A Dissertation Submitted to Department of Creative Informatics,
Graduate School of Information Science and Technology,
The University of Tokyo

A Study on a Single Construct for Events, Aspects, and Behaviors

（イベント、アスペクト、ビヘイビアのための
単一言語機構に関する研究）

By

荘　永裕
YungYu ZHUANG

Supervisor:

千葉　滋
Shigeru CHIBA

December 2013,

Copyright © 2013 YungYu ZHUANG.
All Rights Reserved.

Abstract

Programming paradigms are so important that a lot of research activities are devoted to the support for them. How to support the implementation of paradigms can be classified into three types of approaches: by design patterns, by dedicated constructs, and by generic constructs. However, none of them are sufficient. Design patterns can be used to implement most paradigms but not all. Furthermore, without language support the code tends to scatter and tangle. Dedicated constructs greatly improve the modularity of code, but also increase the number of constructs in a language supporting multiple paradigms; a large number of constructs complicates the language design. Generic constructs can be considered as a potentially good approach, but the number of supported paradigms in current research is quite limited; existing generic constructs are not flexible enough to support more paradigms.

To overcome the problem that existing generic constructs are not flexible enough, this thesis proposes a new generic construct, *method slots*, based on our observation of the common ground among the implementations of three important paradigms in the real world: OOP, the event-handler paradigm (event-driven programming), the aspect paradigm (aspect-oriented programming). The common ground has never been noticed before this thesis since the dedicated constructs for these paradigms were individually developed from the beginning. The observation on the similarities motivates us to extend the methods in JavaScript to *method slots*, which can be used as methods, events, and advices. To demonstrate how *method slots* can be used in practice, a Java-based language named *DominoJ* is proposed with a compiler implementation. We then evaluate *DominoJ* by comparing with existing languages, running benchmarks for it, and rewriting programs as case studies.

The concept of *method slots* is very simple and easy to extend. To support

this argument, we demonstrate how to extend *method slots* by taking the example of the reactive paradigm (functional-reactive programming). We first compare the reactive paradigm with the most similar one in the paradigms supported by *method slots*: the event-handler paradigm. We find that the major difference between them is whether the event composition is automatic or not. Then we discuss the definitions for event composition in existing event mechanisms, and get the conclusion that existing event mechanisms lack an *inference-based definition* to automatically select events for a higher-level event. This thesis proposes such an *inference-based definition* by adding only one more operator for *method slots*. How the operator can be used for the reactive paradigm is presented with a feasible implementation and discussed in detail to clarify the limitations.

Acknowledgments

First of all, I would like to express my deep gratitude to my supervisor, Prof. Shigeru Chiba. He always gives me great advice on doing research and guided me through the process of this thesis. I learned a lot from him, especially how to study and present. He is also the person who kindly gave my the chance to start my research again in Japan. I also would like to thank the thesis committees: Prof. Hiroshi Esaki, Prof. Kei Hiraki, Prof. Mary Inaba, Prof. Hideki Nakayama, Prof. Hidehiko Masuhara, and Prof. Shigeru Chiba. They gave me valuable comments on not only the contents of this research but also the arguments of this thesis.

My grateful thanks go to all my colleagues in Chiba's group. In particular, I would like to thank Prof. Yoshiki Sato, Michihiro Horie, Masayuki Ioki, Shumpei Akai, Fuminobu Takeyama, Maximilian Pascal Scherr, Kazuhiro Ichikawa, and Hiroshi Yamaguchi. They gave me many help in both life and research.

I would like to thank all my family. I always get encouragement from my parents, my elder brother, and my wife's parents. I am very grateful to my wife; it is she who encouraged me to come to Japan. She takes care of our daughter alone to let me concentrate on my research. I was even unable to accompany her when our son was born. Without her support it is impossible to finish this thesis.

Finally I appreciate the Development of System Software Technologies for post-Peta Scale High Performance Computing project from JST CREST and the CompView Global COE program from JSPS for the financial support (as a research assistant), and the scholarship from Interchange Association (Japan). They supported me to complete my doctoral program.

Contents

1 Introduction **1**
1.1 Motivating problems . . . 2
1.2 Solution by this thesis . . . 4
1.3 Position of this thesis . . . 6
1.4 Structure of this thesis . . . 7

2 Backgrounds **9**
2.1 Programming paradigms . . . 9
2.1.1 Object-oriented programming . . . 10
2.1.2 The event-handler paradigm . . . 17
2.1.3 The aspect paradigm . . . 19
2.1.4 The reactive paradigm . . . 25
2.2 Supporting the implementation of paradigms . . . 26
2.2.1 Design patterns . . . 26
2.2.2 Dedicated constructs and the unification . . . 30
2.2.3 Generic constructs and the integration . . . 31
2.3 Comparing the three approaches . . . 32

3 Method Slots **37**
3.1 Introduction . . . 37
3.2 Motivation . . . 38
3.3 DominoJ . . . 42
3.3.1 Method slots . . . 42
3.3.2 Assignment operators for method slots . . . 45
3.4 Evaluation . . . 51
3.4.1 The implementation . . . 51
3.4.2 Microbenchmarks . . . 56

3.4.3 Compiling and running the DaCapo benchmark suite . . . 59
3.4.4 Method slots and design patterns 64
3.4.5 The event-handler paradigm 73
3.4.6 The aspect paradigm 79
3.4.7 Summary of the coverage 84
3.4.8 Event-handler vs. Aspect 85
3.5 Case study 86
3.5.1 The events and delegates examples in C# tutorials 86
3.5.2 The Observer pattern in JHotDraw 94
3.5.3 The aspects in AspectTetris 102
3.6 Related work 108
3.7 Summary 111

4 An Extension for Supporting the Reactive Paradigm 113
4.1 Introduction 113
4.2 Emulating the reactive paradigm by the event-handler paradigm . 114
4.2.1 An example of the reactive paradigm 115
4.2.2 The event-handler paradigm can do the same thing 116
4.2.3 The major difference between the two paradigms 118
4.2.4 The definitions of higher-level events in existing event mechanisms 118
4.2.5 The lack of inference rules 123
4.3 ReactiveDominoJ 123
4.3.1 Using DominoJ for the event-handler paradigm 123
4.3.2 The braces operator for the inference-based definition . . 124
4.3.3 The implementation of the reactive extension 130
4.4 Discussion 139
4.4.1 Preliminary microbenchmarks 139
4.4.2 Comparing with DominoJ 141
4.4.3 Comparing with AspectJ 142
4.4.4 For the reactive paradigm 143
4.5 Related work 149
4.6 Summary 151

5 Conclusions 153

Bibliography 159

List of Figures

2.1 The objects in the music player example 11
2.2 The model of the music player example 11
2.3 The musicPlayer object can be replaced with the musicPlayerH object 12
2.4 The model of the media player modified from the music player example 13
2.5 The class hierarchy for mediaEngine of the media player example 14

3.1 In JavaScript, both an integer and a function are fields on an object 42
3.2 A method slot is an extended field that can keep more than one function closure 43
3.3 The keywords $caller and $predecessor 51
3.4 The average time of continuously calling a method in Java and DominoJ, including the reflection version 58
3.5 The average time of using the operators for method slots 59
3.6 The average execution time of the 12 DaCapo benchmarks compiled by the three compilers 62
3.7 The box plot of the execution time for the DaCapo benchmarks . 63
3.8 The execution order of the shape example in EScala and DominoJ 78
3.9 Adding another advice to the shape example in AspectJ and DominoJ 83
3.10 Calling proceed in AspectJ and DominoJ 83
3.11 The design decision of EScala, AspectJ, and DominoJ 85

4.1 An example of spreadsheet 116
4.2 The DominoJ version needs the explicit bindings for the event composition of this.changed 128

4.3 According to the default closure of update, this.update depends on b1.getValue and c1.getValue 128
4.4 According to Listing 4.3, getValue depends on setValue 129
4.5 The actual bindings are completed by inference 129
4.6 The dependency graph for Shape 133
4.7 The dependency graph for Meter 133
4.8 The dependency between the objects m, s1, s2, and s3 133
4.9 An endless propagation loop 137
4.10 The average time of selecting a set of method slots in DominoJ and in ReactiveDominoJ 141
4.11 The average time of selecting a method slot through a number of objects in ReactiveDominoJ 141
4.12 The braces operator infers the fields it eventually depends on and selects their writer method slots 143

List of Tables

2.1 Comparing the three approaches to implementing paradigms . . . 33

3.1 The four assignment operators for method slots 45
3.2 The result of compiling the DaCapo-9.12-bach benchmark suite by the three compilers 61
3.3 Method slots can be applied to design patterns 65
3.4 The benefit of applying DominoJ to design patterns 66
3.5 The roles and bindings of the event-handler paradigm in EScala and DominoJ 74
3.6 The mapping of language constructs for the aspect paradigm in AspectJ and DominoJ 80
3.7 A summary of the significant characteristics of the two paradigms and the support in DominoJ 84
3.8 The 8 subject-observer pairs defined in JHotDraw 97
3.9 Rewriting the Observer pattern in JHotDraw by DominoJ 99
3.10 Rewriting the Observer pattern in JHotDraw by DominoJ (continued) 100
3.11 All the files modified in the DominoJ version of JHotDraw 101
3.12 The aspects in AspectTetris 104
3.13 Rewriting the aspects in AspectTetris to the DominoJ version . . 105

4.1 A comparison among the three types of definitions 119

List of Listings

2.1 MediaEngine and its subclasses in Java 15
2.2 MediaEngine and its subclasses in JavaScript 16
2.3 Switching the icon of the play button without events 18
2.4 Switching the icon of the play button using events in EScala . . . 20
2.5 Printing log messages for each method 21
2.6 Try to modularize the code for the logging concern 22
2.7 Modularizing the logging concern to an aspect 24
2.8 Switching the icon of the play button using the Observer pattern 28
2.9 MediaPlayer is the observer of both EngineSubject and FileSubject 29
3.1 Defining a reaction in EScala 40
3.2 Defining a reaction in AspectJ 41
3.3 A sample code in DominoJ 44
3.4 The algorithm of calling a method slot 45
3.5 The algorithms of the four assignment operators 47
3.6 The algorithm of checking the types 48
3.7 The algorithm of binding method slots 49
3.8 The applications/libraries that are not compiled by the DominoJ compiler in this experiment 60
3.9 The Chain of Responsibility pattern example in Java 68
3.10 The Chain of Responsibility pattern example in DominoJ 68
3.11 The Template Method pattern example in Java 70
3.12 The Template Method pattern example in DominoJ 70
3.13 The State pattern example in Java 71
3.14 The State pattern example in DominoJ 72
3.15 Using DominoJ for the event-handler paradigm 75
3.16 Using DominoJ as the aspect paradigm 81
3.17 Rewrite UpdateDisplay for pertarget 81
3.18 The example in C# events tutorial 87

3.19 The DominoJ version of the example in C# events tutorial 88
3.20 The Bookstore namespace of the first example in C# delegates tutorial 90
3.21 The BookTestClient namespace of the first example in C# delegates tutorial 91
3.22 The DominoJ version of the first example in C# delegates tutorial (the bookstore package) 92
3.23 The DominoJ version of the first example in C# delegates tutorial (the booktestclient package) 93
3.24 The second example in C# delegates tutorial 94
3.25 The DominoJ version of the second example in C# delegates tutorial 95
4.1 Using Flapjax to implement the spreadsheet example 117
4.2 Using DominoJ to implement the spreadsheet example 117
4.3 The source code of IntCell 118
4.4 The semantics of event selection 126
4.5 Using ReactiveDominoJ to implement the spreadsheet example . 127
4.6 The shape and meter example in DominoJ 131
4.7 The shape and meter example in ReactiveDominoJ 132
4.8 The event selection in the ReactiveDominoJ compiler 135
4.9 Selecting events for printing two fields in AspectJ 144
4.10 Selecting events for printing two fields in ReactiveDominoJ 144

Chapter 1

Introduction

The art of programming is the art
of organising complexity, of mastering multitude
and avoiding its bastard chaos as effectively as possible.
—E. W. Dijkstra

Unless you can support the paradigms I use when I program,
or at least support my extending your language
into one that does support my programming methods,
I don't need your shiny new languages.
—R. W. Floyd

If you never write programs, programming paradigms are nothing to you; but if you are a programmer, programming paradigms are the air, the water, the must-have. It is not impossible to write a program without programming paradigms, but it sounds like building a house without any plan or strategy: it might be possible, but no one knows if it will success. Even it succeeds, nevertheless, how to maintain and extend is the next problem. Repairing it is a nightmare, and possibly tougher than rebuilding it. Maybe the term programming paradigm is not familiar to everyone, but anyone interested in programming should have heard object-oriented programming (OOP) or functional programming; they are programming paradigms. Programming paradigms are methodologies for writing programs, which help programmers to resolve a certain implementation issue or modularize their code.

As what Robert W. Floyd mentioned in his Turing Award lecture [29], a shiny new language does not attract programmers unless the language supports the programming paradigms they use. It is not wrong to say that we are always choosing between programming languages for implementing a program, but saying that we are always choosing between programming paradigms might be more proper. To a programmer, programming paradigms are even important than programming languages.

Someone might not agree that programming paradigms are more important than programming languages: "choosing a language to use is always our first step to write a program, isn't it?" Yes, it is, but I believe that which programming paradigms supported by the language is always the first concern for choosing a language. If you are a functional programming lover, an OO language never attracts you even it owns a lot of great libraries. Similarly, a defender of OOP might learn what functional languages are but never use them. Programmers care about the languages but care about the programming paradigms more.

Since programming paradigms are important, how to support the implementation of programming paradigms in a language is an important issue. Maybe we can say that the value of a language is determined by how it can be used to implement a programming paradigm. For example, we might choose Java to implement a program if we want to use OOP. Object instances can be automatically created according to the class declaration by the new operator, so we do not have to copy the memory for fields. When a method on an object instance is called, it is dynamically dispatched to proper method implementation. Such functionalities are built-in and there is no need to implement additional code; Java satisfies our needs. Suppose that now we want to use event-driven programming (the event-handler paradigm) in Java we could implement by design patterns, though the pattern code might be repeatedly implemented. In other words, Java does not give built-in events, so we have to write some pieces of code for the event-handler paradigm. It is not a perfect solution, but also works for us. However, we have no idea how to use aspect-oriented programming (the aspect paradigm) in Java since it is a limitation. Which programming paradigm we can take advantage of is always a concern when we choose a language.

1.1 Motivating problems

Even though programming paradigms are so important, current support for the implementation of programming paradigms is not sufficient at all. Generally speaking, there are three approaches to supporting the implementation

of programming paradigms—we will discuss them later—but none of them is sufficient. Here we focus on three paradigms and discuss these approaches to supporting the implementation of the three paradigms. The three paradigms are OOP, the event-handler paradigm, and the aspect paradigm, which are very important programming paradigms in the real world. Object-oriented programming (OOP) has been developed for more than thirty years and now widely used in academia and industry. Event-driven programming (the event-handler paradigm) was introduced to loosen the coupling of subjects and observers and has been used over the past several decades. At the beginning it was independent of OOP but now is usually used along with OOP. The event-handler paradigm can be found in almost every GUI library on either MacOSX, Windows, or X11. In a system such as Android and iOS all operations are composed of events and handlers. Aspect-oriented programming (the aspect paradigm) [43], which is relatively younger, was developed for modularizing cross-cutting concerns. The aspect paradigm can modularize the code that cannot be modularized by OOP.

As we slightly mentioned above, there are three approaches to implementing such a programming paradigm. The first approach is implementing by design patterns, the support of which is undoubtedly insufficient since the modularity of code is not good. We should regard design patterns as the last approach we can take rather than the best approach. As what we learned from the famous “GoF” design patterns book [30], paradigms can be implemented by the pattern code. For example, the event-handler paradigm can be implemented by the Observer pattern. The techniques revealed by programming books that explain how to write code in a specific pattern belong to this approach. In other words, writing code pieces in a certain style to resolve a specific implementation issue that cannot be directly represented by language constructs. This approach makes it possible to use the paradigms that are originally not supported by the language. However, the pattern code might cause code scattering and code tangling.

The second approach is implementing by dedicated constructs. This approach lacks the flexibility and results in a large number of constructs in a language supporting multiple paradigms; it complicates the language design and makes the language hard to learn. Nowadays more and more dedicated constructs are designed for a specific paradigm since implementing a paradigm by a design pattern is difficult to maintain. The code of implementing the pattern scatters the program. This drawback makes it difficult to separate the pattern code from the main concern. As the discussion above, the Observer pattern can be used to implement the event-handler paradigm, but the code for managing the list of observers and notifying the observers is mixed with the main concern. Furthermore, maintaining two sets of observers

might cause code tangling. This approach, implementing by dedicated constructs, sounds good, but there is a drawback: the lack of flexibility. Dedicated constructs cannot be shared by different paradigms. As a result, a language supporting multiple paradigms must provide all the constructs for different paradigms. For example, [54, 70, 27, 40, 32] give both dedicated constructs for OOP and the event-handler paradigm, and [42, 3] give both dedicated constructs for OOP and the aspect paradigm. Such a unification of dedicated constructs increases the number of constructs in a language and makes the code analysis complicated. Every time a new dedicated construct is added to a language, the number of the construct combinations needed to analyze is increased as well. Furthermore, it might not be able to apply the optimization for existing constructs to the new construct. If the number of language constructs is smaller, the design of the compiler and the debugger is simpler.

The third approach is implementing by generic constructs. This approach shows a good direction, but current support cannot be considered as sufficient. There are several research activities devoted to develop generic constructs supporting multiple paradigms, but the number of supported paradigms is not larger than two as far as we know. This approach integrates the dedicated constructs for multiple paradigms into generic constructs. Such an integration can reduce the number of constructs in a language and keep the design simple, while make it possible to support multiple paradigms without writing pattern code. This approach is the best one to implement paradigms. Unfortunately, it is really hard to extend a single construct to support multiple paradigms. The coverage of the paradigms supported by existing generic constructs is usually limited to two paradigms. For example, OOP, the event-handler paradigm, and the aspect paradigm are major pardigms today. For these paradigms, there have been a number of constructs proposed but their coverage are limited. Several research activities [35, 75, 10] support OOP and the aspect paradigm, but cannot cover the event-handler paradigm. On the other hand, other research activities such as [62, 38] support only the aspect paradigm and the event-handler paradigm. If there is a more simple, generic, and flexible construct, we can use it for more paradigms and make the language design simpler.

1.2 Solution by this thesis

Although all the three approaches to supporting the implementation of programming paradigms are not sufficient, the third approach (generic constructs) is a potentially good approach. Supporting multiple paradigms by

generic constructs can not only improve the modularity of code, but also avoid increasing the total number of constructs needed in a language. The problem of this approach is that the generic constructs in current research are not flexible to support more than two paradigms. However, supporting more paradigms by generic constructs is a challenge but not a limitation. This thesis proposes a new generic construct named *method slots* for supporting multiple programming paradigms. We first present how *method slots* can be used for three important paradigms: OOP, the event-handler paradigm, and the aspect paradigm. Then how *method slots* can be extended by a small extension is demonstrated by adding the support for the reactive paradigm.

