Java Layers:

Extending Java to Support Component-Based Programming

Richard Cardone Don Batory Calvin Lin
Department of Computer Sciences
University of Texas at Austin
{richcar, batory, lin} @cs.utexas.edu

June 28, 2000

Abstract

Java Layers extends the Java programming language by implementing a software component model based on layer composition. Each layer implements a single design feature and may contain code that crosscuts multiple classes. Layer composition enables large software applications to be constructed in a more modular way, and with a higher level of semantic checking, than is typically achieved using current programming techniques such as object-oriented frameworks. This paper describes the Java Layers language extension.

1 Introduction

A fundamental goal of software engineering is to reduce the complexity of creating and maintaining large software applications. The need to accommodate variation has made this a difficult and elusive goal. There are at least two types of variation. First, the need to change applications over time tends to degrade software quality. As features are added, removed or modified, unanticipated interactions and co-dependencies between feature implementations decrease the overall modularity of the software. The design decays with each change until an expensive redesign is forced or until the application becomes so resistant to change it must simply be discarded. Second, variation over feature sets allows different versions of an application to satisfy different users, execution environments or market segments. The desire to easily provide distinct feature sets leads to the development of families of applications or *software product lines* [9]. The challenge here is to build and maintain product lines that maximize code reuse without sacrificing performance or maintainability.

An ideal solution to both of these problems would be a programming model in which applications could be trivially composed from *components*. Components would be composed in various ways to easily produce the multiple application instances of a software product line, each with precisely the desired set of features. Similarly, as an application's requirements evolve, the application would be modified by either composing components in different ways or by writing new components that would then be combined with those that already exist.

We speak of *design features*, or simply features, as high-level requirements that define some application attribute or capability. For example, applications might have features that make them secure, portable, fail-safe, able to use multiple protocols, dependant on certain libraries, etc. A key attribute of our software component model is that each component encapsulates exactly one design feature. This property maximizes code reuse, since each feature is implemented only once. This property also facilitates the composition of components, making it easy to include or exclude individual features. And of course, the one-feature/one-component property preserves code and design modularity.

This paper introduces Java Layers (JL), an extension of Java that provides a software component model based on components called *layers*. Each layer supplies the code for a single design feature, and layers can be composed using *type equations*. Layers are restricted forms of Java classes, so the construction of layers is similar in complexity to the construction of standard Java classes. Type equations are sufficiently succinct that layer composition is trivial. From layers and type equations, the JL compiler produces a collection of Java class definitions and interface definitions.

JL is a Java source code generator based on the GenVoca model of layered software development [3][32]. JL differentiates itself from previous GenVoca efforts by combining domain independence with a central role for interface definitions in the code generation process, language support for high-level semantic checking, and use of a class hierarchy optimization algorithm.

This paper proceeds as follows. Section 2 describes in more detail the problems that JL is designed to solve, contrasting JL with current practice. Section 3 gives a high-level overview of JL and Section 4 provides an illustrative example. Sections 5 and 6 describe the JL language extension and its compiler optimizations, respectively. Related work and conclusions are discussed in Sections 7 and 8.

2 Motivation

Current programming technologies do not have the power to encapsulate complete design features in one language construct. Code that implements a single design feature is often dispersed throughout multiple classes or functions (*scattering*), and code at a single location often participates in the implementation of multiple features (*tangling*) [14][23][35]. When scattering and tangling happen, there no longer is a clean *separation of concerns* [26] as code performing various functions is intermixed. This intermixing diminishes both the ability to reuse the feature code and the modularity of the program.

Software libraries, macros, parameterized types, and the object-oriented concepts of inheritance and polymorphism are examples of current programming technologies that exhibit code tangling and scattering. The first three techniques can only promulgate a single change at a time and, therefore, cannot easily encapsulate features that change different procedures, methods or classes in different ways. Object-oriented technology provides a more powerful refinement capability: The class is the basic unit of reuse and subclassing allows multiple methods to be modified in a single refinement. Unfortunately, many design features require changes to multiple collaborating classes [20][23][25] and cannot be accommodated by the creation of a single subclass.

JL programming avoids code tangling and scattering by encapsulating all of the modifications that implement a design feature inside a single layer. Since the addition or removal of a layer can induce changes throughout an application's code base, layers can be described as *large-scale refinements*. Not only do layers preserve the modularity of an application, but composing features at the design level leads to the kind of configurability necessary for building software product lines. We note that JL only provides first order protection against code tangling/scattering: Feature code that requires changes to multiple layers would become scattered. Since the features encapsulated by layers are for the most part independent of each other, code scattering among layers should not be a serious problem in practice.

Object-oriented frameworks, a common reuse technology for developing large applications and software product lines [29][28], exhibit their own limitations. Frameworks are a set of abstract classes that embody an abstract design. Applications are built by extending these general, abstract classes with application-specific, concrete implementations. The rigid distinction between framework and application often leads to problems as both evolve [7]. Putting too many features in the framework leads to *overfeaturization* that complicates the use of the framework and ultimately bloats applications with unneeded feature code [15]. Conversely, omitting features from a framework can lead to *code replication* across applications and the predictable maintenance problem that this causes.

Object-oriented frameworks are also susceptible to the *feature combinatorics* problem [4]. Given a domain with ***n*** optional features, the feature combinatorics problem occurs when all valid feature combinations must be predefined or in some way materialized in advance. In the worst case, ***n!*** concrete programs would have to be instantiated. In frameworks, each abstract class that defines a variation point can be implemented by a unique concrete class for each desired mix of features. The use of standard subclassing alone can lead to an exponential explosion in the number of required classes. One alternative is to employ dynamic composition to avoid the combinatorial problem at the expense of higher runtime overhead.

