A Compositional Model for Software Reuse

R. K. RAJ AND H. M. LEVY
Department of Computer Science, FR-35, University of Washington, Seattle, Washington 98195, U.S.A.

Emerald is a strongly-typed object-oriented language designed for programming distributed applications. Among other things, it provides abstract typing, type conformity, and complete separation of typing from implementation. While Emerald supports type inheritance, it does not support behaviour sharing among objects for simplifying distribution. To increase Emerald's utility in general-purpose programming, some support for software re-use is needed. Our research reveals that inheritance-based techniques commonly used in other object-oriented systems for obtaining re-use are inappropriate for Emerald. As an alternative to traditional inheritance, a compositional model, in which objects are composed from simpler entities, is proposed, outlined and analysed in this paper.

Received April 1989

1. INTRODUCTION

This paper discusses our research into promoting software re-use in Emerald, a strongly-typed, object-oriented programming language for concurrent, distributed programming.[5,6] Emerald is unusual in that it supports type inheritance but not implementation inheritance. To permit implementation sharing*, we propose a *compositional* model, in which objects are composed from simpler entities, and use this model to extend Emerald. This compositional approach to reuse is evaluated and compared with primarily inheritance-based techniques used in other object-oriented languages.

Emerald is intended to support the development of software for use in multi-user, distributed environments,[15] and not to be a prototyping language such as Smalltalk.[9] Although originally designed for distributed applications, Emerald is a general-purpose programming language. It is different from most other object-oriented languages in that it *unbundles* the several functions performed by *classes* in Smalltalk-like languages:[8] objects are classified by *abstract types*, types (which are interface specifications) are organized into a hierarchy by *type conformity*, and objects are created using *object constructors*. These features enable Emerald to support object encapsulation, subtyping, and complete separation of typing from implementation, and yet provide much of the flexibility usually associated with untyped languages.[25]

Making software truly *soft*, i.e. reusable in either an enhanced or modified form, is important because the process of adding features to a system not originally anticipated by the designers accounts for at least 40% of the total software life-cycle costs.[21] Thus, the design of systems that facilitate the *use* and *reuse* of software, both elegantly and efficiently, has become exigent, with several approaches having been proposed in the literature.[2,28]

Techniques used to provide software reuse in programming languages have included the use of procedures, modules, generic and polymorphic entities, and the use of class-based inheritance (as in Smalltalk and C++[30] or prototype-based delegation[8,20] in object-oriented systems. This paper views both delegation and inheritance as mechanisms for sharing *implementation*, and uses the term inheritance, unless qualified, to refer to implementation sharing by two or more objects.

For reusability, Emerald provides the constructs of operations and objects, and supports both genericity and polymorphism using parameterised object creation. What it does not support extensively is the reuse of implementation, i.e. the use of the same implementation in different objects. Thus, Emerald lacks 'implementation' inheritance but supports 'type' inheritance via type conformity.

It will be shown in Section 1.3 that implementation inheritance and its variants are found to be inadequate for application to Emerald. Our research into alternative techniques resulted in the following two-part solution: first, Emerald is extended using a compositional model that requires the creation of reusable units called *components*; second, a programming environment is provided to support reusability. The new system, called Jade, treats Emerald entities such as operations, objects and types as components, and provides mechanisms in both the programming language and the environment to compose components together to create larger components.

The rest of this section overviews the Emerald object constructor, outlines our specific goals, and discusses why we consider traditional inheritance-based techniques to be unsuitable for Emerald. Section 2 introduces the compositional model being used to promote software reuse in Emerald, presents examples of its usage, and overviews the Jade programming environment being prototyped. In Section 4, we evaluate Jade's support for reuse, comparing it with other models that have been used in the past. Finally, Section 5 comments on the current status of our research.

1.1. The Emerald object constructor

To make this paper self-contained, we present an overview of the Emerald object constructor, glossing over peculiarities of the Emerald language. The rationale for Emerald's design is described by Hutchinson and Jul,[11,14] and a complete description is available in the language report.[12]

* Sharing implementation and behaviour are generally viewed as synonymous in this paper, i.e. we are concerned more with the conceptual sharing of implementation than with actual physical sharing of code.

```
object aPoint
  export setXY, getXY
  var x, y: Coordinate
  operation setXY[newX: Coordinate, newY: Coordinate]
    x ← newX
    y ← newY
  end setXY
  operation getXY → [newX: Coordinate, newY: Coordinate]
    newX ← x
    newY ← y
  end getXY
end aPoint
```

Figure 1. An object constructor expression that creates a point object.

Objects are created using the object constructor, an Emerald expression that, when evaluated, creates a new object. This expressions defines the object's representation and its public and private operations. The object constructor has the following syntax:

```
object aConstructorName
  export public operation names
  % private state declarations
  % private and public operation declarations
end aConstructorName
```

Figure 1 shows an object constructor that can be used to create a point object. The constructor, *aPoint*, when evaluated creates an object, whose public operations are *setXY* and *getXY*, and whose private state consists of x and y, which reference objects of type *Coordinate*. Note that we have ignored the concurrency aspects of Emerald to keep our examples small; Emerald objects actually use a Hoare-style monitor to regulate concurrent access to their private state.

The constructor *aPoint* being an Emerald expression can be assigned to a variable or used in another expression. For example, in the following sequence of statements:

```
var p: PointType
...
p ←
  object aPoint
    % as in Figure 1
  end aPoint
```

the identifier p is assigned the object created by the constructor expression *aPoint*.

