Chapter 1

REDUCING CODE REPLICATION IN DELEGATION-BASED JAVA PROGRAMS

Martin Kuhlemann*
School of Computer Science, University of Magdeburg,Germany
Christian Kästner†
School of Computer Science, University of Magdeburg,Germany
Sven Apel‡
Department of Informatics and Mathematics, University of Passau,Germany

Keywords: Java language, generative programming

Abstract

Interfaces and delegation are fundamental concepts in OO languages. Although both concepts have been shown to be beneficial in software development, sometimes their implementation is cumbersome. Both result in numbers of forwarding methods or numbers of empty methods for respective classes. These trivial methods distract the user from non-trivial methods the class comprises. This increases complexity and decreases maintainability. In its current form, Java does not provide sufficient mechanisms to avoid this boilerplate code. Instead, all the methods that are empty or only

*E-mail address: mkuhlema@ovgu.de
†E-mail address: ckaestne@ovgu.de
‡E-mail address: apel@uni-passau.de

forward calls have to be coded manually. In this paper, we introduce a new lightweight mechanism, that improves the implementation of interface-based and delegation-based programs. We show, though this mechanism is very simple, it solves these problems that are well-known in object-oriented software development. In three open-source Java programs of up to over 25.000 lines of source code, we show how we use this mechanism to generate up to 5.7 % of all methods per case study that were empty or only forwarding calls.

1. Introduction

Interfaces and delegation are fundamental concepts of *object-oriented programming (OOP)* and Java [12, 11, 15]. An interface defines a protocol how to access a class and hides the class' implementation [6, 2]. Delegation is a mechanism that forwards message calls from a *delegating class* to an associated *delegatee class* for the purpose of reuse [20]. Delegation and interfaces are used frequently in design patterns [11] and black-box frameworks [15] to improve flexibility and extensibility. However, there are several known problems related to the implementation of methods that forward calls to a delegatee class and that implement abstract methods of implemented interfaces – we name both types of methods *default methods*. These methods are trivial and repetitious but tedious to write. These default methods also distract from the non-trivial methods of a class that we call *core methods*. Default methods are replicated throughout the code [11, 15] and the resulting code is criticized as bad design (a.k.a. 'code smell') [10]. They may reduce software maintainability and understandability [14] which is in contrast to the promised benefits of reuse through interfaces and delegation.

We show that default methods constitute 2.2 % to 5.7 % of all methods in three analyzed medium-sized Java programs. In order to avoid the implementation of these methods, we propose a lightweight language mechanism for Java, called *Implicit Defined Methods (IDM)*, that takes care of generating default methods when required. Our aim is to improve code quality and to take the burden from the developer who otherwise has to implement and maintain a high number of default methods (e.g., up to 107 in *JHotDraw*). We illustrate IDM using the delegation-based design patterns Decorator and Visitor, and evaluate it using three open-source case studies of different size and purpose, *JHotDraw*[1] [8, 26], *Java Class File Editor (Jcfe)*[2], and the Java package *java.io* [24].

Note, with only 2.2 % to 5.7 % of all methods in our analyzed programs, default methods occur rather infrequently. In many programs that do not use delegation it may not occur at all. Nevertheless, it is a problem that – when it occurs – inflicts code quality and is tedious for the developers. Our studies show how, even though default methods do not occur frequently, we gain considerable benefit from IDM.

2. Problem Statement

Delegation is the process of forwarding a request to a delegatee class that deals with the request instead of the delegating class that received the request originally. Delegation is

[1] http://sourceforge.net/projects/jhotdraw/

[2] http://sourceforge.net/projects/classeditor/

```
1  interface Command{
2   void execute();
3   boolean isExecutable();
4   DrawingEditor getDrawingEditor();
5   String name();
6  }
7
8  class UndoableCommand implements Command{
9   Command myWrappedCommand;
10  void execute() {/* ... */}
11  boolean isExecutable() {
12   return myWrappedCommand.isExecutable();
13  }
14  DrawingEditor getDrawingEditor() {
15   return myWrappedCommand.getDrawingEditor();
16  }
17  String name() {
18   return myWrappedCommand.name();
19  }
20 }
21
22 class ZoomCommand implements Command{
23  void execute() {/* ... */}
24  boolean isExecutable() {/* ... */}
25  DrawingEditor getDrawingEditor() {/* ... */}
26  String name() {/* ... */}
27 }
```