There have been several research activities devoted to the generic constructs for two of OOP, the event-handler paradigm, and the aspect paradigm, since the three paradigms are so important in the real world. However, the common ground among their implementations was not clearly recognized, thus no one could propose a generic construct that is flexible enough to support all of them. Our observation on the similarities among their implementations motivates us to propose a more flexible generic construct named *method slots* to support them. *Method slots* are very simple and can replace the methods in OOP. With the assignment operators proposed in this thesis *method slots* can also be used as the events in the event-handler paradigm and the advices in the aspect paradigm. Any language providing this generic construct and its operators will naturally support the three paradigms.

Since a *method slot* is a simple construct, its expressive power can be augmented to cover other paradigms by only adding a small extension. In this thesis we take the reactive paradigm as an example of how to extend *method slots*. We first find out the most similar paradigm in the paradigms that are already supported by *method slots*, the event-handler paradigm. Then we can consider the reactive paradigm from the viewpoint of the event-handler paradigm, and know that the major difference between them is whether the event composition is automatic or not. By discussing the ways to compose a higher-level event in existing event mechanisms we can get the conclusion that the event-handler paradigm lacks inference rules for event composition. Finally a new operator, the braces operator, is proposed to automatically select events for a higher-level event as what the reactive paradigm does.

Method Slots

A *method slot* is an object's property like a field that can keep more than one function closure at the same time. With the assignment operators given by *DominoJ*, the function closures in a *method slot* can be manipulated,

for example adding a closure that calling another *method slot* to a *method slot*. We analyze the similaries and the differences between the event-handler paradigm and the aspect paradigm, and then show how to use *DominoJ* for the two paradigms. A prototype compiler is implemented and preliminary microbenchmarks are run for showing the feasibility. We also compile the DaCapo benchmark suite [7] by the *DominoJ* compiler and compare the running performance with the one compiled by the plain Java compiler. For OOP we analyze how *method slots* can be used by applying to the GoF design patterns. For the event-handler paradigm we show how to translate C# events to *method slots* by examples [56, 55] and rewrite the Observer pattern code implemented for events in the famous Java GUI framework, JHotDraw [39]. As to the aspect paradigm, a real AspectJ program named AspectTetris [28] is rewritten as a case study.

An Extension for Supporting the Reactive Paradigm

We point out the major difference between the reactive paradigm and the event-handler pardigm is event composition, in other words how the events are selected for a higher-level event. The existing three types of definitions of higher-level events, enumeration-based definition, pattern-based definition, and predicate-based definition, are not as automatic as the reactive paradigm. We then propose a new kind of predicate-based definition, *inference-based definition*, which can select the events by inference as what the reactive paradigm does. A small extension named *ReactiveDominoJ* is proposed to give the braces operator for providing the *inference-based definition* for event composition. The braces operator takes a *method slot* and infers all the *method slots* that affect the output of that *method slot*. Then all the events behind an event can be automatically selected. How to realize the event selection while preserve the dynamic method dispatch in OOP is shown by the semantics along with a compiler implementation. To evaluate how *ReactiveDominoJ* works for the reactive paradigm we show several examples.

1.3 Position of this thesis

This thesis extends the approach of generic constructs to cover more than two paradigms. The generic construct proposed in this thesis is very simple and flexible, which can be extended to support four paradigms by proposing operators. This thesis takes the approach of generic constructs to support the implementation of programming paradigms since implementing generic

constructs is the most potential approach as we explained above. Unlike the approach of design patterns, generic constructs are supported by languages and thus no pattern code is necessary. Unlike dedicated constructs, generic constructs can be shared among paradigms and thus do not complicate the language design.

Before this thesis there is no existing research activity that can cover all the following paradigms as far as we know: OOP, the event-handler paradigm, the aspect paradigm, and the reactive paradigm. They are important and widely-used paradigms in the real world. Although there have been several research activities focused on the constructs of part of them, as far as we know no one covers all of them. Furthermore, most proposals are unification work on the constructs for these paradigms, which means that all dedicated constructs for these paradigms are given in a language. For example, [70, 27, 40, 32] are the research activities devoted to unifying the dedicated constructs for the event-handler paradigm and OOP; [43, 3] are the ones unifying the dedicated constructs for the aspect paradigm and OOP, and [52, 53, 49] are the ones unifying the dedicated constructs for the reactive paradigm and OOP. Their approaches are either proposing new dedicated constructs or dedicated libraries for supporting special types of classes. As to the integration work, [35, 75, 10] propose generic constructs for the aspect paradigm and OOP, and [62, 38] propose generic constructs for the event-handler paradigm and the aspect paradigm. Although a joint work of the research might support three paradigms, for example [3, 62] or using [49] with [32], no one covers all of the four paradigms with an integration solution. On the other hand, the relation between the event-handler paradigm and the reactive paradigm is also discussed in [50]. As to the relation between the aspect paradigm and the reactive paradigm, as far as we know, there is no publication yet. This thesis analyzes the essentials of OOP, the event-handler paradigm, the aspect paradigm, and the reactive paradigm to show the possibility of integrating the dedicated constructs proposed for them in existing research into a single generic construct.

1.4 Structure of this thesis

From the next chapter we explain the programming paradigms this thesis focuses on, the issues addressed by this thesis, and then present the details of our proposals with evaluation. The rest of this thesis is organized as follows:

Chapter 2: Backgrounds

This chapter gives an overview of the four paradigms this thesis focuses. What the paradigms are and what the issues they resolve are explained. Then the three approaches to supporting the implementation of the four paradigms are shown. Why this thesis takes the approach of generic constructs is explained by comparing the advantages and the disadvantages of using the three approaches. The languages proposed for the four paradigms and their language constructs are also discussed. The unification work and integration work on part of the four paradigms are briefly introduced as well.

Chapter 3: Method Slots

This chapter proposes the new generic language construct, *method slots*, to simplify the language design and the constructs programmers have to learn. The semantics of *method slots* and the assignment operators are presented with a Java-based language named *DominoJ*. How to use *DominoJ* for the event-handler paradigm and the aspect paradigm are also shown by comparing with other languages. The evaluation is performed by running benchmarks and case studies.

Chapter 4: An Extension for Supporting the Reactive Paradigm

This chapter demonstrates how to add a small extension to extend *method slots* to support one more paradigm: the reactive paradigm. First the similarities and the differences between the reactive paradigm and the event-handler paradigm are analyzed to clarify what we need for emulating the reactive paradigm by the event-handler paradigm. Then we summarized what are provided in existing research of the event-handler paradigm to reveal the lack of inference rules in existing event mechanisms. This chapter proposes the braces operator to automatically select a set of *method slots* that affect the output of a *method slot*. The extension of *DominoJ* named *ReactiveDominoJ* is presented in detail. The semantics of such a new kind of predicate-based definition, *inference-based definition*, in *ReactiveDominoJ* is shown along with a compiler implementation. Then how the reactive paradigm can be supported by *ReactiveDominoJ* is discussed and compared with other languages.

Chapter 2

Backgrounds

Since this thesis is an integration work on the constructs of several programming paradigms, what programming paradigms are must be explained first. The programming paradigms this thesis focuses on: OOP, the event-handler paradigm, the aspect paradigm, and the reactive paradigm, must be explained as well. After that, the three approaches to implementing a programming paradigm: by design patterns, by dedicated constructs, and by generic constructs, are discussed to point out the reasons why this thesis takes the approach of generic constructs.

2.1 Programming paradigms

Programming paradigms are the methodologies for building the code blocks in a program, and thus very important to both programmers and language designers. Properly using programming paradigms can help to prevent oversights in a program. The first contribution to this subject might be "Notes on Structured Programming" [16] written by Edsger W. Dijkstra, which clarifies the understanding of the program structure. Programming paradigms are so important that every programmer must learn how to properly use programming paradigms in their programs. By using programming paradigms programmers can benefit from the solutions that have been invented and proved.

Nowadays a lot of programming paradigms have been developed for resolving different problems. Several of them are discussed from a very basic view such as how to describe the computation, the example of which are imperative programming and declarative programming. Several of them further discuss the logic, the expression, and the structure in detail, for example object-oriented programming (OOP) and functional programming. Several programming paradigms take a step further to target at specific issues, for example event-driven programming (the event-handler paradigm), aspect-oriented programming (the aspect paradigm), and functional-reactive programming (the reactive paradigm).

In order to support different paradigms, especially the paradigms for specific issues, various language constructs and design patterns are introduced. Constructs provide built-in language support for paradigms, while design patterns make it possible to implement the paradigms that are not directly supported by the language. As a consequence, there is a trend towards proposing new constructs for paradigms and naturally providing all constructs for different paradigms in a language. Unless the proposed constructs are generic and can be used for different paradigms, the number of language constructs is always increasing.

Below we will continue explaining the backgrounds: the programming paradigms this thesis focuses on and the language constructs for them by examples, in order.

2.1.1 Object-oriented programming

Undoubtedly object-oriented programming (OOP) has been one of the most well-known programming paradigms in either academia or industry. OOP improves code modularity and makes code reusable. In OOP how a program works is described as interactions between objects. An object represents a component in the program, which has fields and methods. The fields are the properties of the object, and the methods are the actions that the object can perform. When an object gets a message sent by another object, its corresponding action will be performed. For example, we can roughly describe a music player by three components: the frontend, the backend, and the audio file. An audio object wraps an audio file in the file system to allow reading data and getting information such as the title of the music. The backend object is responsible for loading an audio file, decoding the data, and sending to the hardware device. The frontend object, which is a graphical user interface (GUI), keeps a playlist and has several buttons such as play and next. In the music player we might have a frontend object named musicPlayer, a

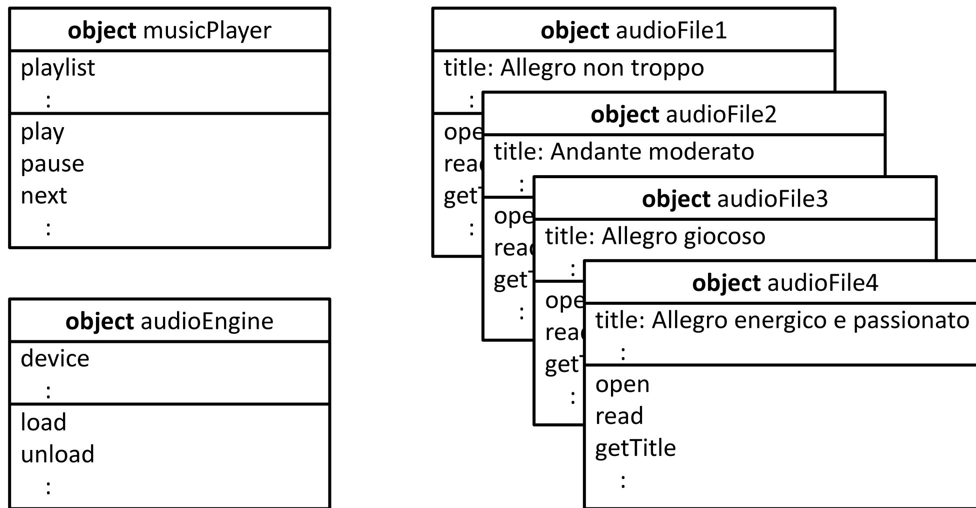

Figure 2.1: The objects in the music player example

Figure 2.2: The model of the music player example

backend object named audioEngine, and a lot of audioFile objects as shown in Figure 2.1. The model is shown in Figure 2.2.

When the user clicks on the play button on the musicPlayer object, the corresponding method named play will be called. In the play method the musicPlayer object sends a message load to the audioEngine object to ask it loading an audio file. Then the audioEngine object sends messages open and read to the specified audioFile objects in sequence for getting the data. Everything in the model is an object, and every functionality of the program is represented by the interactions between objects.

In OOP an object does not and should not know the implementation of other objects; the encapsulation is important. Every object encapsulates its details and exposes only the information about how to use it; how it does for a message is hidden from its client. In the music player example, the musicPlayer object does not know how the audioEngine object controls the hardware device, and the audioEngine object does not know how the data are arranged in the audioFile object. This makes the responsibility of every object clear and thus easy to implement. Furthermore, replacing or improving an

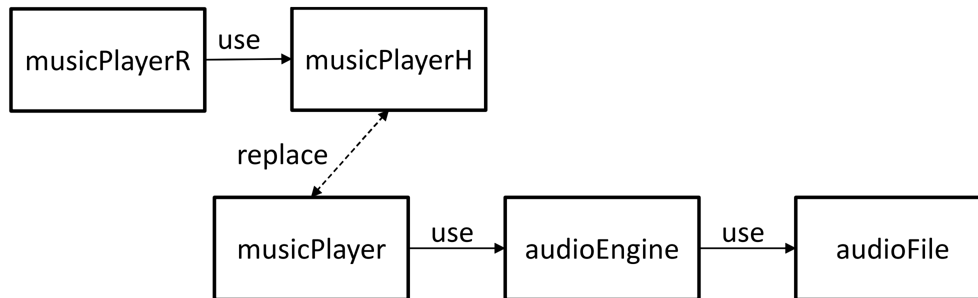

Figure 2.3: The musicPlayer object can be replaced with the musicPlayerH object

object is also easier. For example, if the hardware device is changed, we can replace the audioEngine object with a new one without modifying the musicPlayer object and the audioFile objects. If we need a remote client for the music player, the musicPlayer object can be replaced with a modified one named musicPlayerH and a new object musicPlayerR can be added to the model as shown in 2.3; the audioEngine object and audioFile objects are unaware of this change.

The dynamic method dispatch is also an important concept in OOP. A message sent to an object, in other words the call to a method on an object, can be dispatched properly according to the object's type—the class type in class-based OOP or the prototype in prototype-based OOP. For example, if we want to extend the music player discussed above to a media player which can play not only audio files but also video files, we can modify the model in Figure 2.2 to Figure 2.4. Here the mediaPlayer is an extended version of musicPlayer, which has an additional window for showing the video besides the buttons provided by musicPlayer such as the player button. The mediaEngine object can be either the audioEngine in Figure 2.2 or a new object videoEngine which accepts the same messages as the ones accepted by audioEngine, for example load. The videoEngine object can load a video file when it receives the message load. To the mediaPlayer it only knows the receiver is an object which is named mediaEngine and accepts the message load. When the program is executed and the user clicks on the play button, the message load sent by mediaPlayer will be dispatched according to the type of the receiver dynamically. If the receiver is the audioEngine object, the method implemented in audioEngine will be executed; if the receiver is the videoEngine object, the method implemented in videoEngine will be executed.

The features of OOP such as encapsulation and dynamic method dispatch greatly improve the modularity of programs, and make it easier to implement

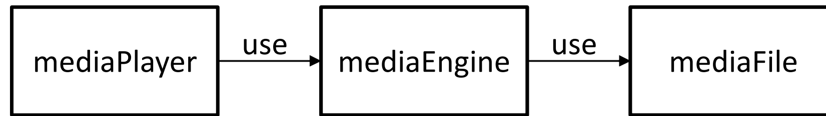

Figure 2.4: The model of the media player modified from the music player example

and maintain a large-scale application. With the encapsulation objects can be unaware of the implementation inside each other; every object focuses on its own functionality. The dynamic method dispatch allows to delay the selection of the object implementation for a message; a message can be flexibly sent to the appropriate object implementation at runtime. By using OOP the code can be better modularized. It is believed that SIMULA [18, 17] is the first language with the object design. The concept of OOP was going mature in the 1970s, and the famous Smalltalk-80 [33] is regarded as the most elegant OO language.

Class-based OOP In Smalltalk classes are declared to create object instances at runtime. All object instances of a class share the same implementation; they are the same kind of components in the program. However, each object owns its memory space for holding its states. The class is a template for creating object instances. Furthermore, a class is allowed to extend another class for inheriting their implementation and overriding methods. For the media player example in Figure 2.4, we can declare a class named MediaEngine, which is inherited by the classes AudioEngine and VideoEngine as shown in Figure 2.5. Both AudioEngine and VideoEngine are kinds of MediaEngine; such is-a relation is called the polymorphism, which properly handles the dynamic method dispatch according to the inheritance. Listing 2.1 is the sample code in Java for the three classes MediaEngine, AudioEngine, and VideoEngine. Note that VideoEngine overrides load but does not override unload, while AudioEngine does not override any of them. When the methods load and unload are called as follows:

```
MediaEngine engine = new AudioEngine();
  :
engine.load();
  :
engine.unload();
```

The proper implementation for the load method will be executed according to the actual type of the engine object. Here engine is an AudioEngine object,

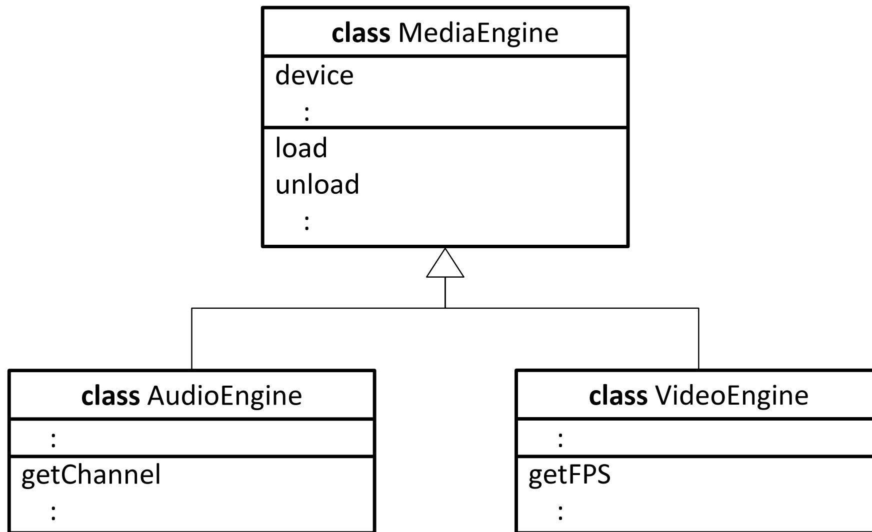

Figure 2.5: The class hierarchy for mediaEngine of the media player example

so the following line will be printed:

```
MediaEngine: loading
MediaEngine: unloading
```

If engine is a VideoEngine as follows:

```
MediaEngine engine = new VideoEngine();
  :
engine.load();
  :
engine.unload();
```

it prints:

```
VideoEngine: loading
MediaEngine: unloading
```

If the subclass overrides the method, the implementation in the overriding method is executed; otherwise the original one is executed. Such style of OO design is called class-based OOP, which is followed by C++ [79], Objective-C [2], and Java [64]. Classes, fields, and methods are necessary language constructs for supporting class-based OOP.

```java
public class MediaEngine {
  public void load() {
    System.out.println("MediaEngine: loading");
      :
  }
  public void unload() {
    System.out.println("MediaEngine: unloading");
      :
  }
    :
}

public class AudioEngine extends MediaEngine {
    :
}

public class VideoEngine extends MediaEngine {
  public void load() {
    System.out.println("VideoEngine: loading");
      :
  }
    :
}
```

Listing 2.1: MediaEngine and its subclasses in Java

Prototype-based OOP On the other hand, prototype-based OOP is introduced by Self [84], which is influenced by Smalltalk but simplifies the design by integrating the classes and objects in Smalltalk. An object can be generated by duplicating from prototype objects. In other words, using an object as the template of another object. This style of OO design is followed by JavaScript. Listing 2.2 shows the sample code in JavaScript for the three classes discussed above. Line 16 and Line 25 specify the prototypes for AudioEngine and VideoEngine, respectively. If the method called by its clients is not implemented in the object, the method call is delegated to its prototype object. For example, calling load on an AudioEngine object will call the implementation in MediaEngine and print:

```
MediaEngine: loading
```

The result is the same as the Java version showed above. Note that in the Java version we have classes and objects, while in the JavsScript version we have only objects. The dynamic method dispatch is supported by the delegation rather than the inheritance.

