A Case for Custom, Composable Composition Operators

Wilke Havinga, Christoph Bockisch, Lodewijk Bergmans
Software Engineering group – University of Twente
P.O. Box 217, 7500 AE Enschede, The Netherlands
{w.havinga,c.m.bockisch,l.m.j.bergmans}@ewi.utwente.nl

ABSTRACT

Programming languages typically support a fixed set of composition operators, with fixed semantics. This may impose limits on software designers, in case a desired operator or semantics are not supported by a language, resulting in suboptimal quality characteristics of the designed software system. We demonstrate this using the well-known State design pattern, and propose the use of a composition infrastructure that allows the designer to define custom, composable composition operators. We demonstrate how this approach improves several quality factors of the State design pattern, such as reusability and modularity, while taking a reasonable amount of effort to define the necessary pattern-related code.

1. INTRODUCTION

One of the most important quality characteristics of source code is its modularity. Good modularity is achieved, when each distinct piece of behavior (also called concern) in a program is encapsulated in one or only a few modules; and when it is possible to extend and refine this behavior in a way that requires no changes to existing code. A high degree of modularity in source code, thus, favors its re-usability and maintainability.

The degree of modularity that can be achieved, is significantly influenced by the composition power of operators offered by a language to compose modules. Therefore, research in the field of programming languages is intensively concerned with providing new composition operators. Examples are method or function calls, aggregation, inheritance, mixin-composition, or aspect-oriented composition.

However, we have observed that in each programming language only a few of the known composition operators are at the developer's disposal as language features. This hinders the modularity of source code. Thus, the ability to modularize source code is limited by the choice of composition operators made by the language designers. In our research, we want to enable the developer to freely use and mix all existing—and future—composition operators.

For most of the composition operators, different variations exist, e.g., when inheriting from a class, either the parent (e.g., in Beta) or the child implementation (e.g., in Java) may have precedence. While approaches exist to support different variants of single operators simultaneously (e.g., Beta-style and Java-style inheritance in [12]), the developer is typically provided with very limited choice. That is, different concerns may be well modularizable in different composition styles; but if no language exists that supports all necessary styles, not all concerns can be optimally modularized.

To avoid this limitation, often domain-specific languages (DSL) are developed that provide composition operators tailored toward a specific program domain. However, developing a DSL only pays off, when it is used sufficiently often. If this approach, thus, is not feasible and a general-purpose language is used, often the missing composition operators are emulated by a specific programming style, e.g., in terms of design patterns [10], which encode interactions between (and thus compositions of) objects.

To emulate composition operators, design patterns typically require some pieces of code which are application-independent (possibly tailored with element names from the application program) but cannot be localized in one module; we refer to these code pieces as *boilerplate code*. Boilerplate code entails several disadvantages.

- Firstly, it *obfuscates the design*; instead of specifying the relation of two or more modules explicitly, this code defines their composition imperatively. Because this code is scattered over multiple participating modules, the design intention becomes even more implicit.

- Secondly, boilerplate code is *difficult to write*. While it is not very sophisticated, its correctness is not easily enforced; for example, consider the Visitor pattern, where each `Element` class must implement the method `void accept(Visitor visitor){ visitor.accept (this); }`[1]. Each class contains the same line, but it is not possible to factor it out into the superclass.

[1]In languages like Java with an overloading semantics for methods, the static type of the argument distinguishes between the `accept` methods for different `Element` types in a `Visitor`.

- Finally, while it is sometimes necessary to combine multiple composition operators, *not all required operators can be emulated* by design patterns. As an example, consider the expression problem [8], where the building blocks of the application are data types and operations on them. With an object-oriented language, the data types are easily extensible, but not the operations. The Visitor pattern emulates a functional composition style, which makes it easy to extend the operations, but in turn the data types cannot be easily extended. Different language-level solutions to this problem have been proposed that are all founded on combining multiple composition operators [5, 8].