(a)

```
1  interface ClassVisitor{
2   void visitClass(ClassFile classFile);
3   void visitInstructions(Instructions instrs);
4   void visitInstruction(Instruction instr);
5   void visitVersion(Version ver);
6  }
7
8  abstract class NavigatingClassVisitor implements ClassVisitor {
9   void visitClass(ClassFile classFile) {/* ... */}
10  void visitInstructions(Instructions instrs) {}
11  void visitInstruction(Instruction instr) {}
12  void visitVersion(Version ver) {}
13 }
```

(b)

Figure 1. Code repitition in delegating and delegatee classes (ecxerpts of case study's classes).

a reuse mechanism beside inheritance in Java and other OOP languages [35], i.e., different delegating classes may reuse the implementation of one delegatee class by forwarding requests to it. Both, delegating and delegatee classes, may include repetitive method implementations.

In delegating classes, a high number of forwarding methods makes their implementation tedious. This may hamper reasoning about that class because the core methods, i.e., the reason for implementing it, may be hard to identify among all forwarding methods. Consequently, the class design may be considered problematic because it mainly forwards messages (known as Middle Man code smell) [10] and contains numerous methods that do not communicate with each other [15].

In Figure 1a, the delegating class *UndoableCommand* (Lines 8-20, taken from *JHotDraw*) includes numerous forwarding methods (we underlined default code) which may hamper finding the core method *execute* and reasoning about it. In this class *UndoableCommand* six of nineteen methods simply forward calls.

In delegatee classes, a high number of empty methods has to be implemented; the empty methods distract from the core methods of this class. Oftentimes the delegatee classes disable methods of their super-type with empty methods [14, 27, 9] because these methods do not relate to the delegatee's concern (A typical solution is to introduce an abstract class, but as discussed below this cannot be used generally.). Consequently, also the disabling methods do not contribute to the delegatee's concern. Classes that include many empty methods are considered as bad code design (code smell Lazy Class) [10].

In Figure 1b, the delegatee *NavigatingClassVisitor* (Lines 8-13, taken from *Jcfe*) includes mostly empty methods because it needs to define all methods of its interface *ClassVisitor* – we again underlined default code. (Note, using an abstract class is problematic as outlined later in this section but avoids harder problems of empty methods for its subclasses.)

Relevance of the Problem

Design patterns define solutions for recurring design problems in OOP with the goal of an improved reusability and variability of code [11]. Numerous design patterns rely on delegation (in particular, Adapter, Bridge, Decorator, Chain Of Responsibility, Proxy, Strategy, Visitor, or Iterator) and these patterns are known to occur frequently compared to others (McNatt et al. reported that 30 of 99 Gang-of-Four-pattern occurrences in their studies are delegation-based [22].). Moreover, pattern code has to be repeated every time a pattern is used [23, 3, 11, 5], thus, worsening problems of repetitive delegation methods. The excerpts of Figure 1, taken from *JHotDraw* and *Jcfe*, show implementations of the design patterns Decorator and Visitor.

Black-box frameworks use interfaces to provide a common architecture for a family of related applications [15, 19]. A framework pinpoints delegatee interfaces (so-called *hot spots* [15]) that user classes have to implement to configure and obtain a complete application; thereby, every user class has to implement also the methods of its hot spot interface that do not contribute to the class' concern – respective methods often remain empty. While larger hot-spot interfaces may provide more variability for framework applications than small hot spot interfaces, they also may include more methods to learn by the developer

but not needed for single applications [7]. Thus, black-box frameworks may force a user to implement numerous empty default methods.

Abstract Classes – Not Always a Solution

Abstract classes [15] are often used to cope with default methods. An abstract class provides a protocol for subclasses and may implement that protocol in parts. This way, abstract classes localize a method that had been repeated across multiple classes before. However, abstract classes cannot be used as a general solution for reducing default methods, because of several problems and limitations.