OO languages has become one of the mainstream languages for more than twenty years. Several of them also support other paradigms based on OOP

```
1  function MediaEngine() {
2    this.load = function() {
3      print("MediaEngine: loading");
4        :
5    }
6    this.unload = function() {
7      print("MediaEngine: unloading");
8        :
9    }
10     :
11 }
12
13 function AudioEngine() {
14     :
15 }
16 AudioEngine.prototype = new MediaEngine();
17
18 function VideoEngine() {
19   this.load = function() {
20     print("VideoEngine: loading");
21       :
22   }
23     :
24 }
25 VideoEngine.prototype = new MediaEngine();
```

Listing 2.2: MediaEngine and its subclasses in JavaScript

to further improve the modularity of code. For example, C# [54] includes the support of the event-handler paradigm, and AspectJ [77] adds the support of the aspect paradigm to Java. OOP are widely used in either business software or open source projects, especially to implement libraries.

OOP can be directly supported by language constructs as the OO languages mentioned above, but it is also possible to be supported by design patterns. Even with non-OO languages such as C some libraries like GTK+ [81] are also implemented in OOP. However, it is expected that every program has to implement the pattern code.

2.1.2 The event-handler paradigm

Now the event-handler paradigm (event-driven programming) is usually used along with OOP, though it is independent of OOP at the beginning. The event-handler paradigm can be used to loosen the coupling between two components in a program. It lets both sides of components to focus on the task that should be handled rather than who the creater is and who the handler is. At first the event-handler paradigm was used alone to replace the polling style of programming. Its concept is similar to the interrupts in hardware and thus massively adopted by system libraries such as the socket library on Linux, Xlib [85] for X Window system, and Microsoft Windows API [60]. After OOP was getting popular, people found that the event-handler paradigm can make the relation between two objects more flexible in OOP. The Observer pattern can be regarded as a standard of how to use the event-handler paradigm in OOP. Today most OO libraries heavily rely on the event-handler paradigm. Almost all GUI libraries, for example AWT [65], Cocoa [1], MFC [58], Swing [67], SWT [80], and Qt [21], are implemented by the event-handler paradigm. In a system such as Android the event-handler paradigm is also massively used; an Android application might be regarded as a composition of handlers, which reacts to various events in the system.

Recently several research activities such as EScala [32] and EventCJ [40] propose a new language construct named events for using the event-handler paradigm in OOP. Events are also supported by C# [54]. An event is a kind of field, which marks the happening of something and allows methods to be bound as handlers. When the event is triggered either explicitly or implicitly, the handlers will be executed. In EScala `evt` can be used to declare events and the operator += can be used to bind a method to an event. Taking the media player in Figure 2.4 as an example, if the frontend `mediaPlayer` needs to switch the icon of the play button according to the status of the backend `mediaEngine`: showing the pause icon when it is playing and showing

```
1  public class MediaPlayer {
2    public void onPlay() {
3      System.out.println("it is playing.");
4        :  // switch to the pause icon
5    }
6    public void onPause() {
7      System.out.println("it is paused.");
8        :  // switch to the play icon
9    }
10     :
11 }
12
13 public class MediaEngine {
14   MediaPlayer mediaPlayer = null;
15   public void setPlayer(MediaPlayer player) {
16     mediaPlayer = player;
17   }
18   public void load() {
19       :
20     mediaPlayer.onPlay();
21   }
22   public void unload() {
23       :
24     mediaPlayer.onPause();
25   }
26     :
27 }
```

Listing 2.3: Switching the icon of the play button without events

play icon when it is paused. We can implement such a switching by Java as shown in Listing 2.3. Note that here the Observer pattern is not used. When the mediaEngine object finish loading the media file, it calls the onPlay method on the mediaPlayer object for switching to the pause icon. Similarly, onPause is called for switching to the play icon when the media file is finished unloading. It works, though events are not used.

However, there are several drawbacks in this sample code. First, MediaPlayer is explicitly specified in MediaEngine. The one to be notified of the switching must be an object instance of MediaPlayer or its subclasses. Second, we hard code the two methods in MediaEngine, although MediaEngine does not have to know the methods onPlay and onPause. It is not reasonable since MediaEngine also needs to be modified when the names of the two methods in MediaPlayer are modified. MediaEngine should be unaware of the two methods. On the other hand, it is not reasonable that MediaPlayer need to know the methods onPlay and onPause will be called for getting the notifications after setPlayer, either.

We can use the events in EScala to describe such a switching as shown

in Listing 2.4. Line 22–23 declare two events loaded and unloaded, which are triggered after the execution of load and unload, respectively. Similarly, we can describe another switching: if an mediaFile is removed, disabling the corresponding item in the playlist of mediaPlayer. Line 34–37 define that removed and restored are triggered for remove and restore, respectively. Then we can bind the methods to the events as follows:

```
val mediaPlayer = new MediaPlayer()
val mediaEngine = new MediaEngine()
val mediaFile   = new MediaFile("track01")
  :
mediaEngine.loaded   += mediaPlayer.onPlay
mediaEngine.unloaded += mediaPlayer.onPause
mediaFile.removed    += mediaPlayer.disablePlaylistItem
mediaFile.restored   += mediaPlayer.enablePlaylistItem
```

Using events can effectively loosen the coupling between objects since the object triggering the event can be unaware of the owner of the handler. For example, after the method load on the object mediaEngine is executed, the event loaded will be triggered and the handler method onPlay on the object mediaPlayer will be called. In the class MediaEngine we can avoid explicitly specifying the handler: the method onPlay on the object mediaPlayer. It is more flexible.

2.1.3 The aspect paradigm

The aspect paradigm (aspect-oriented programming) is developed for modularizing what OOP cannot modularize: crosscutting concerns. With OOP programmers can modularize different concerns by classes, but cannot modularize those concerns that crosscut classes. Here we take the media player in Listing 2.8 as an example again to explain this issue. Suppose that we want to debug the program and give a profiling of the media player. In order to know how much time each method takes we add a log message at the beginning and the end of each method as shown in Listing 2.5. When the method play in MethodPlayer is executed, the output looks like:

```
[1390566365503] MediaPlayer: the beginning of play track01
[1390566365603] MediaEngine: the beginning of load track01
[1390566365903] MediaEngine: the end of load track01
[1390566366903] MediaPlayer: the end of play track01
```

```
1  class MediaPlayer() {
2    def onPlay() {
3      System.out.println("it is playing.");
4        :  // switch to the pause icon
5    }
6    def onPause() {
7      System.out.println("it is paused.");
8        :  // switch to the play icon
9    }
10   def enablePlaylistItem(filename: String) {
11     System.out.println("enable the item in playlist: " + filename);
12       :  // enable the list item
13   }
14   def disablePlaylistItem(filename: String) {
15     System.out.println("disable the item in playlist: " + filename);
16       :  // disable the list item
17   }
18     :
19 }
20
21 class MediaEngine() {
22   evt loaded[Unit] = afterExec(load)
23   evt unloaded[Unit] = afterExec(unload)
24   def load() {
25       :
26   }
27   def unload() {
28       :
29   }
30     :
31 }
32
33 class MediaFile(filename: String) {
34   evt removed[String] = afterExec(remove)
35                         map ((_: (Unit, Unit)) => filename)
36   evt restored[String] = afterExec(restore)
37                         map ((_: (Unit, Unit)) => filename)
38   def remove() {
39       :
40   }
41   def restore() {
42       :
43   }
44     :
45 }
```

Listing 2.4: Switching the icon of the play button using events in EScala

```
1  public class MediaPlayer {
2    private MediaEngine mediaEngine = new MediaEngine();
3    public void play(String filename) {
4      System.out.println("[" + System.currentTimeMillis() + "]"
5                         + " MediaPlayer: the beginning of play "
6                         + filename);
7        :
8      mediaEngine.load(filename);
9        :
10     System.out.println("[" + System.currentTimeMillis() + "]"
11                        + " MediaPlayer: the end of play "
12                        + filename);
13   }
14     :
15 }
16
17 public class MediaEngine {
18   public void load(String filename) {
19     System.out.println("[" + System.currentTimeMillis() + "]"
20                        + " MediaEngine: the beginning of load "
21                        + filename);
22       :
23     System.out.println("[" + System.currentTimeMillis() + "]"
24                        + " MediaEngine: the end of load "
25                        + filename);
26   }
27     :
28 }
```

Listing 2.5: Printing log messages for each method

The sample code in Listing 2.5 works, but is ugly and hard to manage. The code for printing log messages scatters everywhere. Programmers might create a Logger class and try to move the log message to the Logger class as shown in Listing 2.6. The latter version looks better since now we can control the log messages in a place. For example, changing the format of log messages or managing them by logging levels. However, this issue is not really resolved. Indeed we have collected related methods into the Logger class, but when to call the methods and how to call the methods are left in other classes. Calling the debug method of Logger class still have to be explicitly stated in the methods of MediaPlayer class and MediaEngine class (Line 4, 8, 15, and 17 of Listing 2.6). Such a crosscutting concern, Logger, crosscuts other classes and cannot be modularized by only OOP.

It has been more than a decade since the issue was raised [43], and the aspect paradigm has been adopted by enterprise software frameworks such as VMware Spring [86] and Red Hat JBoss AS [71]. In the media player example there are two issues. First, code scattering: the code for the logging concern

```
public class MediaPlayer {
  private MediaEngine mediaEngine = new MediaEngine();
  public void play(String filename) {
    Logger.debug("MediaPlayer: the beginning of play " + filename);
      :
    mediaEngine.load(filename);
      :
    Logger.debug("MediaPlayer: the end of play " + filename);
  }
    :
}

public class MediaEngine {
  public void load(String filename) {
    Logger.debug("MediaEngine: the beginning of load " + filename);
      :
    Logger.debug("MediaEngine: the end of load " + filename);
  }
    :
}

public class Logger {
  public static void debug(String str) {
      System.out.println("[" + System.currentTimeMillis() + "] " + str);
  }
    :
}
```

Listing 2.6: Try to modularize the code for the logging concern

is scattered in several classes. The calls to the methods in Logger class such as debug are stated in a lot of places out of Logger class. It is hard to check the meaning and the correctness of Logger. When to call the methods in Logger class is not described in one place with a clear statement. Second, code tangling: the code for the Logger concern is tangled with other concerns. Although in this case we know that the first statement and the last statement in a method are for the Logger concern, we might have other concerns that adds statements at the beginning or any other places of a method. The code for different concerns are tangled. This increases the overheads of verifying one of the two concerns.

The aspect paradigm makes it possible to modularize crosscutting concerns. In a language supporting the aspect paradigm such as AspectJ [42], the code of debug can be implemented in a language construct named advices. How to attach the advice to methods can be defined by another language construct named pointcuts. Then advices and pointcuts can be modularized in another language construct, aspects. The media player example can be rewritten by AspectJ as shown in Listing 2.7. The calls to the debug method of Logger are totally removed from MediaPlayer and MediaEngine. Instead, we add an aspect named Logger, which contains a pointcut named allNonStaticMethods, a before advice, an after advice, and a method. The allNonStaticMethods pointcut specifies the timing: either when any non-static methods in MediaPlayer is executed or when any non-static methods in MediaEngine is executed. Then the two advices are attached to the pointcut allNonStaticMethods for being called before/after the timing specified by allNonStaticMethods. Note that thisJoinPoint is a special variable provided by AspectJ for using the reflection. In the two advices thisJoinPoint and a string are passed to the same method for printing the log messages.

With the aspect paradigm the media player example can be elegantly separated into three classes and an aspect. At the compile-time or load-time the code is woven according to the pointcuts and let the program work as the original one (Listing 2.6). With the Logger aspect all the log messages are printed. If we remove the Logger aspect, all the log messages will be removed. The aspect paradigm greatly increases the modularity of a program and makes it easier to read and maintain.

The aspect paradigm also influences other programming paradigms such as feature-oriented programming [69, 41] and context-oriented programming [12, 37, 72]. Maybe we could say that feature or context is some sort of "aspect", which cannot be modularized by only OOP.

```
1  public class MediaPlayer {
2    private MediaEngine mediaEngine = new MediaEngine();
3    public void play(String filename) {
4        :
5      mediaEngine.load(filename);
6        :
7    }
8      :
9  }
10
11 public class MediaEngine {
12   public void load(String filename) {
13       :
14   }
15     :
16 }
17
18 public aspect Logger {
19   pointcut allNonStaticMethods()
20     : execution(!static * MediaPlayer.*(..))
21       || execution(!static * MediaEngine.*(..));
22   before() : allNonStaticMethods() {
23     debug(thisJoinPoint, "the beginning of");
24   }
25   after() : allNonStaticMethods() {
26     debug(thisJoinPoint, "the end of");
27   }
28   public void debug(JoinPoint joinPoint, String str) {
29     String className =
30       joinPoint.getSourceLocation().getWithinType().getName();
31     String methodName = joinPoint.getSignature().getName();
32     String output = "[" + System.currentTimeMillis() + "] "
33                     + className + ": " + str + " " + methodName;
34     Object[] objs = joinPoint.getArgs();
35     for(int i=0; i<objs.length; i++) {
36         output += " " + objs[i];
37     }
38     System.out.println(output);
39   }
40 }
```

Listing 2.7: Modularizing the logging concern to an aspect

2.1.4 The reactive paradigm

Recently the reactive paradigm (functional-reactive programming) also attracts attentions from OOP community. The reactive paradigm is originally based on functional programming and proposed for building richly interactive programs. In the reactive paradigm the time model is quite different from the traditional programming style. Variables are not no longer constant values but time-varying values. Such variables are called behaviors, which are also named signals in several languages. An expression involving behaviors in the reactive paradigm is similar to the expression assigned to a cell in a spreadsheet program: the expression will be evaluated again when anything affects its result is changed. For example, if we assign an expression "B1 + C1" to the cell A1 in the spreadsheet program, whenever the value of B1 or C1 is changed, A1 will be updated automatically. In other words, the expression "A1 = B1 + C1" can always be regarded true. Another concept in the reactive paradigm is the event stream, which is an infinite stream of discrete events in the time line. An event in an event stream marks something happens in the time line and can help to get the constant value of a behavior at that time, which is called a snapshot of the behavior. Fran [24] is regarded as the first publication on programming with the reactive paradigm.

The behaviors in the reactive paradigms can simply express data flows. This might remind readers of data-flow programming [87, 9, 6, 4], which have a long history. Their concepts are similar, but data-flow programming are more close to hardware design since the time is totally hidden from programmers. In the reactive paradigm the time is an explicit factor, although its value at a specified time is not explicitly used. Recently the reactive paradigm attracts attentions again since the need for interactive programs is increaing. For example, the GUI applications for mobile devices or Web applications.

At the beginning the reactive paradigm was proposed to be used with functional programming since its features can be smoothly represented by functional programming languages. Several languages such as Elm [15] belong to the traditional style. However, in the recent years several research activities are devoted to use the reactive paradigm with OOP since OOP has become one of the mainstream languages as mentioned above. Massive OO libraries and large-scale OO applications make researchers notice the integration of the reactive paradigm and OOP. Several of the research activities propose a library providing behaviors and event streams for OOP, and several of them propose new language constructs in OO languages. How to use the reactive paradigm in an OO language is still a challenge.

2.2 Supporting the implementation of paradigms

How the implementation of programming paradigms is supported in a language has been known as an important issue since programming paradigms are so important to programmers. Generally speaking, programming paradigms can be implemented by design patterns, dedicated constructs, or generic constructs. This section explains what the three approaches are. Then we analyze their advantages and disadvantages of using them to support multiple paradigms in a single language. The conclusion on their advantages and disadvantages led us to challenge the approach of generic constructs.

2.2.1 Design patterns

The language constructs for OOP can fit the needs of a lot of program design but not all. In 1994 the famous "GoF" design patterns book [30] was published, which shows 23 classic design patterns for resolving various design issues. This book demonstrates how to write pattern code for resolving those issues with the language constructs provided by a typical OO language such as C++ and Smalltalk. In other words, the pattern code utilizes the capability of OOP with existing language constructs.

However, using design patterns to implement a programming paradigm might be a good solution, but not the best solution. It enables the possibility of using a paradigm that is not directly supported by the language, but programmers have to write the pattern code repeatedly. For example, the Observer pattern loosens the coupling of subject and observer. In a subject object a list is maintained for remembering its observer objects, and then other objects can be registered as the observer by asking the subject object to add them to the list. When something the observer objects concern happens, the subject object will notify all the observer objects in the list. The Observer pattern can be used to implement the event-handler paradigm without any dedicated constructs. In other words, the pattern code consists only the constructs for OOP; no construct for events is used. For example, the switching icons example discussed above can be described by the Observer pattern as shown in Listing 2.8. Implementing the Observer pattern can help to loosen the coupling of MediaPlayer and MediaEngine. The MediaEngine object is the subject and the MediaPlayer object is the observer. First we declare two abstract classes EngineSubject and EngineObserver for the subject and the observer, respectively. In EngineSubject a list named observers is declared for holding the observers, and an observer can add itself to the list by calling addObserver or remove itself from the list by calling removeObserver. Then in

a subclass of EngineSubject such as MediaEngine, the methods notifyLoaded and notifyUnloaded can be used to notify its observers. On the other hand, in EngineObserver two methods updateLoaded and updateUnloaded are defined in order to ask its subclasses to implement how to handle them. Using the Observer pattern the relation between the subject and the observer is more flexible: any class extending EngineSubject can be the subject and any class extending EngineObserver can be the observer. They are not limited to MediaPlayer and MediaEngine. Furthermore, the subject just notifies the observers to update and is unaware of how the observers update their status. Similarly, the observer can add itself to any one that agrees the protocol. The coupling between the subject and the observer is loose and flexible.

Unlike the EScala version in Listing 2.4, this code does not use any additional construct except the constructs for OOP. Using the Observer pattern there is no need to hard code the reference to the observers in the subject and the registration can be done in advance. Unlike the polymorphism such interactions cannot be described by using only the constructs for OOP. In an class-based OO language programmers can simply write an overriding method in the subclass to desctibe the dynamic method dispatch; no additional code is necessary. However, the design issue resolved by the Observer pattern, the break of encapsulation caused by tight-coupling, cannot be described without additional code. Programmers need to implement the code of observer management and notification. Design patterns show how to write pattern code using existing language constructs to resolve specific design issues.

