JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2004**

Vol. 3, No. 7, July-August 2004

Extending the Java Language with Dynamic Classification

Liwu Li, University of Windsor, Canada

Abstract

The *dynamic classification* feature of an object-oriented programming language allows an object to change its class membership without changing its identity at runtime. The new membership of the object can be signified with a role, which is taken on by the object and which can be implemented as an object of the target class. Here, we propose an approach to extend the Java language with a dynamic classification mechanism, which can be implemented by extending the Java language, compiler, and standard library. We present a prototypical implementation of the mechanism to show the feasibility of the approach to dynamic classification.

1 INTRODUCTION

In real life, a person may be employed by a company and registered as a student and, later, promoted or graduated. Following the literature, we say the person plays the role of an employee and that of a student dynamically. In an object-oriented language such as Smalltalk, C++, Java, or C#, an object of class `Person` can be used to represent the person. But, the object cannot represent the employee or student role of the person. It cannot be reclassified into class `Employee` or `Student`. Possible solutions to representing dynamic roles of a person object are:

R1. Using objects of classes `Employee` and `Student` to represent employee role and student role of a person. Thus, we need to represent a person with multiple independent objects, each having a distinctive identity.

R2. Using slots or fields in class `Person` to keep information on the employee status, student status, and other possible status of the person [Bachman and Daya 1977].

R3. Melting data and functionality of person, employee, and student into a class and using an object of the new class to replace the person object. The new class may inherit classes `Person`, `Employee`, and `Student`.

The first solution needs extra efforts for a programmer to maintain and coordinate dynamically changing memberships and multiple objects for a real-world evolving entity.

Cite this article as follows: Liwu Li: "Extending the Java Language with Dynamic Classification", in *Journal of Object Technology*, vol. 3, no. 7, July-August 2004, pp. 101-120.
http://www.jot.fm/issues/issue_2004_07/article2

Similar ideas have been explored in design and analysis patterns for programmers to cope with dynamic classification manually [Bäumer et al. 2000, Fowler 1997]. The second solution burdens class `Person` with fields that are not essential or necessary for all real-world persons and, thus, blurs the responsibility of the class. The third solution needs a large number of classes to meet the different combinations of roles of persons. It needs multiple inheritance feature of an object-oriented programming language to ease the task of class programming.

In this paper, we propose extending the Java language with a dynamic classification mechanism and present a prototypical implementation of the mechanism to show the feasibility of the mechanism. In the next section, we extend the Java language with dynamic classification constructs in the forms of declaration and expression, which can be used in programs. In Section 3, we describe how the Java compiler and the Java standard library can be extended to support the dynamic classification mechanism. A prototypical implementation of the mechanism is described in Section 4. We conclude the paper in Section 5.

In the following presentation, we distinguish identifiers and terminal symbols used in programs from non-terminal symbols used in grammar rules. The identifiers and terminal symbols appeared in programs are printed in `blue` color. The non-terminal symbols and other symbols that are used in grammar rules are printed in `green` color. In a grammar rule, we use a pair of braces followed by an asterisk `{}*` to enclose a component that may be repeated zero or more times. We use a pair of square brackets `[]` to enclose an optional component in a grammar rule.

2 EXTENDING THE JAVA LANGUAGE

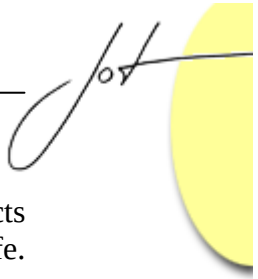
We propose introducing the following constructs to the Java language [Arnold et al. 2000] for programmers to code dynamic classification in applications:

J1. *Dynamic classification declaration* for supporting programmers to specify dynamic classification associations, which indicate which types of objects can play which types of roles and which also specify labels to identify roles;
J2. *Dynamic reclassification* and *declassification expressions* for objects to take on new roles and drop the roles; and
J3. *Expressions of role access*, *dynamic field access*, and *dynamic method invocation* for retrieving roles, accessing fields of roles, and invoking methods of roles.

2.1 Dynamic Classification Declaration

The dynamic classification mechanism proposed here is based on two tenets:

D1. Any (non-abstract) class in an object-oriented program can be instantiated to create roles as well as create objects.
D2. The kinds of roles that can be played by objects of a class are labeled with identifiers, which can be used to access roles.



The first tenet allows using any class to create roles that are taken on by existing objects and roles. The second allows naming roles with labels (identifiers) used in our real life. For example, we may use class `Employee` to create `employee` roles that are played by objects or roles of class `Student`. Here, the roles are named `employee`. The two tenets are reified with the notion of a dynamic classification declaration, which is defined as follows.

A *dynamic classification association* (DCA), denoted with a triplet ⟨D_1, `l`, D_2⟩, relates classes D_1 and D_2 with a label `l`. It indicates that an object or role of class D_1 can play, and thus take on, a role of class D_2 and the role is named `l`. Based on the DCA ⟨D_1, `l`, D_2⟩, we call class D_1 a *dynamic parent* (class) of class D_2, which is a *dynamic child* (class) of class D_1. For example, the DCAs ⟨`Student`, `employee`, `Employee`⟩, ⟨`Employee`, `deptManager`, `Manager`⟩, and ⟨`Employee`, `projectManager`, `Manager`⟩ indicate that a student object or role can have an add-on `employee` role of class `Employee`, and an employee object or role can play two roles of class `Manager`, which are called `deptManager` and `projectManager`, respectively.

We could introduce DCAs in an object-oriented program by declaring DCAs directly. Instead, we propose construct *dynamic classification declaration* (DCD) to introduce DCAs that share a dynamic parent class in a program. A DCD is coded in a source code file in one of two forms: *stand-alone* or *dynamic-clause*.

A stand-alone DCD can be placed anywhere outside class definitions in a source code file. Using keywords `dynamic` and `label`, the syntax of a stand-alone DCD is

```
<parent> dynamic <child> {, <child>}*
[label [ <label> {, <label>}* ] {[ <label> {, <label>}* ]}* ] ;
```

Non-terminals `<parent>` and `<child>` denote class names, `<label>` denotes an identifier. Keyword `dynamic` is followed by a sequence of dynamic child names that share the dynamic parent denoted by `<parent>`. The keyword `dynamic` and dynamic child names compose a *dynamic-clause*. Keyword `label` is followed by a sequence of *label block*s, each of which is enclosed with a pair of square brackets `[]`. The keyword `label` and label blocks compose a *label-clause*, which is optional. The number of label blocks in a DCD must be less than or equal to the number of dynamic child names in the DCD. A stand-alone DCD is terminated with a semicolon. For example, the DCD

```
Employee dynamic Manager
         label [deptManager, projectManager];
```

associates dynamic parent `Employee` and child `Manager`. It indicates an employee object can play manager roles called `deptManager` and `projectManager`.

