2` (an object of class `Rule2`). Similarly, an instance of `Rule2` is either an instance of `pattern2` (an object of `C2`) or an instance of `pattern3` (an object of `C3`). From this information, the inheritance hierarchy of Figure 4.2(b) is constructed: classes `C1` and `Rule2` are subclasses of `Rule1`, and `C2` and `C3` are subclasses of `Rule2`.

Additionally, for each production Bali infers the constructors for syntax tree node classes. Each parameter of a constructor corresponds to a token or nonterminal of a pattern.[2] For example, the constructor of the `IfStm` class has the following signature:

2. The tokens need not be saved. However, Bali-produced precompilers presently save all white space—including comments—with tokens. In this way, JTS-produced tools that transform domain-specific programs will retain embedded comments. This is useful when debugging programs that have a mixture of generated and hand-written code, and is a necessary feature if transformed programs will subsequently be maintained by hand [TB95].

```
IfStm( Token iftok, Token lp, Expression exp, Token rp,
       Statement stm )
```

Methods for editing and unparsing nodes are additionally generated.

Although Bali automatically generates an inheritance hierarchy and some methods for the produced Jak compiler, there are obviously many methods that cannot be generated automatically. These including type checking, reduction, and optimization methods. Such methods are syntax-type-specific; we hand-code these methods and encapsulate them as subclasses of Bali-generated classes.

In essence, Bali takes the grammar specification and uses it to produce a skeleton for the compiler of the language. The skeleton has the form of a set of classes organized in an inheritance hierarchy, together with the methods that can be automatically produced (that is, constructors, editing, and unparsing methods). In other words, Bali produces an *application framework* [JF88] for a compiler. As we explain in the next section, the framework itself has the form of a component that occupies the root of a mixin layer composition—i.e., a class with many nested classes that will be subsequently refined. The refinements determine the semantics of each syntax type and are expressed as mixin layers.

4.2 Bali Components and Mixin Layers in JTS

Apart from its parser generator aspect, Bali is also a tool that synthesizes language implementations from components. Bali can create compilers for a family of languages, depending on the selection of components used as its input. We use the name *Jak* for any Bali-generated compiler. Currently available Bali components support the base Java language, meta-programming extensions (e.g., code template operators), general purpose extensions (e.g., syntax macros for Java), a domain-specific language for data structure programming (P3 [BCRW98]), and more. Compositions of these components define different variants of Jak: with/without

meta-programming constructs, with/without extensions for data structure programming, with/without CORBA IDL extensions, and so on. This a classical example of the *library scalability problem* [BSST93, Big94]: there are *n* features and often an exponential number of valid combinations (because most components are optional). It is not possible or practical to build all combinations by hand. Instead, the specific instances that are needed can be composed from components encapsulating orthogonal units of functionality.

A *Bali component* has two parts: The first is a Bali grammar file (which contains the lexical tokens and grammar rules that define the syntax of the host language or language extension). The second is a mixin layer encapsulating a collection of multiple hand-coded classes that contain the reduction, type-checking, etc. methods for each syntax type defined in that grammar file.

To illustrate how classes are defined and refined in Bali, consider four concrete Bali components: `Java` is a component implementing the base Java language, `SST` implements code template operators like tree constructors and explicit escapes[3], `GScope` supplies scoping support for program generation, and `P3` implements a language for data structures. The Jak language and compiler can be defined by a composition of these components. We use the `[...]` operator to designate component composition—for instance, `P3[GScope[SST[Java]]]`.

The syntax of a composed language is defined by taking the union of the sets of production rules in each Bali component grammar. The semantics of a composition is defined by composing the corresponding mixin layers. Figure 4.3 depicts the class hierarchy of the Jak compiler. `AstNode` belongs to the JTS ker-

3. Our code template operators are analogous to the backquote/unquote pair of Lisp operators. Unlike Lisp, however, multiple operators exist in JTS—one for each syntactic type (e.g., declaration, expression, etc.). Multiple constructors in syntactically rich languages are common (e.g., [WC93], [Chi96]). The main reason has to do with the ease of parsing code fragments.

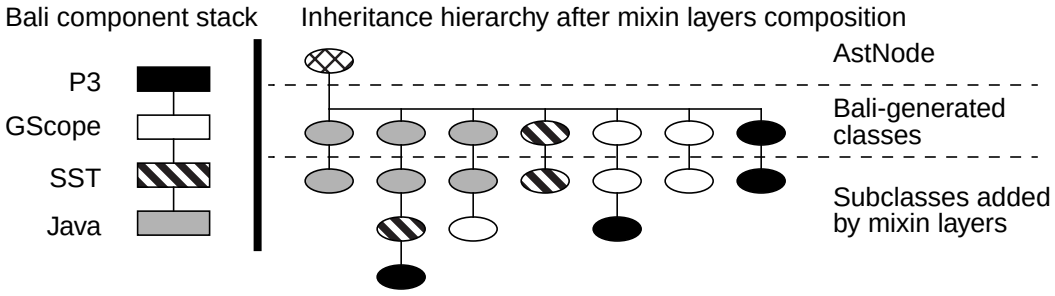

Figure 4.3: The Jak Inheritance Hierarchy

nel, and is the root of all inheritance hierarchies that Bali generates. Using the composition grammar file (the union of the grammar files for the `Java`, `SST`, `GScope`, and `P3` components), Bali generates a hierarchy of classes that contain tree node constructors, unparsing, and editing methods. Each mixin layer then grafts onto this hierarchy its hand-coded classes. These define the reduction, optimization, and type-checking methods of tree nodes by refining existing classes. *The terminal classes of this hierarchy are those that are instantiated by the generated compiler.*

It is worth noting that Figure 4.3 is not drawn to scale. Jak consists of over 500 classes. The number of classes that a mixin layer adds to an existing hierarchy ranges from 5 to 40. Nevertheless, the simplicity and economy of specifying Jak using component compositions is enormous: to build the Jak compiler, all that users have to provide to Bali is the equation `Jak = P3[GScope[SST[Java]]]`, and Bali does the rest. To compose all these classes by hand (as would be required by Java) would be very slow, extremely tedious, and error prone. Additionally, the scalability advantages of mixin layers can easily be obtained: when new extension mechanisms or new base languages are specified as components, a subset of them can be selected and Bali will automatically compose a compiler for the desired language variant.

4.3 Java Mixin Layers for JTS

In Chapter 2, we discussed the applicability of mixin layers in various programming languages. There we explained that Java already supports nested classes but the language currently specifies no parameterization mechanism. Furthermore, some of the proposed parameterization mechanisms for Java (e.g., Pizza [OW97] or Thorup's virtual types [Tho97]) do not support parameterized inheritance. In order to support mixin layers for Bali components in JTS, we implemented our own Java language extensions for parameterization. This section gives a brief overview of the main language constructs.

Our parameterization extensions to Java are geared towards mixin layer development (as opposed to general-purpose genericity). Our approach in designing and implementing these language constructs was motivated by pragmatic and not conceptual considerations: We needed a layer mechanism to facilitate our own development efforts—not to supply the best-designed and robust parameterization mechanism for Java. Therefore, our implementation was straightforward, adopting a heterogeneous model of transformation: for each instantiation of a mixin layer, a new Java class is created at the source code level. Thus, our approach resembles C++ template instantiation and does not take advantage of the facilities for load-time class adaptation offered by the Java Virtual Machine (see, e.g., the approach of Agesen et al. [AFM97] and the work on binary component adaptation [KH98]). Nevertheless, in our context our approach is not necessarily at a disadvantage. Mixin layers in Bali component compositions are never reused in the same application (i.e., a single Jak compiler can use at most one instance of a mixin layer). Therefore, code bloat (redundancy in generated classes) is not a problem. At the same time, our straightforward approach made for an easier implementation which contributed to the quicker development of JTS.

