Dynamic Class Methods in Java

Christian Heinlein

Dept. of Computer Structures, University of Ulm, Germany
heinlein@informatik.uni-ulm.de

Abstract. The concept of *dynamic class methods* in Java, constituting a specialization of a general new programming language concept called *dynamic routines*, is introduced and applied to a simple case study. Its advantages over standard object-oriented programming techniques including design patterns are demonstrated. Furthermore, an implementation of dynamic class methods as a precompiler-based language extension to Java is described.

1 Introduction

Similar to *aspects* in AOP [9, 10], *dynamic routines* are a general new concept to make programming languages more flexible and to support extensibility of software systems along various dimensions beyond the capabilities of object-oriented solutions. *Dynamic class methods* presented in this paper, are a specialization, adaptation, and integration of this general concept into the programming language Java [7]. To demonstrate their usefulness, a simple case study is presented in Sec. 2, whose implementation with standard object-oriented programming techniques including design patterns [5, 2] is briefly sketched in Sec. 3 and then contrasted with a functionally equivalent, but significantly shorter and less complex implementation based on dynamic class methods in Sec. 4.

To complement the rather informal introduction of dynamic class methods there, Sec. 5 provides a more detailed description of the concept and its integration into the Java language, while Sec. 6 describes the basic ideas of its transformation to pure Java code by means of a simple precompiler. Finally, Sec. 7 concludes the paper with a discussion of the concept itself and of related work.

An accompanying Technical Report [8] provides a more elaborate presentation of the object-oriented implementation of the case study (cf. Sec. 3) as well as a more detailed description of the transformation to Java (cf. Sec. 6).

2 A Simple Case Study

In the following, the evolution of a simple software package for the management of arithmetic expressions is described. In every stage of the development, none of the source code produced in previous stages shall be modified or recompiled. Furthermore, while developing the code of the current stage, the requirements of

subsequent stages might not be known. Therefore, appropriate techniques have to be employed from the very beginning to support a maximum of flexibility and extensibility of the system.

1. Develop a class hierarchy for the representation of arithmetic expressions consisting of *variables* (with a name and an integer value) and *binary operators* for the four basic arithmetic operations.
   Implement methods to *evaluate* a given expression, i. e., determine its value, and to *print* an expression on the standard output stream.
2. Add methods for the *symbolic differentiation* of expressions.
   According to Fig. 1, which depicts the evolution of the system through the various stages by listing the supported categories of expressions on the vertical axis and the operations applicable to them on the horizontal axis, this kind of extension is called a *horizontal extension*.
3. Add a new category of expressions to represent *negation*, i. e., application of the unary minus operator.
   According to Fig. 1, this kind of extension is called a *vertical extension*.
4. Modify the behaviour of the evaluation method for divisions in such a way, that it catches the `ArithmeticException` arising from a division by zero and returns a special *null value* in that case (which might be represented, for instance, by the smallest available integer value).
   Also modify the behaviour of the other evaluation methods to return that null value if one of their operands is equal to it.
   This kind of extension, whose proper visualization would actually require a third dimension added to Fig. 1, is called a *behavioural extension* or *modification*.

	`eval`	`print`	`diff`
`Var`			
`Add`			
`Sub`		stage 1	stage 2
`Mul`	stage 4		
`Div`			
`Neg`		stage 3	

Fig. 1. Evolution of the system

3 Achieving Extensibility with Design Patterns

To achieve the aims described in the previous section with a typical object-oriented programming language such as Java, two well-known design patterns

[5, 2] have to be employed from the very beginning: The *Visitor pattern* is needed to support later horizontal extensions by means of visitor classes (cf. stage 2 below), while some kind of *Factory pattern* is used to implement later behavioural extensions by introducing subclasses with modified behaviour and adjusting the factory to instantiate these subclasses instead of the original classes (cf. stage 4 below).

Using the same grayscales as in Fig. 1, Fig. 2 depicts all classes and interfaces developed in the various stages and their basic relationships to each other. Due to space limitations, no source code is presented in this section; the interested reader is referred to [8].