A Java class definition can be extended with a *dynamic-clause*, which designates the defined class as the dynamic parent of the DCAs introduced by the dynamic-clause. The syntax of a dynamic-clause in a class definition is

```
dynamic <child> {, <child>}*
```

```
[label [ <label> {, <label>}* ] {[ <label> {, <label>}* ]}* ]
```

A dynamic-clause in a class definition is placed between the head and body of the class definition. It is a shorthand of a stand-alone DCD. For example, the following source code defines class `Employee` and presents a stand-alone DCD that declares dynamic parent `Employee`.

```
Employee dynamic Manager
         label [deptManager, projectManager];
public class Employee extends Person {
   // class body is omitted
}
```

The above code is equivalent to the following source code, which uses a dynamic-clause in the definition of class `Employee` to replace the above stand-alone DCD.

```
public class Employee extends Person
    dynamic Manager label [deptManager, projectManager] {
   // class body is omitted
}
```

We now explain how to extract DCAs from a stand-alone DCD that has a label-clause. A stand-alone DCD with no label-clause is a special case for the following algorithm. A dynamic-clause in a class definition is processed similarly. We shall use a container named `DCAs` to hold all the DCAs declared in a program. Assume a DCD that declares m dynamic child classes $childName_i$ with $0 \le i < m$ and n label blocks L_j with $0 \le j < n \le m$. Also assume each label block L_j encloses $s_j \ge 1$ labels. The DCD has the form

$$parentClass\ \mathbf{dynamic}\ childName_0, \ldots, childName_{m-1}\ \mathbf{label}\ L_0, \ldots, L_{n-1}$$

Each label block L_j with $0 \le j < n$ has the form

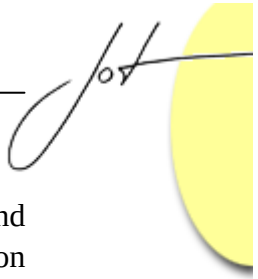
$$[label_{j1}, \ldots, label_{js_j}]$$

where $s_j \ge 1$. The DCAs introduced by the above DCD can be extracted with the following pseudo-code, where expression `lower(childName`$_i$`)` changes the first letter of identifier $childName_i$ to lowercase and returns the changed identifier.

```
for (int i=0; i < n; i++)
 for (int j=1; j <= s_i; j++)
   place DCA ⟨parentClass, label_ij, childName_i⟩ into DCAs;
for (int i=n; i < m; i++)
   place DCA ⟨parentClass, lower(childName_i), childName_i⟩ into DCAs;
```

For example, an execution of the above algorithm for the DCD

```
Person dynamic Student, Employee
       label [undergrad, grad];
```



places DCAs ⟨Person, undergrad, Student⟩, ⟨Person, grad, Student⟩, and ⟨Person, employee, Employee⟩ into the container DCAs. The DCAs allow a person object to play an undergrad, a grad, and an employee role, which are created by instantiating class Student or Employee.

A Java program may be composed of multiple source code files, each of which may include stand-alone DCDs and dynamic-clauses. All the DCAs implied by the DCDs in the program are collected into the container DCAs. We require the DCAs ⟨C_1, l_1, D_1⟩ and ⟨C_2, l_2, D_2⟩ of a program satisfy condition

($C_1 = C_2$) ∧ ($l_1 = l_2$) ⇒ ($D_1 = D_2$).

The condition means that a dynamic parent C and a label l uniquely decide a dynamic child D and a DCA ⟨C, l, D⟩. Thus, each label l signifies only one type D of roles for a given dynamic parent C. The condition is a reasonable constraint for an application. No other constraints are enforced for DCDs or DCAs in a program.

In object-oriented programming, objects of a subclass of a class are objects of the class. A DCA ⟨C, l, D⟩ implies that an object or role instantiated with a subclass C' of class C can take on a role that is instantiated with a subclass D' of class D and that is named l.

2.2 Dynamic Reclassification and Declassification

By the dynamic classification mechanism, a class can be used to create objects and roles. An *independent* object, which is not a role, can be created as specified in the Java language [Arnold et al. 2000]. For instance, expression `new Employee()`invokes the default constructor of class Employee to create an object. We introduce keyword newChild in the following expressions for creating roles:

```
o1 newChild(D1, l, D2, C2)
o1 newChild(D1, l)
o1 newChild(l)
```

A role o_2 created with a newChild-expression is associated with an object or role o_1, which is called the *parent* of o_2. It is is a *child* of its parent o_1. We detail the expressions as follows.

The first newChild-expression is based on a DCA ⟨D_1, l, D_2⟩ and a subclass C_2 of the dynamic child D_2. Its execution creates a role o_2 by instantiating class C_2 and links the created child role o_2 to its parent o_1 through DCA ⟨D_1, l, D_2⟩. Based on the fact that dynamic child D_2 is uniquely determined by dynamic parent D_1 and label l, the second newChild-expression is a shorthand of expression o_1 `newChild(`D_1`, l, `D_2`, `D_2`)`, which creates a role o_2 by instantiating the dynamic child class D_2. If the runtime class of object o_1 is class C_1 and ⟨C_1, l, D_2⟩ is a DCA for a class D_2, the last newChild-expression denotes a shorthand of expression o_1 `newChild(`C_1`, l, `D_2`, `D_2`)`. The above three

newChild-expressions return (references to) the created roles o_2. Like objects, created roles o_2 can be assigned to variables and parameters.

In Section 2.3, we shall explain how to access fields and invoke methods defined in a child role o_2 through the parent o_1 of o_2. Hence, a newChild-expression reclassifies an object or role o_1 into a class C_1 or D_2 effectively without changing the identity of o_1.

The dynamic classification mechanism permits selective removal of roles associated with an object or role o_1. Thus, we declassify o_1 from some classes. Declassifications are realized with following expressions, which use keywords `removeAll`, `removeSelf`, and `removeChild` to signify intended operational semantics.