The implementation of our Java extensions for mixin layer support occurred concurrently with the development of JTS. In fact, an early version of JTS was used to implement the first version of our Java mixin layers. The Java mixin layers were, in turn, used to evolve and further develop JTS, resulting in a bootstrapped implementation. (Actually, this is not the only reason why JTS is based on a bootstrapped implementation. Another reason is that the meta-programming capabilities added to Java have been used in the code that implements JTS itself. The entire JTS system is compiled using a basic version of the Jak compiler, composed of only a few layers that specify the basic Java language, code template operators, syntax macros, etc.)

The syntax of our Java mixin layers is straightforward and resembles their C++ counterparts. Two new keywords are introduced: `layer` and `realm`. The `layer` keyword is analogous to `class` but defines a mixin layer (i.e., an outer class that may be parameterized with respect to its superclass). The `realm` keyword is used to specify interface conformance for mixin layers (see Section 3.2), in analogy to the Java `implements` keyword. (The reader may recall from Section 3.2 that "realm" is another name used in the literature for interfaces of multi-class components.) Finally, the `[...]` operator is used to specify layer composition. The (slightly simplified) general form of a layer definition is shown below, with the terminal symbols appearing in bold for clarity:

```
layer_definition :
  layer layer_name (param_list) realm realm_name [super]
  {
    declaration_list
  }
```

The syntax for non-terminals in the above definition is straightforward. `param_list` is a list of type parameters for the mixin layer. If the parameter list contains layers, the parameterization can be constrained by specifying the

expected realm of these layers. The optional `super` construct designates an `extends` clause (in much the same way as for regular Java classes). The contents of a mixin layer can only be Java type declarations.

The actual details of our mixin layers implementation in Java are not important, however. We consider of much greater importance the general approach that this implementation represents. What we did in JTS is a prime representative of a *domain-specific languages* approach to software construction. In the course of creating our medium-size software project (JTS is implemented in about 30K lines of code), we recognized that mixin layers would facilitate our task significantly. That is, we saw an opportunity for improving our implementation through extra language support. It then proved cost-effective to add the extra linguistic constructs that were needed (i.e., mixin layers), in the course of implementing the original project (i.e., JTS). Our language support for mixin layers is not perfect, but it fulfills its task of facilitating the implementation of JTS.

It is our belief that the domain-specific language approach to software construction is a promising way to building better software. The designer of a software application can (and should) be thinking about language constructs that can have a significant impact in the application's efficiency, maintainability, or reusability. Often such constructs can be readily identified, but they are not available in the implementation language of choice. With the advent of language extensibility tools, as well as extensible/reflective programming languages, supplying special-purpose (or *domain-specific*) language support may be the right approach in fighting software complexity. JTS itself is a tool aiming at facilitating the implementation of domain-specific languages and language extensions. The use of mixin layers in the implementation of JTS is a vivid demonstration of the same paradigm that JTS promotes.

Chapter 5

Related Work

The focus of this dissertation is on the implementation of large-scale object-oriented components. Such components give rise to a layered model of software construction: components form building blocks and entire software applications are built through component composition. Thus, our ideas are similar to many other research efforts on modular software implementations. In the previous chapters we concentrated on the concrete elements of our approach and demonstrated their *novelty.* In this chapter we concentrate on the *conceptual similarities* of our ideas to work in the literature. Hence, the discussion in this chapter is at a more abstract level than that of previous chapters and the emphasis is on positioning our work in the greater software systems literature.

There are two main axes around which this chapter's discussion revolves. First, our ideas are an outgrowth of a large body of work on the *GenVoca* model of software design and implementation. Mixin layers were originally inspired by GenVoca and are now an essential part of the GenVoca arsenal of implementation techniques. Second, modular software construction has been studied extensively (often under many different names) and there are clear connections between such work and ours. Section 5.1 discusses the GenVoca model and mixin layers within it. This provides a "local perspective" of mixin layers and the closely related ideas

that led to their development. Section 5.2 positions our work in the overall spectrum of modular software implementation.

5.1 The GenVoca Model

GenVoca is a design and implementation model for defining families of hierarchical systems as compositions of reusable components. GenVoca has been employed in the implementation of several *application generators* (that is, compilers for domain-specific programming languages). Indeed, the name GenVoca is derived from the first two GenVoca generators that were recognized as such: Genesis [Bat88, BBG+88] and Avoca [OP92]. Many other independently-designed generators in different domains exhibit the characteristics captured by GenVoca: Rosetta in data manipulation languages [Vil94, Vil97], Ficus in distributed file systems [HP94], Brale in host-at-sea buoy systems [Wei90], and ADAGE in real-time avionics software [CS93, BCGH95]. Thus, GenVoca is based on factoring out the common, domain-independent principles that underlie many different generators. These principles give rise to design techniques as well as implementation guidelines for the construction of GenVoca-based software. Mixin layers form a concrete implementation technique that follows the GenVoca implementation guidelines and is applicable to a wide subset of GenVoca designs.

The following subsections describe GenVoca in detail before discussing the connections between GenVoca and various elements of the mixin layers approach.

5.1.1 Elements of GenVoca

GenVoca is a methodology—not a programming language or a tool. Thus, it is best expressed as a collection of ideas that aim at influencing software designers and

implementors. The same ideas have formed the general themes in previous chapters of this dissertation. We summarize them briefly below:

- *Subsystems are the building blocks of generated systems*. Effective software synthesis requires that systems be constructed from combinations of *subsystems* (a.k.a., *components* or *layers*) consisting of suites of interrelated functions and/or classes. It is too unwieldy to construct large software by selecting and assembling hundreds or thousands of functions and classes from a reuse library. Thus, larger units of software encapsulation are needed.
- *GenVoca is both a design and an implementation methodology.* One of the characterizing features of GenVoca is that designs are straightforwardly mapped into implementations. That is, the modularity of GenVoca components should be preserved at the implementation level, with each design component being represented by a distinct implementation entity.
- *Components import and export standardized interfaces*. The key to software synthesis is composition. Composition is much easier when component interfaces correspond to fundamental abstractions of the target domain and these interfaces have been standardized. Standardization encourages functionally similar components to be plug-compatible and interchangeable.
- *Component interfaces are explicitly expressible at the implementation level*. In a GenVoca implementation, interfaces are explicit actual implementation entities. A GenVoca *realm* is a set of components that implement a compatible interface in different ways. That is, all the components in a realm share what is, to a first approximation, the same interface, but have different implementations. Because their interfaces are compatible, all the members of a realm are plug-compatible and interchangeable. Realms play the role of type signatures in GenVoca implementations and conformance of a component to a realm is declared explicitly.

- *Relationships between components can be complicated but effort should be made to keep them simple*. Even though GenVoca components can be parameterized in arbitrary ways, ideally components should have very little knowledge of other components' characteristics. In many cases component interdependencies collapse into a simple *virtual machine* model. This means that one component is expressed in terms of the operations supplied by another, without knowing how this functionality is implemented.

5.1.2 The GenVoca Notation

To better express component compositions, GenVoca offers a simple notation for representing components, realms, and systems. If a component imports another component's interface, it is designated as a parameter. Thus, in the GenVoca notation, a component is denoted by its name, followed by a bracketed list of the names of the realms it imports, followed by a colon and the name of the realm it exports. For example, a component `c` that imports realm interface `S` and exports realm interface `R` is expressed as `c[S] : R`.

A realm is denoted as a set of elements, where each element represents a component belonging to the realm. For example, Figure 5.1 shows three realms: `R`, `S`, and `T`.

```
R = { a, b[R], c[S] }
S = { d[T], e, f[S, T] }
T = { g }
```

Figure 5.1: Example of three realms expressed in the GenVoca notation

Realm `R` has three components: `a`, `b`, and `c`; realm `S` also has three: `d`, `e`, and `f`; and realm `T` has one: `g`. Component `b` imports realm interface `R` and component `c` imports realm `S`. Because it has two parameters, component `f` imports the

two realm interfaces S and T. In essence, this notation treats a realm as if it were a type. A component from a realm is simply a function of some type, and a component that imports an interface has a parameter of some type. So, d is an object of type S, where d has a parameter of type T.