An important goal of JL is to improve upon the current use of object-oriented frameworks in building applications. JL avoids the problems of overfeaturing and code replication by using layer composition to tailor each application with only those features that it actually requires. Moreover, the code for a specific feature needs to be implemented only once in its own reusable layer. JL also avoids the feature combinatorics problem by generating code with the required combination of features only on demand. JL's static compositional approach solves the scalability problem without incurring the runtime overhead of dynamic composition.

In the idealized plug and play environment described in the Introduction, code specification is a feature selection and composition activity. The ability to compose high-level components in this manner must be accompanied by an equally high-level way to restrict invalid component combinations. To illustrate this point, consider the semantics of a feature that implements mutual exclusion in JL. Since locks typically need to be acquired only once, a layer that implements synchronization should appear at most once in a layer composition. Multiple appearances may not violate type correctness, but would certainly violate the design intent of the feature. This paper will describe a preliminary version of JL's *design rules*, which are used to express this type of semantic constraint on layer composition.

Lastly, JL is motivated by the desire to raise the level of programming abstraction while still producing efficient code. The goal in JL is that the effects of design-time layering should not significantly affect the performance of the runtime code.

3 Java Layers Overview

JL extends standard Java and has the following design goals:

- *Domain independence* – One model and language should suffice for all domains.
- *Easy component composition* – Even novice programmers should be able to specify layer compositions.
- *Easy component creation* – Layer writing should be similar in difficulty to writing Java classes.
- *Efficiency* – Design-time layering should not appreciably impact runtime performance.
- *Effectiveness* – JL should provide compelling advantages in handling program variation.

JL is used to define and compose layers. Each JL layer can supply the code for a single, complete, design-level feature and, in doing so, may generate code that ultimately resides in multiple classes. A layer exports (implements) one or more Java interfaces and imports zero or more types (classes or other layers). Two layers can be composed or stacked only if they have compatible import/export interfaces.

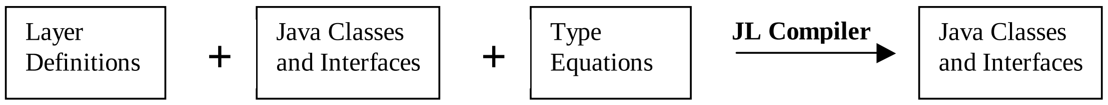

Figure 1 – JLC Input/Output

Figure 1 shows that the JL compiler accepts layer definitions, Java classes and interfaces, and *type equations* as input and produces Java class and interface definitions as output. Type equations specify layer compositions and are typically extremely succinct. We will refer to the compilation of these type equations as *layer composition*, and we will refer to the time at which this compilation occurs as *composition time*. After layer composition occurs, the code generated is automatically compiled with a standard Java compiler to produce the final executable program.

4 Examples

This section introduces the basic concepts of JL by developing a running example. We begin by defining a standard Java interface and three layers that export it. We then show how those layers can be composed, how the compositions can be annotated with semantic checks, and how interfaces can be easily extended. Finally, we use a client/server example to illustrate how JL can use nested classes and interfaces to implement large-scale refinements and build applications in a stepwise manner.

4.1 Transport Example

To understand how an application can be constructed from a set of layers, consider the following Java interface which contains only public methods (throws clauses are not shown):

```
public interface TransportIfc
{
 int send(byte[] outBuf);
 int recv(byte[] inBuf);
 void disconnect();
}
```

Figure 2

Suppose we develop layers TCP, UDP, and UnixPipe, which export TransportIfc and provide the basic connectivity suggested by their names. We might also develop a Secure layer, which provides data authentication and encryption, and a KeepAlive layer which automatically sends liveness notifications between communicating peers. Assume that the last two layers both import and export the TransportIfc interface.

```
layer TCP exports TransportIfc
{
 public int send(byte[] outBuf){…}
 public int recv(byte[] inBuf){…}
 public void disconnect(){…}
}

layer Secure<mixin M implements TransportIfc>
 exports TransportIfc
{
 public int send(byte[] outBuf){…}
 public int recv(byte[] inBuf){…}
 public void disconnect(){…}
}
```

Figure 3

Figure 3 shows that the TCP and Secure layers both explicitly export the TransportIfc interface. The Secure layer also imports either a class that implements TransportIfc or a layer that exports TransportIfc (which, we will see shortly, amounts to the same thing). The **mixin** keyword indicates that the imported type will be used as a superclass in the generated code. The ellipses represent code that actually implements the required function and may include calls to superclass methods. The actual generated code is described in the next section.

Mixins [10][32] are an important concept in JL that are available in C++ [34] but not Java. Mixins are types whose supertypes are parameterized. Mixins are useful because they allow a set of classes to be specialized in the same manner, with the specialized code residing in a single class definition. For example, suppose we wish to extend three unrelated classes–`Car`, `Chest` and `House`–to be "lockable" by adding two methods, `lock()` and `unlock()`. Without mixins, we would define subclasses of `Car`, `Chest`, and `House` that each extended their respective superclasses with the `lock()` and `unlock()` methods. The result is code replication. With mixins, we would instead write a single class called `Lockable` that could extend `Car`, `Chest`, `House` or any other superclass. In C++ syntax this class would be defined as follows, where `lock()` and `unlock()` would only have to be defined once:

```
// Mixins in C++
template <class T>
 class Lockable : public T
{
 public lock();
 public unlock();
}
```

The syntactic resemblance between layer definitions and standard Java class definitions hints at their deeper connection. Intuitively, each layer definition can be thought of as generating a Java class and a Java interface in which the semantics of import, export and composition constraints are enforced.