1. In the first stage, an abstract root class `Expr` with abstract subclasses `Atom` and `Binary` and concrete subclasses `Var`, `Add`, `Sub`, `Mul`, and `Div` is declared. The methods implemented by these classes are `eval` (to evaluate an expression), `print` (to print it on the standard output stream), and `accept` (the generic entry point for visitors).
   Furthermore, an accompanying factory class `Factory` implementing methods such as `createVar`, `createAdd`, etc. as well as an interface `Visitor` declaring methods such as `visitVar`, `visitAdd` etc. are declared in this stage.
2. In the second stage, a visitor class `DiffVisitor` implementing the interface `Visitor` is introduced to add methods for symbolic differentiation of expressions.
3. In the third stage, classes `Unary` (abstract) and `Neg` (concrete) are declared as new subclasses of `Expr` to introduce negation as a new category of expressions.
   Furthermore, an extended factory class `Factory3`[1] implementing an additional method `createNeg` as well as an extended visitor interface `Visitor3` declaring an additional method `visitNeg` are declared.
   Finally, to extend the `DiffVisitor` class introduced in the second stage to the new category of expressions introduced in this stage, an extended `DiffVisitor3` class implementing the extended `Visitor3` interface is needed.
4. In the fourth stage, subclasses `Add4`, `Sub4`, etc. of the original classes `Add`, `Sub`, etc. are declared which override the method `eval` in the way described in Sec. 2.
   To actually "install" the corresponding behavioural modification, a subclass `Factory4` of `Factory3` is introduced that overrides the methods `createAdd`, `createSub`, etc. to instantiate the new classes `Add4`, `Sub4`, etc. instead of their original counterparts `Add`, `Sub`, etc.

The resulting system consists of 10 "essential" classes (`Expr` and its first two levels of subclasses) and 12 "helper" classes whose sole purpose is to support the system's modular extensibility.

[1] To simplify "terminology," the current stage number 3 is used as a suffix for some names introduced in this stage, even though corresponding names with suffixes 1 and 2 do not exist.

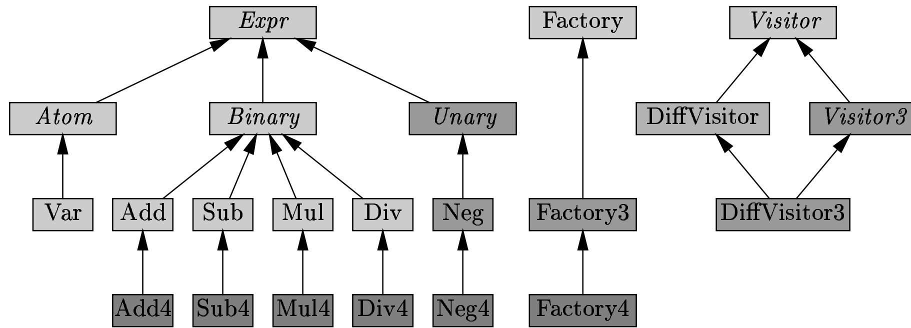

Fig. 2. Overall view of the system

4 Achieving Extensibility with Dynamic Class Methods

Simply speaking, *dynamic class methods* are class methods (i. e., "static" methods in Java terminology) which can be overridden in other classes (and therefore are actually not "static"). Due to these properties, they constitute a generalization of both class and instance methods, and even "virtual constructors" (i. e., constructors which can be overridden in other classes) can be implemented with them. Therefore, by strict application of dynamic class methods, factory and visitor patterns (and probably some other design patterns, too) become obsolete, because the required flexibility and extensibility can be achieved much more directly and easily. This will be demonstrated in the following by re-implementing the software system of Sec. 3 with dynamic (class) methods.[2]

Stage 1: Basic Class Hierarchy and Methods

Figures 3 and 4 show the code written in stage 1 to support evaluation and printing of variables and the four basic arithmetic operations.

