Dynamic Class Methods in Java[1]

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 [11, 13], *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 [8]. *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] (Sec. 3) is contrasted with a functionally equivalent, but significantly shorter and less complex implementation based on dynamic class methods (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 sections 6 and 7 explain the basic idea and the accompanying details, respectively, of transforming classes containing dynamic class methods to pure Java code by means of a simple precompiler. Finally, Sec. 8 concludes the paper with a discussion of the concept itself and of related work.

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, either because it might be simply not available or to strictly apply the principle of *modular extensibility*. 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.

[1] This is an extended version of [9].

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*.

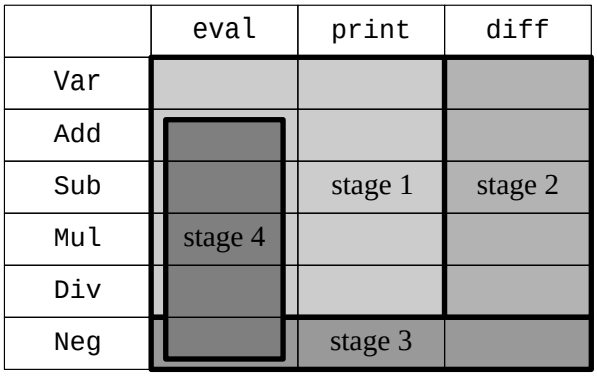

Figure 1: Evolution of the system

3. Achieving Extensibility with Design Patterns

In the following, design patterns [5, 2] will be employed to achieve the aims described in the previous section. Afterwards (Sec. 4), a new language concept called *dynamic class methods* will be employed to achieve the same aims much more easily and directly.

Stage 1: Basic Class Hierarchy and Methods

Figures 2 and 3 show the code written in Stage 1 to support evaluation and printing of variables and the four basic arithmetic operations.

To prepare the system for later horizontal extensions, the *Visitor pattern* is employed from the beginning by defining a `Visitor` interface declaring `visit` methods for all concrete subclasses of the abstract class `Expr` constituting the root of the expression class hierarchy (cf. Fig. 2). Furthermore, a simple kind of *Factory pattern* is employed by defining a `Factory` class providing factory methods for all concrete subclasses of `Expr` as well as a singleton instance of itself. By replacing this factory object with an instance of a factory subclass whose methods create instances of subclasses of the original classes, later behavioural modifications of expressions become possible.

The classes shown in Fig. 3 are abstract and concrete subclasses of `Expr` representing different categories of expressions such as variable expressions (`Var`) or additions (`Add`). Classes which possess private data fields (e. g., `Var`) provide a protected constructor to initialize them and public methods for

```
public abstract class Expr {
  public abstract int eval ();          // Evaluate.
  public abstract void print ();           // Print.
  public abstract void accept (Visitor v); // Visit.
}

public interface Visitor {
  // Visit methods for all concrete subclasses of Expr.
  void visit (Var x);
  void visit (Add x);
  ......
}

public class Factory {
  // Factory methods for all concrete subclasses of Expr.
  public Var createVar (String n, int v) { return new Var(n, v); }
  public Add createAdd (Expr l, Expr r) { return new Add(l, r); }
  ......

  // Singleton factory instance.
  public static Factory f = new Factory();
  protected Factory () {}
}
```

Figure 2: Root class `Expr` and pattern types `Visitor` and `Factory`

reading them (e. g., `name` and `val`). If the `Factory` class belongs to the same package as these classes, its methods are allowed to call the protected constructors, while client code residing in different packages is forced to use the factory methods to create instances of expressions, e. g., `Factory.f.createVar("x", 5)` to create a new variable expression.

To simplify the overall view, Fig. 4 depicts all classes and interfaces developed in this and subsequent stages (using the same grayscales as in Fig. 1) and their basic relationships to each other.

Stage 2: Horizontal Extension

Since the classes developed in Stage 1 shall not be modified, the new methods for symbolic differentiation of expressions cannot be added as instance methods like `eval` and `print` to `Expr` and its subclasses.

For a rather straightforward solution, `diff` could be implemented as a single static method of a new class `Diff`, as shown in Fig. 5. The obvious problem with this solution is, however, that it is not vertically extensible, i. e., if new subclasses of `Expr` are added later, this method must be extended and recompiled.