4.1.1 Composing Layers

```
public class Tr1 = UnixPipe;
class Tr2 = Secure[UDP];
class Tr3 = KeepAlive[Secure[TCP]];
```

Figure 4

Once compatible layers have been defined, they can easily be composed. Figure 4 depicts three JL classes that are defined using type equations. A type equation consists of an identifier on the left, followed by an equals sign, followed by a layer composition on the right. The Tr1 class is publicly accessible and only includes the functionality built into the UnixPipe layer. The UnixPipe layer, like the TCP layer shown in Figure 3, exports the TransportIfc interface and doesn't import any interface. Layers without imports are called *terminal* layers.

The Tr2 class has package scope and allows for secure communication over UDP. As described above, the UDP layer is a terminal layer that exports TransportIfc. The Secure layer, shown in Figure 3, is said to be *symmetric* because it both imports and exports the same interface (TransportIfc). The type parameter M in the Secure layer's mixin clause is bound to the class generated from the UDP layer. The binding of type parameters in JL is similar to the binding that takes place during C++ template processing.

The Tr3 class implements the automatic keep alive feature over a secure TCP connection. The TCP layer is terminal; KeepAlive and Secure are symmetric. Type parameters in KeepAlive and Secure are bound to the classes generated from their imported layers, Secure and TCP, respectively.

Tr3 could, alternately, have been defined with the Secure and KeepAlive layers in reverse order. The result would be a class in which liveness messages would be sent in the clear rather than encrypted as in the original Tr3 configuration. In this example, each of the three terminal transport layers can be combined with any combination of Secure and KeepAlive layers. If we discount type equations with duplicate layers, there are still 15 possible feature combinations that we can easily define using this small number of layers.

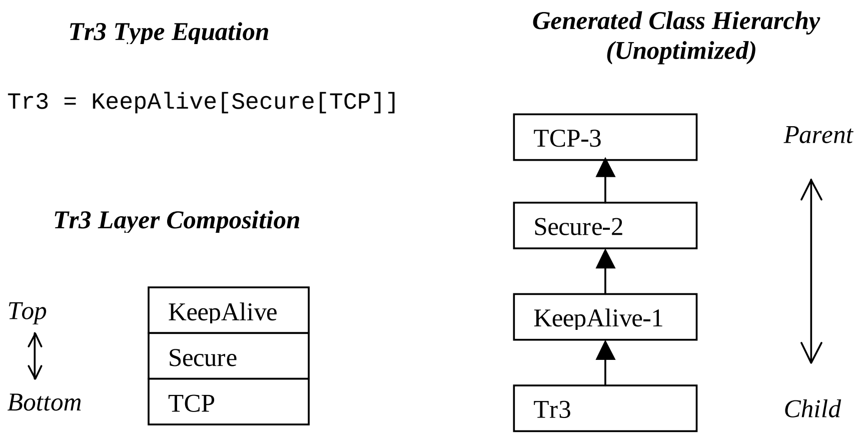

Figure 5

Figure 5 shows the relationship between the Tr3 type equation, the layer composition it represents, and the Java class hierarchy generated to implement it. The ultimate result of compiling the Tr3 type equation is a Java class named Tr3 that implements the generated Tr3Ifc interface (not shown in the figure).

Compilation can also be seen as generating classes and interfaces for each of the layers appearing in the Tr3 layer composition. Figure 5 depicts the unoptimized class hierarchy generated for the Tr3 type equation, assuming that the KeepAlive layer also mixes in its imported type. By default, each generated class name contains a unique layer number that allows layers to appear more than once in a composition. Each of these generated classes implements a custom interface that extends TransportIfc, and each class body is derived from its corresponding layer body.

It's common for a method to perform some work and then invoke the superclass method with the same signature for further processing. For example, the send() method in Secure-2 could encrypt the data then call send() in TCP-3 for data transmission. In this way, each layer can perform its feature-specific work and then pass control to its mixed-in superclass. JL uses mixins as a design pattern in support of the *stepwise refinement* of a program. Of course, the choice to call a superclass method is purely at the discretion of the layer programmer who chooses how each method will be implemented.

Note that the bottom mixin layer in the layer composition (TCP) becomes the root of the generated class hierarchy (TCP-3). Layers can be thought of as a stack a of virtual machines with the highest level service being exposed on top and the most basic service residing at the bottom. When translated into an object-oriented model, the most basic service occupies the root of a class hierarchy and each subclass provides a more specialized implementation. When laid out pictorially as in Figure 5, generated classes appear in an order opposite that of the associated type equation.

Compiler optimizations that eliminate artifacts of layer specification from the generated class hierarchy are described in Section 6. When optimized, the four classes shown in Figure 5 would be collapsed into a single, optimized Tr3 class.

4.2 Adding Design Rules

Design rules allow semantic information to be manipulated, propagated and checked at layer composition time. The leftmost layer in a layer composition is said to be the *top* node in a tree that contains all other parameters in the composition. The KeepAlive layer is the top node in the Tr3 composition shown in Figure 5. Design rule processing is modeled as two separate flows of design rule variables within a type equation. The *upflow* begins at leaf nodes and propagate upwards to the top node. The *downflow* begins at the top node and propagates downwards to all leaf nodes.

```
layer TCP exports TransportIfc
 upflow { transport = true; }
{
 public int send(byte[] outBuf){…}
 public int recv(byte[] inBuf){…}
 public void disconnect(){…}
}

layer Secure<mixin M implements TransportIfc>
 exports TransportIfc
 uptest (exists(transport)&& !exists(secure))
 upflow { secure = true; }
{
 public int send(byte[] outBuf){…}
 public int recv(byte[] inBuf{…}
 public void disconnect(){…}
}
```

Figure 6

In Figure 6, the TCP and Secure layers have been augmented with **upflow** definitions; **downflow** definitions can be similarly defined. The TCP layer creates and assigns the boolean variable `transport` in the upflow. This variable is propagated up to the next layer in any type equation in which the TCP layer appears. In the case of the Tr3 class in Figure 5, the Secure layer receives its upflow from the TCP layer, which contains only the `transport` variable.