In the above example, *aPoint* creates only one object. However, it can in general be placed in a context where it is used repeatedly, e.g. the body of a loop or the result of an operation. Figure 2 illustrates how object constructors may be used to create a class-like (from the object creation viewpoint) object that creates new points. *PointCreator* references an object, created by the object constructor *PC*, that exports the operation *new*. When this object is invoked, the body of *new* gets executed, and constructor *aPoint* gets evaluated, thus resulting in the creation of a new point. The following sequence of statements illustrates how a new point r is created, and is then assigned the point (10, 40).

```
r ← PointCreator.new
r.setXY [10, 40]
```

```
const PointCreator ←
  object PC
    export new
    const PointType ←
      type PT
        operation setXY [Coordinate, Coordinate]
        operation getXY → [Coordinate, Coordinate]
      end PT
    operation new → [a: PointType]
      a ←
        object aPoint
          % as in Figure 1
        end aPoint
    end new
  end PC
```

Figure 2. A class-like object using an object constructor.

Recapitulating, the object constructor is a language mechanism for specifying the implementation of an object. While an object constructor creates one object, it can be nested in loops or used within operations to create many objects, and can be used to build traditional classes. Since objects can be constructed with the same external interface (type) using different object constructors, multiple implementations of the same abstract type are supported. Locality of definition is maintained because all intra-object entities are defined within the same object constructor. The notion of type conformity, which subsumes subtyping, leads to a type hierarchy that determines when one object may be used in place of another. Finally, all this flexibility is obtained in a compile-time type-checked language.

1.2. Re-use in Emerald

A general discussion of software reusability is beyond the scope of this paper: an introduction may be found in some recent issues of *IEEE Software*.[28] We focus only on those technical aspects of reuse that we are interested in providing in Emerald. To make the reuse of software components a reality in Emerald, it must become cost-effective to:

(1) *Create reusable components.* The design of reusable components is difficult: first, component correctness becomes more critical since errors are replicated whenever a component is reused; second, components must be understandable, i.e. well-written and well-documented; third, components must be easily adaptable for different uses, either in original or in modified form.

(2) *Find existing components.* Components must be organized 'properly' so that they can be rapidly found when needed by the programmer to help in their reuse.

(3) *Share components among different users.* To truly facilitate reuse, it must be possible for several programmers to co-operate by sharing software components. Suitable sharing policies need to be established by the co-operating users to derive the most from such a shared environment. Sharing is aggravated when users are working in a distributed programming environment.

Johnson and Foote point out: 'software reuse does not happen by accident...system designers must plan to

reuse old components and must look for new reusable components... the keys to successful software reuse are attitude, tools, and techniques'.[13] While programmer attitudes are not quite within the control of the programming language/environment designer, it is obvious that an easy-to-use programming system can go a long way in proselytising programmers to reusability. Tools must be provided by the programming environment to support goals 2 and 3 above, and also to help in achieving goal 1.

Our major concern was that extensions made for providing implementation reuse should not violate Emerald's fundamental features:

- support for abstract typing,
- the notion of type conformity,
- the locality of object definition,
- the separation of implementation from typing, and
- compile-time type-checking

We also wanted to minimize the changes made to the Emerald language, and to retain backward compatibility if possible.

We strongly believe that the issues of implementation reuse and abstract typing are, and should be, kept orthogonal, i.e.

- implementations are used to compose objects, and
- dynamic interaction between objects is governed by their type.

These two aspects of programming are disjoint by their very nature, and attempts to link the two are inappropriate, as we explain next.

1.3. Problems with inheritance

Our evaluation of inheritance in several languages, including Simula-67,[3] Smalltalk, Trellis/Owl,[27] C++, BETA,[18] SELF[31] and Eiffel,[24] revealed that several properties of inheritance are inappropriate for extending Emerald. These problems may be broadly classified as (*a*) those generally applicable to most systems, and (*b*) those specifically applicable to Emerald. We summarize these problems here, and as noted below, some of them have been previously discussed in the literature.

(1) *Encapsulation is violated.* Inheritance may violate encapsulation in at least three ways: a subclass may (*a*) refer to an instance variable in the superclass, (*b*) call a private operation of its superclass, and (*c*) refer to superclasses of its superclasses.[29]

(2) *Class organization is insufficient.* Since classes do not provide the support for finding methods (and other classes), object-oriented systems need to provide cross-referencing and other organizing tools to locate entities 'out of turn', i.e. outside of the class hierarchy.

(3) *Multiple inheritance is problematic.* The need for multiple inheritance usually indicates that simple class organization is inadequate for the application at hand. Additionally, there are problems associated with understanding and implementing inheritance.[29]

(4) *There is little support for* IS-PART-OF *hierarchies.* Inheritance does not provide support for the notion of components, that is, for IS-PART-OF hierarchies. Blake and Cook discuss the usefulness of such hierarchies and show how they may be implemented in Smalltalk.[7] Attempts to address this problem have also been made in VULCAN,[16] and in object-oriented databases.[17]

(5) *Classes are not automatically reusable.* For successful reuse, inheritance requires the use of a set of coding rules[26] and a set of design rules.[13]

(6) *Re-use via inheritance is not scalable.* Inheritance is successful in environments where the software is written by few people and the number of classes is in the hundreds at most. We believe that software reuse only through inheritance is not scalable.

(7) *Versioning is difficult.* Since inheritance is implicit, it is difficult to determine which version of a superclass will actually be used.

(8) *Supporting distribution could be difficult.* When objects can be mutated and distributed, inheritance imposes substantial difficulties as demonstrated in Distributed Smalltalk.[1]

(9) *Supporting multiple users is difficult.* Inheritance works best in single-user systems. Where there are multiple users, successful co-operative use of inheritance is more a matter of policy than of language design.

Although the problems mentioned below are 'universal' they become more noticeable when inheritance is applied to the Emerald language; of course, the problems mentioned above also affect reuse in Emerald.

(1) *Locality is lost.* Inheritance introduces object dependence on 'unknown' inherited operations (and variables), thus violating the Emerald emphasis on locality of object definition and construction.

(2) *There should be no linkage between typing and implementation.* Emerald's extensive support for multiple implications of the same abstract type is violated if abstract types are also used as a basis for inheritance (see Section 4.2).