First, in languages like Java, inheritance from different superclasses is not possible. Thus, a class that inherits from an abstract class to work around the default method problem, can no longer inherit from any other class. For instance, we observed the problem in several programs that use different abstract classes of Java's AWT framework, abstract AWT classes that define default implementations for AWT interfaces. While it is possible to write abstract classes that implement combinations of interfaces, as done in AWT for *WindowAdapter*, there is an exponential number of possible combinations.

Second, abstract classes may face problems of *putting methods to high in an inheritance hierarchy* [14, 27, 9]. That is, for the purpose of reuse, a method may be located so high in the inheritance hierarchy that some subclasses that inherit the method must disable the method by overriding. Therefore, the abstract class should cover exactly the methods that are repeated in every subclass, e.g., forwarding or empty methods. If an abstract class does not provide all forwarding and empty methods for a set of subclass, these certain subclasses still have to define those methods which results in code replication. If an abstract class provides too many methods, some subclasses have to disable inherited methods by overriding them such that they have no effect [27]. A system of fine-grained abstract classes may help for that limitation but complicates the inheritance structure [14].

Summary. Delegation is frequently used in OOP, especially, when the software is considered well-designed, i.e., using design patterns or frameworks. To implement this well-designed software, the developer is forced to implement numbers of default methods. Contemporary techniques to work around this problem, like abstract classes, are limited.

3. Implicit Defined Methods

We propose a new language mechanism for Java that instructs the compiler to generate default methods implicitly to either forward their arguments or do nothing. Of course, the mechanism does not generate core methods which have to be implemented by the developer.

Delegating as well as delegatee objects follow one common interface which provides the signatures for the methods to implement. We augment the *implements* declaration of Java classes regarding an interface with an argument list. This argument list signalizes the body of default methods (delegating or delegatee methods) that are generated in the class for the given interface. In Figure 2, we show a refactored implementation of the *UndoableCommand* example using IDM that corresponds to Figure 1a. In Line 1, we instruct the IDM

```
1 class UndoableCommand force implements Command(myWrappedCommand){
2  Command myWrappedCommand;
3  void execute(){/* ... */}
4 }
```

Figure 2. Declaration to generate default methods.

```
1 public class ZoomDrawingView {
2   public final double getScale() {
3     return scale;
4   }
5 }
6 public interface IZoom{
7   public double getScale();
8 }
9 public class SpecialZoomView extends ZoomDrawingView force implements IZoom() {}
```

Figure 3. Declaration to generate default methods.

compiler using the underlined *force implements* clause to generate default methods for class *UndoableCommand* such that this class fulfills the interface *Command*.

The compiler compares the set of core methods in the class, implemented by the developer, with the interface the class should follow; after that, the compiler generates those methods in the class that are missing to provide the interface. Thus, we *force* the compiler to make the class in question implement the according interface by generating the missing default methods.[3] In Figure 2, the forwarding methods are not defined by the developer but the core methods are, like *execute*. After compiling, the class *UndoableCommand* implements the interface *Command* exactly as shown in Figure 1a. If an augmented *implements* declaration takes a parameter (like *myWrappedCommand* in Figure 2) forwarding methods are generated and empty methods otherwise; in the first case, the *implements* declaration's argument is used as a reference to forward messages to. If the forwarding target does not accept the forwarded message, the forwarding method will not be generated.

If different interfaces of one class overlap in declared methods, the compiler will not generate these overlapping methods. However, if a class implements different interfaces, the default methods regarding each interface may differ. Special rules also apply for abstract methods and final methods (all generated methods are non-abstract and non-final). Abstract methods may be defined in abstract classes instead of interfaces; since we propose IDM only for interface methods, abstract methods of superclasses cannot be defined implicitly. Final methods of superclasses will not be generated in subclasses (and thus not overridden) to prohibit compiler errors. As an example, in Figure 3 the method *getScale* of the interface *IZoom* will not be generated for class *SpecialZoomView* because this class inherits a finalized method of this signature from class *ZoomDrawingView*.