The Secure layer definition has been augmented with an **uptest** clause that must evaluate to true before layer composition is allowed to proceed; **downtest** clauses are similar in structure. In this example, we only attach meaning to the existence of variables in a flow and ignore their values. Secure's uptest clause can be interpreted as requiring that some lower layer provide the actual transport function and that no lower layer already provide the secure function. Secure's uptest would fail, for instance, if two Secure layers appear along the same path in a type equation. In Tr3, the uptest expression evaluates to true, so design rule processing would proceed. The next step would be to process Secure's upflow clause by adding the new variable, `secure`, to the existing upflow. The cycle of uptest/upflow processing continues in this way up to the top node in a type equation. Though not shown, downtest/downflow processing operates in essentially the same manner.

4.3 Adding Methods

Up to this point, we have focused on implementing existing interfaces and we have omitted the methods needed to establish a communication session. Figure 7 shows the signatures of new session-oriented methods (excluding throws clauses). The `connect()` method allows a client to initiate a session. The `createPassive()` method allows a server to create a communication endpoint that accepts new sessions using the `accept()` method.

```
public boolean connect(Address addr);
public boolean createPassive(Address addr);
public TransportIfc accept();
```

Figure 7

```
layer Active<mixin M implements TransportIfc>
 exports TransportIfc
{
 public boolean connect(Address addr){…}
}
```

Figure 8

There are a number of ways in which a layer can augment the set of methods that it exports. The Active layer in Figure 8 introduces the new connect() method by simply defining the public method in its body. JL will automatically incorporate new public methods into the interface generated for the layer. Also note that layers do not need to implement all methods of their exported interfaces.

```
public interface PassiveIfc
{
 boolean createPassive(Address addr);
 TransportIfc accept();
}

layer Passive<mixin M implements TransportIfc>
 exports TransportIfc, PassiveIfc
{
 public boolean createPassive(Address addr){…}
 public TransportIfc accept(){…}
}
```

Figure 9

A layer may also export new public methods by exporting multiple interfaces. The Passive layer in Figure 9 exports both the TransportIfc and PassiveIfc interfaces.

```
class TrClt = Active[Secure[TCP]];
class TrSvr = Passive[Secure[TCP]];
```

Figure 10

Figure 10 shows two new class definitions that we will use in the next example. The TrClt class provides secure TCP transport to clients that can initiate sessions. The TrSvr class allows servers to create TCP endpoints and accept new secure TCP sessions.

4.4 Client/Server Example

We now illustrate how a layer can represent a large-scale refinement that modifies multiple classes simultaneously. We describe the central role that nested interfaces play in building JL applications and the expressive power that nested structures bring to the JL programming model.

```
public interface CSIfc
{
 public interface Client
 {
  Address findServer();
 }

 public interface Server
 {
  boolean start();
 }
}
```

Figure 11

Complex applications can be built in a stepwise fashion in JL by using nested interfaces. The CSIfc interface in Figure 11 contains two nested public interfaces, Client and Server. Consider the following layer that exports the CSIfc interface and uses definitions developed in our previous examples:

```
layer CSBase<class CltTransport implements TransportIfc,
             class SvrTransport implements TransportIfc, PassiveIfc>
 exports CSIfc
 upflow { dispatchLoop = true; }
{
 public class Client
 {
  private CltTransport ctran;      // Private field using type parameter
  public Client(){…}               // Constructor
  public Address findServer(){…}   // Query a server location
 }

 public class Server
 {
  private SvrTransport stran;      // Private field using type parameter
  public Server(){…}               // Constructor
  public start(){…}                // Command dispatch loop
 }
}
```

Figure 12

The CSBase layer in Figure 12 provides the base layer for a client/server application. The imported type bound to the CltTransport type parameter must be a class that implements TransportIfc or a layer that exports TransportIfc. The SvrTransport type parameter is similarly constrained by both TransportIfc and PassiveIfc. The CSBase Layer provides the base implementation for the methods declared in the CSIfc interface. We can assume that the nested class constructors in this base layer would initialize their transport objects. Other layers that export CSIfc can provide application-specific function and rely on CSBase for common functionality.

Type parameters introduced by the **class** keyword, such as those in the CSBase layer, do not imply the inheritance relationship that the **mixin** keyword does. Class type parameters are simply bound at composition time and available for use within the layer body wherever a class type is permitted.

```
layer CSCmd1<mixin M implements CSIfc>
 exports CSIfc
 uptest (dispatchLoop)
{
 public class Client
 {
  public Boolean cmd1(){…}
 }

 public class Server
 {
  private void cmd1Processor{…}
 }
}
```

Figure 13

The CSCmd1 layer represents an application-specific layer that provides both the client-side method `cmd1()` and its associated server-side command processor. Though no code details are given, assume that `cmd1Processor()` is invoked from the dispatch loop in the server's `start()` method upon receipt of a client `cmd1` request. CSCmd1 uses a design rule to guarantee that another layer, such as CSBase, implements the core dispatch loop. Any number of new commands can be added in the same way to build a more complex client/server application.

```
class CS = CSCmd1[CSBase[TrClt, TrSvr]];
```

Figure 14

In Figure 14, the CS class defines a client/server application that implements the `cmd1()` function. The Transport layer's TrClt and TrSvr classes, as defined in Figure 10, are used to connect the application's front end to its back end. To create a CS server, one would instantiate an object of type CS.Server. To obtain a runtime instance of the CS client interface, one would instantiate an object of type CS.Client. Invoking the cmd1() method on the client object would cause the associated server object to execute the cmd1Processor() method. JL guarantees that the nested interfaces (Client, Server) exported by the composition's top layer (CSCmd1) correspond to public nested classes with the same names in the generated class (CS).