2. THE COMPOSITIONAL MODEL

As an alternative to traditional inheritance, we have developed a compositional approach to software reuse in Emerald. This section presents the compositional model used in Jade, first intuitively and then more concretely with several programming examples. Following this, we discuss how composition affects programming style, contrasting it with the style used in more traditional object-oriented systems.

2.1. Intuition

The model for construction used in the children's educational toy, LEGO™, provides a reasonable analogue of our model. LEGO provides children with a basic set of building blocks, which can be put together to form more complex objects. The restriction placed on composition is that only *matching* blocks can be fitted into one another, i.e. the depressions in one block must match the projections in the other. As composed entities become larger, matching becomes more difficult because there are fewer ways of combining large composite blocks together.

The notion of components is quite often used in the

construction of many hardware systems, where interfaces between modules are clearly defined so that one module may be replaced by another. This idea has been extended and used in programming languages such as Modula-2 or Emerald at the module (object) level, where one module (object) may replace another only if their specifications match, i.e. they have the same *type*.

In the Jade compositional model, software components are analogous to LEGO pieces, and the *compositional* interface analogous to the notion of matching. The compositional interface permits appropriate components to fit together, leading to larger components. The compositional interface between software components is different from the invocation interface used to regulate messages between objects. What makes the Jade model different from standard LEGO is that the player or programmer gets to design his own pieces and to reuse (by re-creating) existing pieces as often as needed.

We complete this intuitive introduction to Jade by presenting some simple programming examples.

Composing a stack

Fig. 3 illustrates the compositional model by explaining how a stack object, *aStack*, is composed from simpler components. At this stage, we keep our discussion high-level and defer low-level details to Section 2.2. The different components needed here are shown in Figure 3(*a*):

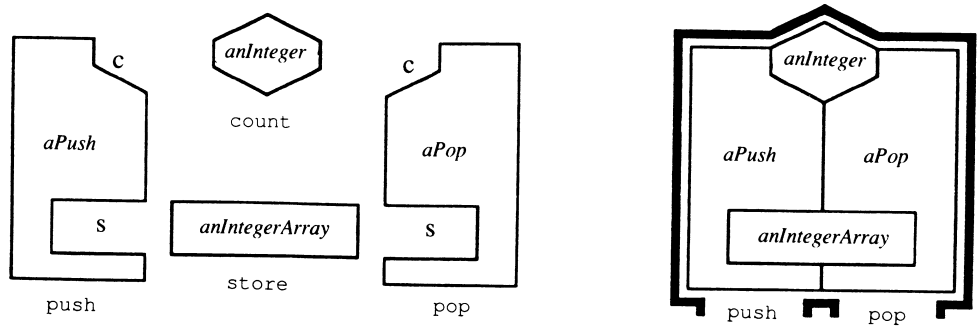
- store, which names an implementation, *anIntegerArray*, used to store the various stack elements.
- count, which names an implementation, *an Integer*, used to keep count of the stack elements.
- push, which names an operation, *aPush*, used to push its argument onto the stack, and
- pop, which names an operation, *aPop*, used to remove and return the topmost stack element.

Fig. 3(*b*) shows how these components can be combined to form a stack object; when the components are placed together, the vacancies *c* and *s* in *aPush* and *aPop* are occupied by *count* and *store* respectively, thus resulting in a complete Emerald object that understands invocations push and pop, which are handled by the implementations *aPush* and *aPop* respectively.

Composing a queue

Our notion of component is useful for reuse because each component, as in Fig. 3, completely describes itself; even *aPop* and *aPush* describe themselves, although in terms of the vacancies *c* and *s*. In other words, it is possible to reuse components in different contexts, as long as all vacancies are filled appropriately. For example, we can create a queue object, *aQueue*, by defining a new component *aRemove* and reusing three of the previously defined components (see Fig. 4).

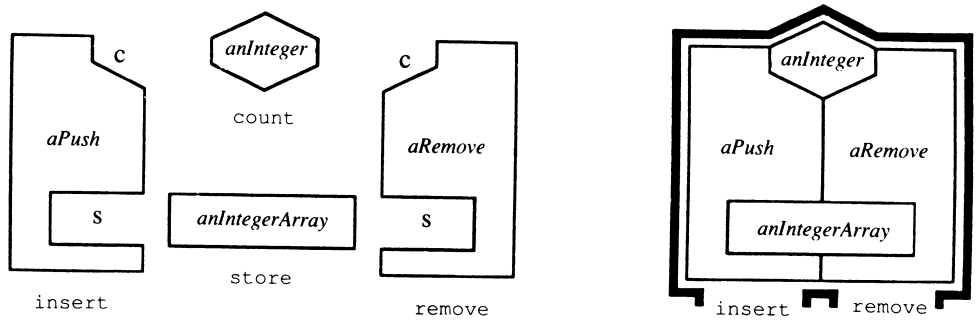
The programmer composing an object may choose components to have names that are different from those used for the actual implementation. This permits greater flexibility in naming abstractions appropriately; for example, the *aQueue* object calls its components `insert` and `remove`.



(*a*) Components needed for a stack

(*b*) The composed stack

Figure 3. Object composition in Jade.



(*a*) Reusing some old components with a new aRemove

(*b*) The composed queue

Figure 4. Reusing components in Jade.

2.2. A preview of Jade

Our earlier discussion indicates that the compositional model is conceptually simple. Informally, composition involves only the following concepts:

(1) The *component* is the basic reusable unit in Jade, and is a completely self-defined entity. Components are first-class entities in Jade and define Emerald entities such as objects, operations and types.

(2) The *compositional interface* has two aspects: (1) the *habitat*, which is defined as a signature specifying the formal names of other components that must be provided to state external dependencies, and (2) the *use* construct, which instantiates a new copy of a component with its habitat completely provided. These two extensions are the only additions made to the Emerald language.

(3) *Composers* are the mechanisms for combining simpler components to form larger components. Such mechanisms are provided both in the language and in the environment; the former is discussed in this section, and the latter in Section 3.