To gain more flexibility, the Visitor pattern can be exploited to implement `diff`, as shown in Fig. 6. Here, `diff` creates an instance `v` of the `DiffVisitor` class, which implements the `Visitor` interface defined in Fig. 2, and passes it to the `accept` method of the expression `x`. This method in turn calls the appropriate `visit` method of `v` which performs the actual differentiation and stores the result in an internal variable of `v`.

To remain open for later vertical extensions, the original Visitor pattern as described in [5, 2] is combined with the *Prototype pattern* which allows later replacements of the prototypical visitor instance `proto` with an instance of a subclass (cf. Stage 3).

```
// 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; }

  public int eval () { return val; }
  public void print () { ..... }
  public void accept (Visitor v) { v.visit(this); }
}

// 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); }

  public int eval () { return left().eval() + right().eval(); }
  public void print () { ..... }
  public void accept (Visitor v) { v.visit(this); }
}

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

Figure 3: Abstract and concrete subclasses of `Expr`

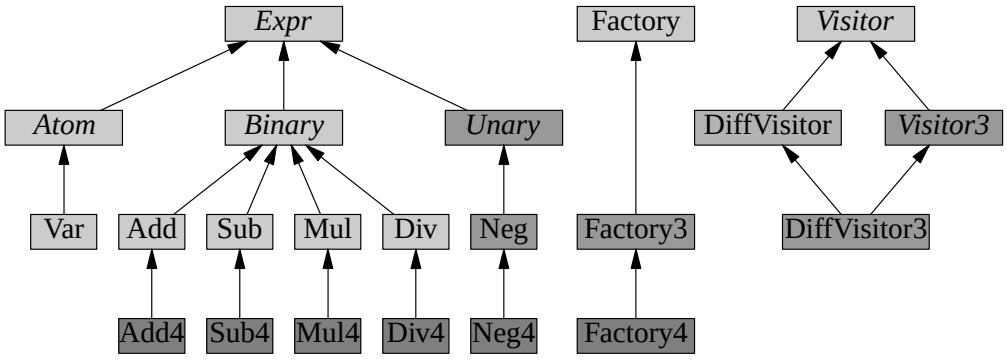

Figure 4: Overall view of the system

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

Figure 5: Direct implementation of `diff`

```
public class DiffVisitor implements Visitor, Cloneable {
  // Differentiate expression x along variable n.
  public static Expr diff (Expr x, String n) {
    DiffVisitor v = null;
    try { v = (DiffVisitor)proto.clone(); }
    catch (CloneNotSupportedException e) {}
    v.name = n;
    x.accept(v);
    return v.result;
  }

  // Prototypical visitor instance.
  protected static DiffVisitor proto = new DiffVisitor();

  // Internal variables.
  protected String name;
  protected Expr result;

  // Visit methods for all concrete subclasses of Expr.
  public void visit (Var x) {
    int c = x.name().equals(name) ? 1 : 0;
    result = Factory.f.createVar("", c);
  }
  public void visit (Add x) {
    result = Factory.f.createAdd(diff(x.left(), name),
     diff(x.right(), name));
  }
  ......
}
```

Figure 6: Visitor implementation of `diff`

Stage 3: Vertical Extension

Adding a new subclass `Neg` (plus an intermediate abstract class `Unary`) to the existing class hierarchy is trivial (cf. Fig. 7), except for implementing its `accept` method. The stereotyped implementation

```
public void accept (Visitor v) { v.visit(this); }
```

used in all other concrete subclasses of `Expr` would be erroneous, since the `Visitor` interface − which has been defined before the introduction of `Neg` − does not contain a matching `visit` method with a parameter of type `Neg`.

```
// Extended visitor interface.
public interface Visitor3 extends Visitor {
  void visit (Neg x);
}

// Extended visitor class.
public class DiffVisitor3 extends DiffVisitor implements Visitor3 {
  static { DiffVisitor.proto = new DiffVisitor3(); }
  public void visit(Neg x) {
    result = Factory3.f.createNeg(diff(x.body(), name));
  }
}

// Extended factory class.
public class Factory3 extends Factory {
  public Neg createNeg (Expr b) { return new Neg(b); }
  public static Factory3 f = new Factory3();
  protected Factory3 () {}
}