5 Java Layers Language

This section describes in more detail the JL language constructs and programming model introduced in the examples of the previous section. To explain the semantics of layer composition, we define the concepts of interface propagation, genericity, constructor propagation, deep subtyping, and deep interface conformance. The presentation should be of practical use to those interested in JL programming.

5.1 The Layer Definition

The simplest form of a layer definition is as follows:

layer L1 **exports** Ifc {}

The interfaces that a layer exports define the layer's *type signature.*[1] L1's type signature consists of Ifc.

layer L2<**mixin** M **implements** Ifc2> **exports** Ifc {}

The L2 layer's *imports clause* consists of everything that appears within the angular brackets. We've seen how JL uses mixins [18][32], or parameterized superclasses, to generate new class hierarchies. The class generated from the L2 layer will inherit from a superclass specified at composition time and bound to the type parameter M. L2's superclass must implement the Ifc2 interface. A layer may have at most one mixin clause. Mixins allow for stepwise refinement as described in Section 4.1.1, Composing Layers, on page 5.

A layer's type signature is automatically augmented with all interfaces specified in its mixin clause. This is called *static interface propagation* because it can be computed before composition time with just the layer definition. The L2 layer's generated class will implement Ifc, as specified in the exports clause, and the interface of the mixed-in superclass, Ifc2. *Dynamic interface propagation* takes place at composition time and ensures that all interfaces actually implemented—not just those specified as constraints in the layer definition—are reflected in a layer's type signature.

layer L3<**mixin** M **implements** MIfc1, MIfc2,
        class C **extends** CBase **implements** CIfc1, CIfc2>
  exports Ifc, Ifc2 {}

We now consider non-public constraints that may be specified using the **extends** subclause. The L3 layer imports both a mixin type parameter and a class type parameter. A class type parameter may be constrained by an implements subclause and/or an extends subclause. The type parameter, C, in the L3

[1] Remember that Java interfaces can only contain non-static public methods and public constant fields.

layer defines both kinds of constraints. A mixin type parameter must by constrained by an implements subclause and may also be constrained by an extends subclause.

An extends constraint can only be satisfied by the specified class or one of its subclasses and, therefore, can be used to specify required constructors, non-static fields, static methods, protected members, etc. In the L3 layer, only classes that extend CBase, or the CBase class itself, can be bound to the type parameter, C. In this case, the actual parameter bound to C is also constrained to implement the CIf1 and CIfc2 interfaces.

The class generated from the L3 layer's definition will implement the explicitly exported interfaces, Ifc and Ifc2, as well as the statically propagated interfaces, MIfc1 and MIfc2. Interface propagation never involves class type parameters or their interface constraints.

5.1.1 Importing Literals and Generics

JL layers can be parameterized by any primitive Java type (boolean, byte, char, double, float, int, long, short) or by strings (java.lang.String).

layer L4<**mixin** M **implements** Ifc, **int** _i, **string** _s> **exports** Ifc {}

JL will insert private instance fields _i and _s in the class generated from the above definition of layer L4. The actual parameters specified in type equations must be literals that conform to the types declared in the layer definition. The inserted fields are initialized with their actual literal values at object construction time.

layer L5<**mixin** M **implements** <T>> **exports** T {}

Layer L5 is a *generic layer* because its imports clause contains a parameterized **implements** subclause. L5 imports a single class or layer parameter as prescribed by its single mixin clause. The type parameter T will bind at composition time to any non-empty set of interfaces implemented or exported by its actual parameter. T is a *generic interface type parameter* that acts as a placeholder for a set of interfaces determined at composition time. Once T is bound, type equation processing continues as if the interface constraints were explicitly specified in the layer definition.

Layers such as L5 that mix in a generic interface type parameter can apply the same refinement to any type (layer or class) and still preserve that type's exported interface. The scope of a generic interface type parameter is the layer definition *excluding* the layer body. Generic interface type parameters can also be used in the implements clauses of class type parameters. If the same generic interface parameter appears in multiple implements subclauses in a layer definition, the leftmost instance (ignoring new lines) is bound first and all other instances are immediately bound to the same set of interfaces.

5.1.2 Layer Body Contents

The essential principle of layer programming is that each layer in a composition conceptually generates (1) a Java class in which the layer body becomes the class body and (2) a Java interface. The transformation of a layer body into a class body leaves most statements unchanged and requires that a number of naming conventions and other restrictions be observed.

All Java control flow statements, data manipulation statements, and expressions are available to the layer programmer. A programmer may add initializer blocks and *non-public* fields, methods, constructors, nested class declarations and nested interface declarations to a layer definition without restriction.

Public members of a layer are treated specially because they help define a layer's signature, its public interface. New public methods and public static final fields can be added without restriction to extend the layer's type signature, as we saw in Section 4.3. The declaration of non-constant public fields in layers is discouraged but not prohibited by JL. Such public fields are a part of a public interface that cannot be captured in Java interfaces and, therefore, cannot formally contribute to a layer's signature. Note that the

extends constraint in an imports clause can often be used with an appropriate abstract class definition to make up for this deficiency.

A layer may declare at most one public constructor, which can take any number of arguments of any type. JL accumulates formal parameters of public constructors across the mixed-in layers of a type equation. These accumulated parameters are used to create a chain of public constructor invocations in the classes generated from the layer definitions. Each class' public constructor is modified to take all the parameters it needs plus all those needed by its superclasses. This process, called *constructor propagation*, also propagates throws clauses.

