

# Ph.D. Proposal: Language Support for Layered Refinement

Richard Cardone  
Department of Computer Sciences  
University of Texas at Austin  
[richcar@cs.utexas.edu](mailto:richcar@cs.utexas.edu)

December 14, 2000

## Abstract

The main goal of the research proposed in this document is to reduce the cost of developing and maintaining large software applications. Our approach augments existing object-oriented languages with a small number of domain-independent features designed to enhance code reusability, thus making applications easier to build. This proposal describes the contributions we expect our research to make.

## 1 Introduction

The 1968 NATO conference [42] on the “software crisis” popularized the term *software engineering*, and ever since that time researchers have been trying to deliver on the promise of systematic and efficient software development that the term implies. In 1979, 43% of the federal project officers surveyed by the General Accounting Office [24] reported that it was fairly or very common for software developed under federal contract to be unusable as delivered. In 1995, a comprehensive survey [53] of over eight thousand public and private software projects reported that more than 30% of the projects were cancelled during development, and of those that were completed, three-quarters were late, over-budget or didn’t meet their specifications. The cancelled projects alone were estimated to cost American industry and government \$81 billion in 1995.

In addition to the results from studies, well-publicized failures of critical software projects reinforce the perception, both inside and outside the industry, that any large-scale software development is an expensive and risky undertaking. Recent project losses in dollars range from the tens of millions (California DMV’s license and registration application), to the hundreds of millions (American Airline’s new reservation system, Denver’s airport luggage system), to the billions (FAA’s air traffic control system) [32,53]. As the reliance on software becomes more pervasive in society, the chronic software crisis becomes more acute.

The research proposed in this document is an attempt to make software easier to reuse and, in doing so, to reduce the risk and expense of developing large applications. If existing software can be reapplied in new applications, then the cost of developing and maintaining new code can be avoided. The more code is reused, the more cost-effective it becomes to guarantee the code’s correctness, which raises the quality of applications that reuse the code.

Our research focuses on designing a domain-independent language, called Java Layers (JL), in which applications are assembled from reusable *software components*. The goal of JL is to simplify application development and application evolution by increasing the flexibility and reusability of code. JL extends Java [2] by implementing a software component model in which applications are constructed incrementally in layers. We propose to empirically evaluate JL, the compiler that implements it, and the model on which it depends through a number of programming experiments. Our contribution is to show that by augmenting

existing object-oriented languages with a small number of features, large applications can be more easily built.

We recognize, however, that our research represents only one step towards a solution to the software crisis described above. Our work addresses the way in which applications are designed and coded, but does not address other aspects of software engineering such as architecture, requirements gathering, modeling, people management, documentation, testing, formal methods, installation, operation, and versioning.

This proposal proceeds as follows. The Problem Section focuses on problems encountered when programming large applications using current technologies; the Solution Section describes our approach to alleviating these problems. The Contributions Section details the goals of our research, and the Related Work Section provides context with respect to other research. The Proposed Work Section discusses the whole of the planned work, and the Work Completed discusses the tasks already completed. The Evaluation Section describes how success will be gauged. Finally, the Plan Section lays out our work plan.

## 2 Problem

An important reason why large application development and maintenance is difficult is because of the *complexity* of application code. In this section, we describe two factors that complicate application code, limiting reuse and driving up the cost of software.

### 2.1 Separation of Concerns

One of the main sources of application complexity is the inability to *separate concerns* [44] or *modularize* code. *Design features*, or simply features, are high-level requirements that define the characteristics or capabilities of an application. The inability to localize feature implementations in an application makes maintenance more difficult because the code is harder to understand and more complex due to non-local interactions.

In general, application code is organized using some encapsulation mechanism provided by the implementation language. We call the organization of an application's code its *functional decomposition* [38], which is independent of the type of language used (functional, object-oriented, procedural, etc.). Once a functional decomposition is chosen, it's always possible to define features whose implementations crosscut the discrete units of code that make up the application [57]. These crosscutting features are called *aspects* [38] and they occur commonly. For instance, features that define global properties such as security, thread safety, fault tolerance, or performance are usually aspects because their implementations crosscut the code base. Error handling and tracing are also examples of features that are usually aspects.

Current programming technologies do not have the power to encapsulate complete design features in individual language constructs. This routinely forces programmers to choose between code comprehensibility and other application requirements, such as performance. For example, in object-oriented programming, code that implements an aspect is often dispersed throughout multiple classes or methods (*scattering*), and code at a single location often participates in the implementation of multiple features (*tangling*) [35]. This intermixing of code diminishes both the ability to reuse feature code and the modularity of the program, increasing the cost of development and maintenance.