As also others have noticed [7, 26, 13], we claim that quality characteristics of design pattern implementations can often be improved if the implementation language supports particular composition operators. However, while this decreases the complexity of programs using a supported pattern, providing such support by extending the syntax of a language will increase the language's complexity [19].

In our research, we are concerned with developing a composition infrastructure, where the developer can choose from different composition operators and from different variations thereof. However, we do not simply aim at providing a fixed set of composition operators to choose from; but we aim to provide an infrastructure in which composition operators can be user-defined. In addition, our approach allows to re-use and combine implementations of composition operators—thereby developing new composition operators—because they are first-class. This also makes our approach open for future developments in the research of composition operators.

In this paper we present our approach through the example of the State pattern and our prototypical *Co-op* language. We have chosen this pattern because it is suitable to demonstrate the interplay between different composition operators, namely forwarding and delegation semantics as well as aspect-oriented composition. By this example we show that customizable composition operators can lead to a re-usable implementation of design patterns as well as to improved modularity of source code.

2. COMPOSITION ISSUES DEMONSTRATED

In this section, we demonstrate the occurrence of issues caused by language limitations, based on a concrete example. For this purpose, we discuss the object-oriented "State" design pattern [10], which realizes a state machine.

Figure 1 shows a concrete instance of the State pattern that (partially) implements the TCP/IP protocol. Based on this figure, which is very similar to the example found in the Design Patterns book [10], we identify several issues.

First, the State pattern has to be instantiated and tailored to this specific application. In general, many patterns (including the State pattern) specify roles that have to be mapped to implementation-specific classes. However, parts of these classes represent common pattern-defined behavior, made specific for a particular instantiation of the pattern. This obfuscates the generality of the design pattern, decreases the

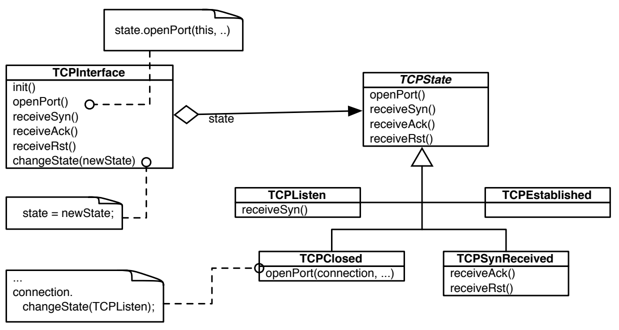

Figure 1: State pattern instantiation partially representing TCP/IP

separation between application-specific and pattern-generic code, and makes it harder to reuse common parts of the pattern implementation ("boilerplate" code).

Specifically in this example, the most important occurrence of boilerplate code is found in the methods defined in class `TCPInterface`. For each action (e.g., `openPort`, `receiveSyn`, etc.) supported by the state machine, this interface class has to define a method that forwards calls to the currently active state. As shown in figure 1, the forwarding method has to pass the `this` reference to the state object, such that it can, e.g., call `changeState` on the context object. Similarly, whenever a new action has to be added to the state machine, additional boilerplate code has to be added to both the State superclass `TCPState`, as well as `TCPInterface`. Both issues can be addressed more concisely in languages that support explicit delegation [22].

Second, pattern implementations may impose limitations on the way a particular concept is expressed. In the case of the State pattern implementation as shown in figure 1, the behavior associated with each state is modularized. However, state transitions are encoded as part of the actions, and thus become scattered over multiple State implementation classes.

As an alternative, the State pattern therefore also explicitly suggests that all state transitions can be kept in a single location, e.g., a transition table. This addresses the scattering of transition statements over the program, thus making it easier to, e.g., check whether an instantiation of the State pattern matches a corresponding state diagram, or to modify several transitions in one go. However, in many languages this alternative requires additional boiler-plate code, as is shown in figure 2.

