Comparing Frameworks and Layered Refinement

Richard Cardone and Calvin Lin
Department of Computer Sciences
University of Texas at Austin
Austin, TX, 78712 USA
{richcar, lin}@cs.utexas.edu

ABSTRACT
Object-oriented frameworks are a popular mechanism for building and evolving large applications and software product lines. This paper describes an alternative approach to software construction, Java Layers (JL), and evaluates JL and frameworks in terms of flexibility, ease of use, and support for evolution. Our experiment compares Schmidt's ACE framework against a set of ACE design patterns that have been implemented in JL. We show how problems of framework evolution and overfeaturing can be avoided using JL's component model, and we demonstrate that JL scales better than frameworks as the number of possible application features increases. Finally, we describe how constrained parametric polymorphism and a small number of language features can support JL's model of loosely coupled components and stepwise program refinement.

Keywords
Frameworks, parametric polymorphism, mixins, layers.

1 INTRODUCTION

Surveys show that nearly three quarters of all large software projects are cancelled, over budget, or late [20]. To address this problem, various methods of reusing code and reducing design complexity have been proposed. In terms of reusing both code and design to build large applications, object-oriented frameworks [2,17,27] represent the current state of the art when using general-purpose programming languages. Frameworks are *starter kits* that use abstract classes to provide partially implemented applications. Different applications can be created from a single framework by providing different implementations of these abstract classes, so frameworks are ideal for supporting *software product lines*, which are families of related software products.

This paper introduces a language-based alternative to frameworks called Java Layers (JL). JL [12] is an extension of Java that supports a layered software component model. Like frameworks, JL can be used to provide partially implemented applications. Unlike frameworks, starter kits in JL consist of a set of components, or *layers*, that are then composed to generate applications. The key idea behind the JL component model is that each layer encapsulates exactly one *design feature*, which is a high-level requirement that defines some application attribute or capability. This one-feature-per layer property maximizes code reuse since each feature is implemented only once. This property also facilitates the composition of layers, making it easy to include or exclude individual features. Finally, this property preserves modularity in terms of both code and design.

To compare JL against object oriented frameworks, we use JL to re-engineer the Adaptive Communication Environment (ACE), an object oriented framework developed in C++ by Schmidt and colleagues [28]. ACE is a well-documented, well-engineered framework that has been used in dozens of commercial and academic applications. Thus, ACE represents proven and mature framework technology and provides a standard against which new technologies can be measured. In this paper, we compare application development in ACE and in JL using the following qualitative measures:

Usability – How easy is it to develop applications?
Application Flexibility – How easy is to customize applications?
Starter Kit Flexibility – How easy is it to evolve the starter kit?

Contributions
This paper makes the following contributions.

1. We present the first experimental comparison of JL's component model against a large, mature, object-oriented framework (ACE).

2. Compared to frameworks, we describe how JL employs simpler, more precise interfaces that reduce memory overhead, runtime overhead, and code complexity. We also show how JL provides better support for evolution, and how JL avoids the framework problem of overfeaturing.

3. We briefly describe JL's novel features, which enhance usability and efficiency, and how these

features can be integrated into Java.

This paper proceeds as follows. Section 2 explains the JL component model, and Section 3 describes the foundation of the JL language. Section 4 provides context by sketching the ACE architecture and its key design patterns. Section 5 then uses ACE to compare JL against frameworks. In Section 6, we describe novel JL features that simplify component-based programming. Finally, we present related work and conclusions.

2 THE JL COMPONENT MODEL

JL is based on the GenVoca software component model [7]. This component model encourages a programming methodology of *stepwise refinement* in which types are built incrementally in layers. Stepwise refinement is important because it allows design features to be *mixed and matched*, allowing applications to be flexibly and precisely customized.

Another advantage of stepwise refinement is that it solves the *feature combinatorics* problem [10]. For 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. With stepwise refinement, only those feature combinations that are needed are materialized.

The key to stepwise refinement is the use of components, called *layers*, that encapsulate the complete implementation of a single design feature. This encapsulation often includes code that would be packaged separately using today's programming language technologies. For example, a layer in JL can contain Java code for multiple methods or even multiple classes, as we briefly describe in our discussion of *deep conformance* in Section 6.

Once layers have been defined, the features that they encapsulate can be composed if the layers have compatible interfaces. Layers *export* an interface and *import* zero or more interfaces. New types are defined by matching the exported interface of one layer to the imported interface of another layer.

To see how layer composition works, consider interface `TransportIfc`, which declares methods `send()` and `recv()`:

```
interface TransportIfc {
  send(Data d);
  recv(Data d);
}
```

Assume that three layers use this interface: The `TCP` layer provides data transport using TCP; the `Secure` layer provides data encryption/decryption; and the `KeepAlive` layer automatically exchanges liveness notifications between communicating peers. Assume that all three layers export the `TransportIfc` interface and that `Secure` and `KeepAlive` also import `TransportIfc`. The declaration below of variable `trans` uses a new type defined by composing these three layers:

```
KeepAlive<Secure<TCP>> trans;
```

We say that the type of `trans`, which implements a secure TCP transport with the automatic keep-alive feature, is *generated* in the above composition. This generated type implements `TransportIfc` because that is the interface exported by the leftmost, or *top*, layer in the composition stack.