A *type expression* (a.k.a., *equation*) is a named composition of components that form a composite system. For example, a type expression that specifies how components c, d, and g are combined to form a composite system is:

```
A = c[d[g]]
```

Note that the components' syntactic compatibility is easily checked by verifying that each parameter's imported interface matches the corresponding component's exported interface. Thus, A is syntactically valid, because c's imported and d's exported interface are both realm S, and d's imported and g's exported interface are both realm T.

Component semantic compatibility is a more complicated issue. Note that some combinations of components may be syntactically but not semantically correct. That is, each pair of components in the system imports and exports compatible interfaces, but the resulting algorithms may be invalid for some reason. To verify the semantic correctness of a system, each component must supply domain-specific information that describes the assumptions and restrictions on the use of the component (see [BG97] for details).

Consider the meaning of type expression A, above. GenVoca components are relatively sophisticated, which makes a *refinement* model appropriate for understanding component combinations. That is, when two components are interconnected, they exchange function, data type, and customization information with one another. In GenVoca, the refinements of A start at the top component c, which provides data type information to component d; d, in turn, provides its own data types to g; which then supplies implemented data types and functions back to d;

and so on. Note that the refinements start at the top component, work their way down to the bottom component, and then back up to the top component. In this way, the presence of one component can alter the behavior of any other—regardless of whether they are "above" it or "below" it in a layer hierarchy.

In addition to imported and exported realm parameters, components often take additional imported parameters called annotations. *Annotations* (a.k.a., *non-realm parameters* or *configuration parameters*) are instantiated by key field names, predicates, timestamp field names, file names, and other constants.

5.1.3 Variations in the GenVoca Design Space

As discussed in Section 5.1.1, GenVoca components map to separate entities at the implementation level. Nevertheless, the model does not specify a particular form for these entities. Thus, GenVoca components could correspond to language modules, classes, binary objects (e.g., COM or CORBA components), etc.

Generally, the spectrum of GenVoca implementations varies along two axes [Bat97]: components may be either *compositional* or *transformational*, and either *dynamic* or *static*. Compositional components define the source code that an application will execute; transformational components define code that, when executed, will generate the source code that an application will execute. The dynamic/static attribute refers to the time of component composition. When components are composed at application run-time, they are dynamic. When composed at compile-time, they are static.

The choice of how components should be implemented and when they should be combined reflects a trade-off between factors like optimization potential, implementation effort, and binary compatibility. In particular, transformational components offer more opportunities for optimization but are harder to implement than compositional components. At the same time, composing components stati-

cally eliminates the dynamic overhead of operation dispatch across components (which could be significant for fine-grained components), but dynamic composition allows reusing binary components without modification.

5.1.4 GenVoca, Mixin Layers, and Collaborations

At this point, our description of GenVoca is complete and we can see the correspondence between mixin layers, collaboration-based design, and GenVoca.

The main concepts of GenVoca design and collaboration-based design are identical. The central idea in both cases is that of a component that interrelates many object classes. Classes themselves become of secondary importance. A single class, however, has functionality that results from the combination of several components. The terminology is slightly different (for instance, GenVoca *layers* correspond to collaborations; GenVoca has no name for roles).

Under this light, mixin layers are a way to offer programming language support for implementing GenVoca designs. Since layer composition occurs at application compilation time, and layers specify executable code, mixin layers are ideally suited for implementing *static compositional GenVoca designs*. All of the elements of GenVoca are immediately identifiable in mixin layers:

- Mixin layers correspond to GenVoca components and form the building blocks of entire software applications (as discussed in Chapter 2). Mixin layers are larger units of encapsulation than single classes or functions, exactly as GenVoca prescribes.
- Mixin layers map design components (collaborations) into implementation entities. In this way, the GenVoca property that the design modularity be preserved in the implementation is guaranteed.
- Mixin layers can import and export standardized interfaces. As discussed in Section 3.2, language support for layer interfaces can be provided (e.g., as an

extension to the Java interface mechanism). Such explicit interfaces represent types for mixin layers and correspond directly to the GenVoca concept of a "realm".

- Mixin layers provide a simple model of component interaction. Each component can receive type information from other components and rely on the functionality other components provide. The propagation of types and operations in both directions (up and down) in a GenVoca composition is effected through dynamic binding (for operations) and virtual types (for type information, as discussed in Section 3.3).
- The GenVoca notation for type equations is remarkably similar to the notation used for mixin layer instantiation. For instance, a mixin layer composition of the form

  ```
  typedef Collab1 <Collab2 <Collab3 <FinalCollab> > > T ;
  ```

  is directly analogous to a GenVoca type equation

  ```
  T = Collab1 [ Collab2 [ Collab3 [ FinalCollab ] ] ]
  ```

  except for minor syntactic variations ("`[...]`" replaces "`<...>`", etc.).
- Non-realm parameters (GenVoca annotations) are directly expressible in the mixin layers framework as layer parameters that are not themselves layers. Thus, an arbitrary type could be passed as a parameter to a layer—see, for instance, the `element` parameter of the `ALLOC` layer in Section 3.1.1.

5.1.5 Mixin Layers and Dynamic GenVoca Designs

Mixin layers are a straightforward implementation technique for static compositional GenVoca designs. An interesting question, however, is whether similar ideas can be applied to transformational and/or dynamic GenVoca designs.

To answer the first part of the question, the classification of GenVoca designs into transformational and compositional is rather arbitrary and orthogonal

to the actual implementation of these designs. A transformational GenVoca design for an application entails a compositional GenVoca design for the program *generating* the application. In other words, transformational GenVoca components can be viewed as compositional components for a GenVoca generator. The components contain code that is *executed* during the generator runtime, but *transforms* (or just generates) the code of the final application. Regarding mixin layers, the above observation means that their compositional character does not prevent mixin layers from being employed in a transformational setting. Mixin layers can be building blocks for generators, just as well as for target applications.

Therefore, the interesting question is whether mixin layer principles can be applied to *dynamic* GenVoca designs, where components are composed during application run-time, when objects only exist in binary form. Clearly, in a dynamic setting there can be no language support for layer specification and composition (e.g., no type checking or scoping). Nevertheless, there may be benefits from organizing dynamic components in a GenVoca-like fashion. First, many related objects can be grouped together and used as a unit. Second, GenVoca components are very flexible as they can be parameterized by other components to form several different combinations.

Indeed, some of the ideas behind mixin layers can be applied in a dynamic context. In this case, the counterpart of mixin layers is a *design pattern* for organizing objects into large scale, composable components. Recall our discussion in Section 2.3, where we identified encapsulation and mixin-based inheritance as the two essential elements of mixin layers. Although data hiding cannot be achieved in a dynamic setting, encapsulation without data hiding is possible through the use of factory methods. That is, a factory object (the dynamic counterpart of a mixin layer) may be used to group together many other kinds of objects by defining methods to create objects of these kinds. In this way, each factory method can be

viewed as representing the class of the objects it creates. The factory object itself groups together these “classes”.

Mixin-based inheritance can also be approximated among binary objects. Instead of inheriting members and methods from a superclass, however, a dynamic object (called the “outer” object) can only reuse the operations (methods) of another object (called the “inner” object), exporting them as its own without redefining each operation individually. This dynamic counterpart of inheritance is commonly called *aggregation* and is, for instance, supported by the COM object model for binary components [Bro95b]. In fact, because of the dynamic character of such objects, aggregation is analogous to mixin-based inheritance (i.e., the “super-object” is not statically specified at object definition time).

Putting together the above two mechanisms, we can obtain a dynamic counterpart of mixin layers—let us call it *dynamic layers*. A dynamic layer is a factory object (i.e., an instance of a concrete factory class [GHJV94]) that creates objects which can be aggregated. Dynamic layers themselves can be written so that one layer can *delegate* its factory methods to the layer following it in a composition. (The difference between delegation and aggregation is that in delegation the “inner” object’s methods are not automatically exported as methods of the “outer” object.) Figure 5.2 shows such a composition of dynamic layers. Note how each factory method calls the corresponding method of the next layer, while the object created by the outermost layer is passed as a parameter to each factory method. (We use the names `A`*`i`*, `B`*`i`* for the objects created by layer *i*.) The reason is that the generated objects will be aggregated and the inner object of the aggregation needs to know what the outermost object is, so that it can dispatch methods appropriately

(that is, the aggregated object needs to know which object it is a part of, so that it can direct self-methods accordingly).