[3] We discuss possible problems in Section 5.

Measurement	**JHotDraw**	**Jcfe**	**java.io**
Methods	4 868	780	1 193
Generated methods	107	18	69
Generated methods in %	2.2	2.3	5.7
SLOC	29 026	10 672	10 131
SLOC with IDM	28 756	10 640	9 926
Generated SLOC	270	32	205
Classes and interfaces	589	66	119
Pruned classes using IDM	27	1	11
Pruned classes in %	4.5	1.5	9.2

Table 1. Measurements from three case studies.

4. Case Studies

We implemented IDM for Java5 as a pre-compiler but consider our solution a proper and lightweight mechanism to be integrated in a Java compiler.[4] In order to assess its relevance, we measure the impact of the proposed mechanism in three open-source software projects: *JHotDraw*, a GUI framework; *Jcfe*, an editor for Java binaries; and *java.io*, a standard Java package providing different streams, like *PrintStream*. The package *java.io* is based on abstract classes instead of interfaces (with the side effects discussed in Sec. 2.) which forced us to prepare the classes with Extract Interface refactoring [10]. We show the study's results in Table 1.[5]

We removed forwarding and empty methods that implement an interface declaration. After removing these methods in the programs, we instructed the compiler to re-generate the formerly removed methods – this way, we generated from 18 to 107 methods (2.2 to 5.7 % of all methods) and saved 32 to 270 lines of sourcecode. Using IDM, we pruned 27 classes for *JHotDraw* (4.5 % of all classes), we pruned one class and 11 classes respectively (1.5 % and 9.2 % respectively) for *Jcfe* and *java.io*. Notably, in *JHotDraw* using IDM, we generate 24 methods for one class that overall comprises 52 methods (*NullDrawingView*). In *Jcfe*, we generate 18 methods for one class (*NavigationClassVisitor*) that comprises 30 methods at all and in *java.io* we generated 15 methods for one class (*ObjectInputStream* which contains 53 methods at all). Some classes even become empty with IDM which allowed further simplifications with refactorings; in *JHotDraw* two classes become completely empty, four classes (one in *JHotDraw* and three classes in *java.io*) become empty of methods except of one delegatee access method to forward to.

We observed that some methods could not be generated. Default methods of an interface cannot be generated if they are not homogeneous (i.e., they do not perform the same actions) or are different from forwarding or empty methods. That is, in most cases (95 % in our case studies) a method is not a default method and cannot be generated. However, IDM saved implementing code of up to 5.7 % of all methods.

[4] To download the pre-compiler visit: http://wwwiti.cs.uni-magdeburg.de/∼mkuhlema/idm/

[5] SLOC are the lines of source code without empty and comment lines.

5. Discussion

In our case studies, we observed that all default delegation methods could be generated. The developers of *JHotDraw*, *Jcfe*, and *java.io* could have saved writing up to 5.7 % of all methods with IDM.

IDM only can generates homogeneous default methods that either simply forward messages or are empty. Although the tackled problem of repetitive default methods is rather small, to solve it is beneficial because it is typical for systems that actually use delegation. Notably, our solution is lightweight, only makes minimal changes to the language, and is backward compatible completely for legacy applications.

IDM does not impair type safety because only methods are generated that the class has to provide according to its interfaces. However, when used without care IDM may impair semantical correctness – when the class developer does not provide a core method for an augmented class, this method may be accidentally generated as a default method.

IDM allows programmers to implement delegation more easily because default methods are generated implicitely. Default methods can be generated using a parameter of an subtype declaration, i.e., only one method body definition needs to be defined to generate different methods of a class. Default methods are no longer replicated across multiple classes and methods within single classes are no longer repetitive. Furthermore, disabling methods, that for reuse were placed to high in an inheritance hierarchy, is avoided. That is, IDM exactly generates the methods needed by every single class. Some classes even may become empty and allow further simplifications using refactorings.