Using the Observer pattern cannot satisfy library developers. Inside the library developers have to write the pattern code again and again for different events. For example, in Listing 2.8 besides the orignal implementation we have to write the code of the Observer pattern such as the code in EngineSubject and EngineObserver. As discussed above these code are used to let other objects register themselves as the observers and thus their methods such as updateLoaded and updateUnloaded can be called as handlers for the events such as notifyLoaded and notifyUnloaded, respectively. The pattern code is mixed with other concerns. If there are two categories of events and we need to prepare two lists for separately managing the observers, the pattern code has to be implemented twice. For example, we want to show the status of media files in the playlist of the frontend MediaPlayer: if the media file is removed from the file system, the item for the media file in the playlist should be grayed out. For the new functionality we need to implement the Observer pattern again as shown in Listing 2.9, where MediaFile extends FileSubject and MediaPlayer implements FileObserver. Note that FileObserver has to be defined by an interface rather than an abstract class since multiple inheri-

```
1  public abstract class EngineSubject {
2    ArrayList<EngineObserver> observers = new ArrayList<EngineObserver>();
3    public void addObserver(EngineObserver o) { observers.add(o); }
4    public void removeObserver(EngineObserver o) { observers.remove(o); }
5    public void notifyLoaded() {
6      Iterator<EngineObserver> iter = observers.iterator();
7      while(iter.hasNext())  iter.next().updateLoaded();
8    }
9    public void notifyUnloaded() {
10     Iterator<EngineObserver> iter = observers.iterator();
11     while(iter.hasNext())  iter.next().updateUnloaded();
12   }
13 }
14
15 public abstract class EngineObserver {
16   public abstract void updateLoaded();
17   public abstract void updateUnloaded();
18 }
19
20 public class MediaEngine extends EngineSubject {
21   public void load() {
22       :
23     notifyLoaded();
24   }
25   public void unload() {
26       :
27     notifyUnloaded();
28   }
29     :
30 }
31
32 public class MediaPlayer extends EngineObserver {
33   public void updateLoaded() { onPlay(); }
34   public void updateUnloaded() { onPause(); }
35   public void onPlay() {
36     System.out.println("it is playing.");
37       :  // switch to the pause icon
38   }
39   public void onPause() {
40     System.out.println("it is paused.");
41       :  // switch to the play icon
42   }
43     :
44 }
```

Listing 2.8: Switching the icon of the play button using the Observer pattern

```java
public abstract class FileSubject {
  ArrayList<FileObserver> observers = new ArrayList<FileObserver>();
  public void addObserver(FileObserver o) { observers.add(o); }
  public void removeObserver(FileObserver o) { observers.remove(o); }
  public void notifyEnable(String filename) {
    Iterator<FileObserver> iter = observers.iterator();
    while(iter.hasNext())  iter.next().updateEnable(filename);
  }
  public void notifyDisable(String filename) {
    Iterator<FileObserver> iter = observers.iterator();
    while(iter.hasNext())  iter.next().updateDisable(filename);
  }
}
public interface FileObserver {
  public void updateEnable(String filename);
  public void updateDisable(String filename);
}
public class MediaFile extends FileSubject {
  String filename;
    :
  public void remove() {
      :
    notifyDisable(filename);
  }
  public void restore() {
      :
    notifyEnable(filename);
  }
}
public class MediaPlayer extends EngineObserver implements FileObserver {
  public void updateLoaded() { onPlay(); }
  public void updateUnloaded() { onPause(); }
  public void updateEnable(String filename) {
    enablePlaylistItem(filename);
  }
  public void updateDisable(String filename) {
    disablePlaylistItem(filename);
  }
  public void onPlay() {
    System.out.println("it is playing.");
      :  // switch to the pause icon
  }
  public void onPause() {
    System.out.println("it is paused.");
      :  // switch to the play icon
  }
  public void enablePlaylistItem(String filename) {
    System.out.println("enable the item in playlist: " + filename);
      :  // enable the list item
  }
  public void disablePlaylistItem(String filename) {
    System.out.println("disable the item in playlist: " + filename);
      :  // disable the list item
  }
}
```

Listing 2.9: MediaPlayer is the observer of both EngineSubject and FileSubject

tance is not allowed in Java. The code for the two implementations of the Observer pattern are mixed in MediaPlayer. Using super classes and interfaces is helpful to gather up the code for the same concern, but does not totally resolve code tangling. Programmers have to implement the Observer pattern repeatedly and mix the code for different concerns. Several libraries such as Qt and Boost provide the support of the event-handler paradigm, which is also known as signals and slots, to generate pattern code for programmers.

2.2.2 Dedicated constructs and the unification

Dedicated constructs help programmers to implement programming paradigms with language support, but increase the number of constructs. As mentioned in Section 2.1, a lot of research activities propose dedicated constructs for different paradigms: classes, fields, and methods for OOP; events and methods for the event-handler paradigm; aspects, pointcuts, and advices for the aspect paradigm; behaviors and event streams for the reactive paradigm. These dedicated constructs are proposed either to avoid implementing pattern code again and again, for example events, or make the paradigm available, for example pointcuts. For those paradigms that can be implemented by design patterns, supporting them by dedicated constructs makes the code clear. By comparing the switching icon example implemented by events (Listing 2.4) with the one implemented by the Observer pattern (Listing 2.8 and Listing 2.9), we can know the pattern code for managing and notifying the observers is eliminated. EngineSubject, EngineObserver, FileSubject, and FileObserver are no longer necessary. Some readers might notice that the explicit trigger such as Line 23 and Line 27 in Listing 2.8 are also removed. EScala allows events to be implicitly triggered using afterExec rather than explicitly triggered at the end of method bodies. It also greatly reduces the code scattering caused by event triggers. The source code is more clear and easy to maintain. On the other hand, several paradigms such as the aspect paradigm are only available by dedicated constructs; they cannot be implemented by design patterns. The log message example (Listing 2.5 and Listing 2.6) explains why it is not possible to implement the aspect paradigm by the constructs for OOP with design patterns. As shown in Listing 2.7, AspectJ introduces the dedicated constructs, aspects, pointcuts, and advices, to support the implementation of the aspect paradigm. Without dedicated constructs, such paradigms cannot be implemented.

Although dedicated constructs are ideal for using a specific paradigm, they are not good to a language that supports multiple paradigms. For using multiple paradigms in a language, all sets of dedicated constructs for

the paradigms must be unified into a super set; this make the language design complicated. There are several research activities on the unification of the dedicated constructs for two of the paradigms mentioned above. In this thesis the term unification refers to putting all sets of constructs for different paradigms into a super set. The reason why the constructs are unificated rather than integrated is that the constructs are dedicated to specific paradigms. Most proposals for the aspect paradigm can be regarded as the unification of the constructs for the aspect paradigm and OOP since the issue resolved by the aspect paradigm was raised in OOP at the beginning. For example, AspectJ [42] and CaesarJ [3] give extra dedicated constructs besides the constructs for OOP; two sets of constructs are provided. Similarly, the proposals for the event-handler paradigm such as Ptolemy [70], EventJava [27], EventCJ [40], and EScala [32] unify the constructs for OOP and the event-handler paradigm. There are also several unification work on the constructs for the reactive paradigm and OOP such as Frappé [13], SuperGlue [52], Flapjax [53], Scala.React [49], and the discussion in [74]. For using the reactive paradigm in an OO language, they give dedicated libraries for supporting special types of classes. Since those constructs are dedicated to specific paradigms, all of them must be given in a language that supports multiple paradigms.

2.2.3 Generic constructs and the integration

Generic constructs are the constructs that can be used for differnt paradigms, and in this thesis the term integration refers to using the same constructs for multiple paradigms. With generic constructs the total number of the constructs in a language that supports multiple paradigms can be reduced; this simplifies the language design. As a result of introducing more and more programming paradigms, the number of language constructs is increasing. Unfortunately, it is a trend since supporting programming paradigms by language constructs is better than by design patterns. Every research activity proposes its dedicated language constructs based on a specific programming language for a certain paradigm. This means that programmers have to choose between programming languages according to the programming paradigm they want to use at the design stage. Programmers have to carefully consider that which programming paradigm they want to use for the program in order to decide the programming language to implement. Another choice is that checking whether the paradigm provided in their favorite programming languages is suitable for the program or not. After the program is implemented, it might be necessary to change the programming

language in order to change the paradigm used in the program; it is not an easy job. Furthermore, to simply the design of compiler and debugger the number of language constructs should be as small as possible. First, from the viewpoint of code analysis supporting more language constructs needs to consider more combinations of language constructs. The number of the combinations steeply increases with the number of language constructs. Second, supporting less language constructs helps the code optimization. For example, an idea for optimizating a language construct can be applied to more code pieces. A complicated language design results in a complicated design of compiler and debugger, and the number of language constructs is one of the important factors in language design.

However, it is never easy to integrate the constructs for different paradigms into generic constructs. There are several research activities devoted to the integration of the constructs for different paradigms by giving generic constructs, but the number of paradigms is quite limited. For example, Delegation-based AOP [35, 75] and GluonJ [10] integrate the constructs or models for OOP and the aspect paradigm. ECaesarJ [62] and Context Aspects [38] are the contributions to the integration of the constructs for the event-handler paradigm and the aspect paradigm. It is true that these integration work clarify the essentials of different paradigms and use generic constructs to support the implementation of them, but how to increase the number of supported paradigms is still a challenge.

2.3 Comparing the three approaches

Nothing is perfect; all the three approaches have both advantages and disadvantages. The approach of design patterns is flexible, but asks programmers to implement pattern code repeatedly. Furthermore, several paradigms such as the aspect paradigm are not available by design patterns. The approach of dedicated constructs is very expressive but lacks flexibility. Dedicated constructs increase the readability since it is easy to understand which paradigm is used. However, it is not easy to migrate the source code to a different paradigm. A unification work on the constructs for different paradigms also complicates the language design. The approach of generic constructs reduces the total number of constructs in a language supporting multiple paradigms, but the design intention might not as clear as dedicated constructs. In a statement the generic construct is used for which paradigm might not be obvious. More importantly, how to make a construct generic is a concern.

This thesis challenges the approach of generic constructs since it is potentially the best approach as what we will discuss below. How to choose the

Table 2.1: Comparing the three approaches to implementing paradigms

	Design patterns	Dedicated constructs	Generic constructs
the modularity of code	*Fair*	*Excellent*	*Excellent*
the number of paradigms	*Good*	*Excellent*	*Good*
the number of constructs	*Excellent*	*Fair*	*Excellent*

approach to supporting the implementation of paradigms—especially when we want to let a language support as many paradigms as possible—can be considered from the three points of view: the modularity of code, the numebr of paradigms that can be supported, and the number of constructs that a language must provide. We can roughly evaluate the three approaches from the three viewpoints as shown in Table 2.1, where *Excellent* is given if it can be regarded as pretty good, *Good* is given if it can be regarded as good, and *Fair* means that it is not good enough. The table concludes the potential of the three approaches and shows that the best choice is the approach of generic constructs.

The first concern for choosing the approach to supporting the implementation of paradigms is the modularity of code. The modularity means that the code can be modularized according to their functionalities and properties. From this point of view, the approach of design patterns is not good. Even though all the pattern code are implemented for the same concern, they scatter around the program. On the other hand, the other two approaches are supported by the languages themselves. With language support the implementation of paradigms can be minimized. For example, using the language constructs provided for events can eliminate the code of implementing the Observer pattern.

The second concern is the number of paradigms that can be supported. It will be a good thing if a language can support a large number of paradigms. From this viewpoint the approach of design patterns seems a versatile approach since most paradigms can be implemented by design patterns. The GoF book demonstrates such techniques of implementing the paradigms that are not directly supported by OOP. However, not all paradigms can be implemented by design patterns, for example the aspect paradigm. On the other hand, although we have different sets of dedicated constructs for different paradigms, it is possible to give a super set that unifies all the dedicated

constructs for the paradigms we need. The unification work mentioned above show such a possibility. In other words, if we do not take the complexity of language design into account, the number of paradigms that can be supported by the approach of dedicated constructs is not limited. As to the approach of generic constructs, the question is how many paradigms can be supported by a set of generic constructs. In theory it is not a limitation, but in practice existing integration work, which gives generic constructs, support not more than two paradigms as far as we know.

The last concern we discuss here is the number of constructs that a language must provide. Usually we hope the number can be as small as possible to simplify the language design. To the approach of design patterns, the number will be quite small since the purpose of design patterns is supporting more paradigms by a limited number of constructs in a language. For the approach of dedicated constructs the number of constructs in a language always grows with the number of paradigms supported by the language. Unless the constructs are generic and can be shared among the paradigms, the total number of constructs equals to the sum of all the numbers of the dedicated constructs for the supported paradigms. On the other hand, the approach of generic constructs minimizes the number of constructs needed for the paradigms supported by a language.

According to Table 2.1 we have the conclusion that the best approach is supporting by generic constructs. First of all, we should support paradigms by language constructs to avoid code scattering and code tangling, so the approach of design patterns is not the best choice. Furthermore, not all paradigms can be implemented by design patterns. Then we can compare the remainder two approaches: by dedicated constructs and by generic constructs. To the two approaches, the last two concerns looks a kind of trade-off. If the constructs are dedicated, it is easier to unify but the number of constructs is larger; if the constructs are generic, it is difficult to integrate many paradigms but the number of constructs is smaller. However, how to extend generic constructs to to support as many paradigm as possible is a challenge rather than a limitation. If we take the approach of dedicated constructs, the total number of constructs must grow with the number of supported paradigms; it is a limitation. Moreover, although putting dedicated constructs together increases the power of a language, it does not help the simplification of language design. Programmers have to learn all the syntax of them and the compiler/debugger has to be designed for all of them. In the history of programming languages there are a lot of examples on the simplification of language design since a simpler language design also simplifies the design of its compiler and debugger. For example, in Smalltalk everything is an object including primitives. Self further integrates classes

and objects in Smalltalk into one thing. Another example is the memory allocation in Java, which simplifies the usage of stack and heap, and removes the dereference operator (`*`) and address-of operator (`&`) in C++. Similarly, a generic language construct that can be used for different paradigms can simplify the design of a language supporting multiple paradigms.

The goal of this thesis is supporting the implementations of OOP, the event-handler paradigm, the aspect paradigm, and the reactive paradigm by generic constructs. Since the approach of generic constructs has the best potential, in the next chapter, we further advance this approach and present there is a generic construct supporting more than two paradigms. We first explain our observation: the similarities among the implementations of OOP, the event-handler paradigm, and the aspect paradigm, which is also our motivation for developing a more flexible generic construct. As we have shown in the previous sections existing dedicated constructs for the three paradigms look so different that people rarely consider the common ground among them. Even though there are a very few research on the integration, no one is flexible enough to cover all the three paradigms due to the unclear recognition of the similarities among their implementations. The observation led us to develop a new generic construct that can be used for the three paradigms. The new generic construct is very flexible, and how to extend the new generic construct for more paradigms will be demonstrated later by taking the reactive paradigm as an example.

Chapter 3

Method Slots

Providing language constructs for the following three important and widely-used paradigms is a hot topic: OOP, the event-handler paradigm, and the aspect paradigm. However, language designers tend to redundantly give multiple sets of dedicated constructs to support them separately since these dedicated constructs look so different; they were regarded as totally different things. This chapter first clarifies the common ground among the implementations of the three paradigms, which is our motivation to develop a more flexible generic construct for the three paradigms. We then present how the new generic construct, *method slots*, can be used along with the assignment operators to support the three paradigms. The coverage of expressive ability of *method slots* is analyzed and evaluated as well.

3.1 Introduction

The event-handler paradigm has been recognized as a useful mechanism in a number of domains such as user interface, embedded systems, databases [91], and distributed programming. The basic idea of the event-handler paradigm is to register an action that is automatically executed when something happens. At first it was introduced as techniques and libraries [30, 82, 78] rather than supported at language level. Recently supporting it at language level is a trend since a technique such as the Observer pattern [30] cannot satisfy

programmers' need. The code for event triggers and observer management scatters everywhere. To address the issues, supporting events by a language construct is proposed in a number of languages [54, 19, 70, 27, 40, 32]. Implicit invocation languages [31] might be classified into this category.

On the other hand, the aspect paradigm [43] is proposed to resolve crosscutting concerns, which cannot be modularized by existing paradigms such as object orientation. Although the aspect paradigm and the event-handler paradigm are designed for different scenarios, the constructs introduced for them are similar and can work as each other from a certain point of view.

In order to simplify the language constructs programmers have to learn, we borrow the idea of slots from Self [84] to extend the method paradigm in Java. In Self, an object consists only of slots [76], which may contain either a value or a method. In other words, there is no difference between fields and methods since a method is also an object and thus can be kept in a field. We extend the slot and bring it to Java-like languages by proposing a new language construct named *method slots*. A *method slot* is an object's property that can keep more than one closure at the same time. We also present a Java-based language named *DominoJ*, where all methods in plain Java are replaced with *method slots*, to support both the event-handler paradigm and the aspect paradigm.

Our contributions presented in this chapter[1] are two-fold. First, we propose a new language construct, a *method slot*, to extend the method paradigm. Second, we introduce *method slots* to a Java-based language named *DominoJ*, and demonstrate how to use for the event-handler paradigm and the aspect paradigm.

3.2 Motivation

With the evolution of software, more and more programming paradigms are developed for various situations. During programmers' life, they are always learning new paradigms and thinking about which ones are most suitable for the job at hand. For example, the event-handler paradigm is widely adopted by GUI frameworks [85, 57, 65]. When we write GUI programs with modern GUI libraries, we usually have to write a number of handlers for different types of events. The AWT [65] of Java is a typical example. If we want to do something for mouse events occurring on a button, we have to prepare a mouse listener that contains handler methods for those

[1] This chapter is a further extension to the one we presented at Modularity:AOSD2013 and its journal version that to appear in TAOSD2014 (LNCS volume 8400).

mouse events, and register the listener to the specified button object. A GUI program can be regarded as a composite of visual components, events, and handlers. The visual components and handlers are main logic, and events are used for connecting them. Indeed we have been familiar with using the event-handler paradigm for GUI programs, but it is far from our first "hello world" program. We are told to carefully consider the total execution order when users' input is read. If the event-handler paradigm is used, we can focus on the reaction to users' input rather than the order of users' input. Whether the mouse is clicked first or not does not matter. Another example is the aspect paradigm. The aspect paradigm is developed to modularize crosscutting concerns such as logging, which cannot be modularized by using only object-oriented programming. With the aspect paradigm, crosscutting concerns can be gathered up in an aspect by advices. At the same time, programmers cannot check only one place for understanding the behavior of a method call since advices in other places are possibly woven together. It also takes effort to get familiar with the aspect paradigm since it is quite different from our other programming experience.

To use a paradigm, just learning its concept is not enough. After programmers got the idea of a paradigm, they still have to learn new language constructs for the paradigm. Some paradigms like the aspect paradigm are supported with dedicated language constructs since the beginning because they cannot be represented well by existing syntax. On the other hand, although other paradigms like the event-handler paradigm have been introduced at library level for a long time, there are still good reasons for re-introducing them with direct support at language level [54, 70, 32]. Maybe one reason is that events are complicated in particular when we are not users but designers of a library. Besides GUI libraries, the event-handler paradigm is also implemented in a number of libraries for several domains such as simple API for XML [83] and asynchronous socket programming. Some techniques such as the Observer pattern [30] used in those libraries cannot satisfy the needs of defining events and tend to cause code scattering and tangling. Supporting paradigms by language constructs is a trend since it makes code more clear and reusable. Furthermore, a language supported paradigm may have associated static checks.