Fig. 5 displays the Jade source for component *aPush*, intuitively described in Fig. 3, and a typical example of its usage. Component *aPush* has two parts, the habitat in which it must exist, and the Emerald source code for an operation. The **use** expression creates a new copy of *aPush*, with its habitat names, *s* and *c*, instantiated as *store* and *count* respectively.

The stack example of Fig. 3 can now be completed as shown in Fig. 6. As expected, the four main components, *aPush*, *aPop*, *anInteger*, and *anIntegerArray*†, are respectively named by *push*, *pop*, *store*, and *count*. The first two components are Emerald operations and the other two are full-fledged Emerald objects. Notice how the names *store* and *count* are matched with the habitats of the two operations. This correspondence between the habitat signature and the actual names is analogous to formal parameters being matched with actual parameters in an invocation, but this takes place at component creation time. Fig. 7 displays a simple Jade component that can be instantiated to demonstrate the usage of the stack.

2.3. More about the Jade component

Jade components cover the spectrum from small entities such as Emerald operations to large entities such as full-fledged Emerald objects. Fine-grained reusability is needed to support the differential nature of programming that inheritance is convenient for, while coarse granularity provides generic Jade components that can be instantiated easily as complete Emerald objects.

The Jade source code of a component defines the implementation of Emerald entities such as objects or operations. In conjunction with the **use** expression, this

† Since Jade permits multiple implementations of any abstraction, we use names such as *aPop*, *aStack* and *anExtendedStack* to refer to specific implementations.

```
aPush
habitat [
    var s: IntegerArray,      % store for the elements
    var c: Integer]           % count of the elements
operation aPush [x: Integer] -> []
    s.AddUpper [x]
    c ← c + 1
end aPush
```

```
Usage
...
operation push [Integer] -> []
    <- use aPush [store, count]
...
```

Figure 5. A Jade component and its usage.

```
aStack
object aStack
    export push, pop
    var store : IntegerArray
        <- IntegerArray.create[0]
    var count: Integer <- 0

    operation push [x: Integer] -> []
        <- use aPush [store, count]

    operation pop [] -> [x: Integer]
        <- use aPop [store, count]
end aStack
```

```
aPop
habitat [
    var s: IntegerArray,
    var c: Integer]
operation aPop [x: Integer] -> []
    x <- s.RemoveUpper
    c <- c - 1
end aPop
```

Figure 6. Object *aStack*, with component *aPop*.

```
aStackUser
object aStackUser
    process
        var x: Integer
        const myStack <- use aStack
        myStack.push[5]
        x <- myStack.pop[]
        stdout.PutString["Popped number <- " || x.asString "\n"]
    end process
end aStackUser
```

Figure 7. Using *aStack*.

code forms an extension of the Emerald object constructor. It is a *component* constructor that creates components, both objects and intra-object entities such as operations and functions. There are several viewpoints that one can have about the Jade notion of component:

- A component has the same functionality as a *class* in class-based languages for the purpose of creating instances. Note that component constructors, in addition to creating objects, also create operations, functions and types. Of course constructors are not really invoked so they are not really classes.
- A component such as *aStack* may also be regarded as a prototype, with the **use** construct regarded as a cloning mechanism. The **use** expression does result in an identical copy (either of the source code or of some pre-compiled form) of the component definition. This is different from the cloning mechanism in SELF, where objects are replicated along with their run-time state.
- We can also regard components with habitats to be a higher-level language variant of an assembly language macro, but these components are type-checked for conformity. This kind of parameterisation is also similar to that of parameterised classes in Eiffel.
- The **use** construct can also be viewed as an 'include' mechanism that ensures that the compiler includes the definition of the Jade component much the same way that the C pre-processor incorporates program files. However, **use** is a language construct that provides parameterisation, and may support the inclusion of pre-compiled code rather than just the source text.

We use compile-time instantiation for the **use** expression in Jade; instantiation at run-time or at invocation-time are other possibilities that are viable and have interesting consequences.

2.4. Composition versus inheritance

The Jade composition paradigm can be contrasted with the inheritance paradigm used in other object-oriented languages. In inheritance-based systems, classes tend to be reasonably big, with inheritance used to create new classes that are nearly the same, but have been extended by additional or alternative methods. Such support for variational/extensional/differential programming has made inheritance popular among many programmers. In contrast, Jade components are both small and large. The programming style used here is one of collecting a set of components, and using them to make a larger component. Support for differential programming can only be realised if the components used are fine-grained, i.e. the size of Smalltalk methods.

The support for extensional and differential programming in Jade comes from both the language and the environment. At the abstraction level, the old Emerald language provides the support needed for extensional or differential programming via its type hierarchy. The Jade environment provides similar support at the implementation level, but its compositional approach makes it considerably different from that of inheritance.

Jade provides primarily two ways of creating extensions. For example, let us consider an extension of *aStack* that has an additional operation, *anEmpty*, that merely checks the stack for emptiness. The first way of creating this new stack, *an ExtendedStack*, is by examining the *aStack* implementation, adding the new component, *anEmpty*, and making other suitable changes (see Fig. 8). The second way of creating an extension is by using Jade to create *anExtendedStack* by simply adding component *anEmpty* to *Stack*; the Jade environment takes care of the details.

These techniques for creating extensions are different from simple editing of implementations to create new objects because the environment retains the relationship between the components. For example, the fact that components *aPush* and *aPop* are used both by *aStack* and *anExtendedStack* is known both to the programmer and system. Any changes made to a component can be reported to its clients (or rather, the clients' programmers) for suitable action. When implementations are merely edited to create extensions, such relationships between 'shared' components are not retained.

Variants of the stack can also be designed. Fig. 9, based on the intuition outlined in Section 2.1, shows how the components of the stack can be reused. To obtain *aQueue* from the implementation of *aStack*, all that needs to be done is to subtract component *aPush* and add component *aRemove*.