In this alternative design, the constructor of `TCPInterface` constructs a table of state transitions. The methods in class `TCPInterface` each call the method `changeState(..)`. This method is parameterized by the action that is being executed, which is needed to look up the "next" state in the transition table. Similar to the code that "manually" forwards method calls to the current state object, the invocation of method `changeState`, passing the action, is thus replicated for each action supported by the state machine implementation.

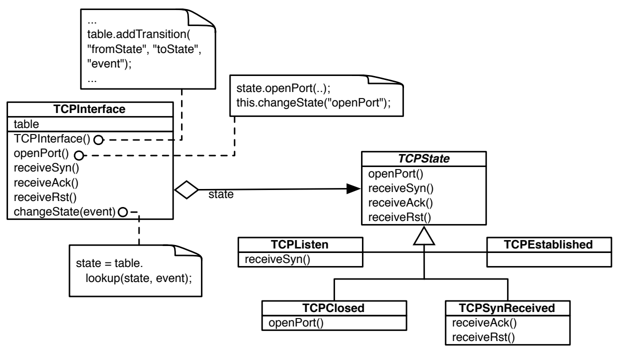

Figure 2: State pattern implementation using a transition table

One way to improve on the situation described above, is by addressing limitations in the underlying implementation language. In this case specifically, both implementations can benefit from a language that supports explicit delegation, whereas the table-based design can additionally benefit from a language that supports pointcut-advice constructs. We demonstrate this in detail in the next section; it should be noted, however, that this example is meant as a demonstrator for the usefulness of more flexible composition operator support in general, even if space limitations prevent us from discussing other examples here.

3. USING COMPOSABLE COMPOSITION OPERATORS

Our approach is based on a composition infrastructure, which supports composition primitives that allow programmers to design custom or domain-specific composition operators. This infrastructure is implemented as an object-based language called *Co-op*, which is discussed in detail elsewhere [14, 15].

Figure 3 shows a schematical overview of a *Co-op*-based design of the State pattern, applied to the TCP/IP example. As is the case with the original OO pattern, it supports both design alternatives discussed in the previous section (i.e., encoding transitions as part of the state implementations, or as a separate transition table). The lower half of the diagram contains the reusable, pattern-generic parts, which should fulfill two main tasks. First, it establishes and controls a delegation relation between a *context* object (as it is called in the original pattern description; in our example, class `TCPInterface` fulfills the role of *context*) and state implementation objects (instances of `TCPClosed`, `TCPListen`, etc.). Second, in case a table-based implementation is desired, it automatically ensures that the specified state transitions are executed at the appropriate moment, i.e., without adding any invocations to the application-specific state implementations (in the upper half of the diagram).

Below, we discuss each module described in the diagram in some detail, and show how this approach reduces the amount of boiler-plate code in the application-specific part.

Listing 1 shows the implementation of the generic, reusable parts of the State pattern. An instance of the State pattern can be created by constructing an instance of module