### 2.2 Variation

Another important source of complexity in application code is the need to support *variation*. Variation can occur along two axes, a time axis and a space axis. On the temporal axis, new or changing requirements tend to degrade application quality over time. 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.

It's hard to over-emphasize the importance of supporting variation over time, or the *maintainability*, of an application. Studies indicate that development organizations spend 60% to 80% of their budget on

software maintenance [47]. Anecdotal evidence also supports this assessment. For example, the Windows NT's code base grew at an annual rate of 33% in the four years after its release, tripling its initial size to surpass 30 million lines of code by 1997 [38]. This level of maintenance activity often accompanies software that supports the changing needs of many users.

Variation on the space axis means that different versions of an application are needed to support different users, execution environments, or market segments at the same time. The desire to provide distinct feature sets in different application instances leads to the development of families of applications or *software product-lines* [13,33]. The challenge here is to build and maintain product lines that maximize code reuse to reduce costs, and to do this without sacrificing performance or maintainability. This task is complex because the requirements of multiple applications must be considered simultaneously.

A particular concern that arises when developing software product-lines is the *feature combinatorics* problem [8]. 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. The ability to efficiently produce software product-lines relies on the easy customization or configuration of applications for specific uses. The goal, therefore, is to maximize the flexibility and reuse of existing code while avoiding the maintenance problems that a combinatoric explosion of code would represent.

To summarize, we've described how the inability to separate concerns and the need to support variation lead to application complexity. Complexity makes maintenance more difficult because code becomes interdependent in non-local ways. Moreover, this interdependence limits opportunities for code reuse because features are not implemented in self-contained units. Thus, an opportunity to reduce future development costs is lost. Developing software product-lines is more complex than developing a single application, and scalability problems due to feature combinatorics further complicate software product-line development. We now describe our proposal for reducing the complexity of application code.

## 3 Solution

Our research focuses on reducing the complexity of programming large applications and software product-lines by addressing the problems described in the previous section. Our goal is to make development easier through reuse, and maintenance easier through modularization. We propose the Java Layers language, which adapts an existing software component model for use in domain-independent, object-oriented programming. We implement JL as an extension of Java. The JL language and the model it supports are described in this section.

### 3.1 The Java Layers Component Model

Java Layers is based on the GenVoca software component model [6,7,49]. The GenVoca model consists of software components called *layers*, a way to specify layer compositions, and a programming methodology that emphasizes the *stepwise refinement* of code. A layer encapsulates the complete implementation of a single design feature.

JL retains the important concepts of the GenVoca model while refining GenVoca for use in domain-independent, object-oriented environments. Thus, the *JL component model* is a refinement of the GenVoca model. JL's component model specializes the GenVoca model by (1) allowing all types to be treated as layers, and by (2) defining layer compositions as instantiations of parametric types. JL's component model extends the GenVoca model by (3) addressing how layers are initialized, and (4) by incorporating semantic checking into subtyping.

#### 3.1.1 Stepwise Refinement

JL supports the programming methodology of stepwise refinement [7] that allows types to be incrementally built by adding features one at a time. Stepwise refinement is important because it allows features to be *mixed and matched*, providing a model that is both flexible and precise. Flexibility is achieved because feature code can be reused in different contexts; precision is achieved because applications can be

customized to support the exact set of features they require. Stepwise refinement solves the feature combinatorics problem described in Section 2.2 by materializing only the feature combinations that are actually needed.

To implement stepwise refinement, software components must completely encapsulate feature implementations. In JL, layers provide this encapsulation. Layers can contain code that would normally be packaged separately using current object-oriented technology. For example, a JL layer can contain code that would typically be scattered across unrelated classes. In JL, applications can be built using these layers because support for stepwise refinement includes a semantically enhanced form of inheritance (Section 3.2.2). The result is that using layers increases the modularity of applications and the reusability of feature code.

## 3.2 The Java Layers Language

We now describe JL, which implements the component model just presented. In JL, layers are types and layer compositions are instantiations of parametric types. A natural implementation choice for our component model is constrained parametric polymorphism. In this section, we describe the parametric implementation of Java that serves as JL's foundation. We then briefly introduce a number of other language features built on top of that foundation, some of which can be applied as standalone features outside of JL. A more detailed discussion of these features is available in specifications and technical reports on the Java Layers home page [21].

### 3.2.1 Parametric Polymorphism

Layer composition in Java Layers is based on the use of *mixins* [14,52]. 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++ [56]. In this section, we describe how mixins support reuse and how they serve as a basis for JL.