Default methods are generated at compile-time and are hidden during development; thus, IDM allows a developer to concentrate on the core implementation of each class and prevents controversial designs of classes (i.e., IDM prevents code smells Middle Man and Lazy Class) and complex class hierarchies (cf. Sec. 2.). The existence of generated methods after compilation eases debugging because every executed method is visible for each single debugging step. Type safety of implicit defined methods is guaranteed because the IDM mechanism generates all methods that a class has to provide while IDM does not invalidate core methods implemented by the developer.

Beside method generation, IDM encodes design information directly in the source code by exposing delegating and delegatee classes and, thus, improves reasoning about classes [4]; even if no method is generated, IDM annotates and explains the meaning of fields (e.g., references to delegatees), classes (e.g., delegatee classes), and interfaces. That way, IDM eases the communication between software designers that use the vocabulary of now exposed design patterns.

IDM additionally supports software evolution. When software evolves and methods are added to a delegating class, the developer only has to adapt the delegating class' interface that is used for method generation; remaining methods of other delegating classes that follow that interface are generated automatically – this is equivalent to adding methods to delegatees. However, if a default method in one subclass of the interface is not sufficient, the developer is needed to implement a new core method instead.

In summary, we achieved the following benefits with IDM: (a) a reduced number of methods, (b) isolation of core methods, (c) isolation of delegating and delegatee classes in the code.

6. Related Work

Related approaches and research is concerned with multiple inheritance, meta-programming, code generation, prototype-based languages, traits, aspect-oriented programming, and design documentation.

Multiple Inheritance. Multiple inheritance allows a class to inherit fields and methods, e.g., default implementations, from different superclasses. Thereby, the inheriting classes may cause the diamond problem[6] [30, 25].

Multiple inheritance may avoid replication of default methods different than forwarding or empty methods which are the only methods IDM provides reuse for, i.e., IDM is more specialized than multiple inheritance. In multiple inheritance, different classes may define default methods inherited by another class. This approach causes plenty of homogeneous methods in the default method's defining class which is a controversial design [10]. Notably, an independent class that does not inherit the default method's defining class may not reuse the default methods but replicates them. Subclasses of a default method's providing class that do not need all inherited methods must disable respective methods by overriding. Again a complex structure of fine-grained superclasses may help.

IDM is possible in Java with single inheritance and thus avoids the problems of multiple inheritance languages [30, 27]. In contrast to multiple inheritance, using IDM, every class may completely define its own implementation (no superclass defines parts of it) without replicating methods across these classes – this increases modularity of each class. Finally, IDM reuses one declaration several times to implement all methods of an interface, i.e., IDM reduces method replication within classes too which is impossible using multiple inheritance.

Meta-programming and code-generation. Meta-programming avoids arbitrary replication of code by code generation. For example, several approaches [36, 4, 31, 13] use meta-programming for the implementation of the patterns Proxy, Adapter, and Visitor.

IDM is a *pre-defined* meta-program. IDM cannot generate arbitrary methods like general meta-programming approaches but forwarding and empty methods, i.e., IDM is limited compared to a general term of meta-programming. But, IDM is very simple to use and integrated into the Java language.

Bosch introduced code generation concepts for structural design patterns [3]. The approach of Bosch does not generate methods based on an interface but needs an explicit definition for every method to generate together with the type of the body to generate. In IDM, code generation is associated to interfaces and improves implementing delegation by quantifying implementations over methods that perform the same actions.

Budinsky et al. customize design pattern descriptions, by arranging pattern roles, and customizing names [5]. Based on these customized descriptions, interfaces and classes are generated. The authors admit that this approach is badly integrated into a development process because generated interfaces and classes are difficult to evolve – changing these

[6] Methods with the same signature that a subclass inherits from different superclasses are ambiguous in that subclass.

classes possibly needs to reincorporate the generation process and to copy all changes to the new generated classes. Development tools, like Eclipse generate default methods, but – as customized design patterns of Budinsky – changes to the generated code need reinvestigation on code. IDM is integrated into the application code (i.e., its classes); hence, code is generated and integrated in the application without actions by the developer – evolving the classes hosting the generated code is possible without restarting a generation process and copying of changes (as in Budinsky's approach); changing all default method at once also does not need to reinvestigate their generation (as in IDEs). IDM does not pollute the classes with default methods during development and maintenance.