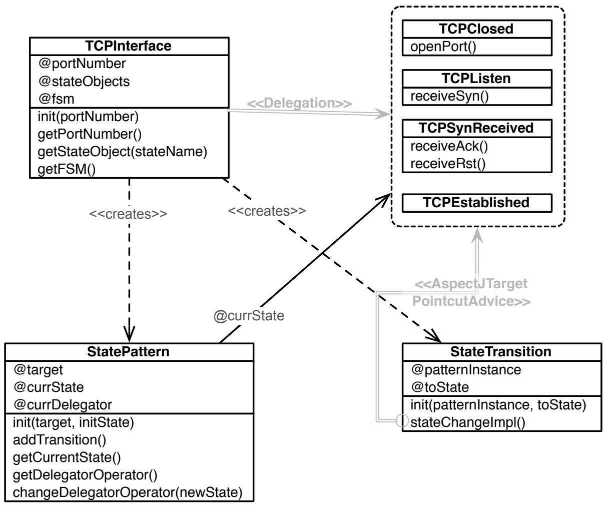

Figure 3: Design diagram of the *Co-op*-based State pattern

`StatePattern`. To facilitate the delegation from the context object to an object representing the current state, each state pattern instance keeps references to the `context` and `currentState` objects, and a reference to the delegation operator[2]. These instance variables of the pattern are defined on line 2. The constructor (lines 4–8) calls the operation that establishes the delegation relation (line 7). This operation, `changeDelegatorOperator(..)` (lines 10–18), which should be invoked whenever a state change is required, deactivates the delegation to the current state object (lines 12–13), and activates a new delegation relation from the context object to the new state object (lines 16–17).

We lack the space to discuss the internal details of the `Delegation` operator in detail, but a discussion of this exact operator can be found in prior work [14]. Here, it suffices to know that the constructor of the `Delegation` module establishes and activates delegation from the object referenced by the first argument (here: `@context`) to the object referenced by the second argument (here: `newState`). Effectively, this means that invocations on the context object are forwarded to the indicated state object, while the "this"-object still refers to the context object (i.e., `this`-calls are all directed to the context object).

Finally, lines 20–22 implement the state transition mechanism used in the table-based pattern implementation: by invoking `addTransition`, a pointcut-advice instance is constructed, which triggers `after` the specified `action` is invoked on the `fromState`. Whenever the pointcut triggers, as an advice the operation `stateChangeImpl` is invoked, as defined on lines 33–35, which changes the state to the selected `toState`. The module `StateTransition` stores references to the pattern instance, as well as the desired `toState`, so that these can be used by the advice[3].

[2]In *Co-op*, operators are themselves implemented as modules, and can be referenced as first-class objects. For a detailed explanation, see [14].

[3]Ideally, these could be supplied as advice parameters, mak-

```
module StatePattern {
  var context, currState, currDelegator;

  initWithContext:aContext initState:initState {
   context = aContext;
   currState = initState;
   this changeDelegatorOperator: initState;
  }

  changeDelegatorOperator:newState {
   // Deactivate the existing delegation (if any)
   (currDelegator isDefined) ifTrue:
    [currDelegator deactivate];

   // Active delegation to new state object
   currState = newState;
   currDelegator = Delegation newFrom: context to:
        newState;
  }

  addTransitionFrom:fromState action:action to:toState {
   AspectJTargetPointcutAdvice new: "after" matchTarget:
        fromState matchOperation:action aspectInstance: (
        StateTransition new:this to:toState) advMethod: "
        stateChangeImpl";
  }
  ... // trivial accessors not shown here
}

module StateTransition {
  var patternInstance, toState;

  init:aPatternInstance to:aToState {
   patternInstance = aPatternInstance;
   toState = aToState;
  }
  stateChangeImpl {
   patternInstance changeDelegatorOperator: toState;
  }
}
```

Listing 1: *Co-op*-based implementation of the State pattern

As mentioned in section 2, we found two types of functionality in the State pattern that can be made more reusable, while also removing the need for a lot of boilerplate code. First, by using delegation, it is no longer necessary to write manual forwarding operations in the context class (here: `TCPInterface`). Second, when using a table-based implementation, the pointcut-advice composition operator removes the need to manually invoke a method that decides about the next state. Note that although our approach *allows* the use of such a transition table, this is by no means obligatory; embedding transitions in action implementations works fine, as well. However, when transition tables are used, our approach removes the need for boilerplate associated with the original implementation.