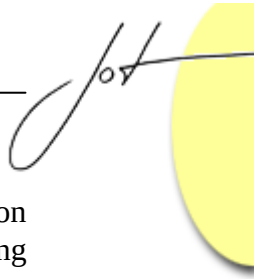
```
o1 removeAll
o1 removeChild(D1, l)
o1 removeSelf
```

The above removeAll-expression removes all the links between o_1 and its child roles o_2. It also removes links between the child roles and the child roles of the child roles recursively. Thus, the removeAll-expression effectively removes all the links between o_1 and roles that are directly or indirectly associated to o_1. If a parent o_1 is linked to a child role o_2 through DCA $\langle D_1, l, D_2 \rangle$ for a class D_2, the removeChild-expression removes the link between o_1 and o_2 and, then, performs o_2 `removeAll`. If a role o_1 is linked to its parent `o'` through DCA $\langle D_1, l, D_2 \rangle$, the removeSlef-expression is equivalent to removeChild-expression `o' removeChild(D1, l)`.

When the dynamic classification mechanism evaluates a newChild-expression such as `o1 newChild(D1, l, D2, C2)`, if o_1 is already linked to a role `o'` through the DCA $\langle D_1, l, D_2 \rangle$, the mechanism performs operation `o1 removeChild(D1, l)` prior to evaluating the newChild-expression. Thus, an object or role o_1 is linked to at most one child role through a given DCA.

For example, the first statement in the following source code creates a graduate student object and assigns the object to variable `tom`. The second statement creates an employee role for the graduate student object and assigns the employee role to variable `e`. The third and fourth statements create a `deptManager` role and a `projectManager` role for the role `e`. Thus, we reclassify object `tom` (and role `e`) into class `Manager`. The last statement removes the links between object `tom` and its roles and, thus, declassifies object `tom` from the classes `Employee` and `Manager`. It also declassifies role `e` from class `Manager` and renders `e` to an independent object. After the last statement is executed, the manager roles are no longer associated with any object or referenced by any variable. They are subject to garbage collection of the Java virtual machine.

```
tom = new GraduateStudent();
e = tom newChild(Person, employee);
e newChild(deptManager);
e newChild(projectManager);
tom removeAll;
```



Only newChild-expressions in a program can create roles. A newChild-expression extends an existing object or role o_1 with a link to a new role o_2. Hence, all the existing objects and roles of a program at runtime are linked with DCAs into a forest. For example, the objects and roles created by the first four statements in the above code form a tree shown in Fig. 2.1. The removeAll-expression changes the tree into a forest shown in Fig. 2.2, in which the two manager objects are no longer referenced by any variables and, therefore, subject to gabage collection.

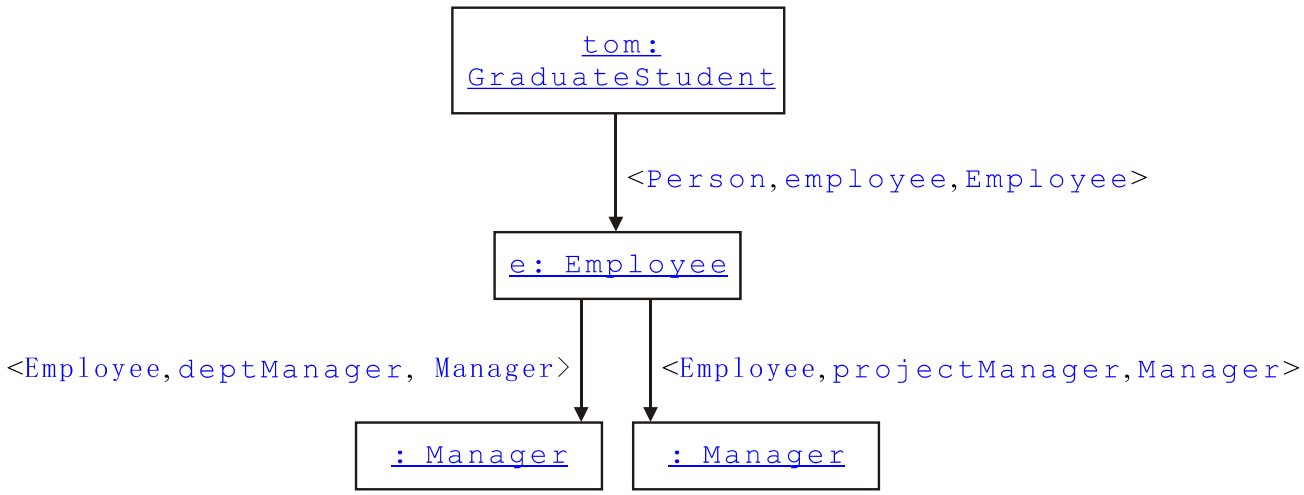


Figure 2.1 An object and its roles

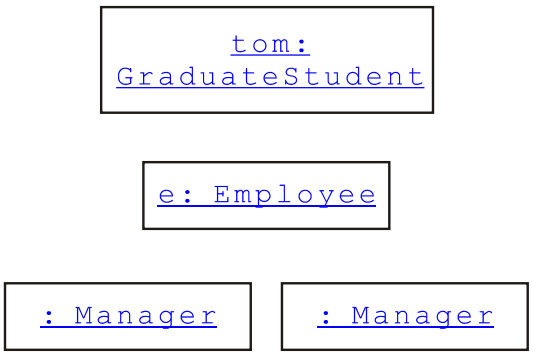


Figure 2.2 Independent objects

2.3 Role Access

The following child-expression retrieves a role o_2 of a given object or role o_1. Keyword `child` introduces dynamic parent D_1 and label `l` of the DCA that links parent o_1 and the intended child o_2. The DCA $\langle D_1, \texttt{l}, D_2 \rangle$ is uniquely determined by the arguments D_1 and `l` of the child-expression.

o_1 **`child`**`(`D_1`, l)`

The following parent-expression retrieves the parent o_1 of a given role o_2. We use the keyword `parent` to signify the operational semantics. As described in Section 2.2, the parent o_1, if exists, of the given role o_2 is unique.

```
o2 parent
```

The above two expressions explicitly request to traverse a parent-child link downward or upward. The dynamic classification mechanism supports implicit access to a field in a role as well. Assume that expression $e_1.v$ is the rvalue of an assignment operation or an argument in a method invocation, expression e_1 denotes an object or role o_1, and v is an identifier. All the roles associated with o_1 form a tree T rooted at o_1. If o_1 encapsulates field v, expression $e_1.v$ is equal to the field v in o_1; otherwise, $e_1.v$ is equal to the field v in the highest role o' in T that encapsulates field v. In the latter case, expression $e_1.v$ accesses the field v in role o' of o_1. When each object or role in tree T does not encapsulate field v or there are multiple highest roles that encapsulate field v, we leave the semantics of expression $e_1.v$ for an implementation of the dynamic classification mechanism to resolve.