Instead of providing instance methods `eval` and `print` which possess an implicit parameter `this` of type `Expr`, equally named dynamic methods are defined in the root class `Expr` which receive an explicit parameter `x` of type `Expr` because dynamic methods, just like static methods, do not possess an implicit parameter. By declaring them `abstract`, it is expressed, similar to abstract instance methods, that they are expected to be overridden in other classes.

Examples of such redefinitions appear in classes `Var` and `Add`, where the qualified names `Expr.eval` and `Expr.print` are used to refer to the methods' original definitions in class `Expr`. Each such redefinition completely replaces the previous definition of the method, i. e., afterwards calls of any client code to `Expr.eval` or `Expr.print` are redirected to the new definition. In the body of such a redefinition, the method's previous definition is available as a parameterless pseudo-method named `dynamic`[3], similar to the way `super` can be used

[2] To simplify the writing, the term "dynamic class methods" will be abbreviated to "dynamic methods" in the sequel.

```
// General expression.
public abstract class Expr {
  public dynamic abstract int eval (Expr x);
  public dynamic abstract void print (Expr x);
}

// Atomic expression.
public abstract class Atom extends Expr {}

// Variable expression.
public class Var extends Atom {
  private String name;
  private int val;
  protected Var (String n, int v) { name = n; val = v; }
  public String name () { return name; }
  public int val () { return val; }

  // Dynamic factory method.
  public dynamic Var create (String n, int v) {
    return new Var(n, v);
  }

  // Redefine dynamic methods of class Expr.
  dynamic int Expr.eval (Expr x instanceof Var) {
    return x.val;
  }
  dynamic void Expr.print (Expr x instanceof Var) {
    ......
  }
}
```

Fig. 3. Basic class hierarchy and methods (part 1)

in a subclass to call an overridden method of a superclass. In contrast to `super` calls, however, a call to `dynamic` does not receive explicit parameters because the original parameter values are implicitly passed unchanged (even if the formal parameters are modified before `dynamic` is called). By that means, a linked list of definitions (also called *branches*) is built up for every dynamic method, with the latest definition at the head and the initial definition at the tail.

Because redefinitions normally want to alter the behaviour of the method only for a subset of its domain while retaining its original behaviour otherwise, they are typically coded as a conditional statement such as:

```
dynamic int Expr.eval (Expr x) {
  if (x instanceof Var) return ((Var)x).val;
```

[3] To avoid the introduction of another new keyword, such as `previous` or `original`, the keyword `dynamic` is reused for that purpose.

```
// Binary expression.
public abstract class Binary extends Expr {
  private Expr left, right;
  protected Binary (Expr l, Expr r) { left = l; right = r; }
  public Expr left () { return left; }
  public Expr right () { return right; }
}

// Addition.
public class Add extends Binary {
  protected Add (Expr l, Expr r) { super(l, r); }

  // Dynamic factory method.
  public dynamic Add create (Expr l, Expr r) {
    return new Add(l, r);
  }

  // Redefine dynamic methods of class Expr.
  dynamic int Expr.eval (Expr x instanceof Add) {
    return Expr.eval(x.left()) + Expr.eval(x.right());
  }
  dynamic void Expr.print (Expr x instanceof Add) {
    ......
  }
}

// Subtraction, multiplication, and division.
......
```

Fig. 4. Basic class hierarchy and methods (part 2)

```
    else return dynamic(); // Call previous branch.
  }
```

To simplify this frequently occurring pattern, it is possible to move the conditional statement acting as a *guard* outside the method body and omit its stereotyped else-part:

```
dynamic int Expr.eval (Expr x) if (x instanceof Var) {
  return ((Var)x).val;
}
```

By that means, the guard becomes a part of the method head which can be interpreted as a *precondition.*

To further simplify the definition of dynamic methods, an *explicit* guard as shown above might be turned into an *implicit* one by integrating it into the parameter list, yielding a *guarded parameter declaration*:

```
dynamic int Expr.eval (Expr x instanceof Var) {
```

```
    return x.val;
  }
```

The same technique works for all relational and equality operators (e. g., `(int i > 0, bool f == false)`), but in conjunction with `instanceof` it has the additional advantage that the static type of the affected formal parameter (`x` in the example) is automatically converted from its formal type (`Expr`) to its actual dynamic type (`Var`) *inside* the method body, thus eliminating the need for explicit casts there.