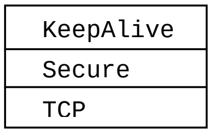

Figure 1 - Transport Layer Composition

Layers in a composition can be thought of as stacked virtual machines that perform feature-specific processing (see Figure 1). Though we haven't shown method implementations, we can walk through a hypothetical invocation of the `send()` method to illustrate this idea of virtual machines. When `trans.send()` is invoked, the `KeepAlive` layer at the top of the stack gets control first. `KeepAlive's send()` simply calls the `Secure` layer's `send()`. The `Secure` layer then encrypts the message and invokes the `TCP` layer's `send()` to transmit the encrypted data. The ordering of layers is important in this scheme—if the `KeepAlive` and `Secure` layers were reversed, then liveness messages would be sent in the clear rather than encrypted.

To demonstrate the flexibility of stepwise refinement, we could now create a new layer, `UDP`, which also exports `TransportIfc` and is analogous to the above `TCP` layer. This new layer could be composed with the `Secure` layer to create a secure UDP transport type. In this way, features are easily selected and composed to create new types.

3 JL'S FOUNDATION

We now introduce JL, which implements the component model just presented. Since layers can be viewed as type parameters in compositions, constrained parametric polymorphism [11] is a natural implementation choice for our component model. In this section, we describe the parametric implementation of Java that serves as JL's foundation. Language features built on top of this foundation, some of which can be applied as standalone features outside of JL, are described in Section 6.

Layer composition in Java Layers is based on the use of *mixins* [3,25]. Mixins are types whose supertypes are parameterized. Mixins are not supported in standard Java, but are available in some languages that support parameterized polymorphism such as C++ [32]. In this section, we describe how mixins support reuse and how they serve as a basis for JL.

Mixins are useful because they allow multiple classes to be specialized in the same manner, with the specializing code residing in a single class definition. For example, suppose we wish to extend three unrelated classes–`Car`, `Box` and `House`–to be "lockable" by adding two methods, `lock()` and `unlock()`. Without mixins, we would define subclasses of `Car`, `Box`, and `House` that each extended their respective superclasses with the `lock()` and `unlock()` methods. The lock code would be replicated in three places. With mixins, we would instead write a single class called `Lockable` that could extend any superclass, and we would instantiate the `Lockable` class to extend `Car`, `Box`, and `House`. The `lock()` and `unlock()` methods would only be defined once. In JL syntax, the `Lockable` mixin would be defined as follows:

```
class Lockable<T> extends T {
 public lock(){…}
 public unlock(){…} }
```

We base JL's implementation on a parametrically polymorphic Java with mixin support. Adding parametric polymorphism to Java is both feasible and desirable, and a number of good solutions have been proposed [1,4,14,22, 24]. The best fit for JL is an extension that supports constrained parametric polymorphism and mixins [1]. To simplify our discussion, we assume such an extended Java exists and we discuss JL in terms of it. This separates the problem of integrating parameterized polymorphism into Java from the problem of supporting JL's programming model, allowing us to concentrate on the latter.

Programming with mixins, however, does have a number of drawbacks. We defer a deeper discussion of mixins until Sections 5 and 6, where we describe additional JL language features that enhance support for our component model.

JL Syntax

We now describe JL syntax that is compatible with most proposals for parameterizing Java, though the current JL implementation [12] uses a different notation. *Layers in JL are simply Java types*, so we will use the terms classes and layers interchangeably in this paper.

Continuing our Transport example from Section 2, we sketch three layer definitions below:

```
class TCP implements TransportIfc {…}
class Secure<T implements TransportIfc>
  extends T {…}
class KeepAlive<T implements TransportIfc>
  extends T {…}
```

The `TCP` class is a standard, non-parameterized class. The `Secure` and `KeepAlive` classes are mixins that inherit from their type parameter, `T`. In both classes, type parameter `T` is constrained by `TransportIfc`—any instantiation of either `Secure` or `KeepAlive` requires an actual type parameter that implements the `TransportIfc` interface. JL also supports parameterized interfaces, F-bounded polymorphism [10], and class constraints on type parameters using the `extends` clause. *Instantiations* of parametric types take the conventional form:

```
KeepAlive<Secure<TCP>> trans;
class TP extends KeepAlive<Secure<TCP>> {}
```

The first statement above declares a variable, `trans`, with an *instantiated* type. We also say that JL *composes* or *generates* this type. The second statement is an idiom used to name an instantiated type, `TP` in this case. In both statements, the use of mixins generates a new class hierarchy with parent `TCP`, child `Secure` and grandchild `KeepAlive`. The second statement also creates the class `TP` as a subclass of `KeepAlive`.

Aside from its support for mixins, we see from this brief description that JL is built upon a fairly standard implementation of constrained parametric polymorphism for Java. We now introduce the ACE framework and then our experiment that re-engineers ACE using mixins.

4 ACE FRAMEWORK

Schmidt and colleagues developed the Adaptive Communication Environment (ACE) [27,28] as a C++ framework for constructing client/server applications. ACE implements a core set of concurrency and distribution design patterns that provides an infrastructure for building customized applications. In general, C++ applications built using ACE require less effort to develop and exhibit greater flexibility, reliability and portability than C++ applications built using ad-hoc methods.