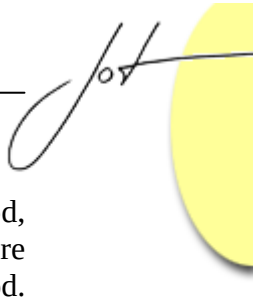
The dynamic classification mechanism supports implicit invocation of a method encapsulated by a role o' of an object or role o_1. Assume a method invocation expression $e_1.m(a_0, \ldots, a_{k-1})$ is executed, where e_1 denotes an object or role o_1, m is a method name, a_0, …, a_{k-1} are arguments with $k \geq 0$. In the execution, expressions a_0, …, a_{k-1} are evaluated to basic values, objects, or roles b_0, …, b_{k-1}. Note that all the roles of o_1 form a tree T rooted at o_1. When evaluating $e_1.m(a_0, \ldots, a_{k-1})$, the dynamic classification mechanism tries to find a highest role o' in T that has a method that can be invoked by the expression $m(b_0, \ldots, b_{k-1})$ in the normal semantics of Java. Particularly, if o_1 encapsulates a method that satisfies methond invocation $m(b_0, \ldots, b_{k-1})$, execution of expression $e_1.m(a_0, \ldots, a_{k-1})$ amounts to executing expression $o_1.m(b_0, \ldots, b_{k-1})$; otherwise, a role o' of o_1 is responsible to execute $m(b_0, \ldots, b_{k-1})$. A Java method that searches for a highest role o' in T that can execute $m(b_0, \ldots, b_{k-1})$ will be described in Section 4.5.

3 EXTENDING THE JAVA COMPILER AND STANDARD LIBRARY

3.1 Extending the Java Compiler

A Java class has at most one *class initialization method*, which is invoked implicitly by the Java virtual machine as part of the class initialization process [Lindholm and Yellin 1999]. A class initialization method cannot be included directly in a Java program. It is never invoked directly from any Java virtual machine instruction. Initialization of a class consists of executing static initializers and static field initializers defined in the class.

The Java compiler translates a class definition to a class file in a format defined in the Java virtual machine specification [Lindholm and Yellin 1999]. In the class file, each



method, including each instance initialization method and the class initialization method, is described by a method_info structure. A code attribute in the method_info structure contains the Java virtual machine instructions and auxiliary information for the method. Particularly, a code array in the code attribute gives the actual bytes of the Java virtual machine instructions that realize the method. A Java virtual machine instruction consists of an opcode that specifies the operation to be performed, followed by zero or more operands that embody values to be operated upon. The latest specification of the Java virtual machine defines 201 opcodes for the Java compiler to compose Java virtual machine instructions and 3 opcodes reserved for implementations of the Java virtual machine.

A support for the dynamic classification mechanism in Java programming does not need to extend the instruction set of the Java virtual machine or change the class file format. It requires the Java compiler to translate the DCDs into static initializers in class definitions before the class definitions are compiled to class files. The Java compiler also needs to translate dynamic classification expressions appeared in a method definition to Java code before it compiles the method definition to a method_info structure. We now explain how to compile DCDs and dynamic classification expressions with the extended Java compiler.

In the next subsection, we propose extending the Java standard library with a class named `Dynamic` for running the dynamic classification mechanism at runtime. Class `Dynamic` contains two static fields named `DCAs` and `dynamicLinks` to hold DCAs and parent-child links, respectively. To simplify the discussion, we assume the static fields hold objects of a subclass of class `Hashtable`. Hash table `DCAs` uses the composite of a dynamic parent D_1 and a label `l` as a key and the dynamic child D_2 of the DCA $\langle D_1, l, D_2 \rangle$ as value to store the DCA. Assume a source code file that declares a DCD, which is either stand-alone or a dynamic-clause. As indicated in Section 2.1, the DCD specifies a set of DCAs. The Java compiler adds a static initializer into each class definition in the source code file. The inserted static initializer includes a statement similar to the following one for each DCA $\langle D_1, l, D_2 \rangle$ implied by the DCD. Thus, the Java compiler can eliminate DCDs from source code files and stores DCAs into the hash table `DCAs` in class `Dynamic`.

```
if (Dynamic.DCAs.containsKey(D1, l)) {
   if (!Dynamic.DCAs.get(D1, l).equals(D2))
      throw new DynamicException();
}
else Dynamic.DCAs.put(D1, l, D2);
```

A DCA is used to link parent o_1 to child o_2 when the child role o_2 is created by a newChild-expression. The newChild-expression instantiates a class C_2 to create the child role o_2. The Java compiler translates the newChild-expression

```
o1 newChild(D1, l, D2, C2)
```

to source code similar to the method invocation expression

```
Dynamic.addLink(o1, new C2(), D1, l, D2)
```

The static method `addLink` of class `Dynamic` removes any previous link that is labeled with DCA ⟨D_1, `l`, D_2⟩ between o_1 and any role, generates a parent-child link between parent o_1 and the newly created role o_2 of class C_2, and labels the link with the DCA ⟨D_1, `l`, D_2⟩. Class `Dynamic` uses hash table `dynamicLinks` to keep parent-child links. The static method `addLink` adds links into the hash table when newChild-expressions are realized. The Java compiler translates other newChild-expressions similarly to method invocation expressions, which can be compiled into Java virtual machine instructions. A static method `removeLink` of class `Dynamic` can be used to remove links for realizing removeAll-, removeSelf-, and removeChild-expressions that appear in method bodies.

A child-expression o_1 `child(`D_1`, l)` placed in a method body accesses a child o_2 of parent o_1 such that the parent and child are linked with DCA ⟨D_1, `l`, D_2⟩ for some class D_2. As indicated above, hash table `dynamicLinks` in class `Dynamic` contains the link if the link exists. Hence, the child-expression can be compiled to Java source code that uses arguments o_1, D_1, and `l` to access hash table `dynamicLinks`. Similarly, the Java compiler translates a parent-expression o_2 `parent` to Java source code that accesses hash table `dynamicLinks` to find the parent of o_2.