Mixins are useful because they allow a set of unrelated 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 superclass with the `lock()` and `unlock()` methods. The lock code would be replicated in three places. With mixins, however, we would write a single class called `Lockable` that could extend any superclass, and we would then instantiate the `Lockable` class to extend `Car`, `Box`, and `House`. The `lock()` and `unlock()` methods would only be defined once in `Lockable`. In JL syntax, the `Lockable` mixin would be defined as follows:

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

JL is based on Java extended with constrained parametric polymorphism [19,55] and mixin support. Researchers have proposed a number of solutions for adding constrained parametric polymorphism to Java [1,11,15,23,40,54]. JL's syntax is similar to that of most of these proposals, though only one other proposal also supports mixins [1]. Let's look at an example layer composition in JL using mixins.

#### 3.2.1.1 An Example of Polymorphism in JL

Consider interface `TransportIfc`, which declares methods `send()` and `recv()`:

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

We now define three layers that 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. Our three layer definitions are shown 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.<sup>1</sup> The `Secure` and `KeepAlive` classes are mixins that inherit from their type parameter, `T`. In both cases, 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 class constraints on type parameters using the `extends` clause, parameterized interfaces, and F-bounded polymorphism [18]. *Instantiations* of parametric types take the conventional form:

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

The statement above declares a variable, `trans`, whose type is an instantiation of parametric types. The use of mixins creates a new class hierarchy with parent `TCP`, child `Secure` and grandchild `KeepAlive`. Semantically, the variable `trans` refers to a secure TCP transport that implements the automatic keep-alive feature. Note that the ordering of layers is important in this scheme—if the `KeepAlive` and `Secure` layers were reversed, then liveness notifications originating in `KeepAlive` code would be sent in the clear.

The flexibility of organizing code this way becomes apparent if we now create a `UDP` layer similar to our `TCP` layer. The `UDP` layer can be composed with the `Secure` layer to create a secure UDP transport type. Thus, features implemented as mixins can easily be mixed and matched to create new types. Feature code gets reused in every instantiation in which the feature appears.

The details concerning JL’s implementation of parametric types, including type parameter usage, constraint checking, scoping rules and nesting rules, are described in JL specifications [21]. Having presented JL’s parametric polymorphism, we now describe the language features built upon this foundation that provide further support for JL’s component model and for stepwise refinement.

### 3.2.2 Deep Conformance

In Java, subtyping is *shallow* because subtypes are not required to implement or extend types nested within their supertypes. For example, consider a class that implements an interface, and that interface contains nested interfaces. The class is a subtype of the interface whether or not the class implements the nested interfaces. In a layering technology such as JL, composition is enhanced when the structure of components is predictable and regular, so JL supports *deep conformance*. A subtype that mirrors the public nested type structure of its supertype is said to deeply conform to its supertype. Mixins that deeply conform to their supertypes provide a convenient mechanism by which a single construct (the mixin) can modify multiple types (the supertypes’ public nested types).

JL introduces the **deeply** modifier for use in `implements` and `extends` clauses to enforce deep conformance. The **propagate** keyword is also introduced to allow the selective application of deep conformance to non-public nested types. The implementation is based on the general notions of *deep subtyping* and *deep interface conformance* [21,51] and could augment Java independent of JL.

### 3.2.3 Virtual Typing

*Virtual typing* [58] is the automatic adaptation of types through inheritance. Virtual typing allows a supertype’s yet-to-be-determined subtypes to be referred to in the definition of the supertype. Virtual typing can lead to better static type checking and less manual typecasting because precise subtypes are used

---

<sup>1</sup> We usually speak of layers when talking conceptually and speak of classes when talking about implementation.

in place of more general supertypes. In JL, the benefits of virtual typing are amplified because it's often useful to refer to the type being instantiated from within the mixins being composed to define that type.

JL supports the **This** virtual type, a limited form of virtual typing that allows the class type of *this* to be specified in many situations. **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.

### 3.2.4 Semantic Checking

By deferring the specification of parent/child type relationships from definition time to instantiation time, mixins offer great flexibility. With this flexibility comes the increased likelihood that syntactically correct compositions will be semantically meaningless. For example, the `trans` type shown in the Section 3.2.1 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 [5,34,45,46]. JL associates an ordered attribute space with each class hierarchy, including hierarchies created by mixin instantiation. 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** clauses to test attributes. Using regular expression pattern matching and a count operator, attributes can be tested for presence, absence, ordering and cardinality.

### 3.2.5 Constructor Propagation

Since the superclass of a mixin is unknown at definition time, mixin instantiation can fail in an attempt to invoke an unavailable superclass constructor. JL supports *constructor propagation* as a way to automatically adjust constructor signatures at instantiation 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 takes place at compile time for non-parametric classes and at instantiation time for parametric classes.

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. This process 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.