Note that this is quite different from directly using the latter type as the formal parameter type, which would yield a different method signature:

```
  dynamic int Expr.eval (Var x) {
    return x.val;
  }
```

Since there is no dynamic method with the signature `int eval (Var)` in class `Expr`, such a declaration would actually be erroneous.

In addition to appropriate redefinitions of the dynamic methods `Expr.eval` and `Expr.print`, every concrete subclass of `Expr` defines its own dynamic method `create` (e. g., `Var.create` and `Add.create`) acting as an overrideable factory method (i. e., a *virtual constructor*), thus eliminating the need for extra factory classes, even though no redefinitions will actually be required in the following.

Stage 2: Horizontal Extension

A key advantage of dynamic methods is the fact that horizontal extensions of a system can be implemented directly and easily, without needing to employ the visitor pattern, by means of additional dynamic methods which are – in contrast to static methods – by nature vertically extensible, too.

Figure 5 demonstrates this by defining a class `Diff` containing branches of a dynamic method `diff` performing symbolic differentiation for all concrete subclasses of class `Expr`. To ensure a uniform syntax, redefinitions of a dynamic method must always use a qualified method name such as `Diff.diff`, even if they appear in the same class as the original definition.

Stage 3: Vertical Extension

The code of Fig. 6, which implements the functionality of stage 3, is another impressive demonstration of the flexibility provided by dynamic methods. Instead of defining an extended `Visitor3` interface and an extended `DiffVisitor3` class as in Sec. 3 to extend the implementation of `diff` to the new `Neg` class, the dynamic `diff` method introduced in stage 2 is simply extended by an additional branch which is naturally integrated into the new class. As this example shows, a dynamic method might be redefined in *any* class where it is accessible, not just in subclasses of the class containing the original definition.

```
public class Diff {
  // Differentiate expression x along variable n.
  public dynamic Expr diff (Expr x instanceof Var, String n) {
    // Constant expressions are represented
    // as variable expressions with empty name.
    int c = x.name().equals(n) ? 1 : 0;
    return Var.create("", c);
  }
  dynamic Expr Diff.diff (Expr x instanceof Add, String n) {
    return Add.create(diff(a.left(), n), diff(a.right(), n));
  }
  ......
}
```

Fig. 5. Horizontal extension

However, to actually *enable* such an additional branch of a dynamic method, the class containing its definition must be *loaded* at runtime. In the present example, this happens naturally and automatically when an instance of class `Neg` is created. (If no such instance is ever created, the additional branch is not needed.)

Stage 4: Behavioural Extension

Finally, retroactive behavioural modifications of a system, which usually require considerable effort if they are possible at all without modifying existing code, can be done in a minute with dynamic methods, as shown in Fig. 7. (Again, it is necessary that the class containing additional branches of dynamic methods gets loaded; in this example, this might be achieved, e. g., by creating a dummy instance of it.)

This example demonstrates that the guards of a dynamic method are not restricted to dynamic type tests, but might test any property of their arguments (or even properties of their environment such as values of static variables). By that means, dynamic dispatching of method calls is not restricted to the dynamic type of *one* (implicit) argument, as with instance methods, but can be based on arbitrary properties of *all* arguments. This is even a generalization of multi-methods [11, 3, 1].

Furthermore, in contrast to the object-oriented solution in Sec. 3, where five new classes with almost identical redefinitions of `eval` had to be defined, only two redefinitions – one for unary and another one for binary operators – are needed with dynamic methods.