When expression e_1`.v` is used as an rvalue or argument, if the Java compiler decides a type C_1 for expression e_1, it can decide whether the class or interface C_1 defines or inherits field `v`. If C_1 does, the Java compiler keeps expression e_1`.v` as is so that it will translate the expression to Java virtual machine instructions directly; otherwise, the Java compiler generates source code to perform the following actions at runtime:

F1. Visit roles associated with o_1 in a breadth-first manner to search for roles `o'` that encapsulate field `v`;
F2. Return field `v` in the role `o'` as the value of expression e_1`.v` if step F1 returns only one role `o'`.

The generated Java code may invoke a method of class `Dynamic` to perform action F1 or F2. In the description of step F2, we assumes action F1 finds exactly one role `o'`. Other cases in which action F1 finds no role or multiple highest roles that have field `v` are left for an implementation of the dynamic classification mechanism to resolve.

We can program action F1 in Java. The action at runtime accesses hash table `dynamicLinks` in class `Dynamic` to find the roles `o'` of o_1 that encapsulate field `v`. It applies Java reflection feature to decide whether a role `o'` encapsulates field `v`. The field `v` is reflected as an object `v_field` of class `Field`. Action F2 is coded by invoking method `get` of the `Field` object `v_field` with argument `o'`. In the description of the prototypical implementation, we shall show how to code actions F1 and F2.

The Java compiler can handle a method invocation expression e_1`.m(`a_0, …, a_{k-1}`)` similarly. Particularly, if the Java compiler can evaluate a type C_1 for expression e_1 and the class or interface C_1 defines or inherits a method named `m` and arguments a_0, …, a_{k-1}

can be substituted for the method parameters, the Java compiler translates the expression to Java virtual machine instructions directly. Otherwise, the Java compiler generates source code from the method invocation expression to invoke the dynamic classification mechanism at runtime. The source code searches for a role `o'` of the object or role denoted by e_1 such that `o'` defines or inherits method `m` that can be invoked by expression `m(`a_0`, ..., `a_{k-1}`)`. The source code executes the found method `m` for object `o'` through the Java reflection feature. It invokes a standard static method of class `Dynamic` to perform the actions.

3.2 Standard Class `Dynamic`

The Java virtual machine starts up by creating an initial class using the bootstrap class loader [Lindholm and Yellin 1999]. It then links the initial class and invokes the `main` method of the initial class. The `main` method drives further execution and may cause additional classes and interfaces to be linked. The creation of a class or interface `C` constructs an internal representation of `C` in the method area of the Java virtual machine. It can be triggered by another class or interface `D` that references `C` or invokes methods of `C`. If `C` is not an array class, it is created by loading a binary representation, compiled by the Java compiler.

The Java virtual machine contains explicit support for objects. An *object* is either a dynamically allocated class instance or an array. A reference to an object is considered to have a Java virtual machine type reference. Values of type reference can be thought of as pointers to objects. More than one reference to an object may exist. Objects are always operated on, passed, and tested via values of type reference.

Roles created at runtime are implemented with objects that are linked to existing objects or roles through DCAs. The `Dynamic` class is responsible to maintain the links, which reference roles. We propose adding the class `Dynamic` to the standard Java library. The class `Dynamic` supports the dynamic classification mechanism with static fields `DCAs` and `dynamicLinks`. The field `DCAs` holds DCAs specified by the source code files of a Java program. The field `dynamicLinks` holds existing parent-child links during the program execution. Any statement or expression that invokes the dynamic classification mechanism triggers the loading of class `Dynamic`, which fills container `DCAs` with DCAs declared in the program and creates container `dynamicLinks`.

In addition to containers `DCAs` and `dynamicLinks`, the class `Dynamic` provides utility methods to serve the code generated from dynamic classification expressions by the Java compiler. The generated code does not access the above containers directly. It invokes the utility methods to fulfill the responsibilities of dynamic reclassification and declassification as described in Section 2.2 and that of role access as described in Section 2.3. For example, a utility method named `invokeMethod` in class `Dynamic` searches a role `o'` for an invoked method. The utility method will be explained in Section 4.5.

4 A PROTOTYPICAL IMPLEMENTATION

The prototypical implementation of the proposed dynamic classification mechanism consists of classes `RoleProcessor`, `Dynamic`, and several other classes. The `main` method of class `RoleProcessor` translates dynamic classification declarations and expressions that appear in a program. Class `Dynamic` is used to support the execution of the source code generated by class `RoleProcessor`. We expect the functionality of class `RoleProcessor` to be incorporated into the Java compiler, class `Dynamic` to be included in the Java standard library. Thus, the dynamic classification mechanism becomes an integrated part of the Java language.

The `main` method of class `RoleProcessor` translates the source code files of a program that apply the dynamic classification feature to Java code files. For example, assume a program is composed of source code files `Employee.txt`, `Student.txt`, and `Experiment.java`. Only the first two files include DCDs and dynamic classification expressions. The command

```
java RoleProcessor Employee.txt Student.txt
```

invokes class `RoleProcessor` to translate files `Student.txt` and `Employee.txt` to Java code files `Student.java` and `Employee.java`. The Java compiler can compile each of the three Java code files of the program. The `main` method defined in class `Experiment` can be executed by command

```
java Experiment
```

which invokes a method defined in class `Student` that applies dynamic classification and, hence, triggers loading class `Dynamic` by the Java virtual machine.

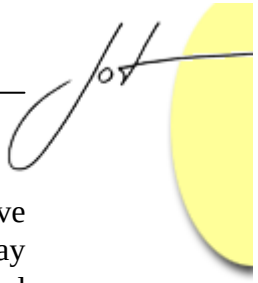
Source code files and compiled class files of the prototypical implementation of the dynamic classification feature for the Java language and sample files `Student.txt`, `Employee.txt`, and `Experiment.java` can be found at [Li 2003]. The prototypical implementation is detailed as follows.

4.1 Class Definition

In a Java class definition, an implements-clause introduces superinterfaces for the class. For simplicity of class `RoleProcessor`, we assume a user-defined class does not include implements-clause. A source code file may define several classes. We allow DCDs to be placed anywhere outside class definitions in a source code file. A DCD in the form of a dynamic-clause can be inserted between the head and body of any user-defined class.

For simplicity of class `RoleProcessor`, we assume the body of a user-defined class consists of variable declarations and method definitions. A *variable declaration* is in the form

```
<modifiers> <type> <varName> ;
```



which has no initialization clause and which is terminated with a semicolon. In the above grammar rule, non-terminal `<modifiers>` denotes a series of modifiers, which may include a visibility keyword such as `public`, `private`, or `protected` and keyword `static`. To facilitate access to the declared variables in a role by invoking the Java reflection feature, we assume the visibility of the variables is `public`. The restriction can be removed by an implementation of the dynamic classification mechanism since the Java reflection can be extended to reflect non-public inherited fields.