### 3.2.6 Optimization

JL's programming methodology of stepwise refinement can create deep hierarchies of small classes. The use of many small classes can increase load time, especially when a network is involved, and requires more memory in the Java Virtual Machine (JVM). Stepwise refinement also results in methods that often call superclass methods with the same signature. When compared to a monolithic implementation, JL's stepwise refinement methodology introduces the runtime overhead of extra method dispatches.

JL's *class flattening* optimization, a feature of the JL compiler, 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. This optimization is designed to work on the bytecode of arbitrary class hierarchies.

## 4 Contributions

Thesis Statement: *A small number of domain-independent language features can support stepwise program refinement and provide an effective way to build large applications and software product lines.*

Our research supports this thesis by making the following contributions:

### 1. Feature Identification

*We identify the essential language features needed to support domain-independent, stepwise refinement.* Our contribution is to show that with the addition of a few well-defined features, Java can support stepwise program refinement. We achieve this by defining the set of necessary and sufficient language features that supports the fundamental concepts of JL's component model. We analyze existing GenVoca implementations to understand how they support stepwise refinement. Where possible, we generalize domain-specific language features for use in domain-independent contexts. We also design new language mechanisms for previously unsupported aspects of the model.

### 2. Feature Implementation

*We show how the identified features can be integrated into an existing object-oriented language.* We describe how constrained parametric polymorphism and a small number of language features can support our model of loosely coupled components and stepwise program refinement. We show how these language extensions integrate well with each other and with Java. We describe the benefits of JL's tight language integration and compare JL to previous domain-independent GenVoca work.

Our contribution includes the design of new language features such as static virtual typing, constructor propagation, and pattern-based semantic checking. We describe how static virtual typing can be implemented in any parametrically polymorphic object-oriented language and how deep conformance can be implemented as a standalone extension to Java. Thus, our contributions have significance in the wider arena of object-oriented languages.

Our contribution also includes a heterogeneous implementation of constrained parametric polymorphism in Java.<sup>2</sup> Our implementation fills an important gap in currently available generic implementations of Java by providing a compiler that supports mixins and other capabilities enabled by heterogeneity.

### 3. Language Application and Evaluation

*We demonstrate the advantages of applying our programming methodology to large application and software product line development.* Using Java Layers, we conduct a number of experiments to evaluate the language's effectiveness and to determine the costs and benefits of its use. We compare programming in JL with programming using object-oriented frameworks and design patterns, the predominant approach to building large applications and software product lines. We evaluate each approach for its flexibility, usability and reusability, and we show how JL can be used to avoid problems common to frameworks. We also compare building software product-lines using JL versus using standard object-oriented techniques.

Specifically, we conduct experiments to investigate the hypotheses that JL:

- avoids problems of framework evolution and overfeaturing,
- scales better than frameworks as the number of possible application features increases,
- supports a higher level of reuse than frameworks, and
- is well suited for developing applications that vary over time and support various feature sets.

---

<sup>2</sup> Heterogeneous implementations generate specific code for each instantiated parametric type; homogeneous implementations generate a single executable for use by all instantiations.

#### 4. Evaluation of Domain Independence

*We compare the advantages and disadvantages of a domain-independent implementation of our model with domain-specific implementations.* In general, domain-specific languages allow new abstractions to be succinctly expressed, optimized and validated, all with compiler support. At best, domain-independent languages can approach the capabilities of domain-specific languages. We compare JL to domain-specific approaches to show that JL is competitive with domain-specific implementations of stepwise refinement, and to determine the extent to which domain-specific capabilities can be incorporated into a domain-independent approach such as JL.

We address domain-specific expressiveness by showing that flexible software components can be defined in various domains using mechanisms already familiar to object-oriented programmers. In JL, domain abstractions are defined using interfaces. Classes that implement these interfaces supply the individual features that are composed to produce applications. Feature compositions are instantiations of parametric types.

We address domain-specific optimization by showing that a domain-independent implementation can approach the performance of domain-specific implementations for an important class of code, i.e., non-scientific, commercial application code. JL's class hierarchy optimization reduces the overhead of design-time layering and we show that this is often enough to approximate the performance of similar code developed in domain-specific environments.

We address domain-specific validation by showing that domain-independent semantic checking is capable of expressing domain-specific constraints on feature composition. In JL, feature compositions are constrained using semantic tags created by programmers. We show that JL's semantic checking provides a simple way to express pre-conditions and post-conditions on feature combinations, no matter what the domain.

### 5 Related Work

Our research draws upon previous work in GenVoca, generic programming and object-oriented programming. Our research can also be compared to other experimental programming paradigms designed to improve application development and maintenance. We discuss these topics in this section.