ACE is implemented in three broad layers [33]. The System Adaptation layer provides operating system portability. The System Services layer provides an object-oriented interface to the Adaptation layer. The Distributed Design Patterns layer implements collaborations useful in distributed applications. In this section, we briefly describe some of the services and design patterns essential to building client/server applications using ACE.

System Services

ACE provides a Timer interface and a set of concrete classes that allow applications to create, schedule, cancel, and expire timers. Timers can be reoccurring and can be stored in specialized data structures for efficient access. ACE also provides Message Queues modeled after those found in UNIX System V [31].

Task

The ACE Task (see Figure 2) is a design pattern for asynchronous processing. In its simplest form, an ACE Task is an object-oriented encapsulation of zero or more threads that perform application-specific work. A Task also contains a Message Queue to store client requests for later processing by the Task's worker threads.

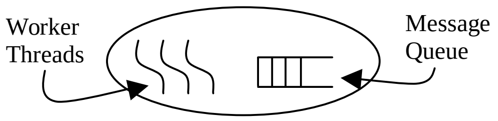

Figure 2 - ACE Task Object

The Task interface includes methods to initialize, activate and terminate a Task. Worker threads execute a virtual call-back method whose implementation is supplied by the user through subclassing. Tasks communicate by queuing requests on each other's Message Queues.

Reactor
The ACE Reactor [30] implements a design pattern for concurrent event dispatching among multiple clients. Clients, who implement the Event Handler interface, register interest in particular events monitored by the Reactor. When an event occurs, the Reactor issues a callback to the appropriate method in registered client objects. Figure 3 shows that Reactors can monitor multiple event sources, including timers, I/O ports, operating system signals, and application level notifications.

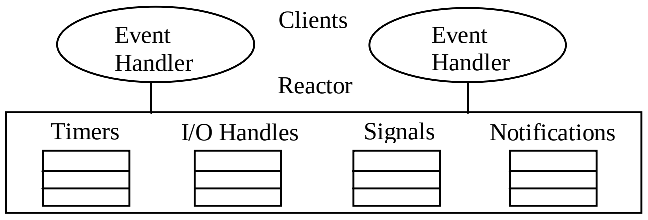

Figure 3 - ACE Reactor and Client Objects

The Reactor interface supports static methods that provide access to a default Reactor instance, as well as methods to create and manage multiple Reactors. Other methods allow clients to register, cancel, suspend and resume interest in events of all types.

Acceptor/Connector
The ACE Acceptor/Connector [29] design pattern decouples session establishment and initialization from application processing in a distributed environment. The pattern also abstracts the underlying transport stream so that different types of streams, such as TCP, Unix sockets, and pipes, can be substituted for one another. Acceptors and Connectors are factory classes [21] that come in complementary pairs: Acceptors handle the passive side of session initiation and Connectors handle the active side. These factory classes orchestrate a session initiation protocol by creating and invoking the other classes that participate in the collaboration.

Collaborators in the Acceptor subpattern are the Acceptor factory itself, a concrete stream-acceptor, a Service Handler, and a Reactor. Similarly, collaborators in the Connector subpattern are the Connector factory, a concrete stream-connector, a Service Handler, and a Reactor. Service Handlers are ACE Tasks that implement the Event Handler interface and have a stream field. Concrete acceptors and connectors provide passive and active session initiation for specific types of transport streams.

The three-phase Acceptor protocol is illustrated in Figure 4. Each Reactor notification is preceded by an appropriate event registration (not shown). The Acceptor factory directs the first two phases of the protocol, the connection initialization and service initialization phases. The Acceptor has no role in the third phase in which the Service Handler communicates independently with its peer, using the Reactor as needed. The three-phase Connector protocol is defined similarly. Both protocols can be customized by overriding methods that implement each phase.

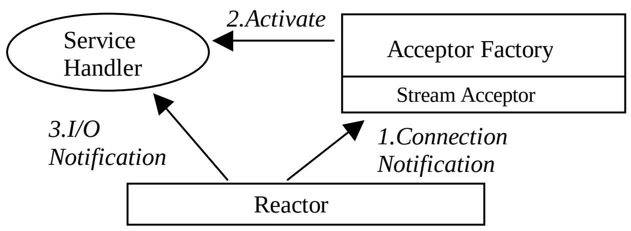

Figure 4 - Acceptor Collaboration

5 COMPARING JL AND FRAMEWORKS

Both JL and frameworks rely on interfaces defined during domain analysis to guide the development process. Both approaches provide starter kits of partially assembled applications, but they differ in the way in which applications are created. Frameworks provide partially assembled applications that use interfaces to define *variation points*; programmers then create applications by supplying concrete classes at all variation points. JL uses interfaces to define groups of interchangeable components that programmers then compose to build complete applications. In this section, we compare these two approaches using the three measures described in the Introduction: usability, application flexibility, and starter kit flexibility.