A *method definition* in a Java program is in the form

```
<modifiers> [ <returnType> ] <methodName> (<parameters>) {
  { <statement> }*
}
```

If a method definition does not specify a return type, it defines a constructor and the method name denoted with non-terminal `<methodName>` is same as the class name. In Java, a constructor is not inherited by subclass. Non-terminal `<parameters>` denotes a series of parameter declarations, each of which consists of a type name and a parameter name. For simplicity, we assume the type name is either `double` or a class name so that class `RoleProcessor` does not have to handle array parameters.

In the Java virtual machine [Lindholm and Yellin 1999, Section 7.6], `n` arguments passed by a method invocation expression are received in the local variables numbered `1` through `n` in the frame created for the method invocation. Local variables declared in the method body are also implemented with local variables in the frame. A method body may use parameters as local variables. Without loss of generality, we assume a method body in a user-defined class does not declare any local variable.

4.2 DCDs and Symbol Table

Class `RoleProcessor` collects variables and method signatures of user-defined classes of a program. It also collects DCAs from the source code files of the program. The first activity of the class `RoleProcessor` is a normal function of the Java compiler. The second activity for handling DCDs can be added to the Java compiler. Class `RoleProcessor` uses static fields `inheritTable`, `symbolTable`, and `DCAs` to collect inheritance, symbols (variables and method signatures), and DCAs declared in user-defined source code files. We explain the three data structures as follows.

Data structure `inheritTable` is implemented in class `RoleProcessor` with a hash table. It stores the name of a user-defined class as a key and the superclass declared in the class definition as the value. We assume that a declared superclass that is not a user-defined class is in package `java.lang`. This assumption can be extended to other packages easily. Note that a user-defined class has class `java.lang.Object` as its default superclass.

Data structure `symbolTable` is a hash table with user-defined class names as keys. For each user-defined class `ownerClass`, we put key `ownerClass` and a hash table

`table` as the value into hash table `symbolTable`. If class `ownerClass` defines a field `v` or inherits field `v` from a user-defined class, we put field name `v` as key and the declared type `T` of `v` as the value into hash table `table`. If class `ownerClass` defines a method `m` or inherits method `m` from a user-defined class, we put method name `m` as a key and a vector object `signatures` as the value into the hash table `table`. We collect the signatures of methods with name `m` that are defined in class `ownerClass` or inherited by class `ownerClass` from a user-defined class into vector `signatures`. Hash table `inheritTable` is used in deciding fields and methods inherited from user-defined classes. It will not be used by class `Dynamic` at runtime.

Data structure `DCAs` is a hash table that uses a dynamic parent D_1 as a key and a hash table `labelChilds` as the value. It takes the advantage of the constraint that a dynamic parent D_1 and a label `l` uniquely determine a dynamic child D_2. For each DCA $\langle D_1, l, D_2 \rangle$ declared in a program, key `l` and value D_2 are saved in the hash table `labelChilds`, which is the value for key D_1 in hash table `DCAs`.

4.3 Field and Method Resolution at Compilation Time

For an expression that accesses a field `fieldName` or invokes a method `methodName` of an object o_1, the Java compiler can decide whether (the type of) object o_1 has the field or method. Class `RoleProcessor` depends on the same decision to determine whether the field or method should be searched for in the roles of o_1. The fields and methods of user-defined classes can be reflected only after the classes are compiled by the Java compiler and loaded by the Java virtual machine. They cannot be reflected at compilation time. In the class `RoleProcessor`, static methods `searchField(objType, fieldName)` and `searchMethod(objType, methodName, argTypes)` decide whether a user-defined class `objType` defines or inherits field `fieldName` or method `methodName`. The method `searchMethod` compares the argument types kept in array `argTypes` with parameter types of the methods of class `objType`. The methods `searchField` and `searchMethod` are explained as follows. Details such as exception handling and utility method invocations are omitted for easy understanding.

Method invocation `searchField(objType, fieldName)` returns the name of the type of the field `fieldName` that is declared or inherited by a user-defined class `objType` if `objType` defines or inherits the field; otherwise, it returns `null`. It implements the following algorithm, which is explained shortly.

```
Hashtable table = (Hashtable) symbolTable.get(objType);
if (table.containsKey(fieldName))
   return (String) table.get(fieldName);
String superclass = objType;
while (inheritTable.containsKey(superclass))
   superclass = (String) inheritTable.get(superclass);
if (symbolTable.containsKey(superclass))
   superclass = "java.lang.Object";
```

```
else
   if (superclass.indexOf('.') == -1)
      superclass = "java.lang." + superclass;
return Class.forName(superclass).
             getField(fieldName).getType().getName();
```

The hash table `symbolTable` contains the name `objType` of each user-defined class as a key and a hash table `table` as the value for the key. The hash table `table` contains each field `fieldName` defined or inherited from a user-defined class by class `objType` as a key and the declared type of the field as the value. In the above algorithm, the first two statements resolve `fieldName` by accessing hash table `table`. If symbol `fieldName` cannot be resolved, the algorithm accesses hash table `inheritTable` to find the first superclass `superclass` that has no entry in the hash table `inheritTable`. If the class `superclass` has an entry in hash table `symbolTable`, it is a user-defined class and it has the default superclass `java.lang.Object`; otherwise, the class `superclass` is a standard Java class. In the latter case, if the class name `superclass` does not include a package name, the algorithm adds the prefix `java.lang.` to the class name. The last statement in the algorithm applies the Java reflection feature to retrieve the declared type of field `fieldName` of the standard Java class `superclass`. The method `searchField` returns `null` when it cannot resolve `fieldName` by applying the above algorithm.

The static method `searchMethod(objType, methodName, argTypes)` of class `RoleProcessor` uses an algorithm similar to the above one to decide whether a user-defined class `objType` defines or inherits a method `methodName` such that the method parameters can be replaced by arguments of types `argTypes`. It returns the return type of the method if class `objType` defines or inherits the method; otherwise, it returns `null`.