// 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 int eval () { return -body().eval(); }
  public void print () { ..... }
  public void accept (Visitor v) { ((Visitor3)v).visit(this); }
}
```

Figure 7: Vertical extension

Thus, it is necessary first to extend the original `Visitor` interface with a new `Visitor3`[2] interface containing an appropriate method. (Since existing source code shall not be modified, the `Visitor` interface cannot be extended in place.) Likewise, the `DiffVisitor` class is extended with a new `DiffVisitor3` class implementing that method for symbolic differentiation of `Neg` expressions. In order to achieve that this subclass is actually used by the `diff` method of `DiffVisitor`, its prototypical instance `proto` is set to an instance of `DiffVisitor3` during initialization of this class. (If there were other visitor classes, they could be extended in the same way.)

Similarly, the `Factory` class defined in Fig. 2 has to be extended with a new `Factory3` class providing an additional `createNeg` method to create `Neg` expressions. (Other categories of expressions might be created either with the old or the new factory, so client code using the old one need not be changed.)

After these preparations, it is possible to implement the `accept` method of class `Neg` as shown in Fig. 7, because all visitors will actually be instances of `Visitor3`.

Stage 4: Behavioural Extension

To achieve the behavioural modifications required in Stage 4, the implementations of `eval` for all unary and binary operators must be replaced by new implementations. This can be achieved by introducing subclasses of all affected classes with appropriately overridden `eval` methods (cf. Fig. 8).

To make these extensions transparent to clients, another new factory class, `Factory4`, is required which overrides all factory methods corresponding to these classes. Furthermore, it is necessary in that case to "close down" all old factories and redirect their clients to the new one to make sure that only instances of the new classes will be created.

Summary

The final system (cf. Fig. 4) 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.

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.[3]

Stage 1: Basic Class Hierarchy and Methods

The code of figures 9 and 10 implements the same functionality as the one shown in figures 2 and 3.

The instance methods `eval` and `print` of the root class `Expr`, which possess an implicit parameter `this` of type `Expr`, have been replaced by equally named dynamic methods receiving an explicit parameter `x` of type `Expr` instead, 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.

[2] 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.

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

```
// Modified division.
class Div4 extends Div {
  protected Div4 (Expr l, Expr r) { super(l, r); }

  public static final int NULL = Integer.MIN_VALUE;

  public int eval () {
    if (left().eval() == NULL || right().eval() == NULL) return NULL;
    try { return super.eval(); }
    catch (ArithmeticException e) { return NULL; }
  }
}

// Modified addition, subtraction, multiplication, and negation.
......

// New factory class.
class Factory4 extends Factory3 {
  // New factory methods for all unary and binary ops.
  public Div createDiv (Expr l, Expr r) { return new Div4(l, r); }
  ......

  // Adjust old factories which might be used by old client code.
  public static Factory4 f = new Factory4();
  protected Factory4 () {}
  static { Factory.f = Factory3.f = f; }
}
```

Figure 8: Behavioural extension

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`[4], similar to the way `super` can be used 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;
  else return dynamic(); // Call previous branch.
}
```

To simplify this frequently occurring pattern, it is possible to move the conditional statement acting as

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

```
// 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) {
    ......
  }
}
```

Figure 9: Basic class hierarchy and methods (part 1)

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.

```
// 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.
......
```

Figure 10: Basic class hierarchy and methods (part 2)

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.

Thus, even though the implementation of symbolic differentiation as a dynamic method `diff` shown in Fig. 11 contains basically the same code as the equally named static method of Fig. 5, it can be easily extended to new categories of expressions without being modified (cf. Stage 3).

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

Figure 11: Horizontal extension

Alternatively, `diff` might be implemented by separate branches of a dynamic method for all concrete subclasses of class `Expr` (cf. Fig. 12). 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.

```
public class Diff {
  // Differentiate expression x along variable n.
  public dynamic Expr diff (Expr x instanceof Var, String n) {
    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));
  }
  ......
}
```

Figure 12: Alternative implementation of `diff`

Stage 3: Vertical Extension

The code of Fig. 13, which implements exactly the same functionality as that of Fig. 7, is another impressive demonstration of the flexibility provided by dynamic methods. Instead of defining an extended `Visitor3` interface and an extended `DiffVisitor3` class 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

```
// 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));
  }
}
```

Figure 13: Vertical extension

method might be redefined in *any* class where it is accessible, not just in subclasses of the class containing the original definition.