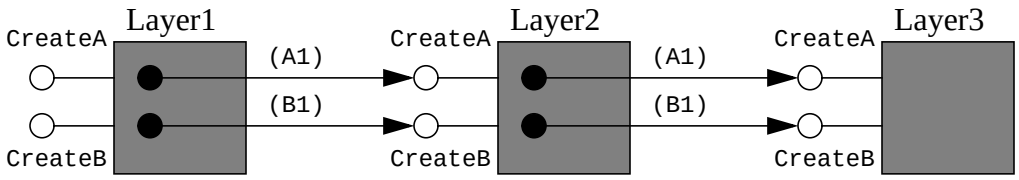

Figure 5.2: Example of a dynamic layer composition

Based on the above scheme for dynamic layers, the corresponding generated objects become simply a collection of aggregated objects, as shown in Figure 5.3. Objects `A1`, `A2`, and `A3` are in a one-to-one correspondence, and so are `B1`, `B2`, and `B3` (but we may have created arbitrarily many such object triples by repeatedly invoking the `CreateA` and `CreateB` methods in the dynamic layers). Note that, for instance, object `A1` aggregates object `A2`, which in turn aggregates object `A3`.

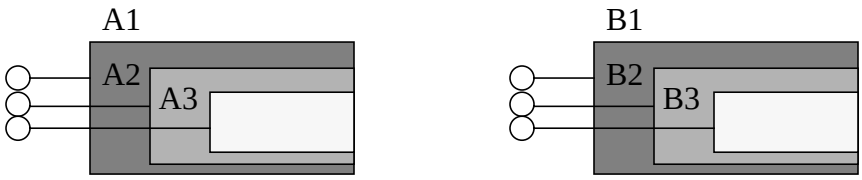

Figure 5.3: Example of the objects created by dynamic layers. Outer objects aggregate inner objects—this is analogous to inheritance in a dynamic setting

This design was actually employed in the DiSTiL generator for data structure programming [SB97]. DiSTiL is implemented as a language extension for the Intentional Programming system of Microsoft Research [Sim95] and follows the GenVoca paradigm. The components in DiSTiL are transformational with respect to the actual application using the DiSTiL data structure code. That is, components collaborate to produce and transform code. From the perspective of the DiSTiL

generator, however, the components are compositional and are put together dynamically (the exact composition of components is determined at generator runtime). Following the pattern shown in Figure 5.2 and Figure 5.3, DiSTiL components are factory objects and create other objects which are aggregated using a simple binary object system.

5.2 Other Related Work

The difficulty of constructing software has been acknowledged early on in the development of Computer Science. In the often-referenced 1968 Software Engineering report of the NATO Science Committee [NR68], the term *software crisis* was used to describe the problems of software development. Given the longevity of the problem, it is not surprising that a wealth of work has been performed in the general area of software construction. Here we selectively discuss some approaches that are closely related to our work but have not been described in detail in the previous chapters.

5.2.1 Modules in High-Level Languages

High-level languages often provide *modules* (a.k.a. *packages* or *namespaces*) as fundamental abstractions. Modules can usually encapsulate static entities, like functions and types. Unlike classes, however, there is usually no notion of separate dynamic instances of a module, each with its own state. Since there is a very large number of languages supporting modules, we selectively discuss a few representative approaches. The Ada *package* mechanism (e.g., [Bar89]) is the prototypical modularization scheme for block structured languages. ML [MTH90] provides a very powerful module system, based on polymorphic types. The C++ equivalent of a module is a *namespace* [Str97].

Mixin layers are probably directly expressible in the latest incarnations of Ada (Ada95 [ISO95]). Standard ML still lacks support for extensible records (i.e., a counterpart of inheritance). Nevertheless, there is nothing fundamental that prevents integrating mixin layers in the language. Recent research has brought some of the mixin layers ideas in a modular language framework. Findler and Flatt's work [FF98] introduces constructs remarkably similar to mixin layers, in an experimental, module-based object system.

The most interesting lesson, however, from comparing mixin layers to traditional modules is simple: classes are a very powerful modularization mechanism. Class nesting allows outer classes to play the role of modules. Using classes as modules offers distinct advantages. First, the mechanism of inheritance can be used to inherit static members (e.g., types) from another class. Second, standard access control (e.g., using the `private` keyword in C++ or Java) can be used for access protection of nested classes. Third, classes are usually better integrated in programming languages than modules (e.g., a C++ namespace cannot be parameterized, while a class can). Having a uniform treatment of classes and modules simplifies a language and results in a more appealing design. Consequently, we believe that the introduction of namespaces in the C++ language should have been avoided, and better support for class nesting should have been provided in the language (enabling classes to fully replace namespaces in all their current functions).

5.2.2 Meta-Object Protocols

Meta-Object Protocols (e.g., [FDM94, KRB91]) are reflective facilities for modifying the behavior of an object system, while the system is being used. Potential modifications include executing arbitrary code around method invocations (method *wrapping*), changing the semantics of inheritance, etc. We will not offer a comprehensive introduction to meta-object protocols here—the interested reader

may consult reference [KRB91] outlining the design and implementation of the CLOS meta-object protocol (the most flexible and powerful representative of meta-object protocols).

Meta-object protocols can be used in several different ways. Method wrappers have been employed to give an object-oriented interface to non-object-oriented legacy systems [JGJ97]. Other applications of wrappers include function tracing, invariant checking, and object locking [FDM94]. Nevertheless, meta-object protocols solve a different problem than mixin layers. Mixin layers intend to address the issue of grouping classes together so they can be treated as a unit and distinguished from other classes or class groups. In contrast, meta-object protocols operate on single classes. Under meta-object protocols, each class has an associated *class meta-object* (an instance of a *meta-class*), which determines the semantics of object system operations on the class. The only grouping that occurs in meta-object protocols is that of methods under a single class: a meta-class can define functionality that affects all methods of a class together (e.g., a single wrapper is applied to all of them).

5.2.3 Aspect-Oriented Programming

A methodology that has gained significant popularity lately is that of *aspect-oriented programming (AOP)* [KLM+97]. Aspect-oriented programming advocates decomposing application domains into orthogonal *aspects*. Aspects are distinct implementation entities and encapsulate code that would otherwise be intertwined throughout an application. In this respect, aspect-oriented programming seems strikingly similar to the GenVoca model. Just like GenVoca, AOP is a design and implementation *methodology*—that is, a set of guidelines and not a set of tools. Hence, it is best described in prose, as a collection of ideas that aim at influencing software designers and implementors.

In this abstract sense, mixin layers qualify as an aspect-oriented implementation mechanism. Nevertheless, the parameterization ability of mixin layers (i.e., the ability to instantiate a layer in multiple inheritance hierarchies, or multiple times in the same inheritance hierarchy) does not seem to be part of standard aspect-orientation.

It should be noted that the foremost application of AOP to date is the AspectJ tool [LK98]: a *transformational* meta-object protocol for Java. Its transformational character means that AspectJ detects actions of the Java object system *in the program text* (e.g., method invocation sites). Then arbitrary code can be executed to modify the program text according to the prescriptions of different aspects.

5.2.4 Adaptive OO Components

Another well-known approach to modular OO software development is Lieberherr's *Demeter* method and adaptive components [Lie96, LP97, ML98]. Adaptive components specify functionality additions based on an abstract pattern of participating classes. The pattern can later be applied to actual classes of an application, so that their functionality is enhanced. This technique is analogous to identifying collaborations in an object-oriented design, only now collaborations are implementation-level entities. Note that mixin layers offer the same flexibility through the concept of adaptor layers discussed in Section 2.4.1. An important difference is that adaptor layers are themselves mixin layers. That is, with mixin layers, both the representation of a collaboration and the representation of a collaboration application are the same (namely, mixin layers).