#### 5.1 GenVoca

JL's lineage includes GenVoca domain-specific implementations [7,8,9,50]. JL's domain-independence leads to more general language features than the domain-specific approach typically does. This generality is often achieved without a loss in expressiveness or efficiency, as in the case of initialization using constructor propagation. On the other hand, domain-independence sometimes limits JL's capabilities when compared to the domain-specific approach, optimization being a case in point.

JL also draws upon previous domain-independent GenVoca work [49,51,52]. JL departs from prior domain-independent research by supporting mixins with integrated virtual typing and deep conformance. JL is also unique in its support for constructor propagation, integrated semantic checking, and class hierarchy optimization. JL's support for stepwise refinement is more complete than that of simple nested mixin classes, or *mixin layers* [52], because of the capabilities of JL's additional language features. Also, JL supports stepwise refinement without introducing specialized constructs, such as P++'s [49] *realm* and *component* constructs, and thus benefits from a close integration with Java's type system.

JL also differs from a proposal by Smaragdakis [51] for adding deep conformance to Java. In JL, deep conformance to a supertype is specified in the definition of a subtype, an approach can be characterized as client-centric. In Smaragdakis's proposal, it's the supertype definition that specifies whether all its subtypes must conform deeply to it, an approach that can be characterized as producer-centric. JL's client-centric approach is less restrictive, though it introduces the **deeply** and **propagate** keywords that the producer-centric approach does not require.

## 5.2 Generic Programming

Extending Java with generics is an active area of research [1,11,15,23,40,54,58]. JL implements genericity using constrained parametric polymorphism with semantics similar to those of most current proposals.<sup>3</sup> JL derives its compositional power from the use of constrained supertype parameterization or mixins [14], which are supported in at least one other proposed extension to Java [1].

To increase its expressiveness, JL also incorporates a restricted form of virtual typing into its implementation of parametric polymorphism. JL's **This** virtual type appears only in parametric types where it can be statically resolved during instantiation. Thorup's proposal [58] for Java virtual types describes a similar construct that can be used in all classes at the cost of increased dynamic type checking.

JL's virtual type is also similar to Bruce's *ThisType* construct [16,17], which is defined as the public interface of *this*. In JL, **This** typically refers to the class type of *this*, not its public interface. Bruce points out two pitfalls to the approach JL takes. First, interfaces used as standalone types have no class type, so **This** cannot be bound. Second, calls to methods requiring an actual parameter of type **This** cannot be statically type checked because the type of the receiving object is not generally known at compile time. JL avoids the first problem by implicitly binding **This** in all situations, including in interfaces, when no explicit binding is given. JL avoids the second problem by limiting the use of **This** to parameterized types, which allows **This** to be bound at instantiation-time prior to compilation.

## 5.3 Object-Oriented Techniques

Object-oriented frameworks [12,28,31,48], 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. A number of framework problems have been documented [20,25,30], including the overfeaturing, framework evolution, and feature combinatoric problems. A goal of our research is to show how problems of framework evolution and overfeaturing can be avoided using JL's component model, and how JL scales better than frameworks as the number of possible application features increases.

## 5.4 Experimental Programming Paradigms

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 modularity in design and code. Examples include subject-oriented programming and hyperslices [57], meta-class programming [27], aspect-oriented programming [38], adaptive programming [39], transformation systems [43], composition filters [10], and associative interfaces [29].

## 6 Proposed Work

The thesis statement and the four contributions described in Section 4 guide our research. We propose to analyze existing component models that support stepwise refinement, with an emphasis on GenVoca. From this analysis, we'll derive the language features needed by JL. For each derived language feature, we'll argue that its inclusion in JL is necessary and sufficient by showing that its absence or diminution would restrict support for stepwise refinement.

We propose to implement all language features described in Section 3.2 except for semantic checking, which will exist in design only. A JL compiler will allow us to evaluate the JL language and its component model by building working programs. We propose that this evaluation take place in three experiments.

In one experiment we'll reengineer an existing object-oriented framework and, in another experiment, we'll build a software product-line. These experiments will allow us to compare JL programming with standard object-oriented programming. The code developed under each approach will be evaluated using the metrics of flexibility, usability and reusability. JL's semantic checking will be evaluated manually.

---

<sup>3</sup> The exception is Thorup's proposal [58], which uses virtual types instead of parametric polymorphism.

We propose another experiment in which we'll compare JL to domain-specific implementations that also support stepwise refinement. In addition to the metrics of flexibility, usability and reusability, we'll also consider the tradeoffs between domain-independent and domain-specific approaches, such as compiler implementation effort and the ability to optimize code. This comparison may or may not include actual programming.