The Jamie pre-compiler targets at the same problem of generating interface methods automatically [34]. However, Jamie introduces a verbose and repetitive syntax and only focuses on the delegating class. IDM allows to generate methods for both, delegating and delegatee classes without language extensions for each of them, using a minimal language extension.

Prototype-based languages. Lieberman defined delegation as a programming concept for OOP [20], in which every object can be used as delegatee (here: *prototype*) of delegating objects. In contrast to Java, in prototype-based languages, delegating objects do not need to provide forwarding methods but delegation is done by the runtime-environment. Ostermann used this mechanism for implementing layered designs [25].

We propose IDM, which improves the use of delegation in Java but not in languages that rely on prototypes, like Self [33]. IDM does not tackle problems of object identity in delegating and delegatee objects as done in prototype-based languages [20, 28]. However, equivalently to prototype-based languages, IDM takes the burden from a developer to implement plenty of forwarding methods for wrapping delegatees. IDM may create different default implementations for classes while in Lieberman's delegation concept, all methods of all objects normally only forward messages.

Traits, mixins, AOP, and vitual classes. Traits define methods and fields to be reused in a class similar to superclasses that also provide reuse for methods and fields [27, 1]. Mixin classes [29] and virtual classes [21] behave similar in this respect – therefore, we will discuss their relation to IDM by means of traits. In contrast to inheritance, traits cannot instantiate objects [27] but can be assigned to classes (that can be instantiated) independently of other classes. Traits provide reusable default implementations for classes even in single-inheritance languages. Methods of traits can be multiplied through alias mechanisms where one method becomes available identically under different names [27].

Traits provide different benefits than IDM. Traits are not applicable for generating different delegating or delegatee methods for single classes (Traits may multiply a method's body with aliases for the method name [27], but the method signature (despite its name) and the body remains unchanged – this is inappropriate for forwarding methods of different names that we focus on.). IDM is more specialized but simpler than traits in that traits can introduce arbitrary methods into classes while IDM only generates forwarding or empty methods. In contrast to Traits, IDM does not divide the implementation of a class into multiple parts just to define default implementations; but, IDM integrates default method definitions into every class that needs to provide them.

Aspect-oriented Programming (AOP) [17] comprises different languages, like AspectJ [18, 16]. To apply standard implementations, AspectJ allows (1) to replace method calls and (2) to introduce methods into interfaces and classes. By replacing method calls (1) standard implementations can be executed instead of the called method. The developer has to avoid that two aspects introduce different methods of the same signature into one class or into different interfaces of one class – if he does he is warned by the aspect compiler but has to change his aspects manually; for IDM the compiler automatically ensures that methods in classes are not ambiguous. With IDM, exclusion of core methods of single classes is done automatically and does not need the adaptation of possibly multiple aspects [32]. One AOP code fragment used for replacing different method calls is identic for all calls – IDM generates *different* method bodies out of one simple statement. Finally, the benefits of IDM are gained from one single language mechanism, instead of a whole language as AspectJ.

7. Conclusions

In this paper, we addressed the problem of default methods in delegation and interface-based programming and discussed its impact in different styles of object-oriented programming – design patterns and frameworks. We proposed a lightweight mechanism, called Implicit Defined Methods (IDM), for Java, that avoids default methods, and discussed its limitations and benefits. The language mechanism generates default methods automatically. Using IDM, we can reduce code replication and simplify the implementation of class hierarchies. We test the language extension using three open source programs and found that we can simplify up to 9.2 % of all classes by generating up to 5.7 % of all methods. These percentages show that even though the proposed generative mechanism is only applicable in some situations, the according lightweight syntax extension is justified.

References

[1] A. Bergel, S. Ducasse, O. Nierstrasz, and R. Wuyts. Stateful traits and their formalization. *Computer Languages, Systems and Structures*, 34(2-3):83–108, 2008.