Nevertheless, the work on adaptive components has revealed an interesting direction of research, with no counterpart in our work. Adaptive components can be applied through a *strategy.* A strategy is a way to specify a path through the

class graph (the graph induced on classes by inheritance and containment relationships among them). Along each node in the strategy, extra functionality can be added. In this way, strategies allow expressing functionality additions for many classes that are grouped together based on their position in the class graph. For instance, one can easily specify new methods to be added to a class *and all its superclasses*. Similarly, assume that class `A` has a member variable that can hold an instance of class `B`, which, in turn, may hold an instance of class `C`. Using strategies, a programmer can describe the path from `A` to `C` in the class graph. (Class `B` does not need to be specified explicitly.) An adaptive component employing this strategy can then define a new method to be added to all three classes. Clearly, mixin layers do not have this ability of identifying classes positionally, but instead rely on explicitly naming the classes that a layer refines.

5.2.5 Design Patterns for Modularization

The *visitor* design pattern [GHJV94] can often serve similar modularization purposes to mixin layers. Visitor is a pattern allowing a *functional* style of programming in object-oriented languages: Multiple definitions of the same operation (applicable to objects of several different classes) can all be grouped together in a visitor class, instead of being distributed in the individual classes. Visitor is a fundamental modularization mechanism and has been used to implement more sophisticated techniques (e.g., [ML98]). Nevertheless, visitors are different from mixin layers in two main ways. First, visitors are dynamic in nature, whereas mixin layers are static. This means, for instance, that mixin layers can be used to add state (i.e., member variables) to the classes they define. Additionally, visitors impose a run-time overhead, unlike mixin layers. Second, visitors are not allowed to access the internals of the classes they are extending. In contrast, mixin layers define subclasses of the refined classes. Hence, mixin layers are often able to

access many more implementation details than visitors. For instance, a C++ class may export a fairly extensive interface to its subclasses (using the `protected` keyword), without making the same interface public so that its visitors can use it.

Overall, many design patterns address some of the same issues as mixin layers. Nevertheless, a mixin layer can be viewed as an elegant way of expressing a collaboration pattern among classes so that it is clear at the language level. Mixin layers are expressed with the aid of the type system, rather than bypassing it, so that more compile-time checking and optimization is possible.

5.2.6 Subjectivity and Views

Objects written for one application may not be reusable in another, because their interfaces are different, even though both applications may deal with what is fundamentally the same object. The principle of *subjectivity* asserts that no single interface can adequately describe any object; objects are described by a family of related interfaces [HO93, HOSU94, OH92, OKH+95]. The appropriate interface for an object is application-dependent (or *subjective*).

Subjectivity arose from the need for simplifying programming abstractions—e.g., defining views that emphasize relevant aspects of objects and that hide irrelevant details. Ossher and Harrison took an important step further by recognizing that application-specific views of inheritance hierarchies can be produced automatically by composing "building blocks" called *extensions* [OH92]. An extension encapsulates a primitive aspect or "view" of a hierarchy, whose implementation requires a set of additions (e.g., new data and method members) to one or more classes of the hierarchy. A customized "view" of an inheritance hierarchy could therefore be defined by composing extensions.

Different and powerful approaches to views and software reuse have been proposed by Goguen [Gog86] and Novak [Nov92, Nov97]. Goguen's work mainly

focusses on mathematical descriptions and axioms, while Novak's work aims at implementing real applications. The essence of both approaches is to define generic abstract components that are automatically specialized to present a customized concrete implementation. A *view* is an isomorphism that defines a mapping of an object to a customized "perspective". Interestingly, Novak's view *clusters* [Nov92, Nov93] encapsulate a suite of interrelated views and map multiple data objects simultaneously. Hence, view clusters are closely related to mixin layers, providing the same essence of grouping classes together. View clusters are probably the first instance of a mixin layer-like pattern to appear in the literature.

5.2.7 Parameterized Programming

Parameterized programming allows generic software to be written once, and instantiated many times for different uses. Goguen identifies two types of parameterization: horizontal and vertical [Gog86]. *Horizontal parameterization* is used to factor out common design elements (e.g., constant values or data types). *Vertical parameterization* is used to layer progressively higher programming abstractions (i.e., abstract machines) in order to progressively implement functionality. Goguen's library interconnection language, LIL, simultaneously provides both horizontal and vertical parameterization. This provides a powerful model of software, allowing maximum reuse of existing software artifacts, and greatly increasing productivity. Other important parameterized programming systems include GLISP [Nov83], LILEANNA [Tra93], PARIS [KRT87], and RESOLVE [SW94].

Our approach to implementing layered designs is not directly comparable to a parameterized programming system. Mixin layers advocate that parameterized modules should be able to encapsulate classes and be viewed themselves as classes (i.e., support inheritance). Clearly, powerful parameterization mechanisms can bet-

ter support mixin layers, but the essential idea is not specific to any parameterization system.

5.2.8 Software Reuse

Software reuse is the process of creating new systems from existing *artifacts* (a.k.a., *assets*) rather than building new systems from scratch [Kru92, Pri93]. Reuse has obvious and significant appeal. It is much easier to reuse existing artifacts than build new ones from scratch [Sel88]. The most obvious example of artifacts that can be reused are source code fragments. But reusable artifacts may be drawn from the full life cycle: requirements, analysis, specifications, designs, documentation, and object code. Potentially reusable design artifacts include specifications written in a design modeling language such as the Unified Modeling Language (UML) and design patterns [GHJV94].

The most naive approach to reuse is scavenging. *Code scavenging* (a.k.a., *leverage*, *cloning* [GW94], *copying*, or *cut-and-paste*) is an ad hoc technique by which software engineers accumulate or locate source code of existing systems *not* specifically designed to be reused (called *legacy systems* if they are *still* in use), find relevant fragments in these systems, and either (1) use them *as-is* (a.k.a., *black-box reuse*) or (2) manually adapt them for use in new systems (a.k.a., *white-box reuse*).[1] Unfortunately, finding relevant source code fragments may require considerable searching, and modifying existing systems for reuse requires understanding them, which itself may require more effort than writing the code from scratch. Thus, although credible, the benefits of scavenging are modest [BR87].

1. The term *design scavenging* is sometimes used to describe scavenging in which a large block of source code is used, but many of the internal details are deleted, while the global template of the design is retained [Kru92]. We avoid using this term, which we consider misleading, because the artifact that is being scavenged is still source *code*, rather than a *design*.

A more successful approach to reuse is based on libraries. A *library* (a.k.a., *repository* or *knowledge base* [Nei94]) is a collection of artifacts, called *components*, specifically designed to be reused. In 1968, McIlroy [McI68] originally envisioned that the components in the library would be functions (a.k.a., subroutines or procedures), because functions were the only suitable language feature available at that time. Since then, however, libraries have been so successful that high level languages have evolved features specifically designed to support components: modules, packages, subsystems, and classes [Kru92].

Reuse can greatly simplify software construction—it has the potential to provide an order of magnitude increase in programmer productivity. Unfortunately, it has three major disadvantages:

- *Difficulty of construction*. It is more difficult to build an object if it is intended to be reused than if it is not. In general, it is 2 to 3 times more difficult [Bro95a]. But the payoff of building for reuse can be substantial.
- *Limited domain of applicability.* The *domain* may be the most important factor in reuse success. The domain must be narrow, well-understood, and slowly changing. Biggerstaff estimates that these properties of the domain account for 80% of the success of software reuse [Big92].
- *The feature combinatorics problem* or *library scalability problem*. This was discussed in Section 2.4 and its essence is that there is an exponential number of component combinations, which makes implementing all combinations by hand infeasible.