## 7 Work Completed

The work plan described below in Section 9 consists of two design-develop-evaluate cycles for JL. The first cycle has been completed. We've completed our initial analysis of language features needed to support our component model and stepwise refinement [3]. We've developed a compiler that implements this language and we've used the compiler to reengineer a mature object-oriented framework. This reengineering experiment has shown that JL offers the advantages over frameworks described in Section 4 [4,20].

Using feedback from our first cycle, we've recently revised the JL language to support the features described in Section 3.2. The preliminary specifications for the revised language and its compiler have been written [21]. The changes we've made to JL are described in Section 9.

## 8 Evaluation

The success of our research depends on making the four contributions defined in Section 4. Taken together, these four contributions support our thesis statement. Since we are analyzing the benefits of a programming methodology and the language that implements it, the evaluation criteria tend to be qualitative. Whenever possible, however, we try to collect empirical evidence for our assessments. We now describe the criteria we use to gauge our contributions.

We measure the success of our language analysis by showing that each identified feature is both necessary and sufficient for support of stepwise refinement. A feature is shown to be necessary by demonstrating that the feature's exclusion would result in incomplete support for stepwise refinement. A feature is shown to be sufficient by demonstrating that a less expressive implementation of the feature would restrict support for stepwise refinement

We measure the success of JL's feature implementation by evaluating how well JL integrates with Java and by evaluating the general utility of JL's features. We assess the impact of JL's new keywords, the ease of their use, the generality of their semantics and the efficiency of their implementation. We evaluate JL's optimization by the performance gains it yields, its applicability outside of JL, and its implementation efficiency.

We measure the success of JL's programming methodology by comparing it to standard object-oriented methodologies, particularly frameworks and design patterns, using the metrics of flexibility, usability and reusability. Flexibility addresses how easily applications can be customized. Usability addresses how easy application programming is in practice. Reusability addresses how easily design and code can be used in multiple applications.

We measure the success of JL's domain-independent approach to stepwise refinement by comparing it to similar domain-specific approaches. We gauge the extent to which JL allows domain-specific abstractions to be succinctly expressed, optimized and validated. We also explore the language tradeoffs between capability and implementation effort under both approaches.

## 9 Plan

The work plan is organized around the four contributions described in Section 4: the identification, implementation, evaluation, and domain-specific comparison of JL's language features. Our approach is iterative: Each cycle of language design, compiler development and application/evaluation corresponds to a further refinement of JL's language features and implementation. We now describe the two cycles that comprise our work plan.

## 9.1 Cycle One

We perform the initial analysis to identify the language abstractions important to our model. We implement most of these language abstractions in a compiler and then use the compiler to reengineer a mature object-oriented framework. This evaluation allows us to demonstrate the benefits of using our model in building large applications and software product lines. In addition, our evaluation includes a reassessment of JL's language features in light of experimental experience. The knowledge we gain from this experience will be fed into the next cycle. All the work in this cycle has been completed.

## 9.2 Cycle Two

The next step in our research is one of integration, generalization and refinement. The main goal of the second design-develop-evaluate cycle is to show that the language abstractions that support our model can be effectively integrated into Java. The goal is to redesign JL so as to introduce into Java the minimum number of new constructs necessary to support our model. We replace specialized language constructs implemented in the first cycle with general constructs that can be used in any class or interface. We now describe our plan for each phase of the second cycle.

### 9.2.1 Design

The most dramatic change to the JL language in the second cycle will be the replacement of the *layer* construct used in the first cycle with parametric classes and interfaces. Constrained parametric polymorphism and mixins will be available independent of other JL language extensions. Deep conformance and constructor propagation, previously only available inside layers, will also be available for use in any Java class or interface. Similarly, semantic constraints will restrict class inheritance in general, whether or not that class represents a layer according to our programming methodology.

An important part of our integration effort will be to generalize the implementation of our abstractions and, thus, increase their appeal outside of JL. For example, the *thisClass* layer construct will be generalized to the static virtual type *This*, which can be applied to any object-oriented language with parametric polymorphism.

Finally, certain capabilities implemented in our first development cycle, such as automatic interface generation, proved not to be useful in practice and will be dropped. Similarly, capabilities that proved valuable in our first cycle experiment, such as the ability to selectively apply deep conformance, will be included in our new design.

### 9.2.2 Development

The new JL compiler will support constrained parametric polymorphism, mixins, static virtual typing, deep conformance, constructor propagation, and the class flattening optimization. The degree of support will be consistent with our experimental goals, but may not reflect a complete implementation of each language feature. Semantic checking based on pattern matching has been designed, but will not be implemented in the compiler. Instead, we will manually perform semantic checking in our experiments.