Summary

Even though the example presented in the previous sections is rather simple and small, the advantages of using dynamic methods are quite obvious: The

```
// Unary expression.
public abstract class Unary extends Expr {
  private Expr body;
  protected Unary (Expr b) { body = b; }
  public Expr body () { return body; }
}

// Negation.
public class Neg extends Unary {
  protected Neg (Expr b) { super(b); }
  public dynamic Neg create (Expr b) { return new Neg(b); }

  dynamic int Expr.eval (Expr x instanceof Neg) {
    return -Expr.eval(x.body());
  }
  dynamic void Expr.print (Expr x instanceof Neg) {
    ......
  }
  dynamic Expr Diff.diff (Expr x instanceof Neg, String n) {
    return Neg.create(Diff.diff(x.body(), n));
  }
}
```

Fig. 6. Vertical extension

code becomes significantly shorter and less complex, since no artificial helper classes like `Factory` and `Visitor` are needed. Furthermore, extending a system by defining additional branches of dynamic methods is straightforward and simple, while extensions based on design patterns are always somewhat tricky and less obvious.

5 Details of Dynamic Methods

To complement the rather informal description of dynamic methods presented so far, Fig. 8 shows the extensions to the Java grammar which have been introduced to integrate the concept into the language.

To simplify the grammar, the new keyword `dynamic` is included into the list of other method modifiers (line 1), even though it is incompatible with `static`, `final`, and `native`. Furthermore, a qualified method name containing one or more dots (line 2) as well as explicit and implicit guards (lines 3 and 4, respectively) are allowed for dynamic methods only. Likewise, usage of the keyword `dynamic` as a pseudo-method name in a primary expression (line 5) is restricted to bodies of dynamic methods.

To declare the *initial* branch of a dynamic method, a normal unqualified method name is used; to declare *subsequent* branches of the same method, its name is qualified by the name of the class containing the initial branch (which

```
public class NullExpr {
  public static final int NULL = Integer.MIN_VALUE;

  dynamic int Expr.eval (Expr x instanceof Unary)
  if (Expr.eval(x.body()) == NULL) {
    return NULL;
  }

  dynamic int Expr.eval (Expr x instanceof Binary) {
    if (Expr.eval(x.left()) == NULL
    || Expr.eval(x.right()) == NULL) return NULL;
    try { return dynamic(); }
    catch (ArithmeticException e) { return NULL; }
  }
}
```

Fig. 7. Behavioural extension

might itself be a qualified name), even if such branches are defined in the same class. Just like normal method declarations, declarations of initial branches must be unique within a class; on the other hand, it is allowed to declare multiple subsequent branches of the same dynamic method within the same class. Subsequent branches of a dynamic method must not throw more exceptions than its initial branch, just like an instance method overriding a method of a superclass must not throw more exceptions than the original method.

For the declaration of an initial branch, the access modifiers `public`, `protected`, none, and `private` determine both the accessibility of the dynamic method to clients and their ability to override it. Therefore, private dynamic methods are not very useful in practice, because they cannot be overridden in other classes. The access modifier of a subsequent branch is ignored if present, because such a branch cannot be called directly by a client.

The body of an initial branch of a dynamic method might be omitted iff the method is declared `abstract` and does not possess any explicit or implicit guards. This is equivalent to a declaration with the unsatisfied guard `if (false)` and an empty body. (Thus, abstract dynamic methods are not restricted to abstract classes.)

6 Transformation to Java

The basic idea of transforming source files containing dynamic method declarations to pure Java code is rather simple (cf. Fig. 9 for an illustration):