Listing 2 shows how the State pattern implementation defined above as a custom, "pluggable" composition operator that can be used in any *Co-op* program, is applied to the TCP/IP example discussed in section 2.

```
 1  module TCPInterface {
 2   var portNumber, fsm;
 3
 4   init:aPortNumber {
 5    var closedState, listenState, synReceivedState,
          establishedState;
 6
 7     portNumber = aPortNumber;
 8   closedState = TCPClosed new;
 9   listenState = TCPListen new;
10   synReceivedState = TCPSynReceived new;
11   establishedState = TCPEstablished new;
12
13     fsm = StatePattern newWithContext: this initState:
           closedState;
14
15     fsm addTransitionFrom: closedState action: "openPort"
           to: listenState;
16     fsm addTransitionFrom: listenState action: "receiveSyn
           " to: synReceivedState;
17     fsm addTransitionFrom: synReceivedState action: "
           receiveAck" to: establishedState;
18     fsm addTransitionFrom: synReceivedState action: "
           receiveRst" to: listenState;
19     // ...additional transitions not shown here
20   }
21   getPortNumber { return portNumber; }
22  }
23
24  module TCPClosed
25  {
26   openPort {
27     Console writeln: "TCPClosed: opening port: " with: (
           this getPortNumber);
28   }
29  }
30
31  module TCPListen
32  {
33   receiveSyn {
34     Console writeln: "TCPListen: received SYN; sending SYN
           -ACK";
35   }
36  }
37  // etc. for other TCP states not shown here
```

Listing 2: Application of the generic State-pattern implementation

In this listing, the constructor of module `TCPInterface`, found on lines 4–21, sets up the state machine: it creates an instance of each TCP state modeled in this example (lines 8–11), and initializes a State pattern instance (line 14), appointing itself as the context object, and setting `closedState` as the initial state object. In this example, we also used the pointcut-advice based transition mechanism, which is initialized in lines 16–19. Note that all the initialization code here is completely application-specific, and also, no boilerplate related to the internal "machinery" required by the pattern implementation is visible. Once the state machine is thus set up, the delegation and pointcut-advice operators automatically take care of effectuating the desired state machine behavior.

The remaining code in listing 2 shows the mock-up state implementations. Note that the state modules do not contain or need any references to the state pattern. Still, because of delegation, you can still use behavior of class `TCPInterface` by means of *this*-calls, such as `this getPortNumber` (line 27).

An example demonstrating how the complete state machine can be instantiated and executed is shown in listing 3. Note that in listing 3, no boilerplate code or references to the state pattern are necessary either.

```
module Main {
 main { var tcpserver;
   tcpserver = TCPInterface new: "80";
   tcpserver openPort; // Request port open
```

ing the module `StateTransition` superfluous, but our much simplified implementation of AspectJ-like pointcut-advice does not support this at present.

```
5     tcpserver receiveSyn; // Receive incoming conn.
6     //etc.
7  } }
```

Listing 3: Using the state machine implementation

When the state machine is initialized (line 5), calls will be delegated to the initial state, an instance of `TCPClosed`. Thus, when `openPort` is invoked (line 6), the call is delegated to the operation `openPort` in `TCPClosed`, as shown before. After this action has been executed, the pointcut-advice that executes the state transition to `listenState`, an instance of `TCPListen`, is automatically invoked, since it triggers *after* the invocation of `openPort` on the instance of `TCPClosed`. Thus the state machine implementation automatically delegates calls to the appropriate implementation, and automatically triggers state changes.

The complete example as well as a prototype *Co-op*-interpreter (a plain jar-file, no installation required) can be downloaded from the *Co-op* website [1].

4. RELATED WORK

The work in this paper is related to a large body of research on defining new languages that support novel composition techniques, especially in the domain of object-based and aspect languages. Many papers also present a (small) set of composition techniques that aim at unifying existing ones. However, *most* of such related research proposes a *fixed set of composition operators*, presented as part of a language, extension of a language, or an application framework. In contrast, our work focuses on a language that has no—or just one—built-in composition operators, but rather is a platform for constructing a wide range of user-defined composition operators.

To the best of our knowledge, there are no other languages that offer *dedicated support* for user-defined composition operators (that can be reused and combined), at least not within the domain of object-oriented and aspect-oriented languages. Please note that this excludes languages that offer generic extension mechanisms—such as macros in Lisp—or allow for the extension and modification of the program through metaprogramming; our work is particularly related to metaprogramming [6] and especially meta-object protocols [21]. As explained, e.g., in [20], the power of metaprogramming comes with more complexity and responsibility.