The compiler implementation effort will require the development of a parser, a name resolution and loading facility, and a source-to-source translator. The majority of the compiler's code will be new, though some code from the old compiler will be reused. In addition, we will adapt existing bytecode tools to perform class file annotation and class hierarchy optimization. A number of candidate tools are under consideration [22,26,36,59].

### 9.2.3 Evaluation

We will build a software product line to evaluate our new language design and compiler implementation. We will apply JL in an application domain in which variability is important, but in which it's difficult to achieve using available technologies. One possible domain is that of web applications that need to run in different network configurations. In this domain, client, server and database components are distributed in various ways depending on available bandwidth and processing resources. A similar experiment [37] was

recently performed using Aspect-Oriented Programming technology, so this represents an opportunity to compare JL to another experimental programming paradigm in addition to conventional object-oriented practice.

The final part of our evaluation will compare our layered, but domain-independent approach, to layered domain-specific approaches. In particular, we will compare JL programming to that of domain-specific GenVoca programming. Our comparison will use qualitative metrics such as flexibility, usability and reusability and, if time permits, head-to-head performance metrics. Our goal is to show that JL's domain-independent approach exhibits many of the benefits of domain-specific languages, but without the need to implement a new language for each domain.

## 10 References

1. O. Agesen, S. Freund, and J Mitchell. *Adding Type Parameterization to the Java Language*. OOPSLA 1997.
2. K. Arnold, J. Gosling and D. Holmes. *The Java Programming Language*, 3<sup>rd</sup> edition. Addison-Wesley, 2000.
3. D. Batory, R. Cardone and C. Lin. *Java Layers: Extending Java to Support Component-Based Programming*. Technical report TR2000-11. CS Dept., University of Texas at Austin, 2000.
4. D. Batory, R. Cardone and Y. Smaragdakis. *Object-Oriented Frameworks and Product-Lines*. First Software Product-Line Conference (August 2000).
5. D. Batory and B. Geraci. *Validating Component Compositions and Subjectivity in GenVoca Generators*. IEEE Transactions on Software Engineering, February 1997, pp. 67-82.
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, and S. O'Malley. *The Design and Implementation of Hierarchical Software Systems with Reusable Components*. ACM Transactions on Software Engineering and Methodology (Oct 1992).
8. 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.
9. D. Batory and J. Thomas. *P2: A Lightweight DBMS Generator*. Technical Report TR-95-26, Department of Computer Sciences, University of Texas at Austin, June, 1995.
10. L. Bergmans. *Composing Concurrent Objects*. Ph.D. dissertation, University of Twente, June, 1994.
11. B. Bokowski and M. Dalm. *Poor Man's Genericity for Java*. Java Informations Tage, Nov. 12, 1998, Frankfurt, Germany.
12. K. Bohrer, A. Christ and B. Rubin, B. *Java and the IBM San Francisco Project*. IBM Systems Journal 37, 3 (1998).
13. J. Bosch. *Product Line Architectures in Industry: A Case Study*. International Conference on Software Engineering, 1999.
14. G. Bracha and W. Cook. *Mixin-Based Inheritance*. OOPSLA-ECOOP 1990.
15. 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.
16. 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>.
17. K. Bruce, M. Odersky and P. Wadler. *A statically safe alternative to virtual types*. ECOOP 1998.
18. Canning, P., Cook, W., Hill, W., Olthoff, W. and Mitchell, J. *F-Bounded Polymorphism for Object-Oriented Programming*. Fourth International Conference on Functional Programming Languages and Computer Architecture, 1989.
19. L. Cardelli and P. Wegner. *On Understanding Types, Data Abstraction and Polymorphism*. ACM Computing Surveys, Vol. 17, No. 4, December 1985.
20. R. Cardone and C. Lin. *Comparing Frameworks and Layered Refinement*. To appear in the Proceedings of the 23<sup>rd</sup> International Conference on Software Engineering, Toronto, 2001.
21. R. Cardone and C. Lin. *Java Layers home page at <http://www.cs.utexas.edu/users/richcar/JavaLayers.html>*.
22. G. Cohen, J. Chase and D. Kaminsky. *Automatic Program Transformation with JOIE*. USENIX Technical Symposium, 1998.
23. C. Cartwright and G. Steel. *Compatible Genericity with Run-Time Types for the Java Programming Language*. OOPSLA 1998