To compare JL against frameworks, we used JL to re-engineer a subset of ACE that captures the sophistication of the original. Thus, we implemented the primary design patterns found in ACE necessary for building ACE-style client-server applications, but we typically did not implement all of the features in an ACE class. The result is a few thousand lines of JL code that delivers a deep slice through ACE' s layered architecture, from the application interface down to the network protocols. While our system does not come close to replicating all the function of ACE' s 125K lines of code, missing functionality can be added by writing additional layers that are conceptually identical to those we have already written.

```
class TimerExtensible<T extends TimerAbstract implements TimerIfc,
                      U implements TimerSortedMapIfc> extends T {…}
class Timer1 extends TimerExtensible<TimerAbstract, TimerTreeMap> {}
```

Figure 5 – Simple JL Timer

For the purpose of comparing development techniques, a complete and exact replication of ACE is not necessary. For example, our implementation uses the standard Java sockets library, which does not support a multiple port I/O call like Unix *select()* [31]. We simulate this capability by using a thread for each port, which is clearly undesirable in real-world applications, but sufficient for studying the structure of JL applications built using ACE design patterns.

We also ignore differences between JL and ACE that stem from disparities between Java and C++. For instance, many ACE classes explicitly declare synchronization parameters and methods to manage concurrency. In JL, this function is largely handled by Java's built-in multithreading support. Similarly, small differences in function, such as support for tracing and inspection during debugging, are also factored out of the comparison.

All of the services and design patterns described in Section 4 have been implemented in JL. Throughout this paper, all ACE C++ classes are prefixed with "`ACE_`." JL classes and interfaces have unprefixed names, though all JL interfaces carry the "`Ifc`" suffix.

ACE and JL Implementations

To provide a concrete basis for comparing JL and ACE, we now discuss the details of the two implementations. We focus on the Timer and Task design patterns, which are representative of how all ACE patterns are implemented in JL: We start with an ACE interface, decompose it into several smaller JL interfaces, and then implement these interfaces in single-feature JL layers. ACE code is described, but not shown, due to its conventional nature.

Timer

In ACE, the C++ class `ACE_Timer_Queue_T` defines the complete Timer public interface. The interface includes methods to schedule, cancel and expire timers; to retrieve and remove the next timer; to calculate the time until the next timer pop; to manage time skew; and to set the time-of-day source. Protected methods are also defined. Classes that implement this interface support all methods.

By contrast, the base JL timer interface, `TimerIfc`, (not shown) declares only four `schedule()` methods. Figure 5 shows the structure of the basic JL timer class, `TimerExtensible`, that implements this interface and takes two type parameters. The first type parameter requires a subclass of `TimerAbstract` that implements the `TimerIfc`. This type parameter is mixed in as the superclass. The second type parameter implements the `TimerSortedMapIfc` interface, which provides a container for timer objects. `Timer1` illustrates a simple use of `TimerExtensible` appropriate for applications that only schedule timers.

In JL, advanced timer features are encapsulated in their own parameterized classes for easy composition. Figure 6 shows the `TimerCancelByTime` class that supports timer cancellation. This class inherits from its type parameter, `T`, which is constrained to implement `TimerIfc`. All instantiations of `TimerCancelByTime` implement interfaces `TimerIfc` and `TimerCancelByTimeIfc`. Features that support query, expiration and other optional operations are defined in a similar way using mixins and constrained type parameters. `Timer2` illustrates a timer that supports both cancellation and query (not shown).

Task

In ACE, the C++ template class `ACE_Task` defines the complete Task public interface. The interface includes public methods to activate and manage threads; to initialize, read, write and manage a Message Queue; and to manage Tasks in the context of a Module. ACE Modules are bi-directional message streams made up of pairs of Tasks.

The JL Task interface is defined in `TaskIfc` and declares only thread activation methods. As with Timers, auxiliary interfaces are defined to support optional features. For example, the `TaskQueueIfc` interface supports Message Queue operations and the `TaskInterruptIfc` interface supports the interruption of threads. Again, features are mixed and matched to customize Tasks as needed.

```
class TimerCancelByTime<T extends TimerAbstract implements TimerIfc>
  extends T implements TimerCancelByTimeIfc {…}
class Timer2 extends
  TimerCancelByTime< TimerExtensible<TimerAbstract, TimerQueryId<TimerTreeMap>> > {}
```

Figure 6 – Complex JL Timer

Interfaces
To understand the differences between JL and ACE, it is crucial to understand how interfaces are used in the two approaches. JL's `TimerIfc` interface is *narrow* because it contains four methods and supports only the most rudimentary features used by almost all applications that require timers. Other narrow interfaces are used to declare optional features whose implementations can be composed.

By contrast, ACE Timers use a one-size-fits-all approach and implement all possible features in every Timer class. Thus, the *wide* `ACE_Timer_Queue_T` interface supports a large number of features, many of which are not needed in most applications. For example, the interface declares 20 methods, some exposing functors and iterators that are not commonly used. In the Analysis Section, we argue that wide interfaces do not stem from poor design, but rather represent an unavoidable technology-based tradeoff.

To summarize, ACE uses a small number of wide interfaces, while JL uses a larger number of narrow interfaces. For each ACE interface used in our experiment, Table 1 shows the number of declared methods, the number of narrow JL interfaces produced, and the average number of methods in the JL interfaces.[1]

	Timer	Queue	Task	Reactor	Acc.	Conn.
ACE Width	20	24	15	66	5	5
No. of JL Interfaces	13	13	10	27	3	4
Avg. JL Width	1.5	1.8	1.5	2.4	1.7	1.3

Table 1 - ACE and JL Interfaces

Comparison

In this section, we compare ACE and JL using the three measures described in the Introduction.