5.1.3 Nested Structures

Nested interfaces and classes hold special significance in JL. The following two properties, defined by Smaragdakis [33], are required in all classes and interfaces that are imported or exported from JL layers:

- *Deep Subtyping*[2] – Type C is a *deep subtype* of another type B if C is a subtype of B, and for every publicly accessible nested type B.N, there is a publicly accessible type C.N that is a deep subtype of B.N.
- *Deep Interface Conformance* – Class C *conforms deeply* to interface I if C implements I, and for each publicly accessible nested interface I.N, there is a publicly accessible class C.N that conforms deeply to I.N.

In JL, deep subtyping applies to both Java classes and Java interfaces. This property preserves nested names within inheritance hierarchies of classes or interfaces. Layer definitions may import and export only interfaces that are deep subtypes. Classes used as actual parameters in type equations must be deep subtypes and all classes generated by JL are deep subtypes.

Deep interface conformance is also enforced by the JL compiler. Classes imported into layers must conform deeply to their implemented interfaces. Layer definitions must conform deeply to their exported interfaces as do the classes generated from those layer definitions. The JL compiler assists the layer programmer by automatically generating missing components of the required nested structure as needed.

The current implementation of JL imposes the restriction that classes and layers may not define public nested interfaces. It also restricts interfaces from defining public nested classes. These restrictions may be lifted in the future.

The restrictions on nested structures described in this section leave the vast majority of existing interface and class definition available for use in layer definitions and type equations. At the same time, these restrictions support the semantics of the JL programming model by guaranteeing the regular structure needed for stepwise refinement.

5.1.4 Layer Body References

The use of the **this** keyword in a layer body refers to the runtime object that instantiates the layer's generated class or one of its subclasses. References to the **super** keyword in a layer body resolve to an actual mixed-in superclass if one is specified or to Object if no mixin is specified.

In a composition, a layer node participates in an *inheritance chain* comprised of the node itself, all nodes recursively mixed into it, and all nodes which recursively mix it in. We've seen that classes actually generated from a layer composition implement their own corresponding inheritance chain using standard Java subclassing.

JL introduces the **thisClass** keyword so that code in layer body can refer to the type of the most refined subclass in a generated inheritance chain. The binding of **thisClass** is type equation specific. For example, the TrSvr class defined in Figure 10 generates a single inheritance chain that includes TrSvr and the classes

[2] Deep Subtyping is a slight generalization of Deep Subclassing in that it encompasses Java interfaces.

generated from the Passive, Secure and TCP layers, in subclass to superclass order. The use of **thisClass** in the bodies of any of these layers resolves to the TrSvr type. The **thisClass** keyword is semantically similar to the proposed **ThisType** construct [12][13].

Finally, all regular type parameters imported into a layer can appear in the layer body wherever class types are appropriate, including allocation statements. Layer programmers are also free to read and write any imported literal fields declared by the layer.

5.1.5 Design Rules

We have seen how a large number of layer compositions are possible given even a small number of compatible layers. Easy layer composition would be impossible if an understanding of code level interactions between layers was required for each possible combination. Layers should be treated as black boxes during feature selection and composition. Import/export type checking provides basic syntactic safeguards, and design rules support the semantic checking needed to detect and restrict invalid feature combinations [2][27]. The simple design rules illustrated in Section 4.2 were used to enforce composition constraints that would be difficult if not impossible using type checking alone.

The JL design rule sublanguage is still actively being researched and is not currently implemented. Our first approximation is an imperative style language, though we are looking into more declarative approaches, including those using pattern matching. We are also considering ways to avoid scattering design rule information among multiple layers. Work continues on various approaches of associating design rules variables with standard Java classes, incorporating the correct abstractions into the constraint language, standardizing the set of built-in functions, and providing meaningful error reports when a semantic check fails.

As currently conceived, design rules are specified in layer definition clauses using the **uptest**, **upflow**, **downtest** and **downflow** keywords. Two separate flows are modeled; the upflow and the downflow are independent streams of variables, values and tests. In any type equation, all tests on both flows must evaluate to true in order for the layer composition to proceed.

The algorithm below is used to evaluate each flow in a type equation independently. The algorithm terminates successfully when the last layer in the flow has been processed or terminates with failure if any boolean test clause returns false.

<u>Design Rule Flow Algorithm</u>

1. Evaluate the boolean test expression, if false return failure.
2. Evaluate flow statements.
3. Propagate remaining variables.
4. Go to step 1 for next layer if it exists else return success.

The design rule language uses a simplified Java syntax to support constraint definition and checking. Design rule variables can be added, removed, modified or checked for existence in a flow. Variables are typed and always bound to a non-null value. Constraint testing (step 1 in the flow algorithm) is side-effect free.

Type inference is used to create and initialize variables if they don’t already exist in the flow. Methods, interfaces and classes cannot be defined, though records allow programmers to structure their data. Relational, logical and arithmetic expressions are supported. Most Java control flow statements are available along with a small number of additional constructs and built-in functions.

Space limitations prohibit describing the design rule language in full, but we will use the following example to convey the basic idea:

```
layer L6<mixin M implements Ifc>
 exports Ifc
 uptest (i > 2 || i < 0)
 upflow {i--; sarray = {“string1”, “string2”};}
 downtest (rec1.i > 0)
 downflow nopropagate {rec2 = {j = 1, bool = false};}
{}
```

Variables can be of type int, boolean, char, string, and arrays and records of these types. Layer L6’s uptest will only return true if the variable `i` exists in the upflow, is of type integer, and satisfies the boolean expression. If the uptest succeeds, the upflow clause decrements i and creates or replaces the string array named `sarray`.

Layer L6’s downtest checks the value of an integer field in the record `rec1`. If this succeeds, the downflow clause uses the nopropagate option to discard all current variables on this flow and then creates a new record variable with integer and boolean fields.

Variables that exist after the upflow or downflow clauses execute are propagated to the next layer in the flow. Since layers may import multiple types, the design rule language and implementation must accommodate the merging of upflows and the splitting of downflows [2].