However, to actually *enable* such an additional branch of a dynamic method, the class containing its definition must be loaded and *initialized* 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. 14. (Again, it is necessary that the class containing additional branches of dynamic methods gets initialized; 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 [14, 3, 1].

Furthermore, in contrast to Fig. 8, 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.

```
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; }
  }
}
```

Figure 14: Behavioural extension

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 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. 15 shows in boldface 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 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

```
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
```

Figure 15: Extensions to the Java grammar

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. 16 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).
   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.

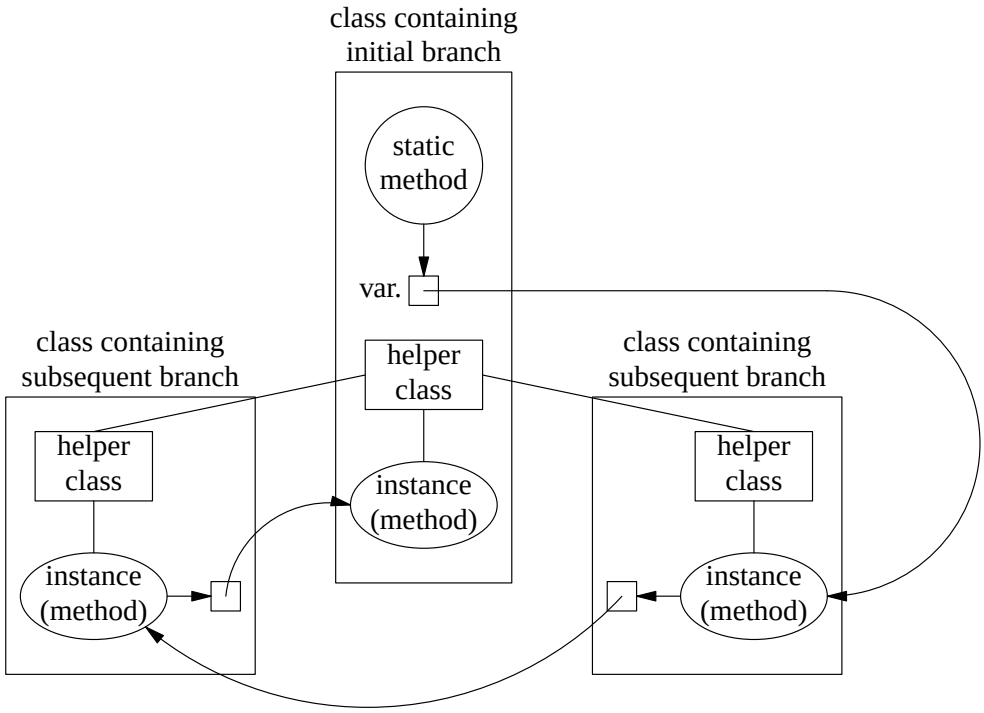

Figure 16: Illustration of the transformation process

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

- 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.

7. Transformation Details

Having explained the basic idea of the precompiler's work in the previous section, the transformation process is described in more detail in the present section.

7.1 Initial Branch of a Dynamic Method

The initial branch of a dynamic method is distinguished from subsequent branches by the fact that its method name is not qualified by a class name. Such a method definition is transformed as follows (cf. Fig. 17 showing the beautified result of transforming class `Expr` shown in Fig. 9):

1. A private static helper class is generated which contains a public instance method possessing exactly the same signature (i. e., name, parameters, return type, and exceptions) and body as the dynamic method. If the dynamic method is declared `strictfp`, this method will be too.
   Since a class might contain several dynamic methods, unique names such as `dynamic$1`, `dynamic$2`, etc. are chosen for these helper classes. Furthermore, each such helper class except the first

```
public abstract class Expr {
  // Transformation of:
  // public dynamic abstract int eval (Expr x);
  private static class dynamic$1 {                                  // 1
    public int eval (Expr x) {
      // Abstract initial branch always throws this exception.
      throw new IncompleteDynamicClassMethodException(
       Expr.class, "eval", new Class [] { Expr.class }
      );
    }
  }
  private static dynamic dynamic$1 = new dynamic();                 // 2
  public static int eval (Expr x) {                                 // 3
    return dynamic$1.eval(x);
  }
  public static dynamic eval (Expr x, dynamic $) {                  // 4
    try { return dynamic$1; }
    finally { dynamic$1 = $; }
  }

  // Transformation of:
  // public dynamic abstract void print (Expr x);
  private static class dynamic$2 extends dynamic$1 {                // 1
    public void print (Expr x) {
      throw new IncompleteDynamicClassMethodException(
       Expr.class, "print", new Class [] { Expr.class }
      );
    }
  }
  ...... // Analogous to steps 2, 3, and 4 of `eval'.

  // At the end of the enclosing class.
  public static class dynamic extends dynamic$2 {}
}
```