This means that the difficulty of language design—except for the concrete syntax—is now on the MOP designer. Indeed, our work might just as well have been presented as a novel design of a MOP, but for practical reasons we chose to use a concrete language, *Co-op*. We are not aware of any MOPs (or languages, or frameworks) that offer similar generic abstractions and structure as we presented in this paper. In particular, we do not know any MOPs that provide abstractions for defining new composition operators with similar variety, expressiveness and composability. For example, *Co-op* explicitly supports a variety of object-oriented as well as aspect-oriented composition operators.

Of the research that aims at providing frameworks for higher-level languages through reflection or meta object protocols, we just mention COLA [24], AspectS [18], MetaClassTalk [4]: please refer to [14] for a discussion of these. There are several frameworks that aim at offering a generic platform for OO and AOP language implementations. For such platforms, the designers have typically made efforts to find a small set of generic constructs that typically serve as a target 'language' for a compiler/code transformation. An important distinction with our work is that these platforms do not aim at, and hence do not support, the ability of creating user-defined composition operators within the same language.

We have used the example of a modular, reusable implementation of a design pattern to exemplify that a single fixed composition technique is insufficient, while at the same time demonstrating that a design pattern implementation can in fact be modeled as a composition operator that 'extends' the language.

In [3], Bosch argues that language support is needed for explicit representation of design patterns in programming languages. The LayOM language offers a number of common design patterns as built-in constructs. These can be extended by growing the language, which supports modular extension of the lexer, parser and code generators for a new pattern: in contrast to our approach, the extension is not specified in the programming language itself. Also in [17], techniques for explicit representation of design patterns are proposed that are based on extension of the language and, consequently, the compiler.

Rajan and Sullivan [25] argue that design patterns are a suitable test case for evaluating and comparing aspect languages, because (1) design patterns are standard, well-documented design structures, and (2) existing examples [13] of design pattern implementations in AOPLs are available. They base their evaluation of the EOS language on a comparison with the AspectJ implementations of patterns in [13], following the metrics that have been proposed by Garcia et al. in [11].

Several efforts have been made to represent design patterns as first-class entities. For example, in [9], the fragment model is introduced to represent design patterns and their components. The FACE approach [23] extends the OMT notation with pattern-specific entities. Similarly, [26] proposes a modeling notation for representing design patterns—specifically for the support of the design and integration of object-oriented frameworks. All of these approaches build on the assumption that a design pattern has roles, which must be filled in by entities that use the design pattern. These roles are called participants in [10].

Hanneman and Kiczales [13] shows how to implement the GoF design patterns using aspect-oriented modularization techniques; in several cases this enables the modularization of all pattern-generic code within a single module (aspect).

5. EVALUATION AND CONCLUSION

Support for flexible, user-definable composition operators can help to improve the modularity and reusability of design pattern implementations, as we have shown for the State design pattern specifically in this paper.

Although this paper shows only one example, the results can be generalized. As has been discussed by the example of the Visitor pattern in section 1, or by the example of other patterns in [16, 15, 2, 13], rich composition operators in the language provide a powerful way to solve problems which are typically only "worked-around" by means of Design Patterns, i.e., requiring boilerplate code in several locations.

In addition, our approach of using a composable composition infrastructure (called *Co-op*) allows the definition of new composition operators that reuse existing ones. We have shown this by expressing the State design pattern as a custom composition operator, which reuses two existing operators that implement explicit delegation and a basic pointcut-advice mechanism. Also as a result of this, we could define such a relatively complex and reusable operator in less than 50 lines of code.

6. REFERENCES

[1] Co-op homepage, `http://wwwhome.cs.utwente.nl/~havingaw/coop/`, 2008.