5.2 Aliases

JL allows layer compositions to be assigned a name or alias using the **layerdef** keyword. The **importlayerdef** keyword allows aliases defined in other files to be visible in the importing file. Aliases are macro-expanded by the JL compiler when they are encountered in type equations or layerdefs.

```
layerdef TRANPORT  KeepAlive[Secure[TCP]];

layerdef SERVER_TRANPORT  Passive[TRANSPORT];

importlayerdef mydir.myfile;
```

6 JL Compiler Optimization

JL optimizes the class hierarchy generated from an extended class definition through a process of class integration and a type of semantic expansion [37]. Each generated inheritance chain is collapsed into a single class. The associated interface definitions are similarly collapsed. This reduces JVM load time and runtime memory overhead by minimizing the number of types.

Before classes are collapsed, superclass methods called from subclass methods with the same signatures are usually inlined into the subclass call site(s). This optimization improves runtime performance by eliminating method call overhead for methods that perform stepwise refinement as described in Section 4.1.1 on page 5. Unreachable methods in the inheritance chain are also detected and eliminated at composition time.

Together, the class and method optimizations can be referred to as *class flattening*. Similar optimizations implemented in previous GenVoca generators have proven effective with performance comparable to that of hand-optimized code [22]. Intermediate classes and methods are eliminated during class flattening and, therefore, cannot be guaranteed to exist. After optimization, JL guarantees only that the class generated by the top layer in each inheritance chain (i.e., the layer that generates the most refined subclass in a chain) will be present.

7 Related Work

GenVoca. JL’s lineage includes GenVoca domain-specfic [3][31][5][6] and domain-independent [30][32] implementations. JL departs from earlier GenVoca work in the central role that interfaces play in the code generation process, the flexibility with which these interfaces can be automatically augmented, the emphasis placed on simplifying layer construction, the language support for semantic checking, and the emphasis placed on generating efficient, optimized code in a domain-independent setting.

Object-Oriented Frameworks. Object-oriented frameworks, especially when used in conjunction with design patterns, represent the current state of the art for building large applications and software product lines with standard programming languages [28][29][16]. A number of framework problems have been documented [15][17], including overfeaturing, code replication, and the feature combinatorics problem as described in the Motivation section. JL alleviates some of the problems associated with frameworks and their evolution by eliminating the distinction between framework and application instance [7]. JL augments the syntactic type checking usually found in frameworks with high-level semantic checking. To their credit, frameworks can be developed using any modern object-oriented language without extension.

Domain-Specific Application Generators. Many domain-specific application generators or development environments exist in today's marketplace. Products like Microsoft's Visual Basic GUI builder, the SAS Institute's data analysis suite, or SAP AG's enterprise software can provide application solutions in specific domains. These products can take advantage of their restricted domains to generate efficient code. Application generators, however, often have limited flexibility and openness for technical or proprietary reasons. JL's building block approach is extensible and JL may be applied to any domain that's appropriate for Java. In theory, domain-specific application generators should always be able to generate more efficient code than a domain-independent approach such as JL.

Parameterized Types. JL derives its compositional power from the use of supertype parameterization (mixins) in conjunction with nested classes and interfaces. Currently, Java does not support parameterized types of any kind and extending Java in such a manner is an active area of research [1][11] [24][36]. Since the problem of integrating generics into the Java type system is largely orthogonal to research into large application development, JL supports genericity only within its own layer construct. Native Java genericity, if it becomes available and includes support for mixins, would be a natural basis for a future version of JL.

Experimental Programming Models. JL is one of a number of research efforts that propose a new model of programming to address fundamental software engineering issues. These proposals tend to follow the historical trend of raising the level of programming abstraction to attain better design and code modularity. Examples include subject-oriented programming and hyperslices [21][35], meta-class programming [19], aspect-oriented programming [23], mixins [18], and composition filters [8]. JL distinguishes itself by composing software components using malleable interfaces—the interface actually exported by any component (layer) depends on the composition in which it's included. Unlike most other proposals, JL addresses higher-level compositional correctness by supporting programmer directed semantic checking.

8 Conclusion

We have presented the JL language and shown how complete design features can be encapsulated within a single layer definition. Layer programming is comparable to standard Java class programming, and most existing Java interface types can be used as is in layer definitions. Large applications and software product lines can be built by composing sets of layers that implement precisely the features required in a particular execution environment. Application maintenance is also reduced to an exercise in feature selection. We saw how JL's preliminary version of design rules provide a way for programmers to specify semantic constraints on layer compositions. Based on past experience with GenVoca generators, JL's class hierarchy optimization should produce efficient object-oriented code.

JL alleviates some of the problems associated with object-oriented frameworks and their evolution. By eliminating the rigid distinction between framework code and application code, JL generates applications with only the code actually required. The ability to precisely customize applications for their execution environment reduces the complexity and size of individual applications. The ability to mix and match features on demand allows JL applications to be flexibly configured without imposing runtime overhead.

The basic functionality of the JL compiler has been implemented. JL is being applied in a distributed application environment to test scalability and the applicability of various language features. We expect to feed changes back into JL as we gain more development experience. This experience will also help us understand what new design methodologies are best suited to the JL programming model. Besides investigating the design rule topics mentioned in Section 5.1.5, we are also considering the use of pattern matching in applying the same change to multiple methods, determining the need for *sublayering*, and looking for opportunities to apply more extensive compiler optimizations.

We would like to thank for Kartik Agaram, David Fink, Sam Guyer, Malcolm Haynes, Daniel Jimenez, Dan Tecuci, Thomas Wahl, and Peter Yeh for their comments and lively discussions concerning JL.

9 References

[1] O. Agesen, S. Freund and J. Mitchell. *Adding Type Parameterization to the Java Language.* OOPSLA 1997.