Usability
How easy and effective is software development using the two approaches? We answer this question by comparing interface usage in JL and ACE.

ACE's wide interfaces are more complex and therefore harder to use than JL's narrow interfaces. Wide interfaces not only require users to learn more methods, but the methods themselves sometimes take more parameters. For example, the `ACE_Task` constructor takes a Message Queue parameter, thereby forcing all Task users to understand something about queuing. In JL, the Message Queue type does not appear in Tasks that do not implement the Message Queue feature.

[1] Factoring out differences between C++ and Java.

The use of narrow, less complex interfaces in JL also leads to smaller executables. We saw how JL Timer classes could easily be constructed with the exact set of features required by an application and no more. ACE Timers, on the other hand, have uniformly large executables because of the width of the interface that they must support.

JL's narrow interfaces can also lead to lower execution overhead. For example, JL Tasks that don't implement `TaskQueueIfc` avoid the overhead of allocating and initializing a Message Queue, costs incurred by every ACE Task.

JL's ability to precisely customize code to its application environment leads to simpler interfaces and smaller, faster implementations. All these characteristics increase the likelihood that JL code will meet the needs of application programmers and, as a consequence, be used.

In terms of maintenance, there is a tradeoff between the number and size of interfaces. An excessive number of small interfaces in JL could be just as unmanageable as excessively large interfaces in frameworks. In our experiment, however, we found that reasonable interface design avoids the worst-case management problems in both JL and ACE.

Finally, while frameworks apparently give programmers more functionality by providing partially assembled applications, JL can do the same by delivering predefined or canned layer compositions. These canned compositions can even be packaged as frameworks.

Application Flexibility
To what extent do ACE and JL allow applications to be constructed with precisely the desired set of features?

The use of wide interfaces in ACE means that any implementation of a service, such as the Timer service, must support all possible methods. In addition, applications that use these services do not have the ability to pick and choose optional features, though new optimization techniques may remove unused code from the application after the fact [35].

On the other hand, the use of narrow interfaces in JL allows each optional feature to be implemented in its own class. These optional features can then be composed to yield a great variety of customized types for use in applications. Table 1, for example, shows that any of 27 separately implemented Reactor features can be used to generate a Reactor. This yields 2^{27} possible feature combinations, even if we assume no duplicates and a total ordering among features. In JL, we compose optional features on demand rather than in advance, allowing JL to avoid the feature combinatorics problem described in Section 2.

Starter Kit Flexibility
This section compares the ability of JL and frameworks to support changes to their starter kits. We first consider how

the two approaches support evolving client needs. We then discuss the more specific issue of adding features to the starter kit.

Evolving Client Needs

A well-designed framework strikes a balance between what to include in the framework and what to exclude. The framework will ideally include all code that is common across many applications. If the framework includes too many features, the interface becomes overly complex and the framework becomes less usable. If the framework omits commonly needed code, multiple applications will have to implement the missing features independently. These problems are commonly referred to as *overfeaturing* and *code replication*, respectively [15].

As well designed as ACE is, it still exhibits overfeaturing and code replication. For instance, `ACE_Reactor` includes methods that support the singleton design pattern [21], which is useful in applications that require only one Reactor, but which is confusing in applications that use multiple Reactors. Thus, what is appropriate for one application may appear to be overfeaturing to another. On the other hand, ACE does not support authentication, authorization or data privacy. Unless the ACE framework is updated, each application requiring security must independently develop its own network security solution outside of the framework.

The problems of overfeaturing and code replication are rooted in the fundamental and somewhat rigid distinction that all frameworks make between framework code and application code [5]. Deciding what to include in a framework is always a compromise based on domain knowledge and the requirements of future users, both of which are likely to change over time.

By contrast, JL promotes code reuse with its ability to selectively mix and match features. JL classes are grouped according to the interfaces they implement. Adding a new capability to a set of starter kit classes usually has minimal impact because of the loose coupling between classes and the orthogonal nature of feature implementations. Adding new starter kit classes is no different than adding application classes.

Adding Features to the Starter Kit

Suppose that a framework needs a new feature that requires changes to its core classes. One approach is to modify existing framework classes while maintaining backward compatibility as much as possible. This approach is not feasible if currently supported applications are intolerant of changes in their binary representation. Applications that store objects persistently or that are conservatively managed for safety reasons often fall into this category. This need to maintain compatibility between separately evolving framework and application code is known as the *framework evolution* problem [15].

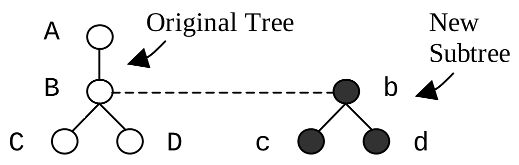

Figure 7 – Framework Evolution

An alternate approach is to implement the new feature in new framework classes. Unfortunately, this approach spawns a new class hierarchy that is parallel to the existing one, creating a potentially large amount of nearly identical new code to maintain. Figure 7 illustrates how a new subtree is created when changes for class `B` are instead implemented in a new class named `b`. Class `b` is a subclass or a copy of class `B`. If child `C` of `B` needs to support the new feature, it does so through its proxy class, `c`, in the new subtree.