[2] C. Bockisch. *An Efficient and Flexible Implementation of Aspect-Oriented Languages.* PhD thesis, Technische Universität Darmstadt, 2009.

[3] J. Bosch. Design patterns as language constructs. *JOOP*, 11(2):18–32, 1998.

[4] N. Bouraqadi, A. Seriai, and G. Leblanc. Towards unified aspect-oriented programming. In *Proceedings of ESUG 2005 (13th international smalltalk conference)*, 2005.

[5] C. Clifton, T. Millstein, G. T. Leavens, and C. Chambers. MultiJava: Design rationale, compiler implementation, and applications. *ACM Transactions on Programming Languages and Systems*, 28(3):517–575, 2006.

[6] P. Cointe. Reflective languages and metalevel architectures. *ACM Computing Surveys*, 28-4, 1996.

[7] M. Dominus. Patterns are signs of weakness in programming languages, `http://blog.plover.com/prog/design-patterns.html`.

[8] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the ACM SIGPLAN international conference on Functional programming*, pages 94–104, New York, NY, USA, 1998. ACM.

[9] G. Florijn, M. Meijers, and P. van Winsen. Tool support for object-oriented patterns. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*, 1997.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-wesley Reading, MA, 1995.

[11] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: A quantitative study. In P. Tarr, editor, *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*, pages 3–14. ACM Press, Mar. 2005.

[12] D. S. Goldberg, R. B. Findler, and M. Flatt. Super and inner: together at last! In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 116–129, New York, NY, USA, 2004. ACM.

[13] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*, pages 161–173. ACM Press, 2002.

[14] W. Havinga, L. Bergmans, and M. Aksit. A model for composable composition operators: Expressing object and aspect compositions with first-class operators. In *Proceedings of the 9th international conference on Aspect-Oriented Software Development*, Mar 2010.

[15] W. K. Havinga. *On the Design of Software Composition Mechanisms and the Analysis of Composition Conflicts.* PhD thesis, University of Twente, Enschede, June 2009.

[16] W. K. Havinga, L. M. J. Bergmans, and M. Akşit. Prototyping and composing aspect languages: using an aspect interpreter framework. In *Proceedings of 22nd European Conference on Object-Oriented Programming (ECOOP 2008), Paphos, Cyprus*, volume 5142/2008 of *Lecture Notes in Computer Science*, pages 180–206, Berlin, 2008. Springer Verlag.

[17] G. Hedin. Language Support for Design Patterns Using Attribute Extension. In *Proceedings of the Workshops on Object-Oriented Technology*, pages 137–140. Springer-Verlag London, UK, 1997.

[18] R. Hirschfeld. Aspect-oriented programming with AspectS. In M. Akşit and M. Mezini, editors, *Net.Object Days 2002*, Oct. 2002.

[19] R. Johnson. Design patterns and language design, `http://www.cincomsmalltalk.com/userblogs/ralph/blogView?entry=3335803396`.

[20] G. Kiczales. It's not metaprogramming. *Software Development Magazine*, (10), 2004.

[21] G. Kiczales, J. des Rivieres, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, Massachusetts, 1991.

[22] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. *SIGPLAN Not.*, 21(11):214–223, 1986.

[23] T. D. Meijler, S. Demeyer, and R. Engel. Making design patterns explicit in FACE: a frame work adaptive composition environment. In *ESEC '97/FSE-5: Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 94–110, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[24] I. Piumarta and A. Warth. Open, extensible object models. In *Self-Sustaining Systems*, volume 5146/2008 of *Lecture Notes in Computer Science*, pages 1–30. Springer, Springer Berlin/Heidelberg, 2008.

[25] H. Rajan and K. Sullivan. Design Patterns: A Canonical Test of Unified Aspect Model. Technical report, Iowa State University, 2005.

[26] D. Riehle and T. Gross. Role model based framework design and integration. *SIGPLAN Not.*, 33(10):117–133, 1998.