As seen above, using habitats provides the ability to define new objects using simple set operations such as union and difference. What is interesting is that several of these operations can be automated (as opposed to

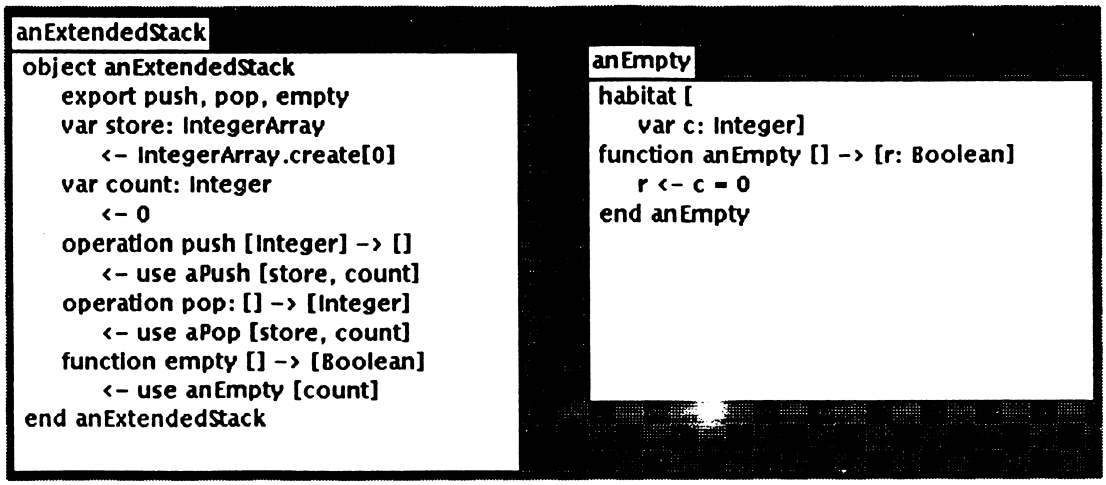


Figure 8. An extended stack.

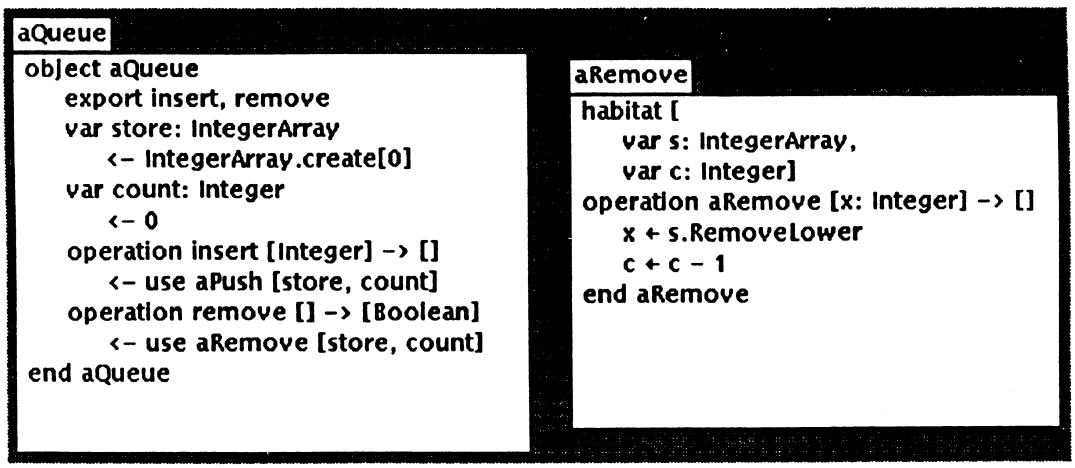


Figure 9. Object *aQueue* and component *aRemove*.

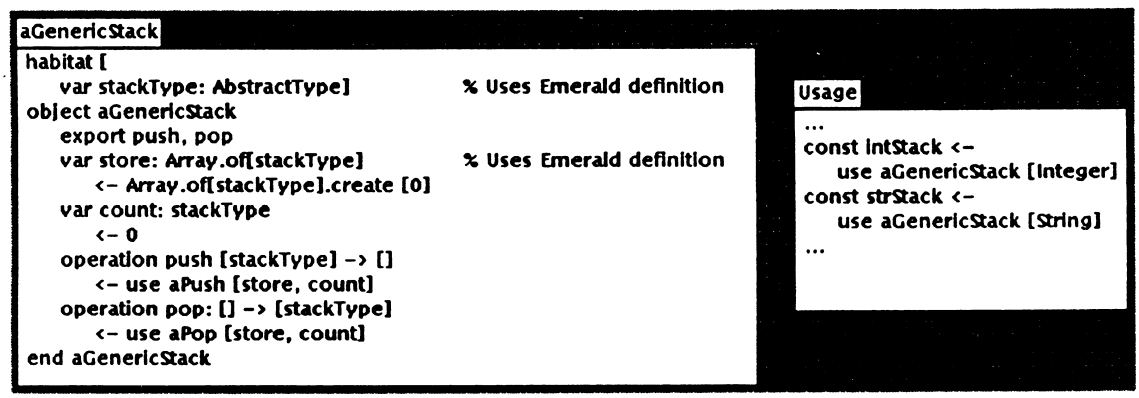


Figure 10. A generic stack and its instantiation.

explicit text-editing by the programmer); we are presently exploring the provision of such operations in the Jade environment.

For a final example, consider the component presented in Fig. 10. Here, we use the expressive power of the habitat construct to create generic entities that can be instantiated by the **use** statement: this demonstrates that Jade also provides parameterised 'classes' in addition to operations and types.

3. THE JADE PROGRAMMING ENVIRONMENT

While the Jade compositional model provides components that are easy to reuse, actual reuse can take place only when existing components are easily accessible by users in a possibly distributed system. The Jade programming environment is designed to support this major aspect of reuse, and to supplement the compositional model by helping to make components more understandable.