24. Comptroller General. *Contracting for Computer Software Development*. General Accounting Office report FGMSD-80-4, 1979.
25. W. Codenie, K. De Hondt, P. Steyaert and A. Vercammel. *From Custom Applications to Domain-Specific Frameworks*. Communications of the ACM 40, 10, (Oct. 1997).
26. M. Dahm. *Byte Code Engineering*. Java Informations Tage, Sept. 20, 1999, Dusseldorf, Germany.
27. S. Danforth and I. Forman. *Putting Meta-Classes to Work*. Addison Wesley Longman, Inc., 1998.
28. D. Doscher and R. Hodges. *Sematech's Experience with the CIM Framework*. Communications of the ACM 40, 10 (Oct. 1997).
29. A. Dube. A Language for Compositional Development of Performance Models and its Translation. Masters thesis, Computer Sciences Department, University of Texas at Austin, 1998.
30. M. Fayad and D. Schmidt. *Object-Oriented Application Frameworks*. Communications of the ACM 40, 10 (Oct. 1997).
31. B. Foote and R.E. Johnson. *Designing Reusable Classes*. Journal of Object-Oriented Programming (June/July 1988).
32. W. Gibbs. *Software's Chronic Crisis*. Scientific American, September, 1994.
33. M. Griss. *Implementing Product-Line Features by Composing Aspects*. Proceedings of the 1<sup>st</sup> Software Product Lines Conference, August 2000, Denver, Colorado. Kluwer Academic Publishers.
34. A. Habermann and D. Perry. System Composition and Version Control for Ada. Symposium on Software Engineering Environments, Bonn, West Germany. June 16-20, 1980. Published in Software Engineering Environments, edited by H. Huenke, North Holland, 1981, pp. 331-343.
35. W. Harrison and H. Ossher. *Subject-Oriented Programming (A Critique of Pure Objects)*. OOPSLA 1993.
36. Jikes Bytecode Toolkit. IBM AlphaWorks, <http://www.alphaworks.ibm.com/tech/jikesbt>.
37. M. Kersten and G. Murphy. *Atlas: A Case Study in Building a Web-Based Learning Environment Using Aspect-oriented Programming*. OOPSLA 1999.
38. 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.
39. K. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, MA, 1996.
40. A. Myers, J. Bank and B. Liskov. *Parameterized Types for Java*. POPL 1997.
41. N. Myhrvold. *The Next Fifty Years of Software*. ACM97 Conference, March 1997. Slides at <http://research.microsoft.com/acm97/>.
42. P. Naur and B. Randell. *Software Engineering: Report on a Conference sponsored by the NATO Science Committee, Garmisch, Germany, October 1968*. Scientific Affairs Division, NATO, January 1969.
43. J. Neighbors. The Draco Approach to Software Construction from Reusable Components. IEEE Transactions on Software Engineering, vol. SE-10, no. 5, September 1984.
44. D. L. Parnas. *On the Criteria to be Used in Decomposing Systems into Modules*. Communications of the ACM, 15(12):1053-1058, December 1972.
45. D. Perry. *Version Control in the Inscape Environment*. 9<sup>th</sup> International Conference on Software Engineering, 1987.
46. D. Perry. *The Inscape Environment*. 11<sup>th</sup> International Conference on Software Engineering, 1989.
47. R. Pressman. *Software Engineering: A Practitioner's Approach*, 3<sup>rd</sup> edition. McGraw-Hill, 1991.
48. D. Schmidt. *An Architectural Overview of the ACE Framework*. USENIX login magazine (Nov. 1998).
49. V. Singhal. *P++: A Programming Language for Writing Domain-Specific Software System Generators*. Ph.D. Dissertation, Department of Computer Sciences, University of Texas at Austin, September 1996.
50. M. Sirkin, D. Batory and V. Singhal. *Software Components in a Data Structure Pre-Compiler*. Proceeding of the 15<sup>th</sup> International Conference on Software Engineering, May 1993.
51. Y. Smaragdakis. *Implementing Large-Scale Object-Oriented Components*. Ph.D. dissertation, Department of Computer Sciences, University of Texas at Austin, December 1999.
52. Y. Smaragdakis and D. Batory. *Implementing Layered Designs with Mixin Layers*. ECOOP 1998.
53. The Standish Group International, West Yarmouth, Massachusetts. *CHAOS*, 1995. <http://www.standishgroup.com/>.

54. J. Solorzano and S. Alagic. *Parametric Polymorphism of Java: A Reflective Solution*. OOPSLA 1998.
55. C. Strachey. *Fundamental concepts in programming languages*. Lecture notes for the International Summer School in Computer Programming, Copenhagen, August 1967.
56. B. Stroustrup. *The C++ Programming Language, 3<sup>rd</sup> Edition*. Addison-Wesley, 1997
57. 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.
58. K. Thorup. *Genericity in Java with Virtual Types*. ECOOP 1997.
59. F. Tip, C. Laffra, P. Sweeney and D. Streeter. *Practical Experience with an Application Abstractor for Java*. OOPSLA 1999.