In JL, evolution can be implemented using the same two approaches available to frameworks. If changing an existing class is not desirable, a new class can be created, typically using inheritance, to incorporate the changes. The loose coupling of JL classes, however, means that the original class is typically not part of a predefined hierarchy, so no parallel subtree is spawned. There is no compatibility problem because applications can be generated using either the new or old classes.

Changes in the Domain Analysis

If new features require the refactoring of important interfaces, then JL and frameworks are equally susceptible to disruption because they both rely on good domain analysis to define interfaces appropriately.

Analysis

In this section, we explain how mixins are the key to JL's power and flexibility. First, mixins allow code to be varied in a new way. In addition to the techniques that support code variation in ACE—subclassing, type parameters and runtime initialization parameters—JL allows a class's supertype to be varied using mixins. In previous work [5], we proposed that frameworks themselves could be implemented more flexibly using a layered component technology.

Second, mixins allow features to be mixed and matched so that new types can be built in a stepwise manner. In JL, we precisely widen interfaces to support the exact feature set that an application requires by encapsulating features in their own classes and composing them. JL uses mixins to solve the feature combinatorics problem without resorting to wider than necessary interfaces. In JL, unused feature combinations are never materialized.

Mixins work because they defer the specification of parent/child relationships from definition time to composition time. This late binding promotes JL's stepwise refinement model that in turn encourages interfaces to be smaller, less complex, and feature-specific. ACE, and frameworks in general, use non-parameterized inheritance to lock in parent/child relationships and create application skeletons. This rigidity forces the use of wide interfaces to avoid the combinatorial explosion in the number of classes that would result from materializing all feature combinations in advance.

There are, however, a number of drawbacks to using mixins in JL. First, deep class hierarchies generated by mixins can increase runtime overhead. Second, superclass initialization is not straightforward because a mixin's superclass is not known when the mixin is defined. Third, compositional flexibility leads to questions of compositional correctness, especially when nested types are used. Finally, defining recursive types can be tricky because expressing the type of a mixin composition from within the mixins themselves is not straightforward. In the next section, we describe JL language features that are designed to address these limitations of mixins.

6 JL'S NOVEL FEATURES

This section briefly describes JL's novel linguistic and compiler support for domain-independent, stepwise program refinement. We introduce language features built upon the foundation of parametric polymorphism introduced in Section 3. The features, described in more detail elsewhere [12,13], are designed to enhance the usability and efficiency of programming with mixins.

Deep Conformance

In Java, subtyping is *shallow* because subtypes are not required to implement or extend types nested within their supertypes. For example, consider class `C` that implements an interface containing nested interfaces. Class `C` is a subtype of the interface whether or not it implements the nested interfaces. In a layering technology such as JL, composition is easier when the structure of components is predictable and regular, so JL supports *deep conformance*. Deep conformance also allows a single layer (mixin) to refine multiple classes if those classes are nested within a lexically enclosing class.

JL introduces the **deeply** modifier on `implements` and `extends` clauses to force the deep public structure of types to be respected during type checking. The implementation is based on the general notions of *deep subtyping* and *deep interface conformance* [12,13,25] and could augment Java in a useful way independent of JL.

Virtual Typing

Virtual typing [34] is the automatic adaptation of types through inheritance. Using virtual types, inheritance causes specialized types to automatically replace more general types. For example, if class `C` uses virtual type `V` in its definition, then subclass `C′` of `C` could cause all occurrences of `V` in `C` to be changed to `V′`, where `V′` is some subtype of `V`. Virtual typing leads to better static type checking and less manual typecasting because precise subtypes are used in place of more general supertypes. In JL, virtual typing allows an instantiated type to be used within the mixins that are composed to define that type.

JL supports the **This** virtual type, which typically gets bound to the class type of "this" when used in mixins. **This** can only be used in parametric types, so it can be treated as an implicit type parameter to all parametric types. **This** integrates a restricted form of virtual typing into a parametrically polymorphic language and, as such, has general application. The code below shows how virtual typing is used in JL:

```
class ReactorSingle<T implements ReactorIfc>
 extends T
 {private static This _inst;
  public static This instance(){
   if (_inst == null) _inst = new This();
   return _inst;} }
```

The mixin above implements the singleton Reactor, which is useful in applications that require only one Reactor instance. The code shows how the **This** virtual type is used to reference subclasses before they are created. The above mixin is used in the following composition:

```
class MyReactor extends
 ReactorSync<ReactorSingle<ReactorBase>>> {}
```

In the `MyReactor` class above, all occurrences of **This** in any layer are replaced by `MyReactor` (assume `ReactorSync` is a mixin). This illustrates how the parametric types used in a composition can refer to the type ultimately generated by the composition.

Semantic Checking

By deferring the specification of parent/child type relationships from definition time to composition time, mixins offer great flexibility. With this flexibility comes the increased likelihood that syntactically correct compositions will be semantically meaningless. For example, the `TP` type in our Transport example in Section 3 could have been defined using three `KeepAlive` and four `Secure` layers, in any order, and still be type correct.

JL supports semantic restrictions on parametric type compositions that go beyond syntactic type checking. JL associates an ordered attribute space with each composition. Attributes are identifiers chosen by the programmer to reflect some semantic characteristic. Class definitions use a **provides** clause to add attributes to the space and a **requires** clause to test attributes. Using regular expression pattern matching and a count operator, attributes can be tested for presence, absence, ordering and cardinality.