Mixin layers complement other approaches to reuse and provide large-scale reusable components. In our experience, it is indeed true that building mixin layers is harder than building non-reusable classes. Nevertheless, the benefits of reusing mixin layers are significant. By employing a static parameterization mechanism,

mixin layers can express an exponential number of combinations without incurring run-time overhead, thus effectively addressing the library scalability problem.

Chapter 6

Conclusions

This dissertation analyzed techniques for implementing large-scale object-oriented components. In this chapter, we review our central results and primary contributions, and discuss a few areas of future research.

6.1 Results and Contributions

Constructing software is a tedious and error-prone task. To alleviate these problems, programming language research has aimed at developing powerful modularization techniques. Using such techniques, a unit of software functionality can be expressed independently of the application in which it is used. In this way, software entities become reusable in multiple environments without having to be reimplemented. This dissertation concentrated on a novel kind of modularization: large-scale object-oriented components. Such components can group together many traditional object-oriented components (classes or binary objects). At the same time, these components act themselves as object-oriented entities, supporting the mechanism of (parameterized) inheritance.

As we demonstrated in previous chapters, large-scale object-oriented components offer several advantages compared to conventional object-oriented programming. We implemented large-scale components by using existing language facilities and showed that they result in much simpler implementations than other

existing techniques. We called our components *mixin layers*, to emphasize their connection to the common *mixin* concept in object-oriented languages. Unfortunately, support for mixin layers is not ideal in any mainstream programming languages. We showed what is missing and how the omissions can be corrected. Finally, we presented a language extension that adds mixin layers to Java and used it to implement an extensible compiler for the Java language. We review the concrete contributions of our research in more detail below:

- In Chapter 2, we introduced mixin layers and described how they can be implemented in multiple programming languages. We showed that mixin layers offer a better way to implement object-oriented *collaboration-based* designs than either application frameworks [JF88] or the technique of VanHilst and Notkin [VN96a-c, Van97]. Mixin layers preserve the advantages of the VanHilst and Notkin implementation method over application frameworks (i.e., maintain design structure, facilitate reuse, and avoid unnecessary dynamic binding). At the same time, mixin layers correct the scalability problems of the VanHilst and Notkin technique yielding simpler code and shorter compositions.
- In Chapter 3, we addressed several programming language issues concerning mixin layers. We showed how type-system support for large-scale components can be provided using two new properties (termed *deep subtyping* and *deep interface conformance*) in order to express constraints for mixin layer parameters. We also showed how type propagation problems (*virtual typing*) can be solved in a mixin layer framework. Other issues addressed include checking the validity of a layer composition through programmer-supplied propositional properties, and analyzing the interaction of mixin layers and other language constructs in mainstream OO languages.

- In Chapter 4 we discussed an actual application that further validates the mixin layers approach. We used mixin layers as the primary implementation technique in a medium-size project (the JTS tool suite for implementing domain-specific languages). Our experience showed that mixin layers are versatile and can handle components of substantial size. The implementation of mixin layers used in that project was itself specified as an extension to the Java language.

6.2 Future Research

Large-scale software components are promising for the future of software construction. The area is relatively young and several interesting directions for future research can be identified.

- *Verifying composition correctness*. The methods discussed in Chapter 3 for verifying the correctness of a composition rely on the programmer supplying simple properties for components. Although this approach can be acceptable, two problems arise. First, the stated properties may not exactly match the component behavior. That is, the checking is not performed on the actual code but on the declared properties of the code. Second, the language for describing requirements may not be expressive enough. Both problems suggest that sophisticated checking mechanisms may be desirable. Properties could be matched to the actual component behavior more closely, perhaps by semi-automatic techniques that will verify that components truly satisfy their stated properties. A richer requirements language could allow the programmer to express declaratively the specification of a "correct" composition, which would later need to be matched to the properties of actual component compositions. Formal verification of computing elements is the focus of a

large body of work in Computer Science. Some of the existing or future results may provide the right balance of automation and expressibility for use in component-based software.

- *Applications and characterization of applicability.* Mixin layers could be applied to several software domains and simplify programming by allowing reusable components to be expressed concisely. Candidate domains include those for which GenVoca designs have been successful in the past. Nevertheless, there is no clear characterization of the domains for which our approach is suitable. The essence of software is its complexity,[1] and software elements often exhibit many interdependencies. Mixin layers rely on isolating orthogonal features of a domain and expressing them independently. Often, separating different software aspects into independent components is impossible, however. Complexity is inherent in such domains and software cannot be decomposed into manageable units. It would be highly valuable to characterize common software domains with respect to their amenability to component-based solutions.
- *Binary components*. Dynamic composition of binary components is a very interesting area for future work. We discussed in Chapter 5 how some of our ideas can be adapted to dynamic components. It remains to be shown, however, whether the expression of mixin-layer-like constructs in a dynamic setting is more advantageous than other patterns of parameterizing binary components. A compelling demonstration (e.g., of the sort presented in Section 2.4) of the advantages of dynamic layers over other actual techniques would be particularly useful in establishing the value of our ideas.

1. To quote Charles Simonyi, “Software is distilled complexity.”

Bibliography

[AFM97] O. Agesen, S. Freund, and J. Mitchell, “Adding Type Parameterization to the Java Language”, *OOPSLA 1997*, 49-65.

[Bar89] J.G.P. Barnes, *Programming in Ada*, 3rd ed., Addison-Wesley, 1989.

[Bat88] D.S. Batory, “Concepts for a Database System Compiler”, in *Proceedings of the Seventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Austin, Texas, March 21-23 1988, ACM Press, pages 184-192.

[Bat97] D. Batory, “Intelligent Components and Software Generators”, *Software Quality Institute Symposium on Software Reliability,* Austin, Texas, April, 1997.

[BBG+88] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, and T. E. Wise. GENESIS: An extensible database management system. In *IEEE Transactions on Software Engineering*, November 1988.

[BC89] K. Beck and W. Cunningham, “A Laboratory for Teaching Object-Oriented Thinking”, *OOPSLA 1989*, 1-6.

[BC90] G. Bracha and W. Cook, “Mixin-Based Inheritance”, *ECOOP/ OOPSLA 1990*, 303-311.

[BCGS95] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer, “Creating Reference Architectures: An Example from Avionics”, In *Proc. ACM SIGSOFT Symposium on Software Reusability,* 1995, 27-37.

[BCRW98] D. Batory, G. Chen, E. Robertson, and T. Wang, “Design Wizards and Visual Programming Environments for Generators”, *Int. Conference Software Reuse*, 1998.

[BG96] G. Bracha and D. Griswold, “Extending Smalltalk with Mixins”, *Workshop on Extending Smalltalk* at OOPSLA 1996. See `http://java.sun.com/people/gbracha/mwp.html`.

[BG97] D. Batory and B.J. Geraci, “Component Validation and Subjectivity in GenVoca Generators”, *IEEE Trans. on Softw. Eng.*, February 1997, 67-82.

[Big92] T.J. Biggerstaff, “An Assessment and Analysis of Software Reuse”, *Advances in Computers*, Volume 34, Academic Press, 1992.

[Big94] T.J. Biggerstaff, “The Library Scaling Problem and the Limits of Concrete Component Reuse”, *3rd Int. Conf. on Softw. Reuse (ICSR ‘94)*.

[BLS98] D. Batory, B. Lofaso, and Y. Smaragdakis, “JTS: Tools for Implementing Domain-Specific Languages”, *5th Int. Conf. on Softw. Reuse (ICSR ‘98)*, IEEE Computer Society Press, 1998.

[BO92] D. Batory and S. O’Malley, “The Design and Implementation of Hierarchical Software Systems with Reusable Components”, *ACM TOSEM*, October 1992.

[BOSW98] G. Bracha, M. Odersky, D. Stoutamire and P. Wadler, “Making the future safe for the past: Adding Genericity to the Java Programming Language”, *OOPSLA 1998*.

[BOW98] K.B. Bruce, M. Odersky, and P. Wadler, “A Statically Safe Alternative to Virtual Types”, *ECOOP 1998*.