1. (a) The *initial branch* of a dynamic method, which is distinguished from subsequent branches by the fact that its method name is not qualified by a class name, is converted to an *instance method* (depicted by an ellipse) of a nested helper class (depicted as a rectangle).

```
MethodDeclaration:
  { "public" | "protected" | "private"
  | "static" | "dynamic" | "abstract" | "final"        // 1
  | "native" | "synchronized" | "strictfp" }
  ResultType <IDENTIFIER> { "." <IDENTIFIER> }         // 2
  "(" [ FormalParameter { "," FormalParameter() } ] ")"
  { "[" "]" } [ "throws" NameList ]
  { "if" "(" Expression ")" } ( Block | ";" )          // 3

FormalParameter:
  [ "final" ] Type VariableDeclaratorId
  [ ( "==" | "!=" ) InstanceOfExpression               // 4
  | "instanceof" Type                                  // 4
  | ( "<" | ">" | "<=" | ">=" ) ShiftExpression ]      // 4

PrimaryPrefix: "dynamic" "(" ")" | ......              // 5
```

Fig. 8. Extensions to the Java grammar

(b) A single instance of this class (graphically merged with the ellipse depicting the instance method) is stored in a static *variable* (depicted as a little square) of the surrounding class.

(c) Finally, a *static method* (depicted as a circle) with the same signature and access modifier as the dynamic method is generated, which calls the above instance method via this variable. This method constitutes the client interface to the dynamic method.

2. (a) A *subsequent branch* of a dynamic method, which is distinguished from the initial branch by the fact that its method name is qualified by the name of the class containing the declaration of the initial branch, is also converted to an *instance method* of a nested helper class which is declared as a *subclass* of the initial branch's helper class. Thus, this instance method overrides the instance method described above.

   (b) A single instance of the helper class defined here is assigned to the variable mentioned in step 1b, while that variable's previous value is saved in a static variable of the current class, i. e., the class containing the declaration of the subsequent branch.

By overriding the value of the variable mentioned in step 1b, this variable will always refer to an object whose instance method represents the *last* branch of the dynamic method. Thus, the static method described in step 1c constituting the client interface of the dynamic method will always invoke this last branch.

By storing the variable's previous value in another variable, the instance method corresponding to a particular branch is able to call the previous branch (pseudo-method `dynamic`) via the latter.

In addition to these basic transformations, the bodies of the instance methods mentioned above are modified as follows:

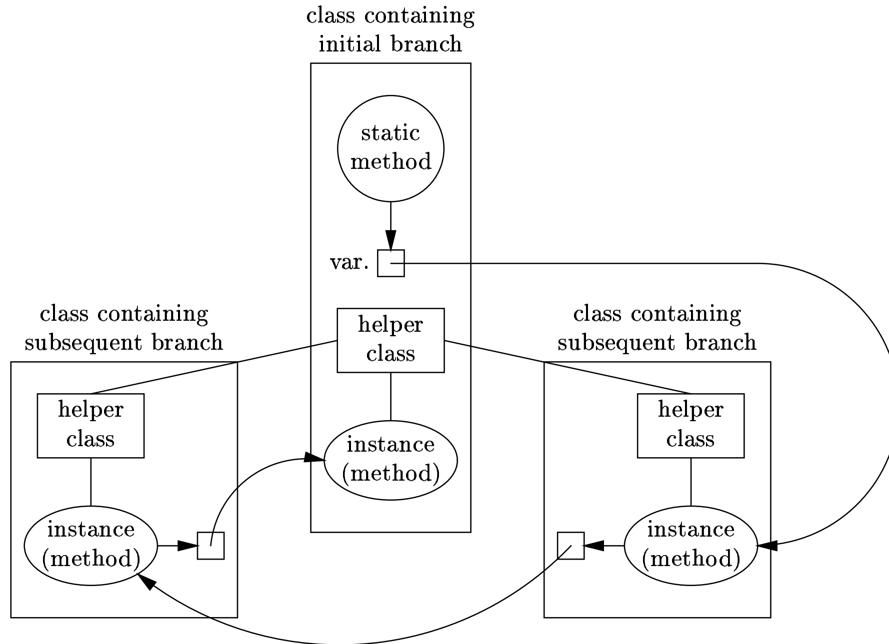

Fig. 9. Illustration of the transformation process

- Explicit and implicit *guards* are transformed to corresponding *conditional statements* whose then-part is the original method body and whose else-part calls the previous branch. (Since the initial branch does not possess a previous branch, it throws an `IncompleteDynamicClassMethodException` instead.)
- If necessary, backup copies of the method's formal parameters are created at the beginning in order to be able to pass the *original* parameter values to calls of the previous branch.
- Calls to the pseudo-method `dynamic` are replaced with normal method calls, as described above.