JL's semantic checking mechanism provides a simple, manual way to restrict feature compositions that are known to be invalid, but it cannot guarantee compositional correctness, much less program correctness. For example, consider the class definition of `TimerCancelByTime` in Figure 6. Augmenting this definition with the "**requires unique**" semantic check limits the class to at most one occurrence per Timer specification. This restriction reflects the fact that adding the same cancel method more than once serves no purpose. This semantic check, however, makes no claim that the cancel method will work correctly.

Constructor Propagation

Since the superclass of a mixin is not known at mixin definition time, mixin composition can fail in an attempt to invoke an unavailable superclass constructor. JL supports *constructor propagation* as a way to automatically adjust constructor signatures at composition time so that all superclasses can be properly initialized.

Only constructors marked with the **propagate** modifier have their parameters propagated and their signatures adjusted. Propagation proceeds in child class `C` with parent class `P` as follows: Each propagated constructor in `C` is replaced by a collection of clones of itself, the number of clones equaling the number of propagated constructors in `P`. Each clone in the collection is uniquely associated with a propagated constructor in `P`. Propagation then occurs in two phases. First, the signatures of the clone constructors are augmented with the parameters of their associated constructors from `P`. Second, a call to the associated constructor in `P` is inserted into each clone constructor.

Constructor propagation allows each class in a mixin-generated hierarchy to call its superclass's constructors with the required parameters. Judicious use of constructor propagation avoids an explosion in the number of constructors. For example, consider the `TaskQueue` mixin, which adds a message queue to a Task:

```
TaskQueue<TaskBase>
```

Assume both classes in the above composition have one constructor specified with the **propagate** keyword. `TaskBase`'s constructor takes a `ThreadMgrIfc` parameter and `TaskQueue`'s constructor takes a `MsgQueueWaitIfc` parameter. A constructor for the instantiated type will be generated that takes both parameters, allowing objects of this type to be completely initialized upon allocation.

Optimization

JL's programming methodology of stepwise refinement can create deep hierarchies of small classes. The use of many small classes increases load time, especially when a network is involved; it also requires more memory in the Java Virtual Machine. Stepwise refinement can also result in methods that often call superclass methods with the same signature, as we saw with the `send()` method in Section 2. When compared to an unlayered implementation, stepwise refinement often introduces the runtime overhead of extra method dispatches.

JL's *class flattening* optimization is designed to address these inefficiencies. Calls to superclass methods with the same signature are aggressively inlined and the whole class hierarchy is then collapsed into a single class. As long as certain constraints are satisfied, this optimization can be applied to the code of arbitrary class hierarchies.

7 RELATED WORK

JL derives its compositional power from the use of supertype parameterization (mixins) [3]. To implement its component model, JL also draws upon recent research into generic extensions of Java [1,4,14,22,24,34].

JL's **This** virtual type combines aspects of both Bruce's *ThisType* [9] and Thorup's general virtual types [34]. Both of these approaches require changes to Java's type system and, in the Thorup proposal, increased dynamic type checking. JL's **This**, though less expressive, avoids these complications by limiting its use to parameterized types.

JL is based on Batory's GenVoca research [6,7,25,26]. JL refines the GenVoca model by incorporating layer initialization and the semantic checking of compositions. JL continues research into mixin programming, which began with VanHilst's [36] work using C++ mixins and was later extended with the idea of *mixin layers* [25,26]. JL's contribution is its novel features that enhance the usability and efficiency of programming with mixins.

Object-oriented frameworks [2,17,27], especially when used with design patterns [21], are a popular way to build large applications and software product lines. A number of framework problems have been documented [15,16], including those described in this paper.

Aspect-oriented programming (AOP) [19] defines *aspects* as encapsulations of code that crosscut multiple units of implementation (classes, methods, etc.). In JL, a mixin can refine multiple classes only if these classes are lexically enclosed inside a common class. In AOP, a new programming construct, the *aspect*, can refine the code in an arbitrary group of classes. Gauging the value of this additional flexibility is the subject of continuing research [18].

8 CONCLUSION

This paper has introduced the Java Layers language and has compared JL against frameworks using ACE. We have shown how JL's method of stepwise refinement provides significant advantages in terms of flexibility, usability, and reusability. JL breaks the static binding among framework classes and delivers instead a collection of composable classes. These classes can be combined in different ways to meet the needs of particular applications. Mixins provide the required compositional flexibility, while other language features enhance usability and efficiency.

Our preliminary experiment with one real-world framework reinforces our belief that new language-based technologies will lead to better-engineered software. Many more experiments are needed, however, to validate whether JL has the right mix of language features and whether programmers will actually use this technology. We are currently enhancing our compiler so that we, and possibly others, can begin a new round of experimentation using JL.

ACKNOWLEDGMENTS
We thank Kartik Agaram, Don Batory, Dewayne Perry and Thomas Wahl for their thoughts and lively conversation concerning JL. We also thank the anonymous reviewers for their extremely thorough and helpful comments.