[BP89] T.J. Biggerstaff and A.J. Perlis, editors, *Software Reusability, Volume 1: Concepts and Models*, ACM Press, 1989.

[BR87] T.J. Biggerstaff and C. Richter, “Reusability Framework, Assessment, and Directions”, *IEEE Software*, July 1987, 41-49. Also in Will Tracz, editor, *Software Reuse: Emerging Technology*, IEEE Computer Society Press, 1988, 3-11. Also in [Big89], pages 1-17.

[Bro95a] F.P. Brooks, *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*, Addison-Wesley, Reading, Massachusetts, 1995, 179-203.

[Bro95b] K. Brockschmidt, *Inside OLE* (2nd. ed.), Microsoft Press, 1995.

[BSST93] D. Batory, V. Singhal, M. Sirkin, and J. Thomas, “Scalable Software Libraries”, *ACM SIGSOFT* 1993.

[BT97] D. Batory and J. Thomas, “P2: A Lightweight DBMS Generator”, *Journal of Intelligent Information Systems*, 9, 107-123 (1997).

[CE99a] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Techniques, and Applications*. Addison-Wesley, to appear in 1999. See also:
`http://nero.prakinf.tu-ilmenau.de:80/~czarn/gmcl/`

[CE99b] K. Czarneci and U. Eisenecker, "Synthesizing Objects", *ECOOP 1999*, 18-42.

[Chi96] S. Chiba, “Open C++ Programmer's Guide for Version 2”, SPL-96-024, Xerox PARC, 1996.

[CS93] L. Coglianese and R. Szymanski, “DSSA-ADAGE: An Environment for Architecture-based Avionics Development”, *Proc. AGARD 1993*.

[CW85] L. Cardelli and P. Wegner, On Understanding Types, Data Abstraction, and Polymorphism, *Computing Surveys*, 17(4): Dec 1985, 471-522.

[ES90] M.A. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[FDM94] I.R. Forman, S. Danforth, and H. Madduri, "Composition of Before/ After Metaclasses in SOM", *OOPSLA 1994*.

[FF98] R.B. Findler and M. Flatt, "Modular Object-Oriented Programming with Units and Mixins", *Int. Conf. on Functional Programming*, 1998.

[FKF98] M. Flatt, S. Krishnamurthi, M. Felleisen, "Classes and Mixins". ACM *Symposium on Principles of Programming Languages*, 1998 (PoPL 98).

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJS96] James Gosling, Bill Joy, Guy L. Steele, *The Java Language Specification*, Addison-Wesley, Reading, Massachusetts, 1996.

[Gog86] J. Goguen, "Reusing and interconnecting software components", *IEEE Computer*, February 1986, 16-28.

[GW94] M.L. Griss and K.D. Wentzel, "Hybrid Domain-Specific Kits for a Flexible Software Factory", *Proc. 1994 ACM Symposium on Applied Computing (ACM SAC '94), Reuse and Reengineering Track*, 47-52.

[HHG90] R. Helm, I. Holland, and D. Gangopadhyay, “Contracts: Specifying Behavioral Compositions in Object-Oriented Systems”. *OOPSLA 1990*, 169-180.

[HO93] W. Harrison and H. Ossher, “Subject-Oriented Programming (A Critique of Pure Objects)”. *OOPSLA 1993*, 411-428.

[Hol92] I. Holland, “Specifying Reusable Components Using Contracts”, *ECOOP 1992*, 287-308.

[HOSU94] W. Harrison, H. Ossher, R.B. Smith, and D. Ungar, “Subjectivity in Object-Oriented Systems: Workshop Summary”, in *Addendum to OOPSLA 1994 Conference Proceedings*, 131-136.

[HP91] N. Hutchinson and L. Peterson, “The *x*-kernel: An Architecture for Implementing Network Protocols”, *IEEE Trans. Softw. Eng. 1991*, pp.64-76.

[HP94] John S. Heidemann and Gerald J. Popek, “File system development with stackable layers”, *ACM Transactions on Computer Systems*, February 1994, 58-89.

[ISO95] ISO/IEC revised international standard 8652:1995, *Ada 95 Reference Manual (Language and Standard Libraries)*.

[Jav97a] Javasoft, *Java Core Reflection Specification*, 1997. In [JavWeb].

[Jav97b] Javasoft, *Java Inner Classes Specification*, 1997. In [JavWeb].

[JavWeb] JavaSoft documentation, available at: `http://java.sun.com/products/jdk/1.1/docs/`.

[JF88] R. Johnson and B. Foote, “Designing Reusable Classes”, *Journal of Object-Oriented Programming*, 1(2): June/July 1988, 22-35.

[JGJ97] I. Jacobson, M. Griss, and P. Jonsson, *Software Reuse: Architecture Process and Organization for Business Success*, Addison-Wesley/ ACM Press, 1997.

[KFFD86] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. “Hygienic Macro Expansion”. In *Proc. of the SIGPLAN ‘86 ACM Conf. on Lisp and Functional Programming*, 151-161.

[KH98] R. Keller, U. Hoelzle, “Binary Component Adaptation”, *ECOOP 1998*.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin, “Aspect-Oriented Programming”, *ECOOP 1997*, 220-242.

[KRB91] G. Kiczales, J. des Rivieres, and D. G. Bobrow, *The Art of the Metaobject Protocol*, MIT Press, 1991.

[KRT87] S. Katz, C. Richter, and K. The, “PARIS: A System for Reusing Partially Interpreted Schemas”, in *Proc. 9th Int. Conf. on Softw. Eng.*, 1987, IEEE Computer Society Press, 377-385. Also in Will Tracz, editor, *Tutorial: Software Reuse: Emerging Technology*, IEEE Computer Society Press, 1988. Also In [Big87], 257-273.

[Kru92] C.W. Krueger, “Software Reuse”, in *ACM Computing Surveys*, Volume 24, Number 2, June 1992.

[Lie96] K.J. Lieberherr, *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*, PWS Publishing Company, Boston, 1996.

[LK98] C.V. Lopes and G. Kiczales, “Recent Developments in AspectJ”, *ECOOP'98 Workshop Reader (Aspect-Oriented Programming Workshop)*, Springer-Verlag LNCS 1543.

[LP97] K.J. Lieberherr and B. Patt-Shamir, "Traversals of Object Structures: Specification and Efficient Implementation", College of Computer Science, Northeastern University, Tech. Report NU-CCS-97-15, July 1997.

[MBL97] A. Myers, J. Bank and B. Liskov, “Parameterized Types for Java”, ACM *Symposium on Principles of Programming Languages*, 1997 (PoPL 97).

[McI68] M. D. McIlroy, “Mass produced software components”, in [Nau68].

[Mez97] M. Mezini, “Dynamic Object Evolution without Name Collisions”, *ECOOP 97*, 190-219.

[Mil78] R. Milner, “A Theory of Type Polymorphism in Programming”, *Journal of Computer and System Sciences*, 17:Dec 1978, 348-375.

[ML98] M. Mezini and K. Lieberherr, “Adaptive Plug-and-Play Components for Evolutionary Software Development”, *OOPSLA 1998*.

[MM89] O. L. Madsen and B. Møller-Pedersen, “Virtual classes: A powerful mechanism in object-oriented programming”, *OOPSLA 1989*, 397-406.

[MMN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard, *Object-Oriented Programming in the BETA Programming Language*. Addison-Wesley, 1993.

[Mon96] T. Montlick, “Implementing Mixins in Smalltalk”, *The Smalltalk Report*, July 1996.

[Moo86] D.A. Moon, “Object-Oriented Programming with Flavors”, *OOPSLA 1986*.

[MTH90] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML*, MIT Press, Cambridge, Massachusetts and London, England, 1990.

[Nei94] J.M. Neighbors, “An Assesment of Reuse Technology after Ten Years”, in *Proc. 3rd Int. Conf. on Softw. Reuse, 1994 (ICSR ‘98)*.

[Nov83] G.S. Novak, “GLISP: A Lisp-based Language with Data Abstraction”, *AI Magazine*, Volume 4, Number 3, Fall 1983, 37-47. Also in Robert Engelmore, editor, *Readings from AI Magazine*, AAAI Press, 1988, 545-555.