[2] D. Batory and B. Geraci. *Composition Validation and Subjectivity in GenVoca Generators*. IEEE Transactions on Software Engineering (special issue on software reuse), February 1997.

[3] D. Batory and S. O'Malley. *The Design and Implementation of Hierarchical Software Systems with Reusable Components.* ACM Transactions on Software Engineering and Methodology, Vol. 1, No. 4, Oct. 1992.

[4] D. Batory, V. Singhal, M. Sirkin and J. Thomas. *Scalable Software Libraries.* Proceedings of the First ACM Symposium on the Foundations of Software Engineering, December, 1993.

[5] D. Batory and J. Thomas. *P2: A Lightwieght DBMS Generator.* Technical Report TR-95-26, Department of Computer Sciences, University of Texas at Austin, June, 1995.

[6] D. Batory, B. Lofaso, and Y. Smaragdakis. *JTS: Tools for Implementing Domain-Specific Languages.* 5th International Conference on Software Reuse, June 1998.

[7] D. Batory, R. Cardone and Y. Smaragdakis. *Object-Oriented Frameworks and Product-Lines.* To be presented at the First Software Product Line Conference, August 2000, Denver, Colorado. Sponsored by the Software Engineering Institute at Carnegie Mellon University.

[8] L. Bergmans. *Composing Concurrent Objects.* Ph.D. dissertation, University of Twente, June, 1994.

[9] J. Bosch. Product Line Architectures in Industry: A Case Study. International Conference on Software Engineering, 1999.

[10] G. Bracha and W. Cook. *Mixin-Based Inheritance*. Proceeding of OOPSLA-ECOOP 1990, ACM SIGPLAN Notices, Vol. 25, No. 10, 1990.

[11] 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.

[12] K. Bruce. *Increasing Java's expressiveness with ThisType and match-bounded polymorphism.* Technical Report, Williams College, 1997, http:// www.cs.williams.edu./~kim/README.html.

[13] K. Bruce, M. Odersky and P. Wadler. *A statically safe alternative to virtual types.* European Conference on Object-Oriented Programming, 1998.

[14] R. Cardone. *On the Relationship of Aspect-Oriented Programming and GenVoca.* Ninth Annual Workshop on Software Reuse, January, 1999.

[15] W. Codenie, K. De Hondt, P. Steyaert and A. Vercammen. *From Custom Applications to Domain-Specific Frameworks.* Communications of the ACM, Vol. 40, No. 10, October 1997.

[16] D. Doscher and R. Hodges. *Sematech's Experience with the CIM Framework.* Communications of the ACM, Vol. 40, No. 10, October 1998.

[17] M. Fayad and D. Schmidt. *Object-Oriented Appplication Frameworks.* Communications of the ACM, Vol. 40, No. 10, October 1997.

[18] M. Flatt, S. Krishnamurthi and M. Felleisen. *A Programmer's Reduction Semantics for Classes and Mixins.* Technical report TR-97-293, Department of Computer Sciences, Rice University.

[19] I. Forman and S. Danforth. *Putting Meta-Classes to Work.* Addison Wesley Longman, Inc., October 1998.

[20] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[21] W. Harrison and H. Ossher. *Subject-Oriented Programming (A Critique of Pure Objects).* OOPSLA 1993.

[22] G. Jimenez-Perez and D. Batory. *Memory Simulators and Software Generators.* ACM Proceedings of the Symposium on Software Reusability, 1997, Boston.

[23] G. Kiczales , J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier and J. Irwin. *Aspect-Oriented Programming.* Proceedings of the European Conference on Object-Oriented Programming, June 1997.

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

[25] A. Myers, J. Bank and B. Liskov. *Parameterized Types for Java.* ACM Symposium on Principles of Programming Languages, 1997.

[26] D. L. Parnas. *On the Criteria to be Used in Decomposing Systems into Modules.* Communications of the ACM, 15(12):1053-1058, December 1972.

[27] D. E. Perry. *The Inscape Environment*. Proceedings of the Eleventh International Conference on Software Engineering, May 1989.

[28] B. Rubin, A Christ and K. Bohrer. *Java and the IBM San Francisco Project.* IBM Systems Journal, Vol. 37, No. 3, 1998.

[29] D. Schmidt. *An Architectural Overview of the ACE Framework: A Case-study of Successful Cross-platform Systems Software Reuse.* USENIX login magazine, Tools special issue, November, 1998.

[30] V. Singhal and D. Batory. *P++: A Language for Large-Scale Reusable Software Components.* Proceedings of the 6th Annual Workshop on Software Reuse, November, 1993.

[31] M. Sirkin, D. Batory and V. Singhal. *Software Components in a Data Structure Pre-Compiler.* Proceeding of the 15th International Conference on Software Engineering, May 1993.

[32] Y. Smaragdakis and D. Batory. *Implementing Layered Designs with Mixin Layers.* European Conference on Object-Oriented Programming, 1998.

[33] Y. Smaragdakis. *Implementing Large-Scale Object-Oriented Components.* Ph.D. dissertation, University of Texas At Austin, Department of Computer Sciences, December 1999.

[34] B. Stroustrup. *The C++ Porgramming Language, 3rd Edition*. Addison-Wesley, 1997.

[35] P. Tarr, H. Ossher, W. Harrison and S. M. Stanley. *N Degrees of Separation: Multi-Dimensional Separation of Concerns.* Proceedings of the International Conference on Software Engineering, May 1999.

[36] K. Thorup. *Genericity in Java with Virtual Types.* Proceedings of the European Conference on Object-Oriented Programming, June 1997.

[37] P. Wu, S. Midkiff, J. Moreira and M. Gupta. *Efficient Support for Complex Numbers in Java.* Proceedings of the ACM Conference on Java Grande, 1999.