To support all aspects of code management, Jade needs tools that assist users in browsing, modifying, compiling, executing and debugging Jade code. Since Jade is still being implemented, we only present the flavour of its environment. The major goal of the current prototype of Jade is to provide a test-bed for experimenting with, and refining our ideas.

3.1. Finding components

The realization that programmers need to find existing components by a variety of techniques depending on varying factors (such as familiarity with the system, programmer background and current requirements) is made explicit in Jade, with components being accessible via a combination of the following:

- *categories*. These are convenient groupings of components, for example, all components used in a payroll application could be kept in a category. More typically, as in Smalltalk, categories are used to keep a collection of related components (classes) together. In Jade, the designer of a component is requested to place it in as *many* categories as are regarded appropriate. For example, the *aStack* component could be entered under *Data Structures* and/or *Repository Objects*. The placement of components in multiple categories makes it different from Smalltalk, where each class belongs to only one category, and each method to only one protocol.
- *synonyms*. These are alternative names used by programmers to access components. Quite often, one programmer's use of a term may correspond to another programmer's use of a synonymous term. How often have we seen identical operations named `delete` or `remove` or `extract`! We assume that programmers will cooperate and use sensible names. The environment also requires them to provide alternative names for entry into a thesaurus-like structure to help in finding components later.
- *role*. This refers to the role played by the component, i.e. is it an `operation` or a `function` or an `object` or a `type`? By permitting access via component functionality, Jade helps the programmer concentrate on only those entities that are of concern.
- *conformity*. The expressive power of type conformity has been discussed in the Emerald papers cited earlier. Conformity of implementations, i.e., the types of the implementations, makes it possible to know what other objects may be used in place of a given component. Note that this is possible because Emerald supports a pure typing hierarchy with multiple co-existing implementations for the same type. Of course, the lack of an explicit connection between implementation and type makes the listing of the type hierarchy more difficult in Jade than, for example, in Trellis/Owl.
- *clients*. These are a collection of components that make use of a given component. Providing rapid access to the clients of a component helps as follows: first, it permits a programmer to examine the component's typical usage and devise new uses for it; second, it allows the programmer contemplating changes to know what updates could affect other users.
- *sub-components*. Here, known components are accessed as the first step to access their sub-components. For instance, in our *aQueue* example, we were able to reuse the sub-component *aPush* of the *aStack* because we knew that the *aStack* was similar to the *aQueue* we were planning on creating.

3.2. Browsers

Central to the Jade environment is the notion of a component library, which may be regarded as a collection of components that is organized for supporting reuse. Every Jade component is stored in a library so that it can be reused later.

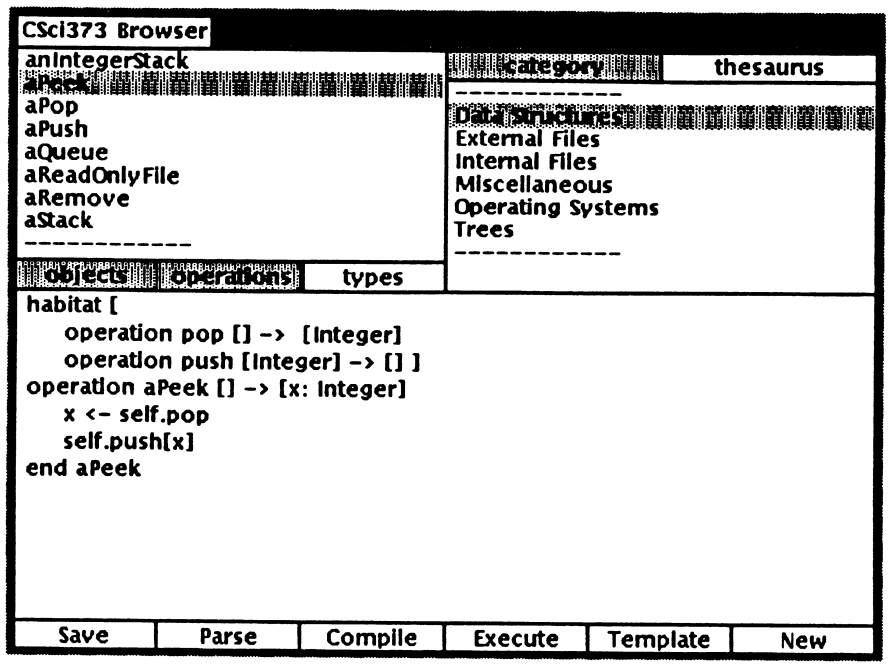


Figure 11. The Jade browser.

Jade has browsers that provide rapid access to components in libraries. A programmer opens a browser on a library for examination and can view the components in it, accessing them using their categories, synonyms and functionality. Figure 11 shows a browser that has been used to examine the contents of a library designed for use in an undergraduate data structures course. Categories are used analogous to their use in Smalltalk to access needed components. Synonyms are used to browse through components the same way that a thesaurus is used to find a suitable synonym. For example, we can search for the component *aPush* by starting with the entry *insert* in the thesaurus, and finding the various components that match until the one sought is found.

There are several ways of cross-referencing information about components. When examining a component, pop-up menus are used to help in finding other components that are clients and sub-components of the one being examined. Additionally, when the component being examined is a full-fledged Emerald object, the pop-up menus also help in listing those Emerald object implementations whose type conforms to that of the object being examined, and vice versa.

Displayed components can be viewed and modified, and when changes are to be recorded, the system can be requested to record the changes. Text-editing support comes directly from the underlying Smalltalk text-editor, and we hope to use all the support we can get from the underlying Smalltalk system.

Libraries provide the basis that can be used to obtain sharing of software among multiple users in the system. To provide effective reuse in a distributed system with 'co-operating' multiple users, setting up proper organizational policies for design and use is important to supplement Jade's concrete support for reuse. That the Smalltalk community practices reuse successfully owes much to the attitude and self-discipline that Smalltalk programmers have toward reusability: they make a concerted effort to write reusable code.