REFERENCES

1. Agesen, O., Freund, S., and Mitchell. Adding Type Parameterization to the Java Language. *OOPSLA 97.*
2. Bohrer, K., Christ, A. and Rubin, B. Java and the IBM San Francisco Project. *IBM Sys. Journal* 37, 3 (1998).
3. Bracha, G., and Cook, W. Mixin-Based Inheritance. *OOPSLA-ECOOP* (1990).
4. Bracha G., Odersky, M., Stoutamire, D. and Wadler, P. Making the future safe for the past: Adding Genericity to the Java Programming Language. *OOPSLA* (1998).
5. Batory, D., Cardone, R. and Smaragdakis, Y. Object-Oriented Frameworks and Product-Lines. *First Software Product-Line Conference* (August 2000).
6. Batory, D., Lofaso, B. and Smaragdakis, Y. JTS: Tools for Implementing Domain-Specific Languages. *ICSE*, (Jun. 1998).
7. Batory, D. and O'Malley, S. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM Transactions on Software Engineering and Methodology* (Oct 1992).
8. Batory, D., Singhal, V., Thomas, J. and Sirkin, M. Scalable Software Libraries. *1st ACM Symposium on the Foundations of Software Engineering* (Dec. 1993).
9. Bruce, K., Odersky, M. and Wadler, P. A statically safe alternative to virtual types. *ECOOP* (1998).
10. Canning, P., Cook, W., Hill, W., Olthoff, W. and Mitchell, J. F-Bounded Polymorphism for Object-Oriented Programming. *Functional Programming Languages and Computer Architecture* (1989).
11. Cardelli L. and Wegner P. On Understanding Types, Data Abstraction and Polymorphism. *ACM Computing Surveys* 17, 4 (Dec. 1985).
12. Cardone, R., Batory, D. and Lin, C. Java Layers: Extending Java to Support Component-Based Programming. *Technical report TR2000-11.* CS Dept., University of Texas at Austin (2000).
13. Cardone, R. and Lin, C. Java Layers home page at *http://www.cs.utexas.edu/user/richcar/JavaLayers.html*.
14. Cartwright, C. and Steel, G. Compatible Genericity with Run-Time Types for the Java Programming Language. *OOPSLA* (1998).
15. Codenie, W. De Hondt, K., Steyaert, P. and Vercammel A. From Custom Applications to Domain-Specific Frameworks. *Comm. ACM* 40, 10 (Oct. 1997).
16. Fayad, M., Schmidt, D. Object-Oriented Application Frameworks. *Comm. ACM* 40, 10 (Oct. 1997).
17. Johnson, R.E and Foote, B. Designing Reusable Classes. *Journal of Object-Oriented Programming* (June/July 1988).
18. Kersten, M. and Murphy, G. Atlas: A Case Study in Building a Web-Based Learning Environment using Aspect-Oriented Programming. *OOPSLA* (1999).
19. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier J. and Irwin, J. Aspect-Oriented Programming. *ECOOP* (1997).
20. King, J. IS Reins in Runaway Projects. *ComputerWorld* (Feb. 24, 1997).
21. Gamma, E., Helm, R., Johnson R., and Vlissides, J. *Design Patterns.* Addison-Wesley (1995).
22. Myers, A., Bank, J. and Liskov, B. Parameterized Types for Java. *POPL* (1997).
23. Parnas, D. On the Criteria to be used in Decomposing Systems into Modules. *Comm. ACM*, 15,12 (Dec. '72).
24. Solorzano, J. and Alagic, S. Parametric Polymorphism of Java: A Reflective Solution. *OOPSLA* (1998).
25. Smaragdakis, Y. Implementing Large-Scale Object-Oriented Components. Ph.D. dissertation, CS Dept., University of Texas at Austin (Dec. 1999).
26. Smaragdakis, Y., and Batory, D. Implementing Layered Designs with Mixin Layers. *ECOOP* (1998).
27. Schmidt, D. An Architectural Overview of the ACE Framework. *USENIX login* magazine (Nov. 1998).
28. Schmidt, D. Home page and links to ACE repository at *http://www.ece.uci.edu/~schmidt*.
29. Schmidt, D. Acceptor and Connector—A Family of Object Creational Patterns for Initializing Communication Services. *Pattern Languages of Program Design 3*, Addison-Wesley (1997).
30. Schmidt, D. and Pyarali, I. Reactor: An Object Behavioral Pattern for Concurrent Event De-multiplexing and Event Handler Dispatching. *Pattern Languages of Programs Conference.* (Aug. 1994).
31. Stevens, R.W. *UNIX Network Programming.* Prentice-Hall (1990).
32. Stroustrup, B. *The C++ Programming Language, 3rd Edition.* Addison-Wesley (1997).
33. Syyid, U. The Adaptive Communication Environment: "ACE". Tutorial at *http://www.cs.wustl.edu/~schmidt/ACE.html*.
34. Thorup, K. Genericity in Java with Virtual Types. *ECOOP* (1997).
35. Tip, F., Laffra C., Sweeney P. and Streeter, D. Practical Experience with an Application Extractor for Java. *OOPSLA* (1999).
36. VanHilst, M. and Notkin, D. Using C++ Templates to Implement Role-Based Designs. JSSST *Int'l Symp. on Object Technologies for Advanced Software*, Kanazawa, Japan, (Mar. 1996).