Figure 17: Transformation of class `Expr` (cf. Fig. 9)

is defined as a subclass of the previous one and, at the end of the enclosing class definition, a public static class called `dynamic`[5] is defined as a subclass of the last helper class. The net effect of these definitions is that the nested class `dynamic` contains corresponding instance method definitions for all initial branches of dynamic methods defined in the enclosing class.
This incremental construction of the class `dynamic` has the advantage that the precompiler has to perform local source code transformations only and need not rearrange source code, which on the one hand simplifies its implementation and on the other hand ensures that line numbers of its output match those of its input (if the output is unbeautified, i. e., does not contain additional line breaks). The latter property is important to localize errors reported by the posterior Java compiler.

2. A private static variable of type `dynamic` is defined which is initialized with a new instance of that type. These variables possess the same unique names as the helper classes mentioned above.

[5] Note that `dynamic` is a keyword in the extended language "Java plus dynamic class methods," but not in Java itself.

3. A static method possessing the same signature and access modifier as the dynamic method is defined which calls the corresponding instance method via the just mentioned variable and returns its result if appropriate. This method constitutes the client interface to the dynamic method.
   As long as no other branches of the dynamic method are defined, calling this static method is equivalent to executing the body of the dynamic method definition.
   If the dynamic method is declared `synchronized`, this method will be too.
4. Another static method possessing the same name, access modifier, and parameters as the the dynamic method plus an additional parameter and a result of type `dynamic` is defined. When called, this method returns the current value of the variable mentioned in Step 2 and replaces it with the value of its last parameter. The other parameter values are ignored and are necessary only to distinguish possibly overloaded variants of this method resulting from correspondingly overloaded variants of dynamic methods.
   This method is called when the dynamic method is overridden in another (or the same) class (cf. Sec. 7.2, Step 2).

It should be noted that it is not possible to encode the information provided by the types of the ignored parameters into the name of the method defined in Step 4, e. g., `eval$Expr` with a single parameter of type `dynamic` instead of `eval` with an additional dummy parameter of type `Expr`, since the same type might be used with different names, e. g., `Expr` and `expr.Expr` if `Expr` is defined in a package named `expr`. Without consulting other source or class files, the precompiler is generally not able to detect such synonyms.

For the same reason, it is not possible in Step 1 to omit the nested class `dynamic` and instead to define a separate public helper class for every dynamic method whose name is derived from the method's signature, even though this would be simpler and more natural at first sight.

The body of the method described in Step 1 above is modified as follows:

- If the dynamic method contains one or more explicit guards, these are moved into the method's body as (possibly nested) conditional statements whose then-part is the original body and whose else-part throws an `IncompleteDynamicClassMethodException` (which is defined as an unchecked exception) receiving the enclosing class, the method's name, and its parameter types as constructor arguments.
  Similarly, calls to the pseudo-method `dynamic()` are replaced by corresponding `throw` expressions, because there is no previous branch for the initial branch of a dynamic method.
  Finally, if the dynamic method is declared `abstract` (and thus does not possess a body), a body throwing the same exception is added.
- If the dynamic method contains guarded parameter declarations, these are transformed to equivalent explicit guards which are in turn transformed to conditional statements as described above.
  As a special case, a guarded parameter declaration with a dynamic type test, such as `Expr x instanceof Var`, is transformed to a conditional statement plus a redeclaration of the parameter `x` with its dynamic type (cf. Sec. 7.2 for an example).

7.2 Subsequent Branches of a Dynamic Method

Subsequent branches of a dynamic method are distinguished from the initial branch by the fact that their method name is qualified by the name of the class `C` containing the initial branch. Such a method definition is transformed as follows (cf. Fig. 18 showing the result of transforming the class `Var` shown in Fig. 9):

1. Similar to Sec. 7.1, a private static helper class is generated which contains a public instance method possessing the same signature and body as the dynamic method, except for the method name which becomes unqualified. Again, if the dynamic method is declared `strictfp`, this method will be too.