However, learning language constructs for a paradigm is never easy, especially for powerful paradigms like the aspect paradigm. Moreover, the syntax is usually hard to share with other paradigms. Even though programmers got familiar with the language constructs for a paradigm, they still have to learn new ones for another paradigm from the beginning. Given that all language constructs we need can be put into a language together, they look too complex and redundant. How to pick up the best language to imple-

```
1  class Display() {
2    def refresh() {
3      System.out.println("display is refreshed.")
4    }
5  }
6  class Shape(d: Display) {
7    var left = 0; var top = 0
8    def setPosition(x: Int, y: Int) {
9      left = x; top = y
10   }
11   evt moved[Unit] = afterExec(setPosition)
12   moved += d.refresh
13 }
14 object Test {
15   def main(args: Array[String]) {
16     val d = new Display()
17     val s = new Shape(d)
18     s.setPosition(0, 0)
19   }
20 }
```

Listing 3.1: Defining a reaction in EScala

ment a program with all required paradigms is always a difficult issue. This motivates us to find out an easy, simple, and generic language construct supporting multiple paradigms.

If we look into the language constructs for the event-handler paradigm and the aspect paradigm, there is a notable similarity between them. Both of them introduce a way to define the effect of calling specified methods. The differences are where the reactions are and what the reactions are targeted at. Listing 3.1 is a piece of code in EScala[2] [32], which is a typical event mechanism, showing how to define a moved event for the setPosition method in the Shape class. Here we specify that refresh method on a Display object should be executed after setPosition method is executed. As shown in Listing 3.2, the reaction can also be represented in AspectJ [77], the most well-known aspect-oriented language.

By comparing the two pieces of code, we can find that pointcuts are close to events and advices can work as the += operator for handlers. They both refresh the display when the specified method is executed, but there is a significant difference between them. In EScala version, one Display object is mapped to one Shape object and the refresh action is performed within the Shape object. On the other hand, in AspectJ version there is only one Display object in the whole program and the refresh action is in UpdateDisplay, which

[2]The syntax follows the example in EScala 0.3 distribution.

```
1  public class Display {
2    public static void refresh() {
3      System.out.println("display is refreshed.");
4    }
5  }
6  public class Shape {
7    private int left = 0; private int top = 0;
8    public void setPosition(int x, int y) {
9      left = x; top = y;
10   }
11 }
12 public aspect UpdateDisplay {
13   after() returning:
14    execution(void Shape.setPosition(int, int)) {
15       Display.refresh();
16   }
17 }
18 public class Test {
19   public static void main(String[] args) {
20     Shape s = new Shape();
21     s.setPosition(0, 0);
22   }
23 }
```

Listing 3.2: Defining a reaction in AspectJ

is completely separated from Display and Shape. From the viewpoint of the event-handler paradigm, such behavior is an interaction between objects, so the reaction is defined inside the class and targeted at object instances; the encapsulation is preserved. From the viewpoint of the aspect paradigm, it is important to extract the reaction for the obliviousness since it is a different concern cutting across several classes. So the reactions are grouped into a separate construct and targeted at the class. Although the two paradigms are developed from different points of view, the language constructs used for them are quite similar. Furthermore, both the two paradigms depend on the most basic paradigm, the method paradigm, since both events and pointcuts cause the execution of a method-like construct. This observation led us to extend the method paradigm to support both the event-handler paradigm and the aspect paradigm. To a programmer, there are too many similar language constructs for different paradigms to learn, so we assume that the integration and simplification are always worth doing.

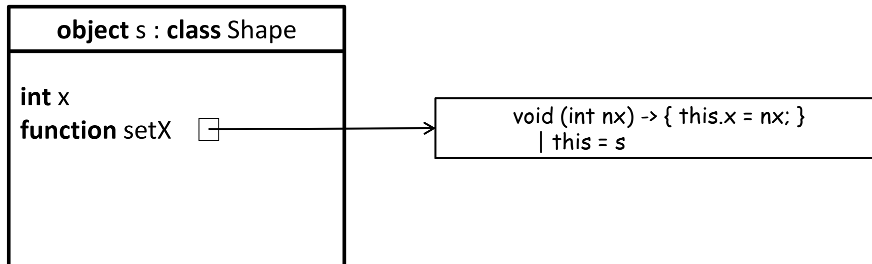

Figure 3.1: In JavaScript, both an integer and a function are fields on an object

3.3 DominoJ

We extend methods to a new language construct named *method slots*, to support methods, events, and advices. We also show our prototype language named *DominoJ*, which is a Java-based language supporting *method slots* and fully compatible with plain Java.

3.3.1 Method slots

Although methods and fields are different constructs in several languages such as C++ and Java, there is no difference between them in other languages like JavaScript. In JavaScript, a method on an object (strictly speaking, a function closure) is kept and used as other fields. Figure 3.1 shows a Shape object s, which has two fields: an integer field named x and a function field named setX. We use the following notation to represent a closure:
⟨*return type*⟩ (⟨*parameter list*⟩) -> { ⟨*statements*⟩ }
 | ⟨*variable binding list*⟩
where the variable binding list binds non-local variables in the closure. The value stored in field setX is a function closure whose return type and parameter type are void and (int), respectively. The variable this used in the closure is bound to s given by the execution context. When we query the field by s.setX, the function closure is returned. When we call the field by s.setX(10), the function closure is executed.

We extend this field in JavaScript to keep an array of function closures rather than just one function closure. As shown in Figure 3.2, the extended field named a *method slot* can keep more than one function closure. DominoJ replaces a method with a method slot in plain Java. All method-like declarations and calls are referred to method slots. A method slot is a closure array and is an object's property like a field. Like functions or other fields, method

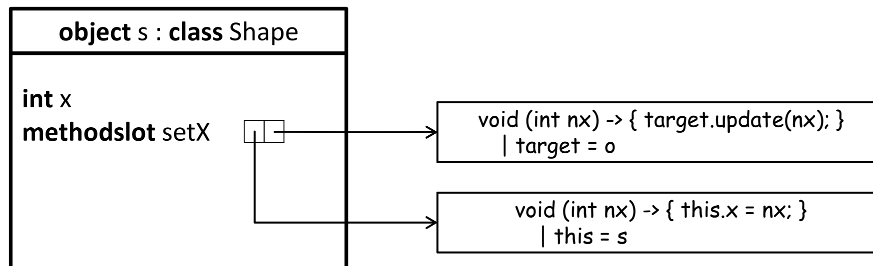

Figure 3.2: A method slot is an extended field that can keep more than one function closure

slots are typed and statically specified when they are declared. The type of method slot includes its return type and parameter types. All closures in it must be declared with the same type.

Listing 3.3 shows a piece of sample code in DominoJ. It looks like plain Java, but here setX is a method slot rather than a method. The syntax of method slot declaration is shown below:
⟨*modifier*⟩* ⟨*return type*⟩ ⟨*identifier*⟩ "(" ⟨*parameter list*⟩? ")" ⟨*throws*⟩?
(⟨*default closure*⟩ | ";")
The default closure is similar to the method body in Java except it is optional. The modifiers can be public, protected, or private for specifying the visibility of the method slot. This ensures that the access to the method slot can be controlled as the methods in plain Java. The modifier static can be specified as well. Such static method slots are kept on the class objects so can be referred using the class name like calling the static method in plain Java. The modifier abstract can also be used to specify that the method slot should be implemented by the subclasses. A method slot can be another kind of "abstract" by being declared without a default closure:

```
public void setX(int nx);
```

Unlike the modifier abstract, this declaration means that the method slot is an empty field and its behavior depends on the closures added to it later. In Listing 3.3, the method slot setX has a default closure, so the following function closure will be created and inserted into setX automatically when a Shape object, s, is instantiated:

```
void (int nx) -> { this.x = nx; }
  | this = s
```

```
1  public class Shape {
2    private int x;
3    public void setX(int nx) {
4      // default closure
5      this.x = nx;
6    }
7  }
8  public class Observer {
9    private int count;
10   public void update(int i) {
11     this.count++;
12   }
13   public static void main(String[] args) {
14     Shape s = new Shape();
15     Observer o = new Observer();
16     s.setX += o.update;
17     s.setX(10);
18   }
19 }
```

Listing 3.3: A sample code in DominoJ

Now there is only one closure in the method slot setX. If we add another closure to setX, the object may look like the s object in Figure 3.2. How to add such a closure to a method slot will be demonstrated in the next subsection.

A method slot can also be declared with the modifier final to specify that it cannot be overridden in the subclasses. Although fields are never overridden in either prototype-based languages like JavaScript or class-based languages like Java, method slots can be overridden in subclasses. Declaring a method slot with the same signature overrides but does not hide the one in the superclass. When a method slot is queried or called on an object, the overriding method slot is selected according to the actual type of the object. It is also possible to access the overridden method slot in the superclass through the keyword super. Note that method slots must be declared within a class and cannot be declared as local variables. Thus the usage of this and super in the default closure are the same as in a Java method, which refer to the owning class and its superclass, respectively. Constructors are method slots as well, and super() is allowed since it calls the overridden constructor.

When a method slot is called by () operator, the closures in it are executed in order. The arguments given to the method slot are also passed to its closures. The return value returned by the last closure is passed to the caller (if it is not the void type). A closure can use a keyword $retval to get the return value returned by the preceding closure in the method slot. If the closure is the first one in the method slot, $retval is given by a default value

```
1  ; call a methodslot
2  (define (call-methodslot object slotname args)
3    (let* ((methodslot (get-field object slotname (get-type args)))
4           (return_type (get-return-type methodslot)))
5      (let execute-closures ((closures (get-closures methodslot))
6                             ($retval (cond ((boolean? return_type) #f)
7                                            ((number? return_type) 0)
8                                            (else '()))))
9        (if (null? closures)
10         $retval
11         (let (($retval (execute-a-closure (car closures) args)))
12           (execute-closures (cdr closures) $retval))))))
```

Listing 3.4: The algorithm of calling a method slot

Table 3.1: The four assignment operators for method slots

Operator	Description
=	*add a new function closure and remove the others from the method slot.*
ˆ=	*insert a new function closure at the beginning of the array.*
+=	*append a new function closure to the end of the array.*
−=	*remove function closures calling the method slot at the right-hand side.*

(0, false, or null). If the method slot is empty, the caller will get the default value and no exception is thrown. It is reasonable since the empty state is not abnormal for an array and just means that nothing should be done for the call at that time. The behavior of a method slot can be dynamically modified at runtime, while still statically typed and checked at compile time. Listing 3.4 uses Scheme to describe how the closures are executed for a method slot call in DominoJ since it is easier to describe closures.

3.3.2 Assignment operators for method slots

DominoJ provides four assignment operators for manipulating the closures in a method slot: =, ˆ=, +=, and −=, as shown in Table 3.1. These operators are borrowed from C# and EScala, and are the only different syntax from Java. It is possible to add and remove a function closure to/from a method slot at runtime.

Their operands at both sides are method slots sharing the same type. Those operators except −= create a new function closure calling the method slot at the right-hand side, and add it to the method slot at the left-hand side.

The method slot called by the function closure will get the same arguments which are given to the method slot owning the function closure. In other words, a reference to the method slot at the right-hand side is created and added to the method slot at the left-hand side. The syntax of using the assignment operators to bind two method slots is shown below:
⟨*expr*⟩"`.`"⟨*methodslot*⟩ ⟨*operator*⟩ ⟨*expr*⟩"`.`"⟨*methodslot*⟩"`;`"
where ⟨*expr*⟩ can be any Java expression returning an object, or a class name if the following ⟨*methodslot*⟩ is static. When the binding statement is executed at runtime, the ⟨*expr*⟩ at both sides will be evaluated according to current execution context and then given to the operator. In other words, the ⟨*expr*⟩ at the right-hand side is also determined at the time of binding rather than the time of calling. The object returned by the ⟨*expr*⟩ at the left-hand side helps to find out the method slot at the left-hand side, where we want to add or remove the new function closure. The object got by evaluating the ⟨*expr*⟩ at the right-hand side is attached to the new function closure as a variable target, which is given to the new function closure along with the execution context at the time of calling. For example, the binding statement in Line 16 of Listing 3.3 creates a new function closure calling the method slot update on the object o by giving target = o, and appends it to the method slot setX on the object s.

```
void (int nx) -> { target.update(nx); }
  | target = o
```

Then the status of the s object will be the same as the one shown in Figure 3.2. When the slot setX on the object s is called as Line 17 in Listing 3.3, the default closure and the slot update on the object o are sequentially called with the same argument: 10. Note that all closures in a method slot get the same execution context except the side effects caused by the preceding closures in the array of that method slot, where this refers to the object owning the method slot, and therefore, the callee method slot in target must be accessible from the caller method slot in this. With proper modifiers, a method slot cannot call and be called without any limitation. The behavior avoids breaking the encapsulation in object-oriented programming.

The −= operator removes function closures calling the method slot at the right-hand side from the method slot at the left-hand side. It is also possible to remove the default closure from a slot by specifying the same method slots at both sides:

```
s.setX -= s.setX;
```

```
1  ; operator =
2  (define (assign-closure methodslot object slotname)
3    (let ((closure `(call-methodslot ,object ,slotname args)))
4      (set-closures methodslot closure)))
5
6  ; operator ^=
7  (define (insert-closure methodslot object slotname)
8    (let ((closure `(call-methodslot ,object ,slotname args)))
9      (set-closures methodslot
10                   (append closure (get-closures methodslot)))))
11
12 ; operator +=
13 (define (append-closure methodslot object slotname)
14   (let ((closure `(call-methodslot ,object ,slotname args)))
15     (set-closures methodslot
16                   (append (get-closures methodslot) closure))))
17
18 ; operator -=
19 (define (remove-closure methodslot object slotname)
20   (let ((closure `(call-methodslot ,object ,slotname args)))
21     (set-closures methodslot
22                   (remove (lambda (x) (equal? x closure))
23                           (get-closures methodslot)))))
```

Listing 3.5: The algorithms of the four assignment operators

Operators manipulate the default closure only when the method slots at both sides are the same one, otherwise operators regard the right-hand side as a closure calling that method slot. Note that the default closure is never destroyed even when it is removed. Again, we use Scheme to describe how the four assignment operators work in DominoJ as shown in Listing 3.5.

Although a method slot at the right operand of the operators such as += must have the same type that the left operand has, there is an exception. If a method slot takes only one parameter of the Object[] type and its return type is Object or void, then it can be used as the right operand whatever the type of the method slot at the left operand is. Such a method slot can be used as a generic method slot. The type conversion when arguments are passed is implicitly performed. Listing 3.6 uses Scheme to describe how the types of two method slots are checked in DominoJ.

DominoJ allows binding method slots to constructors by specifying class name instead of the object reference and giving the keyword constructor as the method slot at the left-hand side. For example,

```
Shape.constructor += Observer.init;
```

means that creating a closure calling the static method slot init on the class object Observer and appending to the constructor of Shape. Here the return

```
1  ; is same type
2  (define (same-type? l_methodslot r_methodslot)
3    (and (equal? (get-return-type l_methodslot)
4                 (get-return-type r_methodslot))
5         (equal? (get-parameter-types l_methodslot)
6                 (get-parameter-types r_methodslot))))
7
8  ; is generic type
9  (define (generic-type? l_methodslot r_methodslot)
10   (and (equal? (get-parameter-types r_methodslot)
11                "Object[]")
12        (if (equal? (get-return-type l_methodslot)
13                    "void")
14          (equal? (get-return-type r_methodslot)
15                  "void")
16          (equal? (get-return-type r_methodslot)
17                  "Object"))))
```

Listing 3.6: The algorithm of checking the types

type of init should be void, and the parameter types must be the same as the constructor. Note that the closures appended to the constructor cannot block the object creation. This design ensures that the clients will not get an unexpected object, but additional objects can be created and bound to the new object. For example, in the default closure of init, an instance of Observer can be created and its update can be bound to the method slot setX of the new Shape object. Using constructor at the right-hand side is not allowed.

Since Java supports method overloading, some readers might think the syntax of method slots have ambiguity but that is not true. For example, the following expression does not specify parameter types:

```
s.setX += o.update;
```

If setX and/or update are overloaded, += operator is applied to all possible combinations of setX and update. Suppose that there are setX(int), setX(String), update(int), and update(String). += operator adds update(int) to setX(int), update(String) to setX(String). If there is update(Object[]), it is added to both setX(int) and setX(String) since it is generic. It is possible to introduce additional syntax for selecting method slots by parameters, but the syntax will be more complicated. Listing 3.7 is the algorithm written in Scheme to show how two method slots are picked up and bound by operators in DominoJ.

Since a language supporting the aspect paradigm must provide a way to retrieve runtime context, for example, AspectJ provides pointcut designators

```
1  ; bind methodslots by operators
2  (define (bind-methodslots
3            operator l_object l_slotname r_object r_slotname)
4    (let ((l_methodslots (get-fields l_object l_slotname))
5          (r_methodslots (get-fields r_object r_slotname)))
6      (for-each
7        (lambda (l_methodslot)
8          (for-each
9            (lambda (r_methodslot)
10             (if (or (same-type? l_methodslot r_methodslot)
11                     (generic-type? l_methodslot r_methodslot))
12               (cond ((equal? operator "=")
13                      (assign-closure
14                        l_methodslot r_object r_slotname))
15                     ((equal? operator "^=")
16                      (insert-closure
17                        l_methodslot r_object r_slotname))
18                     ((equal? operator "+=")
19                      (append-closure
20                        l_methodslot r_object r_slotname))
21                     ((equal? operator "-=")
22                      (remove-closure
23                        l_methodslot r_object r_slotname)))))
24           r_methodslots))
25       l_methodslots)))
```

Listing 3.7: The algorithm of binding method slots

and reflection API for that purpose, DominoJ provides three keywords to retrieve the information about the caller at runtime in the default closure of a method slot. The owner object and the default closure of the method slot at the left-hand side of an operator can be got by using the keywords in the default closure of the method slot at the right-hand side. Unlike AspectJ, which extends the set of pointcut designators available in the language, DominoJ extends the set of special variables such as this and super. In DominoJ a call to the method slot can be regarded as a sequence of method slot calls among objects since a method slot may contain closures calling other method slots. When a method slot is explicitly called by an expression in a certain default closure, the method slots bound to it by operators are implicitly called by DominoJ. Programmers can get the preceding objects in the call sequence. In the default closure, i.e. the body of method slot declaration, the caller object can be got by the keyword $caller. It refers to the object where we start the call sequence by the expression. The predecessor object, in other words, the object owning the preceding method slot in the call sequence, can also be got by the keyword $predecessor. It refers to the object owning the closure calling the current method slot whether explicitly or implicitly. Taking the example of Figure 3.2, suppose that we have a statement calling s.setX in the default closure of the method slot test in another class Client:

```
public class Client {
  public void test(Shape s) {
    s.setX(10);
  }
}
```

If test on an object instance of this class, for example c, is executed, the relationship between the objects c, s, and o can be described as shown in Figure 3.3. Note that calling other method slots explicitly by statements in the default closure of test, setX, or update will start separate call sequences. In Figure 3.3, using $caller in the default closure of setX and update both returns the object c since there is only one caller in a call sequence. However, the predecessor objects of s and o are different. Using $predecessor in the default closure of setX returns the object c, but using $predecessor in the default closure of update returns the object s. Note that both the apparent types of $caller and $predecessor are Object because the caller and the predecessor are determined at runtime. If the current method slot is called in a static method slot, $caller or $predecessor will return the class object properly. The special method call proceed in AspectJ is introduced in DominoJ as well. The keyword proceed can be used to call the default closure of the preceding

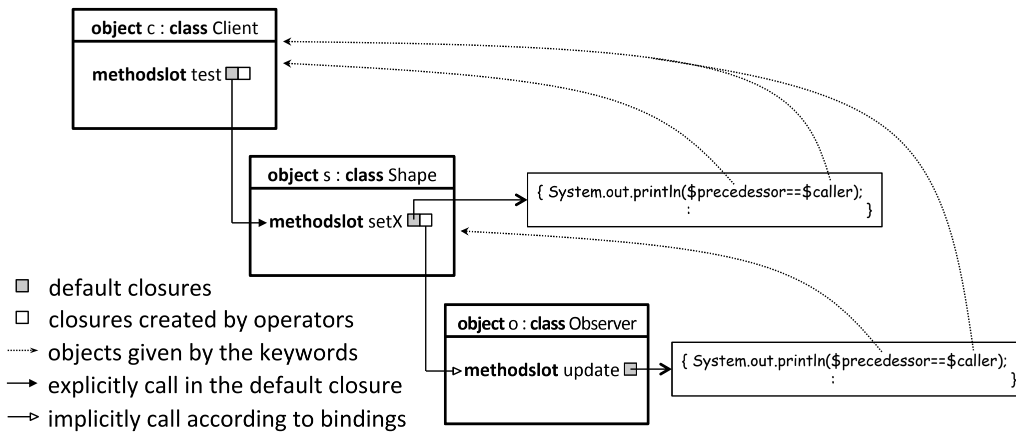

Figure 3.3: The keywords $caller and $predecessor

method slot. In Figure 3.3, calling proceed in the default closure of update on o will execute the default closure of setX on s since s.setX is the preceding method slot of o.update. If there is no preceding method slot for the current one, calling proceed will raise an exception.