3.3. Other tools

As stated earlier, the primary goal of this research, and of the current Jade prototype, is to examine the compositional model from the reuse perspective. However, to make Jade a truly interactive programming environment, Jade would need to support incremental compilation and provide extensive debugging capabilities. These goals, important in their own right, are not the focus of the current research.

4. DISCUSSION

This section discusses several of the issues that influenced the design of Jade, placing our approach in perspective by comparing it with others taken earlier and elsewhere.

4.1. If not composition?

To obtain implementation reuse in Emerald, what other alternatives could we have used? Of the several approaches that we have examined are the following that appear obvious:

- *Using an object-constructor hierarchy.* A simple alternative would have been to extend the functionality of the Emerald object constructor to provide implementation reuse via object constructor hierarchy, e.g. by the use of a 'sub' object constructor that parallels the C++ (derived) subclass. Such a mechanism would be useful because it is almost directly applicable to the Emerald object constructor, and the concept of an implementation hierarchy is well-understood. Unfortunately, this does not solve most of the problems mentioned in Section 1.3.
- *Eliminating object-constructors.* Another simple alternative would have been to get rid of object constructors altogether, and use the popular notions of either class or prototype for object creation:
  - *Using classes.* Since Emerald obtains its expressiveness from abstract typing, this would have meant supporting two hierarchies, one for typing and the other for implementations. This approach has been used by Lunau to create Duo-Talk, an extension of Smalltalk that has these two distinct hierarchies.[23] We found this alternative unsatisfactory for the same reasons we found the object constructor hierarchy unacceptable.
  - *Using prototypes.* We also considered the use of prototypes,[8,20] for object creation, and using delegation for sharing behaviour. Unfortunately, the notion of prototype is significantly different from that of the Emerald object, and making this change would have meant a substantial redesign of Emerald. Moreover, since delegation also does not address several of the inheritance problems (see Section 1.3) such as violation of locality and encapsulation, we did not pursue this idea further.

The alternatives discussed above adversely affect features of Emerald we considered to be inviolable. We therefore conclude that the Jade compositional model to be a useful, or perhaps the only, approach to reuse in Emerald.

4.2. Separation of implementation from typing

Emerald separates implementation from typing, thus allowing multiple implementations of the same type to co-exist and to be used alternatively as desired in an executing system. We did not want any extension to permit implementation reuse to alter this fundamental feature of Emerald. Therefore, Jade treats the reuse issue as completely in the domain of implementations.

We claim that this uncoupling of typing from implementation is the proper approach to take. Liskov has also commented that the use of inheritance to support two different things: to implement a type and to indicate that one type is a subtype of another leads to problems with multiple implementations.[22]

Jade provides a different approach to reuse: implementations are reused by composition, and objects are used by abstract typing. Thus, Jade preserves separation of typing and implementation, but still permits reuse. This however necessitates a different style of programming in Jade, but as seen in Section 2.4, its expressive power is comparable to that of Smalltalk or C++.

4.3. The use of habitats

To our knowledge, the only other programming languages that support something similar to our use of the habitat are Euclid[19] and its descendants, Concurrent Euclid[10] and the Eden Programming Language.[4] Euclid was designed as a language for developing verifiable systems software, and thus includes several constructs that not only aid in verification, but also increase reliability and understandability.

In these languages, scope is controlled by preventing automatic access to global names; explicit specification of names in *import* and *export* lists is required, thus controlling the availability of global names. Procedures, functions, modules and monitors have to specify in the import list any dependencies on all externally declared entities such as names of global variables, other procedures and functions, and other modules. Additionally, variables and modules need to be imported as *var* when they are subject to modification within the scope of the importing entity.

The price paid for the usage of habitats is that programs sometimes become harder to write and generally become verbose. However, habitats provide better detection of errors for both the programmer and the compiler, and thence to more reliable software. Thus, by enforcing controlled access to non-local names, habitats can help Jade programmers write software whose quality can be easily assured.

The parameterisation of these external dependencies in the habitat attribute provides several benefits. It avoids having to deal with the global name space for understanding the names used locally. Parameterisation offers greater flexibility by both permitting and enforcing renaming and typing within each scope. It also makes each component inherently reusable since all names are limited to the component's own scope.

In summary, habitats provide support for genericity. The explicit specification of external dependencies makes it possible to detect what subcomponents may be added or removed. Since a simple preprocessor from Jade to Emerald can easily be constructed, there is no loss of efficiency in the executable code produced by the Emerald compiler. The use of habitats makes it feasible for Jade components to be compiled incrementally. Finally, the use of habitats also helps in reasoning about program correctness, and compilers that automate the checking of verifiable program can be constructed by using the Euclid approach.

4.4. Complete locality of definition

The use of habitat in Jade also helps in achieving one of the primary goals of this work: to retain complete locality in object definition. For reasons mainly to do with distribution, Emerald objects were designed so that their behaviour is self-contained.[6] Our experience has been that such locality is useful for general-purpose programming,[25] particularly from the reusability viewpoint. We therefore ensured that this basic feature was not violated by the Jade compositional model.

Jade uses a 'bottom-up' approach to object construction rather than the top-down model espoused in most class-based systems. Everything must be explicitly defined in a Jade component, although this can be done in terms of other components. This helps localize each component definition, making it easier to understand since there are no implicitly inherited operations to locate. The *export* list of operations helps in understanding the type of an object.

In more theoretical terms, each Jade component may be considered to be a typed lambda expression with no free variables. This follows because each name used is typed within the component, that is, in the habitat, or in argument or result lists (for operations), or simply as a local name. The avoidance of free variables once again means that reusability is enhanced. In the abstractly typed world of Jade, this also leads to greater flexibility.

5. STATUS AND SUMMARY

This paper presented a compositional model as an alternative for achieving implementation reuse in the Emerald system. Composition of implementations, in conjunction with an abstract type hierarchy, provides many of the benefits of inheritance with fewer disadvantages.