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

  // Transformation of:
  // dynamic int Expr.eval (Expr x instanceof Var) {
  //   return x.val;
  // }
  private static class dynamic$1 extends Expr.dynamic {          // 1
    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                          // 2
   = Expr.eval((Expr)null, new dynamic$1());
  private interface dynamic$1i {                                 // 3
    int eval (Expr x) throws Exception;
  }
  private static class dynamic$1c extends Expr.dynamic           // 4
   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'.
}
```

Figure 18: Transformation of class `Var` (cf. Fig. 9)

In contrast to Sec. 7.1, the helper classes generated that way are independent of each other and independent of the helper classes generated for initial branches of dynamic methods of the surrounding class.
Instead, each such helper class is defined as a subclass of the nested class `C.dynamic`, where `C` is the class containing the initial branch of the dynamic method, as described above. By that means, the instance method mentioned above overrides the corresponding method of the class `C.dynamic`.

2. A private static variable of type `C.dynamic` is defined which is initialized by a call to the method described in Step 4 of Sec. 7.1.
   The (qualified) name and the parameter types of this method are exactly those of the dynamic method plus an additional parameter of type `C.dynamic`. Since all parameter values except the last are ignored by that method, arbitrary dummy values might be passed for them. This can be achieved by passing, e. g., `false` for `boolean` parameters, `0` for all numeric types including `char`, and `null` for all other types. To avoid potential ambiguities in the presence of overloaded methods, however, these values are cast to the exact parameter types given in the dynamic method declaration.
   For the last parameter, an instance of the helper class mentioned in Step 1 is passed. By that means, the variable described in Step 2 of Sec. 7.1 is set to this instance, while its previous value is assigned to the variable described above. Consequently, calling the dynamic method (i. e., the static method of class `C` described in Step 3 of Sec. 7.1) is now equivalent to executing the body of the branch defined here.

The body of the instance method described in Step 1 above is modified as follows:

- If the dynamic method contains one or more explicit guards, these are again moved into the method's body as conditional statements whose then-part is the original body and whose else-part invokes the previous branch of the dynamic method, which is accessible via the variable mentioned in Step 2 above.
  In the same way, calls to the pseudo-method `dynamic()` are replaced by calls to the previous branch.
- Since the previous branch shall be called with the original parameter values, even if the formal parameters have been modified, backup copies of all non-`final` parameters are declared and initialized at the very beginning of the method body. Since parameter values might even be changed by the evaluation of a guard, the creation of the backup copies cannot be moved inside the then-part of the conditional statements mentioned before.
- Guarded parameter declarations, in particular those containing dynamic type tests, are transformed in the same way as for initial branches (cf. Sec. 7.1).

The transformation rules described so far are sufficient to transform syntactically and semantically correct dynamic method declarations to legal Java code. Unfortunately, however, they also transform some semantically erroneous declarations to legal Java code, as the following example shows. If the dynamic method `Expr.eval` in class `Var` would be defined as follows:

```
// Error:
// Class `Expr' does not define a dynamic method `eval(Var)'.
// `(Var x)' should be `(Expr x instanceof Var)'.
dynamic int Expr.eval (Var x) { ..... }
```

it would be transformed to the following legal Java code:

```
private static class dynamic$1 extends Expr.dynamic {          // 1
  // This method does not override `Expr.dynamic.eval(Expr)',
  // but defines an overloaded variant of `eval'.
  public int eval (Var x) { ..... }
}