[Nov92] G.S. Novak, “Software Reuse through View Type Clusters”, in *Proc. Seventh Knowledge-Based Softw. Eng. Conf. (KBSE-92)*, 1992, 70-79.

[Nov93] G.S. Novak, “Software Reuse by Compilation Through View Type Clusters”, *University of Texas, AI Lab Tech. Report A192-182A*, September 1993.

[Nov97] G.S. Novak, “Software Reuse by Specialization of Generic Procedures through Views”, in *IEEE Trans. on Softw. Eng.*, July 1997, 401-417.

[NR68] P. Naur and B. Randall, editors, *Software Engineering: A Report on a Conference Sponsored by the NATO Science Committee*, NATO Scientific Affairs Division, Brussels, Belgium, 1968.

[OH92] H. Ossher and W. Harrison, “Combination of Inheritance Hierarchies”, *OOPSLA 1992,* 25-40.

[OKH+95] H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal, “Subject-Oriented Composition Rules”, *OOPSLA 1995,* 235-250.

[OP92] Sean W. O’Malley and Larry L. Peterson. A Dynamic Network Architecture. In *ACM Transactions on Computer Systems*, May 1992, 110-143.

[OW97] M. Odersky and P. Wadler, “Pizza into Java: Translating theory into practice”, *ACM Symposium on Principles of Programming Languages*, 1997 (PoPL 97).

[Pre97] C. Prehofer, “Feature-Oriented Programming: A Fresh Look at Objects”, *ECOOP 1997*, 419-443.

[Pri93] R. Prieto-Diaz, “Status Report: Software Reusability”, *IEEE Software*, May 1993, 61-66.

[RAB+92] T. Reenskaug, E. Anderson, A. Berre, A. Hurlen, A. Landmark, O. Lehne, E. Nordhagen, E. Ness-Ulseth, G. Oftedal, A. Skaar, and P. Stenslet, “OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems”, *Journal of Object-Oriented Programming*, 5(6): October 1992, 27-41.

[Rea90] Reasoning Systems, “Dialect User’s Guide”, Palo Alto, California, 1990.

[Rum94] J. Rumbaugh, “Getting Started: Using use cases to capture requirements”, *Journal of Object-Oriented Programming*, 7(5): Sep 1994, 8-23.

[SB97] Y. Smaragdakis and D. Batory, “DiSTiL: a Transformation Library for Data Structures”, *USENIX Conference on Domain-Specific Languages (DSL 97)*.

[SB98a] Y. Smaragdakis and D. Batory, “Implementing Reusable Object-Oriented Components”, *5th Int. Conf. on Softw. Reuse (ICSR ‘98)*, IEEE Computer Society Press, 1998.

[SB98b] Y. Smaragdakis and D. Batory, “Implementing Layered Designs with Mixin Layers”, *ECOOP 1998*.

[SB98c] Y. Smaragdakis and D. Batory, “Mixin-Based Programming in C++”, University of Texas at Austin CS Tech. Report 98-27.

[SCD+93] P. Steyaert, W. Codenie, T. D'Hondt, K. De Hondt, C. Lucas, and M. Van Limberghen, “Nested Mixin-Methods in Agora”, *ECOOP 1993*, 197-219.

[Sel88] R.W. Selby, “Empirically Analyzing Software Reuse in a Production Environment”, in Will Tracz, editor, *Software Reuse: Emerging Technology*, IEEE Computer Society Press, 1988, 176-189.

[SGIWeb] Silicon Graphics Computer Systems Inc., *STL Programmer’s Guide*. See: `http://www.sgi.com/Technology/STL/` .

[Sim95] C. Simonyi, “The Death of Computer Languages, the Birth of Intentional Programming”, *NATO Science Committee Conference*, 1995.

[Sin96] V. Singhal, “A Programming Language for Writing Domain-Specific Software System Generators”, Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, August 1996.

[SL95] A. Stepanov and M. Lee, “The Standard Template Library”, 1995. Incorporated in ANSI/ISO Committee C++ Standard.

[SPL96] L. Seiter, J. Palsberg, and K. Lieberherr, “Evolution of Object Behavior using Context Relations”, *ACM SIGSOFT* 1996.

[Str97] B. Stroustrup, *The C++ Programming Language, 3rd Ed.*, Addison-Wesley, 1997.

[SW94] M. Sitaraman and B.W. Weide, editors, “Special Feature: Component-Based Software Using RESOLVE”, *ACM Softw. Eng. Notes*, October 1994, 21-67.

[TB95] L. Tokuda and D. Batory, “Automated Software Evolution via Design Pattern Transformations”, *Symp. on Applied Corporate Computing*, Monterrey, Mexico, Oct. 1995.

[Tho97] K. Thorup, “Genericity in Java with Virtual Types”, *ECOOP 1997*, 444-471.

[Tho98] J. A. Thomas, *P2: A Lightweight DBMS Generator*, Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, December 1998.

[Tra93] W. Tracz, “LILEANNA: A Parameterized Programming Language”, in Ruben Prieto-Diaz and William B. Frakes, editors, *Advances in Software Reuse: Selected Papers from the Secomd International Workshop on Software Reusability*, 1993, IEEE Computer Society Press, 66-78.

[Van97] M. VanHilst, “Role-Oriented Programming for Software Evolution”, Ph.D. Dissertation, University of Washington, Computer Science and Engineering, 1997.

[Vil94] Emilia E. Villarreal. *Automated Compiler Generation for Extensible Data Languages*, Ph.D. Thesis. Department of Computer Sciences, University of Texas at Austin, 1994.

[Vil97] Emila E. Villarreal. Rosetta: A Generator of Data Language Compilers, in *Proc. ACM SIGSOFT Symposium on Software Reusability,* Boston, Massachusetts, 1997, 146-156.

[VN96a] M. VanHilst and D. Notkin, "Using C++ Templates to Implement Role-Based Designs", *JSSST International Symposium on Object Technologies for Advanced Software*, Springer-Verlag, 1996, 22-37.

[VN96b] M. VanHilst and D. Notkin, "Using Role Components to Implement Collaboration-Based Designs". *OOPSLA 1996*.

[VN96c] M. VanHilst and D. Notkin, "Decoupling Change From Design", *ACM SIGSOFT* 1996.

[WC93] D. Weise and R. Crew, "Programmable Syntax Macros", *ACM SIGPLAN Notices* 28(6), 1993, 156-165.

[Wei90] David M. Weiss. *Synthesis Operational Scenarios*. Technical Report 90038-N, Version 1.00.01, Software Productivity Consortium, Herndon, Virginia, August 1990.

[WeiWeb] K. Weihe, "A Software Engineering Perspective on Algorithmics", available at `http://www.informatik.uni-konstanz.de/Preprints/`.

[Wil93] D.S. Wile, "POPART: Producer of Parsers and Related Tools", USC/ Information Sciences Institute Tech. Report, November 1993.

[WOS98] P. Wadler, M. Odersky and Y. Smaragdakis, “Do Parametric Types Beat Virtual Types?”, unpublished manuscript, posted in the Java Genericity mailing list (`java-genericity@galileo.East.Sun.COM`), October 1998.

Vita

Ioannis (Yannis) Smaragdakis was born in Athens, Greece on March 31, 1972, the son of Michael Smaragdakis and Eugenia Giannopoulou. After developing his skepticism and cynicism in the Greek public school system, he entered the Department of Computer Science at the University of Crete in 1989. After some of the best years of his life, he received the degree of Bachelor of Science in Computer Science in 1993. In August 1993, he entered the Graduate School of the University of Texas at Austin. There he received the degree of Master of Science in Computer Sciences in June 1995.

Yannis finds referring to himself in the third person to be hilarious and hopes nobody will ever get to read this Vita.

Permanent Address: 38 Plapouta St.
Heraklion, Attiki, 14122, Greece

This dissertation was typed by the author.