4.4 Statements

As indicated in Section 4.1, a method body is a series of statements enclosed within a pair of braces `{}`. Class `RoleProcessor` recognizes the following kinds of statement. Other statements can be handled similarly.

```
<assignment> ::= <lvalue> = <rvalue> ;
<methodInvocation> ::= <varMethodName> ( <arguments> );
<assiMethodInvocation> ::= <lvalue> = <methodInvocation>
<new> ::= [ <lvalue> = ] new <className> ( [ <arguments> ] );
<newChild> ::= [ <lvalue> = ] <lvalue> newChild
     (<className> , <label>, <className>, <className> ); |
              [ <lvalue> = ] <lvalue> newChild
                                 (<className> , <label> ); |
             [ <lvalue> = ] <lvalue> newChild ( <label> );
<removeAll> ::= <lvalue> removeAll ;
```

```
<removeSelf> ::= <lvalue> removeSelf ;
<removeChild> ::= <lvalue> removeChild (<className> ,
                                        <label> );
<parent> ::= <lvalue> parent ;
<child> ::= <lvalue> child ( <className> , <label> );
<return> ::= return [ <rvalue> ] ;
```

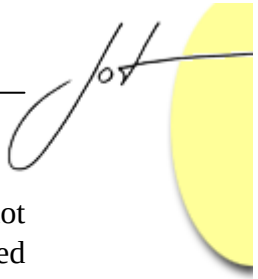
For simplicity of class RoleProcessor, we assume non-terminal <lvalue> in the above grammar rules denotes expressions in three forms:

```
<lvalue> ::=<identifier> | this . <varName> |
             <identifier> . <varName>
```

An <lvalue> expression that consists of more than two identifiers can be handled similarly. An identifier denoted by non-terminal <identifier> in the above rule can be a parameter or (instance or static) variable. Non-terminal <varName> denotes a (instance or static) variable. Keyword this references an object for which the method that includes the <lvalue> expression is invoked. Each <lvalue> expression denotes either a parameter or a (instance or static) field. It simulates the Java compiler to evaluate a type for each <lvalue>. We explain the type evaluation by class RoleProcessor as follows.

Assume an <lvalue> appeared in method m of user-defined class ownerClass. If the <lvalue> consists of a single identifier d, d must be a parameter or a field defined or inherited by class ownerClass. If the <lvalue> is this.v, identifier v must be a variable defined or inherited by class ownerClass. If the <lvalue> is d.v, the identifier d must be a parameter or a field defined or inherited by class ownerClass and, therefore, class RoleProcessor can evaluate a type D from identifier d. Then, class RoleProcessor searches for field v in the class or interface D. The search for field d or v in a class or interface invokes the method searchField, which was described in Section 4.3. Class RoleProcessor does not change any <lvalue> expression, which will be compiled by the Java compiler to an operand of a Java virtual machine instruction for storing a value or reference into the parameter or field denoted by the <lvalue> expression.

For simplicity of class RoleProcessor, we assume an <rvalue> expression in a statement is a double value, a string constant, or an expression in one of the three <lvalue> forms. Assume an <rvalue> that is in an <lvalue> form. Like handling an <lvalue>, class RoleProcessor tries to evaluate a type for the <rvalue> at compilation time. If it can evaluate a type, class RoleProcessor does not modify the <rvalue> expression and the expression will be compiled by the Java compiler to an operand of a Java virtual machine instruction, which retrieves a value from the parameter or field denoted by the <rvalue>. Otherwise, class RoleProcessor invokes class Dynamic at runtime to access parent-child links for retrieving a field in a role.



For example, assume the type `GraduateStudent` of parameter `tom` does not define or inherit field `salary`. The `<rvalue>` expression `tom.salary` is translated by class `RoleProcessor` to method invocation expression `Dynamic.DynamicField(tom, "salary")`, which invokes the method `dynamicField` of class `Dynamic`. Method `dynamicField` searches for field `salary` in a role of object `tom`. The search performed at runtime follows parent-child links in a breadth-first manner. It is described in the next subsection.

In the grammar rule for a `<new>` statement, non-terminal `<arguments>` denotes a series of `<rvalue>` expressions, which are separated with comma `,`. The class `RoleProcessor` processes each argument in a way described above. It also generates Java source code that uses an array to hold the argument values and that invokes the constructor mentioned in the `<new>` statement with the array elements as arguments. Thus, an object is created.

In the grammar rule for non-terminal `<methodInvocation>`, the non-terminal `<varMethodName>` denotes an expression in one of the three forms:

```
<varMethodName> ::= <methodName> | this . <methodName> |
                    <identifier> . <methodName>
```

Non-terminal symbol `<methodName>` denotes the name of an instance or static method, `<identifier>` denotes a parameter or field. Non-terminal `<methodInvocation>` denotes a method invocation in one of three forms: $\texttt{methodName}(a_1, \ldots, a_k)$, $\texttt{this.methodName}(a_1, \ldots, a_k)$, and $\texttt{d.methodName}(a_1, \ldots, a_k)$ with $k \geq 0$. We explain how the static method `invokeMethod` of class `RoleProcessor` handles the three method invocation expressions as follows. We assume the method invocation expressions appear in the definition of class `ownerClass`.

First, method `invokeMethod` decides whether all the `<rvalue>` expressions $a_1, \ldots, a_k$ can be evaluated without invoking the dynamic classification mechanism. Suppose the answer is positive. If the method invocation expression is in the form $\texttt{methodName}(a_1, \ldots, a_k)$, method `invokeMethod` searches the invoked method `methodName` in class `ownerClass`. If class `ownerClass` defines or inherits an instance or static method that can be invoked with the expression, the method `invokeMethod` does not translate the expression; otherwise, the method invocation expression is prefixed with string `"this."` and, thus, method `invokeMethod` needs to handle only the last two forms of method invocation expression. For the last two forms, method `invokeMethod` evaluates a type `varType` for keyword `this` or identifier `d` and, then, searches `methodName` in the class or interface `varType` with the types of arguments $a_1, \ldots, a_k$. If both the evaluation and search succeed, the method `invokeMethod` does not translate the statement. Otherwise, method `invokeMethod` generates Java source code that will

IM1 Create array `arguments` for holding argument values;

IM2 Evaluate each argument and stores the argument value into array `arguments`;

IM3 Invoke static method `invokeMethod` of class `Dynamic`.

At runtime, the method `invokeMethod` of class `Dynamic` searches `methodName` in roles and will be explained in the following subsection. For example, assume class `GraduateStudent` does not define method `setSalary`. Method `invokeMethod` of class `RoleProcessor` translates the statement

```
tom.setSalary(2000);
```

for a graduate student object `tom` to Java source code

```
Object[] arguments = new Object[1];
arguments[0] = new Double(2000.0);
Dynamic.invokeMethod(tom, "setSalary", arguments);
```

which invokes the static method `invokeMethod` of class `Dynamic` at runtime to search for a role of object `tom` that can execute expression `setSalary(2000)`.