3.4 Evaluation

To show the feasibility of DominoJ and measure the overheads caused by method slots, we implemented a prototype compiler[3] of DominoJ built on top of JastAddJ [23]. The source code in DominoJ can be compiled into Java bytecode and run by Java virtual machine. In the following microbenchmark, the standard library is directly used without recompilation due to the performance concern. All methods in the standard library can be called as method slots which have only the default closure, but cannot be modified by the operators.

3.4.1 The implementation

The DominoJ compiler is a source-to-source compiler which translates DominoJ code to plain Java code and then compiles it into Java bytecode. However, implementing the compiler is not easy since closures are not supported by the current Java version (Java 7). In the DominoJ compiler we use the

[3]The prototype compiler of DominoJ is available from the project webpage:
http://www.csg.ci.i.u-tokyo.ac.jp/projects/dominoj/

well-known means of the Java language such as inner classes to represent the closures.

Closure representations in Java

To emulate closures in Java, a naive implementation is using Java reflection. The compiler could generate the code to record the target objects and the method names, and use the reflection API to invoke the methods at runtime. For example, adding a closure calling o.update to s.setX could be represented as adding a pair (o, "update"), which an object instance of the class Pair<Object, String>, to the array for s.setX. When s.setX is called, all the pair stored in the array will be iterated and the methods such as o.update can be invoked by the reflection API. It is not surprising that the overheads are not small. Another idea is to define an interface like Callable then a closure can be represented by an object instance of a class implementing the interface. This class is generated by the compiler for every closure. Such an object can be stored in the array for a method slot, and the method inherited from the interface, which contains the method call such as o.update, can be called when the object is iterated.

The DominoJ compiler

The performance of DominoJ code is determined by how the closures are represented and executed at runtime. Using Java reflection is a naive solution, but the overheads are not negligible. Suppose that we have a method slot setX in DominoJ:

```
public class Shape {
     :
  public void setX(int nx) {
       :  // the default closure
  }
}
```

then the compiler will generate the following Java code in Shape: an array field setX$slot and a method setX for iterating the elements in the array setX$slot. In other words, calling a method slot in DominoJ is translated to calling a method in Java to iterate and invoke the elements in an array as follows:

```
// Java code generated by the compiler
```

```
public void setX(int nx) {
  Iterator iter = setX$slot.iterator();
  while(iter.hasNext()) {
      :  // invoke a method
  }
}
```

If we use the reflection API to invoke the methods in the iteration, the array setX$slot must store the target objects and the method names for invoking them:

```
// Java code generated by the compiler
public class Shape {
  public ArrayList<Pair<Object, String>> setX$slot
    = new ArrayList<Pair<Object, String>>();
    :
}
```

where each element in the array setX$slot holds the target object and the method name. Furthermore, The default closure of setX in DominoJ is translated into a method setX$impl in Java, which contains the statements in the default closure. When an object of Shape, for example s, is instantiated, a pair (this, "setX$impl") is appended to the array setX$slot by default. Suppose that we have another method slot update, the parameter types of which are the same as setX:

```
public class Observer {
    :
  public void update(int i) { ... }
}
```

Then the following binding:

```
s.setX += o.update;
```

where o is an object of Observer, is translated into:

```
// Java code generated by the compiler
s.setX$slot.add(new Pair<Object, String>(o, "update"));
```

When s.setX is called, the pairs (this, "setX$impl") and (o, "update") will be got in order. Here we show the code of setX again for demonstrating how to invoke the methods using the reflection API:

```
// Java code generated by the compiler
public void setX(int nx) {
  Class[] pars = new Class[1];
  pars[0] = Integer.TYPE;
  Object[] args = new Object[1];
  args[0] = nx;
  Iterator<Pair<Object, String>> iter
                             = setX$slot.iterator();
  while(iter.hasNext()) {
    Pair<Object, String> pair = iter.next();
    Object obj = pair.getFirst();
    String mname = pair.getSecond();
    Class c = obj.getClass();
    Method m = c.getMethod(mname, pars);
    m.invoke(o, args);
  }
}
```

where the Class array pars is used to specify the parameter types for finding the correct method, (int) in this example, since there may be several overloaded methods. The Object array args, which contains the arguments given to setX. In this example the only argument nx, an int, is auto-boxed in an Integer instance and put into args. Obviously the cost of finding and invoking a method using the reflection API is not low. A possible improvement is storing Method instances instead of the method names, so that we can avoid spending time on finding the Method instance when a method slot is called. However, the cost of invoking a Method is still quite high.

The idea used in our prototype compiler is using an interface to simulate the function closure in JavaScript:

```
// Java code used by the compiler
public interface Closure {
  public Object exec(Object[] args);
}
```

Then for each method slot the compiler can declare a field, which is an anonymous class implementing Closure. For example, the field update$closure is declared in Observer for calling update:

```
// Java code generated by the compiler
public class Observer {
```

```
    :
  public Closure update$closure = new Closure() {
    public Object exec(Object[] args) {
      this.update((Integer)args[0]);
      return null;
    }
  }
}
```

Note that the individual element in the array args, the arguments to exec, is type-cast properly before giving update. If update is a generic method slot, in other words the only parameter of which is Object[], the array args will be directly given to update:

```
this.update(args);
```

then in the default closure of update programmers need to check the type of each element in the array using instanceof and type-cast them if it is necessary. Furthermore, in this example we simply return null in exec since the return type of setX is void. The array for the method slot setX, setX$slot, is now an array of Closure rather than an array of the pair (Object, String):

```
// Java code generated by the compiler
public class Shape {
  public ArrayList<Closure> setX$slot
                           = new ArrayList<Closure>();
    :
}
```

The binding statement we discussed above is now translated into:

```
s.setX$slot.add(o.update$closure);
```

Similarly, a field setX$impl$closure for calling the method setX$impl, which contains the statements in the default closure of setX, is declated in Shape as well:

```
// Java code generated by the compiler
public class Shape {
    :
  public Closure setX$impl$closure = new Closure() {
    public Object exec(Object[] args) {
```

```
      this.setX$impl((Integer)args[0]);
      return null;
    }
  }
}
```

In the constructor of Shape the following line is added for appending the closure setX$impl$closure to setX$slot by default:

```
// Java code generated by the compiler
this.setX$slot.add(this.setX$impl$closure);
```

When the method slot setX is called, all Closure instances in the array are iterated and their exec methods are called with args, the Object array containing the arguments given to the method slot setX, in this example only nx:

```
// Java code generated by the compiler
public void setX(int nx) {
  Object[] args = new Object[1];
  args[0] = nx;
  Iterator<Closure> iter = setX$slot.iterator();
  while(iter.hasNext()) {
    Closure c = iter.next();
    c.exec(args);
  }
}
```

The iteration is similar to the reflection version, but the code for invoking a method using the reflection API is replaced with a call to the exec method in Closure. In other words, we need more memory to hold the Closure instances, but the overheads of method slots can be reduced to the cost of calling the exec method.

3.4.2 Microbenchmarks

In order to measure the overheads of method slots, we executed a simple program and compared the average time per method call in DominoJ and in plain Java. The method we measure is named test, which calculates $sin(\pi/6)$ by expanding Taylor series up to 100th order a number of times according to the argument as shown below:

```
private double x = 3.141592653589793 / 6;
private double result = 0;
public void test(int count) {
  for(int i=0; i<count; i++) {
    double sum = x;
    double n = x;
    double d = 1;
    for(int j=3; j<100; j+=2) {
      n *= - x*x;
      d *= (j-1)*j;
      sum += n/d;
    }
    result = sum;
  }
}
```

Figure 3.4 shows the results of continuously calling test(10) and calculating the average execution time of calling test(10) every 1000 times of calls until the total amount of calls reaches 30000. For example, the first values we calculate are the average of execution time of 1st 1000th calls in Java and in DominoJ, and the second values are the ones of 1001st 2000th.

This program was compiled by our prototype compiler and run on the JVM of OpenJDK 1.7.0_25 and Intel Core i7 (2.67GHz, 4 cores) with 8GB memory. The result of the naive implementation using the reflection API we mentioned in Section 3.4.1 is also shown for comparison. After the optimization is sufficiently applied by the JIT compiler, the overheads are negligible (2955ns against 2932ns) although it is initially about 34% (9124ns against 6833ns). On the other hand, the overheads of the reflection version are about 20% (3516ns against 2932ns) after the optimization.

To measure the performance of an operation on method slots such as assigning a closure to a method slot using = operator, we repeated the operation and calculated the average time as follows:

```
long start = System.nanoTime();
for(int j=0; j<1000; j++) {
  s.setX = o.update;
}
long estimated = System.nanoTime() - start;
System.out.println(estimated/1000);
```

We also measured other operations by adding one more statement, which uses the other operators such as += operator after the assignment:

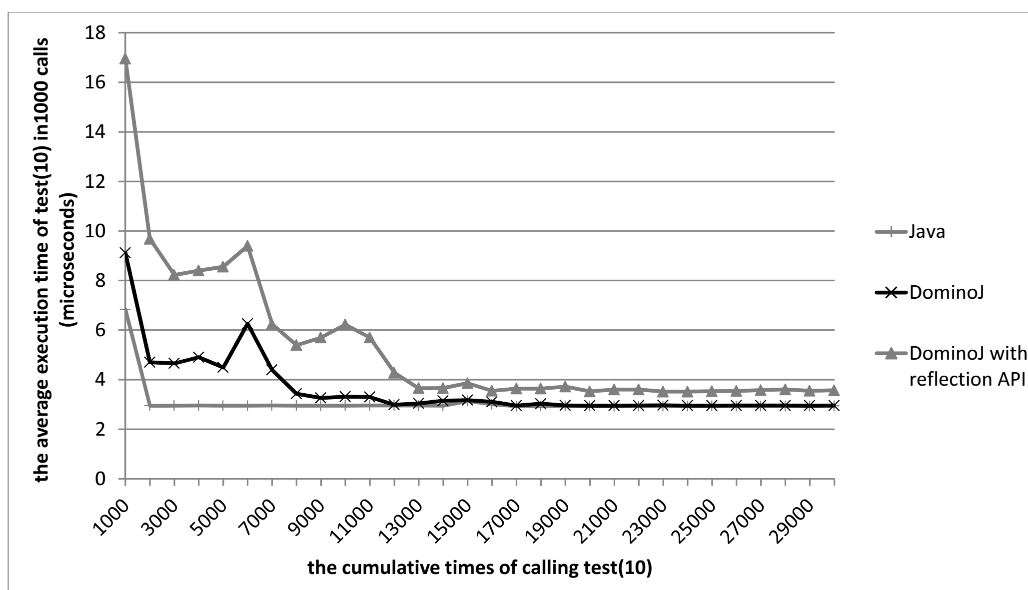

Figure 3.4: The average time of continuously calling a method in Java and DominoJ, including the reflection version

```
  :
for(int j=0; j<1000; j++) {
  s.setX = o.update;
  s.setX += o.update;
}
  :
```

Figure 3.5 shows the result of running such programs one hundred times. According to the average time of the operations in the four programs, we can calculate the time of the four operations: the = operation takes 427ns, the ^= operation takes 483ns, the += operation takes 275ns, and the −= operation takes 726ns. The −= operation might be even slower when the number of closures in the method slot is large since it takes time to check every closure in the array. It is reasonable that the += operation is the fastest one since it simply appends to the array, while the = operation have to clear the array and the ^= operation inserts to the beginning of the array; the performance is relevant to how the method slots are implemented. Finding a more efficient technique to implement method slots is included in our future work. For example, using other structures instead of ArrayList to store the closures or using the new JVM instruction invokedynamic to emulate the closures might be possible solutions to improve the performance of DominoJ code.

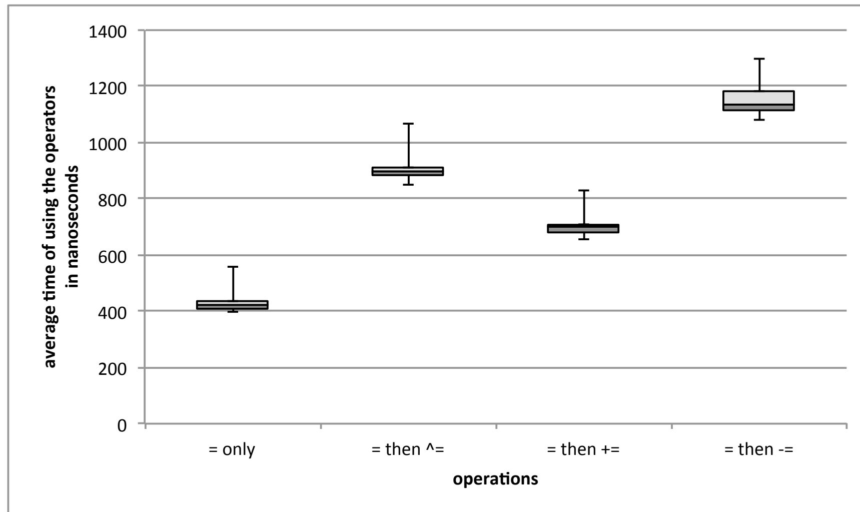

Figure 3.5: The average time of using the operators for method slots

3.4.3 Compiling and running the DaCapo benchmark suite

In order to measure the overheads of method slots more precisely, we use the DominoJ compiler to compile the DaCapo-9.12-bach benchmark suite [7] into bytecode, and compare the running performance with the one compiled by the plain Java compiler. The DaCapo benchmark suite is a tool for Java benchmarking, which consists of 14 client-side Java benchmarks for testing a set of real-world applications. For example, the benchmark batik generates a number of SVG images based on the unit tests in Apache Batik, and the benchmark eclipse runs several non-GUI JDT performance tests for the Eclipse IDE. In this experiment we compiled the benchmarks by assigning the compiler used in the build file, and run the bytecode to get the execution time. Note that only the code of DaCapo benchmarks are compiled; the applications/libraries used by the benchmarks (as shown in Listing 3.8) are either downloaded in the form of binary or compiled by the specified version of the Java compiler according to the build file of the DaCapo benchmark suite.

Table 3.2 shows the result of compiling the 14 DaCapo benchmarks by the three compilers: the plain Java compiler, our prototype version of the DominoJ compiler, and the JastAddJ compiler, which is the one our compiler based on. The version of the plain Java compiler is Oracle JDK 1.6.0_45, and the revision of the JastAddJ compiler is 9317. The benchmarks except avrora and sunflow can be successfully compiled by our prototype compiler and run

```
activemq-core-4.1.2-G20090207
apache-tomcat-6.0.20-src
batik-src-1.7beta1
catalina-6.0.18-G678601
commons-cli-1.2-bin
commons-codec-1.4-sec
commons-collections-3.2-src
commons-daemon-1.0.1
commons-dbcp-1.2.2-src
commons-httpclient-3.1-src
commons-logging-1.1.1-src
commons-pool-1.4-src
daytrader-svn-767347-src
db-derby-10.5.3.0-src
eclipse-Automated-Tests-3.5.1
eclipse-SDK-3.5.1-macosx-cocoa
fop-0.95-src
geronimo-jetty6-minimal-2.1.4-bin
geronimo-jetty6-minimal-2.1.4-plugins
h2-1.2.123
janino-2.5.15
jasper-6.0.18-G678601
jasper-el-6.0.18-G678601
jstl-1.2
jstl-impl-1.2
junit4.7
jython-src-svn-6571
lucene-2.4.1-src
nsis-2.37
oro-2.0.8
pmd-src-4.2.5
python-2.5.4
tomcat-native-1.1.16
tomcat6-deps
xalan-j-2.7.1
xerces-j-bin.2.8.0
```

Listing 3.8: The applications/libraries that are not compiled by the DominoJ compiler in this experiment

Benchmark	plain Java compiler	JastAddJ compiler	DominoJ compiler
avrora	○	fail to run	fail to compile
batik	○	○	○
eclipse	○	○	○
fop	○	○	○
h2	○	○	○
jython	○	○	○
luindex	○	○	○
lusearch	○	○	○
pmd	○	○	○
sunflow	○	fail to run	fail to compile
tomcat	○	○	○
tradebeans	○	○	○
tradesoap	○	○	○
xalan	○	○	○

Table 3.2: The result of compiling the DaCapo-9.12-bach benchmark suite by the three compilers

correctly (12 out of 14). Totally 58 files are compiled and the total number of lines of code is 45,152. Figure 3.6 shows the average time of running the bytecode compiled by the three compilers 100 times, where the JVM of Oracle JDK 1.6.0_45 and the machine has an Intel Core i7 (2.67GHz, 4 cores) and 8GB memory. The benchmarks avrora and sunflow are omitted.