As stated, the current version of Jade is being prototyped in Smalltalk, which we believe still provides the 'best' prototyping environment for such applications as ours. In the next stage of our work, we expect to complete the preliminary design and implementation of Jade, with its support for component libraries. We plan to experiment with different schemes for multiple-user co-operation across the nodes of a distributed system. This will give us an opportunity to evaluate Jade support for scalability and usability, and permit us to propose suitable version management schemes that can be used to keep track of components.

Acknowledgements

We would like to thank Andrew Black, Alan Borning, Norman Hutchinson, Eric Jul, and Ewan Tempero for helping us refine several ideas and for providing feedback on earlier drafts. Thanks are also due to Bill Griswold, David Notkin, N. Shankar, and the referees for providing additional feedback, and John Maloney and Bjorn Freeman-Benson for helping in the prototyping of Jade.

This work was supported in part by the U.S. National Science Foundation under Grants DCR-8420945 and CCR-8700106, Digital Equipment Corporation and External Research Program, by an equipment donation from Tektronix, Inc., and by a dissertation award from Microsoft Corporation.

REFERENCES

1. J. K. Bennett, The design and implementation of distributed Smalltalk. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications* (October, 1987).
2. T. J. Biggerstaff and A. J. Perlis (eds), *IEEE Transactions on Software Engineering*, Special Issue on Software Reusability (September 1984).
3. G. Birtwistle, O.-J. Dahl, B. Myrhaug and K. Nygaard, *Simula Begin*. Petrocelli/Charter (1973).
4. A. P. Black, The Eden programming language. Technical Report 85-09-01, Department of Computer Science, University of Washington, Seattle (September 1985).
5. A. Black, N. Hutchinson, E. Jul and H. Levy, Object structure in the Emerald system. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 78–86, October 1986.
6. A. Black, N. Hutchinson, E. Jul, H. Levy and L. Carter, Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, **13** (1) (January 1987).
7. E. Blake and S. Cook, On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *ECOOP '87 European Conference on Object-Oriented Programming*, pp. 41–50, Paris, France (June 1987).
8. A. H. Borning, Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference* (November 1986).
9. A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implication*. Addison-Wesley, Reading, MA (1983).
10. R. C. Holt, *Concurrent Euclid, The Unix System and Tunis*. Addison-Wesley, Reading, MA (1983).
11. N. C. Hutchinson, Emerald: an object-based language for distributed programming. *Ph.D. Dissertation*, TR 87-01-01, Department of Computer Science, University of Washington, Seattle, January 1987.
12. N. C. Hutchinson, R. K. Raj, A. P. Black, H. M. Levy and E. Jul, The Emerald programming language report. Technical Report 87-10-07, Department of Computer Science, University of Washington, Seattle, October 1987.
13. R. E. Johnson and B. Foote, Designing Reusable Classes. *Journal of Object-Oriented Programming* **1** (2), 22–30, 35 (June/July 1988).
14. E. Jul, Object mobility in a distributed object-oriented system. *Ph.D. Dissertation*, TR 88-12-06, Department of Computer Science, University of Washington, Seattle (December 1988).
15. E. Jul, H. Levy, N. Hutchinson and A. Black, Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems* **6** (1) (February 1988).
16. K. Kahn, E. D. Tribble, M. S. Miller and D. G. Bobrow, The object-oriented classification paradigm. In B. Shriver and P. Wegner (eds), *Research Directions in Object-Oriented Programming*, pp. 75–112. MIT Press, Cambridge, MA (July 1987).
17. W. Kim, J. Banerjee, H.-T. Chou, J. F. Garza and D. Woelk, Composite Object Support in an Object-Oriented Database System. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 118–125 (October 1987).
18. B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen and K. Nygaard, The BETA programming language. In B. Shriver and P. Wegner (eds), *Research Directions in Object-Oriented Programming*, pp. 7–48. MIT Press, Cambridge, MA (July 1987).
19. B. Lampson, J. Horning, R. London, J. Mitchell and G. Popek, Report on the programming language EUCLID (revised). Technical Report CSL-81-12, XEROX PARC, Palo Alto, CA (October 1981).
20. H. Lieberman, Using protypical objects to implement shared behavior in object oriented systems. In *Proceedings of the First ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 214–223 (September 1986).
21. B. Lientz and E. Swanson, *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley, Reading, MA (1980).
22. B. Liskov, Data abstraction and hierarchy. In *Addendum to the Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 17–34 (October 1987).
23. C. P. Lunau, Separation of Hierarchies in Duo-Talk. Technical report, Department of Computer Science, University of Copenhagen, Denmark (1989). To appear in *Journal of Object-Oriented Programming* in 1989.
24. B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall International, London, U.K. (1988).
25. R. K. Raj, E. D. Tempero, H. M. Levy, N. C. Hutchinson and A. P. Black, *The Emerald Approach to Programming*. Technical Report 88-11-01, Department of Computer Science, University of Washington, Seattle, November 1988.
26. R. Rochat, In search of good Smalltalk programming style. Technical Report CR-86-19, Tektronix, Beaverton, Oregon, 1986.
27. C. Schaffert, T. Cooper, B. Billis, M. F. Kilian and C. Wilpolt, An introduction to Trellis/Owl. In *Proceedings of the First ACM Conference on object-oriented programming systems, Languages and Applications*, Portland, OR (September 1986).
28. B. Shriver (ed.), *IEEE Software*, 1987. Several articles on software re-use appear in the January, April and July issues of this year.
29. A. Snyder, Inheritance and the development of encapsulated software components. In B. Shriver and P. Wegner (eds), *Research Directions in Object-Oriented Programming*, pp. 165–188. MIT Press, Cambridge, MA (July 1987).
30. B. Stroustrup, *The C++ Programming Language*. Addison-Wesley, Reading, MA (March 1986).
31. D. Ungar and R. B. Smith, Self: the power of simplicity. In *Proceedings of the Second ACM Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 227–241 (October 1987).