Other kinds of statements for dynamic classification use keywords to identify the intended operational semantics. Their executions depend on the dynamic classification mechanism. They are translated to expressions that invoke proper static methods of class `Dynamic`. For instance, class `RoleProcessor` translates expression o_1 **`child`**`(D, l)` to expression `Dynamic.child(o1, D, l)`. The static method `child` of class `Dynamic` accesses hash table `dynamicLinks` at runtime to find a child o_2 that is linked from parent o_1 through a DCA ⟨`D`, `l`, `D'`⟩.

4.5 Class `Dynamic`

At runtime, class `Dynamic` uses hash tables `DCAs` and `dynamicLinks` to hold DCAs declared in a program and parent-child links created during the program execution. It defines static methods invoked by source code generated by class `RoleProcessor`. Particularly, static methods `dynamicField` and `invokeMethod` of class `Dynamic` access field or invoke method of a role, static methods `addLink`, `removeAll`, `removeSelf`, and `removeChild` manage hash table `dynamicLinks`, and static methods `parent` and `child` access the hash table `dynamicLinks`. We use method `invokeMethod` to illustrate how class `Dynamic` supports dynamic classification.

The method `invokeMethod(obj, methodName, arguments)` of class `Dynamic` starts a breadth-first search for a role, starting from the given object or role `obj`, that defines a method `methodName` that can be invoked with arguments in array `arguments`. The argument types are stored in an array `argTypes`. The breadth-first search uses a hash set `roleSet` to keep object `obj` and its roles to be searched for. During the search, if a role `role` defines a method `meth` whose name is `methodName` and whose parameter types are supertypes of argument types, the method `meth` is placed into hash set `candidateMethods` and `role` is placed into hash set `candidateRoles`; otherwise, the child roles of `role` are placed into hash set `childRoles`. After the roles in `roleSet` are examined, if set `childRoles` is not

empty, set `childRoles` is assigned to `roleSet` so that the breadth-first search can examine roles at the next level. After the breadth-first search is completed, if set `candidateRoles` contains only one role `r` and set `candidateMethods` contains only one method `m`, the `invokeMethod` method invokes the Java reflection to execute method `m` for object `r`.

5 CONCLUSION

A dynamic classification mechanism is proposed for the Java language. It supports programmers to code dynamic classification, which allows objects and roles at runtime to take on new roles and drop the roles. Thus, data members and functions of new roles can be introduced to existing objects and roles dynamically. The proposed mechanism allows a programmer to declare any class, standard or user-defined, as a dynamic parent and as a dynamic child. Thus, standard classes can also be used to create and take roles. Roles are implemented as objects, which are linked to objects or other roles through DCAs.

The prototypical implementation shows that the dynamic classification mechanism can be added to the Java language without changing the Java virtual machine specification. Dynamic classification declarations and expressions can be translated by the extended Java compiler to Java source code. At runtime, a standard class `Dynamic` can be used to support dynamic classification operations.

REFERENCES

[Albano00] A. Albano, G. Antognoni, and G. Ghelli. “View Operations on Objects with Roles for a Statically Typed Database Language”. *IEEE Transactions on Knowledge and Data Engineering*, Vol. 12, July/August 2000, 548-567.

[Arnold00] K. Arnold, J. Gosling, and D. Holmes. *The Java Programming Language–Third Edition*. Addison-Wesley, Reading, Mass., 2000.

[Bachman77] C.W. Bachman and M. Daya. “The role concept in data models”. In *Proceedings of Third International Conference on Very Large Data Bases (VLDB’77)*, pp. 464-476, Tokyo, Japan, October 1977.

[Bäumer00] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. “Chapter 2 Role Object”. In book *Pattern Languages of Program Design 4*, editors N. Harrison, B. Foote, and H. Rohnert, Addison-Wesley, Reading, Mass., 2000.

[Drossopoulou02] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini: “More dynamic object reclassification: FickleII”. *ACM Transactions on Programming Languages and Systems*, 24(2): 153-191, March 2002.

[Fowler97] M. Fowler. "Dealing with Roles". In *Proceedings of the 4th Annual Conference on the Pattern Languages of Programs*, Monticello, Illinois, USA, September 1997. http://www.martinfowler.com/apsupp/roles.pdf.

[Ghelli99] G. Ghelli and D. Palmerini. "Foundations for extensible objects with roles". In *Proceedings of 6th Workshop on Foundations of Object-Oriented Languages* (FOOL 6), San Antonio, Texas, 1999.

[Gottlob96] G. Gottlob, M. Schrefl, and B. Röck. "Extending object-oriented systems with roles". *ACM Transactions on Information Systems*, Vol. 14, July 1996, 268-296.

[Kristensen97] B.B. Kristensen. "Subject composition by roles". In *OOIS'97*, Brisbane, Australia, 1997.

[Li03] L. Li. A prototypical implementation of dynamic classification, http://www.uwindsor.ca/liwu/.

[Lindholm99] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification, Second Edition*. Sun Microsystems, Inc., 1999.

[Richardson91] J. Richardson and P. Schwarz. Aspects: "Extending objects to support multiple, independent roles". In *ACM SIGMOD Conference*, pages 298-307, New York, 1991.

[Rossie95] J. G. Rossie Jr. and D. P. Friedman. "An algebraic semantics of roles". In *OOPSLA'95*, pages 187-199, Austin, Texas, October 1995.

[Rossie96] J. G. Rossie Jr., D. P. Friedman, and M. Wand. "Modeling role-based inheritance". In *ECOOP'96*, pages 248-274, Linz, Austria, July 1996.

[Sciore89] E. Sciore. "Object specializations". *ACM Transactions on Information Systems*, Vol. 7, 1989(April), 103-122.

[VanHilst96] M. VanHilst and D. Notkin, "Using role components to implement collaboration-based designs". In *OOPSLA'96*, pp. 359-369, San Jose, CA, October 1996.

[Wieringa94] R. Wieringa, W. de Jonge, and P. Spruit. "Roles and dynamic subclasses: a modal logic approach". In *ECOOP'94*, pp. 32-59. 1994.

About the author

Dr. Liwu Li is an associate professor in School of Computer Science at University of Windsor, Canada. His research interests include objectoriented language design and implementation and software process design and execution. He can be reached at liwu@uwindsor.ca.