The average execution time of running the 12 DaCapo benchmarks 100 times are shown in Figure 3.6. Although in the benchmarks eclipse and lusearch the average execution time of the DominoJ one is slightly larger than the JastAddJ one and the plain Java one, in general the numbers are very close and within the error. In order to understand the population of the experiment data, the box plots for each benchmark are shown as Figure 3.7. The box plots show that the population of the 100 execution result of running the three versions are very similar but the variance of the DominoJ version tends to be larger than the plain Java version and the JastAddJ version. In other words, the execution time of the DominoJ version varies within a wider range, though its average execution time is almost the same as the plain Java version. The result is reasonable since the execution time depends on the optimization by the JIT compiler. Some readers might notice that in the benchmark fop the variance of the DominoJ version is smaller than the plain Java version. It is not weird if we compare with the JastAddJ version: the variance of the DominoJ version is still slightly larger than the JastAddJ version; we can assume that it is caused by the difference between the JastAddJ compiler and the plain Java compiler. Here we do not discuss

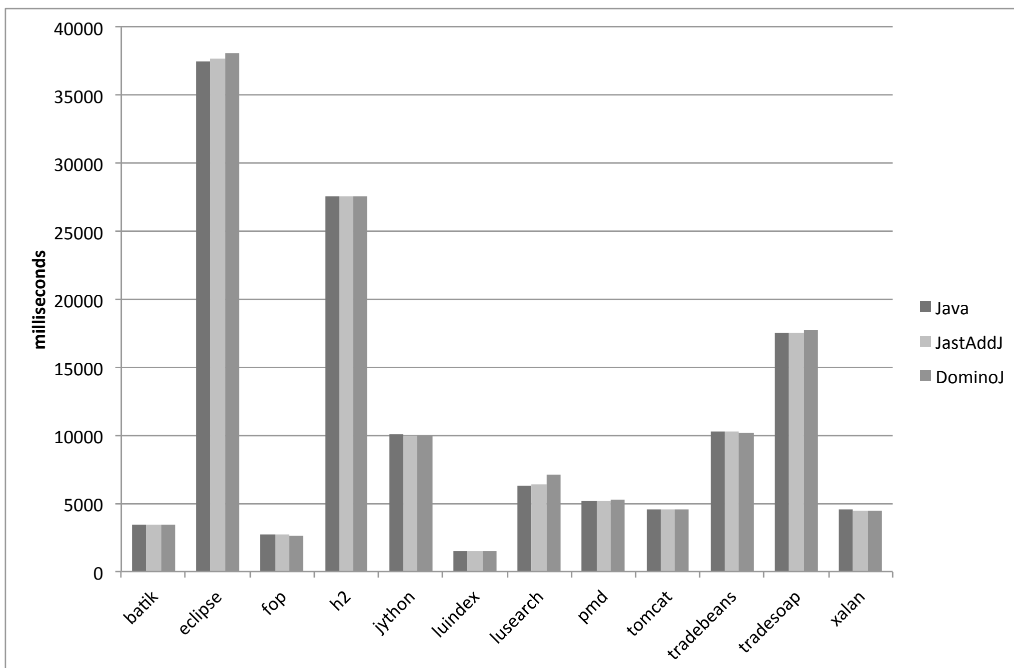

Figure 3.6: The average execution time of the 12 DaCapo benchmarks compiled by the three compilers

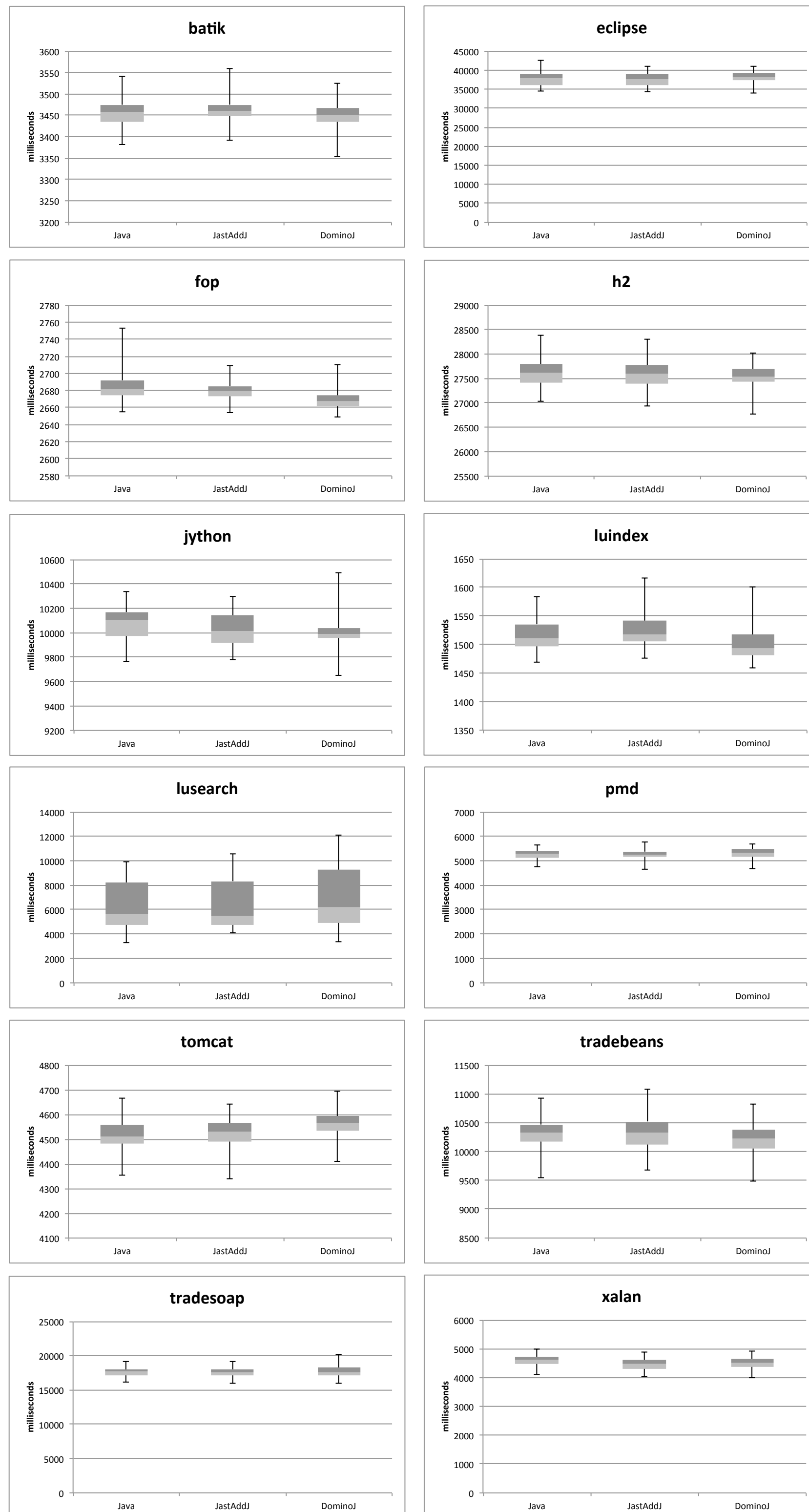

Figure 3.7: The box plot of the execution time for the DaCapo benchmarks

the individual result of running the benchmarks since the DominoJ compiler is not the contribution to a specific metric such as architectures or memory management in Java. Instead, we run the DaCapo benchmarks to get the overview of the performance in practice and check if the result is abnormal in a specific case.

3.4.4 Method slots and design patterns

Method slots extend the method paradigm to support the event-handler paradigm and the aspect paradigm, while still preserve the original behavior in the method paradigm. In DominoJ, if the operators for method slots are not used, the code work as in plain Java. In other words, a Java method can be regarded as a method slot that has only the default closure.

We could regard the inheritance in object-oriented programming as an event mechanism with default bindings. A method declaration in the superclass is an event declaration, and its implementation is the handler bound to the event by default. If the method is overridden in a subclass, the overriding implementation automatically replaces the overridden one and becomes the only handler for the event. In other words, the call to a method on an object is an event, and the method implementation selected by the polymorphism is the handler. The binding from the handler to the event in the inheritance is a one-to-one relation and pre-defined. Method slots extend the default binding in object-oriented programming to allow the binding of more than one handler to an event.

We have also analyzed how method slots can be applied to the GoF design patterns [30], and classify the patterns into four groups as shown in Table 3.3. Furthermore, we implemented the sample code in the GoF book in Java and DominoJ, and compared them with respect to the four modularity criteria borrowed from [34], and a new criterion named non-inheritance, which means that method slots can be used as an alternative to the inheritance solution or not. This might remind readers of the mixin. As an alternative to the inheritance both the mixin and method slots allow to execute an implementation in another class or object for a method call at runtime. However, in several mixin mechanisms both fields and methods are included, but a binding between method slots do not involve fields. The comparison is shown in Table 3.4, where the number of lines of code is listed as well. Note that for the patterns in group III we ignore the comparision on the number of lines of code since in group III method slots do not act a major role as in group I and group II. In this table the locality means the code of defining the relation can be gathered up, the reusability means the pattern code can

Table 3.3: Method slots can be applied to design patterns

	Pattern Name	**Description and Consequences**
I	Adapter, Chain of Resp., Composite, Decorator, Facade, Mediator, Observer, Proxy	*Implicitly propagate events among objects by the bindings.* *GOOD:* *The bindings can be gathered up in one place.* *BAD:* *The method slots which handle the same event must share the same type.*
II	Abstract Factory, Bridge, Builder, Factory Method, State, Strategy, Template Method, Visitor	*Change class behavior at runtime without inheritance.* *GOOD:* *A solution to avoid multiple inheritance.* *BAD:* *Unlike the polymorphism the switch between implementations have to be manually managed.*
III	Command, Flyweight, Interpreter, Iterator, Prototype	*Replace inheritance part in the logic.* *GOOD:* *Provide an alternative for the inheritance part.* *BAD:* *Not helpful except the inheritance part.*
IV	Memento, Singleton	*Not applicable*

Table 3.4: The benefit of applying DominoJ to design patterns

		Modularity Properties					LOC	
	Pattern Name	Locality	Reusability	Composition	Unpluggability	Non-inheritance	J	DJ
I	Adapter	$\surd$					51	48
I	Chain of Resp.	$\surd$	$\surd$	$\surd$	$\surd$	$\surd\star$	38	28
I	Composite		$\surd$	$\surd$	$\surd$	$\surd\star$	41	16
I	Decorator	$\surd$		$\surd$	$\surd$		26	20
I	Facade	$\surd\star$					34	53
I	Mediator	$\surd$			$\surd$		68	49
I	Observer	$\surd$	$\surd$	$\surd$	$\surd$	$\surd\star$	71	32
I	Proxy				$\surd$	$\surd\star$	47	61
II	Abstract Factory	$\surd\star$				$\surd\star$	41	58
II	Bridge	$\surd\star$				$\surd\star$	58	64
II	Builder	$\surd\star$				$\surd\star$	55	69
II	Factory Method	$\surd\star$				$\surd\star$	67	97
II	State	$\surd$					66	69
II	Strategy			$\surd$	$\surd$	$\surd\star$	36	28
II	Template Method	$\surd$				$\surd\star$	31	45
II	Visitor	$\surd$		$\surd$	$\surd$		63	69
III	Command					$\surd\star$	Ignored	
III	Flyweight					$\surd\star$		
III	Interpreter					$\surd\star$		
III	Iterator					$\surd\star$		
III	Prototype					$\surd\star$		
IV	Memento	Same implementation for Java and DominoJ						
IV	Singleton							

The $\surd$ mark means that DominoJ has better modularity than Java when implementing the pattern.
The $\star$ mark means that AspectJ does not provide such modularity when implementing the pattern, while DominoJ does.
J: *the Java version*   DJ: *the DominoJ version*

be abstracted and thus reusable, the composition means the code do not get complicated when applying multiple relationships to the same class, and the unpluggability means it is easy to apply or remove the pattern. The operators for method slots can be used to cause the execution of a method slot on another object when a specified method slot is called. In general such mixin behavior helps to gather up similar implementations in a class and can be an alternative to the polymorphism. Furthermore, the pattern code for propagating events can be expressed by the bindings, which can be gathered up in one place for the locality. For several patterns such as the Chain of Responsibility pattern, the event implementation is almost eliminated and thus the code tangling caused by the composition can be avoided. If the pattern code can be totally eliminated, reusing it is quite easy since no need to implement the pattern every time. It is also possible to make the code easy to plug or unplug for several patterns such as the Proxy pattern since the pattern is applied by a binding rather than passing a different object. However, unlike the polymorphism the switch between different implementations must be manually managed. As to the numbers of lines of code in Java and in DominoJ, basically there are no significantly difference since the explicit triggers for events are removed but the bindings for describing the event propagations are added. However, for several patterns such as the Observer pattern the pattern code of which is totally eliminated. On the other hand, using the mixin behavior in several patterns such as the Factory Method pattern takes

additional lines of bindings to switch the implementations. Below we discuss the four groups by showing concrete examples.

The key idea of the patterns in group I can be considered event propagation—from the outer object to the inner object, or among colleague objects. Using method slots can avoid code scattering caused by the pattern code since event implementation is eliminated. Code tangling caused by combining multiple patterns can be eased as well. The following example is an example of the Chain of Responsibility pattern. In a GUI library a widget such as a button may need a hotkey for showing help. When users are confused with the label of the button, they can press the F1 key to get a pop-up description, which explains the meaning in detail. To implement the help event in Java, the Chain of Responsibility pattern can be used in Widget, the base class for all widgets, as shown in Listing 3.9. Here we assume that the method handleHelp will be called when users press the F1 key on a widget object. Every subclass of the Widget class should override the handleHelp method to implement its own behavior for the help event, and return a boolean value to indicate whether the help event is handled or not. In the Widget class a default implementation is given: propagating the help event to the successor in the chain of responsibility. The successor is kept as a private field and set to its container in the constructor as shown in Line 2–5. If no successor is set, false is returned. When a subclass of Widget such as Button class overrides the handleHelp method, it must explicitly call super.handleHelp for executing the default implementation to propagate the help event to its successor. In DominoJ, the operator += can be used to describe such behavior as shown in Listing 3.10. Note that in Line 10 the keyword $retval is used to check if the help event is handled by the predecessor, and the explicit call super.handleHelp are removed from all subclasses. It makes the code clear, especially when there are several chain of responsibility for different events in the Widget class. Using DominoJ can avoid the tangling caused by pattern code.

Method slots can also be used to improve the transparency to clients. In a class-based object-oriented language such as Java, it is not allowed to change the class membership of objects as discussed in [22]. Suppose that two classes Student and Employee are given to model the students and the employees in a university. If now a student has graduated and employed by the university, we cannot continue using the original Student object. We have to create a new Employee object according to the original Student object and update all references to the object in clients. A solution is using method slots to implement the Proxy pattern for the Student example. In the Proxy pattern usually the clients are aware of the existence of the proxy object. For example, in order to control the access to Student, giving a proxy class

```
public class Widget {
  private Widget successor = null;
  public Widget(Widget container) {
    successor = container;
  }
  public boolean handleHelp() {
    if(successor == null)    return false;
    return successor.handleHelp();
  }
      :
}
public class Button extends Widget {
  public boolean handleHelp() {
      :  // return true if it can offer help,
         // otherwise return super.handleHelp()
  }
}
```

Listing 3.9: The Chain of Responsibility pattern example in Java

```
public class Widget {
  public Widget(Widget container) {
    this.handleHelp += container.handleHelp;
  }
  public boolean handleHelp();
      :
}
public class Button extends Widget {
  public boolean handleHelp() {
      if($retval)    return true;
      :  // return true if it is handled here,
         // otherwise return false
  }
}
```

Listing 3.10: The Chain of Responsibility pattern example in DominoJ

Employee, which owns a reference to its original Student object, then the clients have to use the proxy object instead of the original Student object. In DominoJ the behavior of a Student object such as getInfo can be replaced if it is public:

```
s.getInfo = e.getInfo;
```

where s is a Student object and e is its proxy, an Employee object. Then the clients of s may continue using the reference to s. When s.getInfo is called, the method slot getInfo on its proxy object will be executed for access control. In other words, it is possible to make the clients unaware of plugging or unplugging the proxy.

The patterns in group II use the inheritance to alter the class behavior at runtime. Different implementation for a method slot call can be added to the method slot instead of overriding in subclasses. In that sense, method slots can be used as an alternative to the polymorphism. Although method slots are not perfect replacement for the inheritance, it is convenient in particular when programmers are forced to choose between two superclasses due to single inheritance limitation. For example, Listing 3.11 shows an example of the Template Method pattern in Java. By taking advantage of inheritance, the drawing border step in the class View can be deferred to its subclass FancyView by overriding the method drawBorder. However, unlike mixin or multiple inheritance, in the subclass FancyView we cannot reuse the implementation of other classes due to the single inheritance limitation in Java. For example, the implementation of drawBorder in FancyView may be the same as the one in another class FancyPrint, which is neither a subclass of View nor a subclass of FancyView. In this case we cannot extract the common part of FancyView and FancyPrint into a new class Fancy. In DominoJ such mixin behavior is possible by using the operator =. As shown in Listing 3.12 we move the drawBorder implementation to a new class Fancy and let FancyView own a reference to a Fancy object. Then in the constructor of FancyView we can forward the call to its method slot drawBorder to the one in the Fancy object it refers (Line 22). With DominoJ a subclass can still benefit from another class by the binding as using the mixin. It helps to modularize the code when we want to extract parts of the implementation in the subclass. Programmers can decide to use mixin or inheritance for a feature depending on the design.

Another example we want to show here is the State pattern, which allows an object to alter its behavior by switching between the state objects. Using DominoJ the state transitions can be modularized in another class as using AspectJ [34]. Suppose that we have three state classes for the Queue class:

```
1  public class View {
2    public void display() {
3      drawBorder();
4      drawContent();
5    }
6    public void drawBorder() {
7      System.out.println("View: drawBorder");
8    }
9    public void drawContent() {
10     System.out.println("View: drawContent");
11   }
12 }
13 public class FancyView extends View {
14   public void drawBorder() {
15     System.out.println("Fancy: drawBorder");
16   }
17 }
```

Listing 3.11: The Template Method pattern example in Java

```
1  public class View {
2    public void display() {
3      drawBorder();
4      drawContent();
5    }
6    public void drawBorder() {
7      System.out.println("View: drawBorder");
8    }
9    public void drawContent() {
10     System.out.println("View: drawContent");
11   }
12 }
13 public class Fancy {
14   public void drawBorder() {
15     System.out.println("Fancy: drawBorder");
16   }
17 }
18 public class FancyView extends View {
19   Fancy fancy;
20   public FancyView() {
21     fancy = new Fancy();
22     this.drawBorder = fancy.drawBorder;
23   }
24 }
```

Listing 3.12: The Template Method pattern example in DominoJ

```
1  public class Queue {
2    private QueueState state = new QueueEmpty();
3    public void setState(QueueState s) {
4      state = s;
5    };
6    public boolean insert(Object o) {
7      return state.insert(this, o);
8    }
9      :
10 }
11 public class QueueState {
12   public boolean insert(Queue q, Object o) {
13     return false;
14   };
15     :
16 }
17 public class QueueEmpty extends QueueState {
18   public boolean insert(Queue q, Object o) {
19     QueueNormal nextState = new QueueNormal();
20     q.setState(nextState);
21     return nextState.insert(q, o);
22   }
23     :
24 }
```

Listing 3.13: The State pattern example in Java

QueueEmpty, QueueNormal, and QueueFull. In Java the state transition code scatters across the state classes, for example the transition from QueueEmpty to QueueNormal is checked and performed in the insert method of QueueEmpty class as shown in Listing 3.13. In DominoJ all transitions can be gathered up in another class UpdateQueueState as shown in Listing 3.14. The class UpdateQueueState keeps all the state objects (Line 2–4) and manages the transitions such as emptyToNormal (Line 6–10). For example, the transition emptyToNormal is performed after the method slot insert on the object empty is executed as shown in Line 15. Note that the method slots emptyToNormal and insert share the same type.