private static Expr.dynamic dynamic$1                          // 2
 = Expr.eval((Var)null, new dynamic$1());
 // This actually calls `Expr.eval(Expr, Expr.dynamic)'.
```

Since class `Var` is a subclass of `Expr`, the call to `Expr.eval` with a first argument of type `Var` instead of `Expr` is semantically correct. But since `dynamic$1.eval(Var)` does not override `Expr.dynamic.eval(Expr)`, but rather defines an overloaded variant of `eval`, the former will be actually never called.

In order to keep the precompiler as simple as possible, and in particular to allow it to do its source code transformation without consulting any other source or class file, it is impossible to catch such errors by the precompiler. Instead, it must generate some additional Java code that will cause the posterior Java compiler to report an error in that case. Therefore, the numbered list above is continued as follows (cf. Fig. 18):

3. A private nested interface declaring the same method as the helper class mentioned in Step 1 is generated.
   The only difference is the fact, that the method declared here might throw any exception.
4. A second private static helper class is defined which extends the class `C.dynamic`, implements the interface just mentioned, and has an empty body.

Since the body of this class is empty, the method declared in the interface must match one of the methods defined in the class `C.dynamic`, i. e., one of the dynamic methods initially defined in class `C`. Otherwise, the Java compiler will reject the second helper class as not fulfilling its claim to implement the interface mentioned in Step 3.

Since the method in class `C.dynamic` (i. e., the initial branch of the dynamic method) is allowed to throw more exceptions than the branch defined here, and the precompiler cannot know its exact `throw` clause, the method declared in the interface is allowed to throw any exception. Otherwise, if the throw clause of the branch's definition would be used, the Java compiler would report an undesired error if the branch defined here throws fewer exceptions than the initial branch.

The rule that it must not throw more exceptions than the initial branch is nevertheless enforced by the Java compiler because the method described in Step 1 above overrides the corresponding method of class `C.dynamic` and thus must obey this rule.

8. 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 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].

At first glance, a potential weakness of the concept is the fact that the order of a dynamic method's branches is determined by the order in which the classes containing them get initialized at run time. For many practical applications, however, especially when dynamic methods are just used like instance methods which are overridden in subclasses, the natural "superclass before subclasses" initialization order prescribed by the Java Language Specification [7] is exactly what an application needs. Furthermore, if different additional branches are used to add *orthogonal extensions* to a dynamic method, their precise order is usually irrelevant. So the only remaining critical scenario is a combination of different behavioural *modifications* of a method which influence each other. In such a case, the programmer composing these modifications into a single application should explicitly enforce the semantically correct initialization order, e. g., by creating dummy instances of the classes or by calling `Class.forName` in the desired order.

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 [14], 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. 14). 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 methods, e. g., erroneous or incomplete library methods, are possible, if the concept is applied consistently, i. e., if methods are always defined `dynamic` (cf. Fig. 14).

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 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 [11], dynamic methods provide the same functionality as *inter-type method declarations* (to perform horizontal extensions of classes) and (before, after, and around) *advice* with *pointcut designators* of type `call` or `execution` (to perform behavioural extensions of methods). Thus, AspectJ is obviously more expressive as it offers additional pointcut designators as well as other kinds of inter-type declarations. On the other hand, restricting pointcuts to method calls harmonizes well with the principle of information hiding [12] 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 [11], dynamic methods do not change at all the code of the system that shall be extended. By dynamically loading (and initializing) 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.

The new programming language Timor, in whose development the author is involved, provides concepts called *qualifying types* and *bracket routines* [10] which are quite similar to dynamic methods. An essential difference, however, is the fact that the extensions or modifications implemented by bracket routines are applied only if an object is explicitly associated with an instance of a qualifying type, while the extensions or modifications implemented by additional branches of a dynamic method are applied automatically as soon as the class containing these branches gets initialized. The latter is especially helpful to cope with unexpected behavioural extensions or modifications.

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: *Vertical, Horizontal, and Behavioural Extensibility of Software Systems*. Nr. 2003-06, Ulmer Informatik-Berichte, Fakultät für Informatik, Universität Ulm, July 2003.
http://www.informatik.uni-ulm.de/pw/berichte

[9] C. Heinlein: "Dynamic Class Methods in Java." In: D. Rombach (ed.): *Net.ObjectDays 2003. Tagungsband* (Erfurt, Germany, September 2003). tranSIT GmbH, Ilmenau, 2003, ISBN 3-9808628-2-8.

[10] J. L. Keedy, G. Menger, C. Heinlein, F. Henskens: "Qualifying Types Illustrated by Synchronization Examples." In: M. Aksit, M. Mezini, R. Unland (eds.): *Objects, Components, Architectures, Services, and Applications for a Networked World* (Int. Conf. NetObjectDays, NODe 2002; Erfurt, Germany, October 2002; Revised Papers). Lecture Notes in Computer Science 2591, Springer-Verlag, Berlin, 2003, 330−344.

[11] 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.

[12] D. L. Parnas: "On the Criteria to Be Used in Decomposing Systems into Modules." *Communications of the ACM* 15 (12) December 1972, 1053−1058.

[13] 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.

[14] P. H. Winston, B. K. P. Horn: *LISP* (Third Edition). Addison-Wesley, Reading, MA, 1989.