To illustrate the transformation process, figures 10 and 11 show the result of transforming the classes `Expr` and `Var` shown in Fig. 3. In addition to the basic principles described above, these figures show several technical details (1d, 1e, 2c, 2d) described in [8] which might be helpful for the interested reader, but whose precise description is beyond the scope of this paper.

7 Discussion

Dynamic methods are at the same time a powerful and a dangerous device. When used properly, they offer unique possibilities to extend and retroactively modify software systems, as has been illustrated in Sec. 4. On the other hand, when used

```
public abstract class Expr {
  private static class dynamic$1 {                        // 1a
    public int eval (Expr x) {
      // Abstract initial branch always throws this exception.
      throw new IncompleteDynamicClassMethodException(...);
    }
  }
  private static dynamic dynamic$1 = new dynamic();     // 1b
  public static int eval (Expr x) {                       // 1c
    return dynamic$1.eval(x);
  }
  public static dynamic eval (Expr x, dynamic $) {      // 1d
    try { return dynamic$1; }
    finally { dynamic$1 = $; }
  }

  private static class dynamic$2 extends dynamic$1 {    // 1a
    public void print (Expr x) {
      throw new IncompleteDynamicClassMethodException(...);
    }
  }
  ...... // Analogous to steps 1b, 1c, and 1d of 'eval'.

  public static class dynamic extends dynamic$2 {}      // 1e
}
```

Fig. 10. Transformation of class `Expr` (cf. Fig. 3)

inappropriately, they make it quite easy to cause havoc by overriding dynamic methods in a completely nonsensical way. To limit the chances of accidental or deliberate abuse of the concept, it might be an interesting task to integrate it with the Java Security Framework [6].

Ideas to support aims similar to those of dynamic methods can be found in many different areas. For instance, the concepts of open classes, multi-methods, before- and after-methods, and methods specialized to individual instances, found in different combinations, e. g., in MultiJava [1], CLOS [11], and Dylan [3], offer many of the possibilities of dynamic methods. The latter, however, provide additional flexibility by allowing dispatch strategies that are based on arbitrary properties of their arguments, not just their dynamic types (cf. Fig. 7). Furthermore, even properties of the "environment," such as values of static variables, user preferences read from an application's configuration file, etc., can be incorporated into the dispatch process if appropriate. Finally, complete redefinitions of, e. g., erroneous or incomplete library methods would be possible, if the concept were applied consistently, i. e., if methods were always defined `dynamic`.

At first glance, dynamic methods appear to be just a syntactic variation of methods with *predicate dispatching* [4], but when taking a closer look, several

```
public class Var extends Atom {
  ......

  private static class dynamic$1 extends Expr.dynamic { // 2a
    public int eval (Expr $1) {
      if ($1 instanceof Var) { Var x = (Var)$1;
        return x.val;
      } else return dynamic$1.eval($1);
    }
  }
  private static Expr.dynamic dynamic$1                // 2b
   = Expr.eval((Expr)null, new dynamic$1());
  private interface dynamic$1i {                       // 2c
    int eval (Expr x) throws Exception;
  }
  private static class dynamic$1c extends Expr.dynamic  // 2d
   implements dynamic$1i {}
  // Above class will be rejected by the Java compiler
  // if class Expr does not define a matching dynamic method.

  ...... // Analogous for 'print'.
}
```

Fig. 11. Transformation of class `Var` (cf. Fig. 3)

differences become obvious: First of all, while logical implications between the predicates of a method are used to define an overriding relationship in predicate dispatching, no attempt is made to determine such a relationship between the guards associated with the branches of a dynamic method. The main reason for this decision is the fact that arbitrary Boolean expressions of the host language, even though permitted in predicate dispatching, can in principle not be compared with respect to logical implication, leading to an actually incomplete algorithm in [4]. The second important reason for preferring a simple linear order of branches that is built up dynamically to a statically defined partial order of methods is the inability of the latter to support retroactive behavioural extensions and modifications, i. e., the third dimension of extensibility mentioned in Sec. 2. Welcome side effects of this decision are much simpler semantics and implementation of the concept. In particular, no separate notion of predicate expressions and predicate abstractions is needed since arbitrary Java expressions of type `boolean` can be used as guards, while normal (or even dynamic) methods can be used to encapsulate them.