[2] G. Booch. *Object-Oriented Analysis and Design with Applications*. Addison Wesley Professional, 2nd edition, 1993.

[3] J. Bosch. Design patterns as language constructs. *Journal of Object-Oriented Programming*, 11(2):18–32, 1998.

[4] A. Bryant, A. Catton, K. De Volder, and G. C. Murphy. Explicit programming. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 10–18, 2002.

[5] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, 35(2):151–171, 1996.

[6] L. Cardelli and P. Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–523, 1985.

[7] R. Cardone and C. Lin. Comparing Frameworks and Layered Refinement. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 285–294, 2001.

[8] H. B. Christensen. Frameworks: Putting design patterns into perspective. *ACM SIGCSE Bulletin*, 36(3):142–145, 2004.

[9] W. R. Cook. Interfaces and specifications for the smalltalk-80 collection classes. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 1–15, 1992.

[10] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., 3 edition, 2005.

[13] S. S. Huang and Y. Smaragdakis. Expressive and safe static reflection with morphj. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*, page MISSING???, 2008.

[14] S. Jarzabek and L. Shubiao. Eliminating redundancies with a ”composition with adaptation” meta-programming technique. In *Proceedings of the International Symposium on Foundations of Software Engineering (FSE)*, pages 237–246, 2003.

[15] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[16] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 327–353, 2001.

[17] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[18] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Co., 2003.

[19] R. Lajoie and R. K. Keller. Design and reuse in object-oriented frameworks: Patterns, contracts, and motifs in concert. In *Proceedings of the Congress of the Association Canadienne Francaise pour l'Avancement des Sciences (ACFAS)*, pages 295–312, 1994.

[20] H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 214–223, 1986.

[21] O. L. Madsen and B. Moller-Pedersen. Virtual Classes: A Powerful Mechanism in Object-Oriented Programming. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 397–406, 1989.

[22] W. B. McNatt and J. M. Bieman. Coupling of design patterns: Common practices and their benefits. In *Proceedings of the International Computer Software and Applications Conference on Invigorating Software Development (COMPSAC)*, pages 574–579, 2001.

[23] B. Meyer and K. Arnout. Componentization: The Visitor Example. *IEEE Computer*, 39(7):23–30, 2006.

[24] E. R. Murphy-Hill, P. J. Quitslund, and A. P. Black. Removing duplication from java.io: A case study using traits. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 282–291, 2005.

[25] K. Ostermann. *Modules for Hierarchical and Crosscutting Models*. PhD thesis, Computer Science Department, Darmstadt University of Technology, 2003.

[26] D. Riehle. *Framework Design – A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology Zurich, 2003.

[27] N. Schärli, S. Ducasse, O. Nierstrasz, and A. Black. Traits: Composable units of behavior. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, 2003.

[28] C. Sekaraiah and D. Janaki Ram. Object schizophrenia problem in modeling is-role-of inheritance. In *Proceedings of ECOOP Inheritance Workshop*, 2002.

[29] Y. Smaragdakis and D. S. Batory. Implementing Layered Designs with Mixin Layers. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 550–570, 1998.

[30] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of the International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 38–45, 1986.

[31] M. Tatsubori and S. Chiba. Programming support of design patterns with compile-time reflection. In *Proceedings of the Workshop on Reflective Programming in C++ and Java*, pages 56–60, 1998.

[32] T. Tourw, J. Brichau, and K. Gybels. On the Existence of the AOSD-Evolution Paradox. In *Workshop on Software-Engineering Properties of Languages for Aspect Technologies*, 2003.

[33] D. Ungar and R. B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, 1987.

[34] J. Viega, P. Reynolds, and R. Behrends. Automating delegation in class-based languages. In *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems (TOOLS EUROPE)*, page 171, 2000.

[35] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *ACM SIGPLAN OOPS Messenger*, 1(1):7–87, 1990.

[36] E. V. Wyk, L. Krishnan, A. Schwerdfeger, and D. Bodin. Attribute grammar-based language extensions for java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 575–599, 2007.