In the terminology of aspect-oriented programming (AOP), in particular that of AspectJ [9], dynamic methods provide the same functionality as (before, after, and around) *advice* with *pointcut designators* of type `call` or `execution`. Thus, AspectJ is apparently more expressive as it offers additional pointcut designators as well as other features such as inter-type declarations. On the other hand,

restricting pointcuts to method calls harmonizes well with the principle of information hiding where only the signatures and (formal or verbal) specifications of methods are known outside a class, while employing other kinds of pointcuts usually requires detailed knowledge of method implementations.

While the extensions defined by aspects are woven into the source or byte code of all affected classes by the AspectJ compiler producing augmented class files [9], dynamic methods do not change at all the code of the system that shall be extended. By dynamically loading classes containing branches of dynamic methods, it is even possible to add new branches of dynamic methods at run time. Furthermore, in contrast to aspect-oriented languages, the concept requires only marginal language extensions; to the contrary, when dynamic methods are introduced into an object-oriented language, static and instance methods might be thrown out in principle, actually yielding a simpler language.

Acknowledgement. Many thanks are due to Wolfgang Doll for implementing the Java precompiler.

References

1. C. Clifton, G. T. Leavens, C. Chambers, T. Millstein: "MultiJava: Modular Open Classes and Symmetric Multiple Dispatch for Java." In: *Proc. 2000 ACM SIGPLAN Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '00)* (Minneapolis, MN, October 2000). *ACM SIGPLAN Notices* 35 (10) October 2000, 130–145.
2. J. W. Cooper: *Java Design Patterns: A Tutorial.* Addison-Wesley, Boston, 2000.
3. I. D. Craig: *Programming in Dylan.* Springer-Verlag, London, 1997.
4. M. Ernst, C. Kaplan, C. Chambers: "Predicate Dispatching: A Unified Theory of Dispatch." In: E. Jul (ed.): *ECOOP'98 – Object-Oriented Programming* (12th European Conference; Brussels, Belgium, July 1998; Proceedings). Lecture Notes in Computer Science 1445, Springer-Verlag, Berlin, 1998, 186–211.
5. E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, Reading, MA, 1995.
6. L. Gong: *Inside Java 2 Platform Security.* Addison-Wesley, Reading, MA, 1999.
7. J. Gosling, B. Joy, G. Steele: *The Java Language Specification.* Addison-Wesley, Reading, MA, 1996.
8. C. Heinlein: *Dynamic Class Methods in Java.* Nr. 2003-05, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, July 2003. http://www.informatik.uni-ulm.de/pw/berichte
9. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold: "An Overview of AspectJ." In: J. Lindskov Knudsen (ed.): *ECOOP 2001 – Object-Oriented Programming* (15th European Conference; Budapest, Hungary, June 2001; Proceedings). Lecture Notes in Computer Science 2072, Springer-Verlag, Berlin, 2001, 327–353.
10. O. Spinczyk, A. Gal, W. Schröder-Preikschat: "AspectC++: An Aspect-Oriented Extension to the C++ Programming Language." In: J. Noble, J. Potter (eds.): *Proc. 40th Int. Conf. on Technology of Object-Oriented Languages and Systems (TOOLS Pacific)* (Sydney, Australia, February 2002), 53–60.
11. P. H. Winston, B. K. P. Horn: *LISP* (Third Edition). Addison-Wesley, Reading, MA, 1989.