The patterns classified under group III also use the inheritance as a part of their pattern code, so programmers may use method slots or not depending on the situation. For example, the intent of the Command pattern is wrapping the requests in objects in order to pass around clients, and inheritance is used for overriding the behavior of a request. Suppose that we want to implement a document editor, which allows users to open a document, edit its content, and copy a paragraph. First we declare an abstract class Command, which has a method slot execute, to model the commands supported in the editor:

```
public abstract class Command {
```

```
1  public class UpdateQueueState {
2    private QueueEmpty empty = new QueueEmpty();
3    private QueueNormal normal = new QueueNormal();
4    private QueueFull full = new QueueFull();
5    private Queue queue = null;
6    public boolean emptyToNormal(Object o) {
7      normal.insert(o);
8      queue.setState(normal);
9      return $retval;
10   }
11     :
12   public void setup(Queue q) {
13     queue = q;
14     queue.setState(empty);
15     empty.insert += this.emptyToNormal;
16       :
17   }
18 }
```

Listing 3.14: The State pattern example in DominoJ

```
    :
  public void execute();
}
```

Then we can implement the individual commands such as OpenCommand and CopyCommand by extending the Command class. In the subclasses we can declare necessary parameters and override execute to define the behavior for individual commands. For example, the implementation of OpenCommand looks like this:

```
public class OpenCommand extends Command {
  private File file = null;
    :
  public void execute() {
    file = getFileFromUser();
  }
}
```

Here, the user has to select a file and then the path in the field file will be stored when its execute is called. By creating the command objects, the requests from users can be wrapped and passed to other UI components. The functionalities such as undo and redo can also be implemented easily. In group III the inheritance is not the core of the pattern code, but helps the implementation. As the example of the Template Method pattern shown

above in group II, the inheritance can be replaced with the mixin by using method slots. Again, using the mixin is not always a good choice and it depends on programmers' design decision.

As to the patterns in group IV, DominoJ is not helpful in dealing with object creation as what AspectJ does in [34]. The reason is that DominoJ does not support inter-type declaration and cannot stop the object creation. Further details of this analysis is available in [92].

3.4.5 The event-handler paradigm

There are three important metrics to evaluate an event mechanism. First, the amount of explicit triggers in a program depends on whether the events can be implicit or not. Second, if dynamic binding is not provided, it is not possible to change the handler at runtime. Third, event composition helps the abstraction though it is not absolutely necessary. In an event mechanism the three properties are determined by how the bindings between the event and the handler are presented.

To evaluate how DominoJ works for the event-handler paradigm, first we analyze the bindings between the event and the handler in a typical event mechanism like EScala, and compare them with DominoJ. In languages directly supporting the event-handler paradigm, events are usually introduced as fields, which are separate from methods. In order to associate fields with methods, there are three types of bindings between events (fields) and handlers (methods). The ways used for each type of binding are usually different in an event mechanism, and also different between event mechanisms. Table 3.5 shows the ways provided by EScala. The corresponding DominoJ syntax for the three types of bindings is also listed, but actually there is only slot-to-slot binding in DominoJ since only method slots are involved in the event-handler paradigm. Every method slot can play an event role and a handler role at the same time. Listing 3.15 shows how to use DominoJ for the event-handler paradigm for the shape example mentioned in Section 3.2. Below we will discuss what the three types of binding are, and explain how DominoJ provides the same advantages with the simplified model.

The event-to-handler binding is the most trivial one since it means what action reacts to a noteworthy change. Whether supporting the event-handler paradigm by languages or not, in general the event-to-handler binding is dynamic and provided in a clear manner. For example, in the Observer pattern an observer object can call a method on the subject to register itself; in C# and EScala, += operator and −= operator are used to bind/unbind a method to a special field named events. In addition to the two operators, DominoJ

Table 3.5: The roles and bindings of the event-handler paradigm in EScala and DominoJ

	Type	**EScala**	**DominoJ**
role	*Event*	field (evt)	method slot
	Handler	method	
binding	*Event-to-Handler*	+=	+=
		−=	−=
	Event-to-Event	||	+=, ^=
		&& \ filter map empty any	use Java expression in the default closure of method slots
	Handler-to-Event	afterExec	+=
		beforeExec	^=
		imperative	explicit trigger is possible

```
public class Display {
  public void refresh(int x, int y) {
    System.out.println("display is refreshed.");
  }
}
public class Shape {
  private int left = 0; private int top = 0;
  public void setPosition(int x, int y) {
    left = x; top = y;
  }
  public Shape(Display d) {
    this.setPosition += d.refresh;
  }
}
public class Test {
  public static void main(String[] args) {
    Display d = new Display();
    Shape s = new Shape(d);
    s.setPosition(0, 0);
  }
}
```

Listing 3.15: Using DominoJ for the event-handler paradigm

provides ^= operator and = operator to make it easier to manipulate the array of handlers. In C# and EScala the handlers for an event can be only appended sequentially and removed individually, but in DominoJ programmers can use = operator to empty the array directly without deducing the state at runtime. Using ^= operator along with += operator also makes design intentions more clear since a closure can be inserted at the beginning without popping and pushing back.

The second one is the event-to-event binding that enables event composition and is not always necessary but greatly improves the abstraction. In a modern event mechanism, event composition should be supported. EScala allows programmers to define such higher-level events to make code more readable. An event-to-event binding can be simulated by an event-to-handler binding and a handler-to-event binding, but it is annoying and error-prone. In DominoJ, it is also possible to define a higher-level event by declaring a method slot without a default closure. Then operators += and ^= can be used to attach other events like what the operator || in EScala does. Other operators in EScala such as && and map are not provided in DominoJ, but the same logic can be represented by statements in another handlers and attached by += operator. For example, in Listing 3.1 we can declare a new event adjusted that checks if left and top are the same as the arguments given to setPosition using the operator && in EScala:

```
evt adjusted[Unit] = afterExec(setPosition)
                     && ((left,top) != _._1)
adjusted += onAdjusted
```

where `_._1` refers to the arguments given to setPosition and onAdjusted is the reaction. In DominoJ, we can declare a higher-level event adjusted and perform the check in another method slot checkAdjusted:

```
public void adjusted(int x, int y);
public void checkAdjusted(int x, int y) {
  if(!(x==left && y==top))  adjusted(x, y);
}
```

and then bind them as follows:

```
setPosition += checkAdjusted;
adjusted += onAdjusted;
```

Although the expression in DominoJ is not rich and declarative as in EScala, they can be used to express the same logic. In addition, the event-to-event binding in EScala is static, so that the definition of a higher-level event in EScala cannot be changed at runtime. On the other hand, it is possible in DominoJ since the slot-to-slot binding is totally dynamic.

The last one is handler-to-event binding, which is also called an event trigger or an event definition. It decides whether an event trigger can be implicit or not. In the Observer pattern and C#, an event must be triggered explicitly, so that the trigger code is scattering and tangling. EScala provides two implicit ways and an explicit way: after the execution of a method, before the execution of a method, or triggering an event imperatively. In DominoJ, an event can be triggered either implicitly or explicitly. A method slot can not only follow the call to another method slot but also be imperatively called. More precisely, there is no clear distinction between the two triggering ways. In EScala afterExec and beforeExec are provided for statically binding an event to the execution of a method while DominoJ provides += operator and −= operator for dynamically binding a method slot to the execution of another method slot. This sounds like that a method slot has two pre-defined EScala-like events for the default closure, but it is not correct. In DominoJ's model the only event is the call to a method slot, and the default closure is also a handler like the other closures calling other method slots. This feature makes the code more flexible since the execution order of all handlers can be taken into account together. As to the encapsulation, in EScala the visibility of explicit events follows its modifiers, and the implicit events are only visible

within the object unless the methods they depend on are observable. On the other hand, the encapsulation in DominoJ relies on the visibility of method slots. The design is simpler but limits the usage because a public method slot is always visible as an event to other objects.

There is one more important difference between EScala and DominoJ. In DominoJ, a higher-level event can be declared or not according to programmers' design decision. In order to explain the difference, we use a tree graph to represent the execution order in the shape example by regarding setPosition as the root. As shown in Figure 3.8, we use rectangles, circles, and rounded rectangles to represent methods, events, and method slots, respectively. When a node is called, the children bound by beforeExec or ^= must be executed first, followed by the node itself and the children bound by afterExec or +=. Figure 3.8 (a) is the execution order of Listing 3.1, and Figure 3.8 (b) is the one of Listing 3.15. In the DominoJ version, the event moved is eliminated and its child refresh is bound to setPosition directly since we do not need additional events in such simple case. DominoJ is easier and simpler to apply the event-handler paradigm when events are not complicated but used everywhere. In EScala, events must be created since methods cannot be bound to each other directly. However, such events are still necessary if we want to keep the abstraction. In that case, method slots can be used as the events in EScala by declaring them without a default closure. For example, the event moved in Line 11 of Listing 3.1 can be translated into the following statements:

```
public void moved();
setPosition += moved;
```

Figure 3.8 (c) is another DominoJ version, which has the higher-level event as the EScala version. In DominoJ, programmers can choose between the simplified one and the original one depending on the situation.

Note that the number of lines of Listing 3.15 is one line longer than Listing 3.1 because the syntax of Scala looks more compact than Java. In Java the constructor and the fields used inside a class must be declared explicitly while they are omitted in Scala. In Listing 3.15 the constructor takes two more lines than Listing 3.1. If we do not take this into account, the EScala version is one line longer than the DominoJ version due to additional event declaration.

The line of code can also be analyzed according to Table 3.5. With regard to the roles, additional event declarations are necessary in EScala while they are combined into one declaration in DominoJ as we discussed above. For the event-to-handler binding, both the operators provided by EScala and DominoJ take one line. For the event-to-event binding, the operators provided by

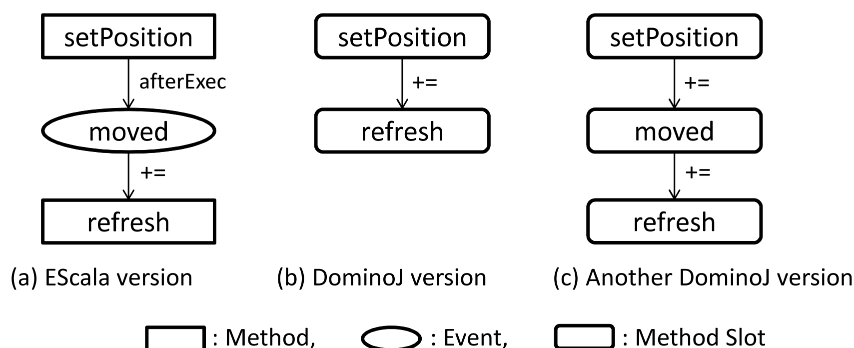

Figure 3.8: The execution order of the shape example in EScala and DominoJ

EScala can be written in the same line, but in DominoJ += operator and ^= operator cannot be merged into one line. In that case the code in DominoJ is longer than the EScala one. For example, a higher-level event changed can be defined by three events resized, moved, and clicked:

```
evt changed[Unit] = resized || moved || clicked
```

but in DominoJ they must be defined as follows:

```
resized += changed;
moved += changed;
clicked += changed;
```

That is why the expression in EScala is richer but complicated. Introducing appropriate syntax sugar to DominoJ to allow to put operators in one line is also possible, but we think it makes the design complicated. However, in this example we can also find passing the event value in EScala takes effort. In EScala, as far as we understand, only a value is kept in an event field. If we want to gather up the arguments x and y given to setPosition, and then pass to moved and changed, we need to declare additional classes such as Point and declare the events with the new type rather than Unit[4]. The additional classes increase the number of lines as well. For the handler-to-event binding, afterExec and beforeExec in EScala can define an anonymous event and share the same line of an event-to-handler binding. To sum up, in DominoJ the event declarations may be eliminated and thus the number of lines of source code can be reduced. On the other hand, the number of code of DominoJ

[4]In EScala, declaring events with Unit type means that no data are passed [32].

version is longer when translating a complex EScala expression composed of a number of operators since DominoJ has less primitive syntax. DominoJ makes code clear because each method slot has a name explicitly, and each line for binding only defines the relation between two method slots.

3.4.6 The aspect paradigm

DominoJ can be used to express the aspect paradigm as well. In order to discuss language constructs concretely, we compare DominoJ with the most representative aspect-oriented language—AspectJ. The call to a method slot is a join point, and other method slots can be bound to it as advices. Note that the aspect paradigm is broader as discussed in [43, 44] and not restricted to the AspectJ style, which is the point-advice model. In AspectJ the important features such as around advices, the obliviousness, and inter-type declaration that an event mechanism cannot provide are all supported by constructs. In this subsection first we analyze the necessary elements in the point-advice model in order to compare the constructs provided by AspectJ and DominoJ. Then we use DominoJ to rewrite the shape example in Listing 3.2 and discuss the differences.

Since the purpose of the aspect paradigm is to modularize the crosscutting concerns, we need a method-like construct to contain the code piece, a way to attach the method-like construct to a method execution, and a class-like construct to group the method-like construct. In AspectJ, the class-like construct is the aspect construct, the method-like construct is the advice body, and the way of attaching is defined by the pointcut and advice declaration. In DominoJ, the method slot and the class construct in plain Java are used and only operators for method slots are introduced for attaching them. The method slots bound by += operator or ^= operator are similar to after/before advices, respectively. The method slots bound by = operator are similar to around advices and `proceed` can be used to execute the original method slot. It is expected that DominoJ cannot cover all expression in AspectJ since DominoJ's model is much simpler. For example, in DominoJ inter-type declaration and the reflection are not provided. According to the three elements, Table 3.6 lists the mapping of language constructs in AspectJ and DominoJ.

In AspectJ programmers need to understand the special instance model for the `aspect` construct, but in DominoJ the `class` construct is reused. Although the instances of the construct for grouping need to be managed manually, there is no need to learn the new model and keywords like `issingleton`, `pertarget`, and `percflow`. In DominoJ programmers can create an instance of

Table 3.6: The mapping of language constructs for the aspect paradigm in AspectJ and DominoJ

Construct	**AspectJ**	**DominoJ**
grouping	aspect	class
code piece	advice body	method slot body (default closure)
pointcut and advice declaration	after returning and execution	+= and $retval
	before and execution	^=
	around	=
	this	$caller
	target	$predecessor
	args	by parameters

the aspect-like class and attach its method slots to specified objects according to the conditions at runtime. If the behavior of issingleton is preferred, programmers can declare all fields including method slots in the aspect-like class as static since static method slots are supported by DominoJ. The shape example of AspectJ in Section 3.2 can be rewritten by DominoJ as shown in Listing 3.16. Here the class UpdateDisplay is the aspect-like class. In Line 14, we attach the advice refresh in a static method slot init, so all Shape objects will share the class object of UpdateDisplay. Furthermore, we let init be executed after the constructor of Shape, so that we can avoid explicitly attaching refresh every time a Shape object is created. Moreover, we do not have to modify the constructor of Shape. If we need to count how many times setPosition is called for each Shape and thus pertarget is preferred, we can rewrite the class UpdateDisplay as shown in Listing 3.17. Every time a Shape object is created, a UpdateDisplay object is created for it implicitly. Note that the object ud will not be garbage-collected since its method slot count is attached to another method slot.

In DominoJ, there is no difference between methods and advices while in AspectJ they are different constructs. Although an advice in AspectJ can be regarded as a method body, it cannot be directly called. If the code of an advice is reusable, in AspectJ we must move it to another method but in DominoJ it is not necessary.

The pointcut and advice declaration in AspectJ and DominoJ are similar

```
1  public class Display {
2    public static void refresh(int x, int y) {
3      System.out.println("display is refreshed.");
4    }
5  }
6  public class Shape {
7    private int left = 0; private int top = 0;
8    public void setPosition(int x, int y) {
9      left = x; top = y;
10   }
11 }
12 public class UpdateDisplay {
13   public static void init() {
14     ((Shape)$predecessor).setPosition += Display.refresh;
15   }
16   static { Shape.constructor += UpdateDisplay.init; }
17 }
18 public class Test {
19   public static void main(String[] args) {
20     Shape s = new Shape();
21     s.setPosition(0, 0);
22   }
23 }
```

Listing 3.16: Using DominoJ as the aspect paradigm

```
1  public class UpdateDisplay {
2    private int total = 0;
3    public void count(int x, int y) {
4      total++;
5    }
6    public static void init() {
7      UpdateDisplay ud = new UpdateDisplay();
8      ((Shape)$predecessor).setPosition += ud.count;
9    }
10   static { Shape.constructor += UpdateDisplay.init; }
11 }
```

Listing 3.17: Rewrite UpdateDisplay for pertarget

but not the same. First, what they target at is different. AspectJ is class-based while DominoJ is object-based. In other words, what AspectJ targets at are all object instances of a class and its subclasses but what DominoJ targets at are individual object instances. However, it is possible to emulate the class-based behavior in DominoJ by the code attaching to the constructor of a class as shown in Line 16 of Listing 3.16. Second, unlike AspectJ that has call and execution pointcut, in DominoJ only execution pointcut is supported. This limits the usage but reduces the complexity. In fact, the relation between advices is quite different in AspectJ and DominoJ. In AspectJ an advice is attached to methods and cannot be directly attached to a specific advice, but in DominoJ a method slot is not only an advice but also a method. For example, if we need another advice for checking the dirty region in Listing 3.2, we may prepare an aspect CheckDirty containing this advice as shown in Figure 3.9 (a). However, the advice can only be attached to setPosition. In DominoJ, the advice can be attached to either setPosition or init as shown in Figure 3.9 (b).

The behavior of proceed in AspectJ and DominoJ is also a little different. The proceed in DominoJ should be used only along with = operator since it calls the default closure in the preceding method slot rather than the next closure. The root cause of the difference is the join point model: what DominoJ adopts is the point-in-time model while the one AspectJ adopts is the region-in-time model [51]. In other words, in AspectJ the arrays of the three types of advices are separate, but in DominoJ there is only one array. If += operator or ^= operator are used after using = operator to attach a method slot containing proceed, the behavior is not as expected as in AspectJ. Figure 3.10 shows an example of around advices in AspectJ and DominoJ. In AspectJ, the around advices localCache and memCache are attached to queryData in order. In DominoJ, we can do it similarly:

```
queryData = localCache;
localCache = memCache;
```

then using proceed in memCache and localCache will call the default closure of their preceding method slot, localCache and queryData, respectively. Another difference is that the args pointcut and the wildcard used in call and execution pointcuts in AspectJ are not supported in DominoJ. Method slots are simply matched by their parameters. If the overloading is not taken into account, the operators in DominoJ only select one method slot in one line statement.

As to the number of lines, the two versions are about the same. Comparing them line by line might not make much sense since there is no simple translation between DominoJ and AspectJ.

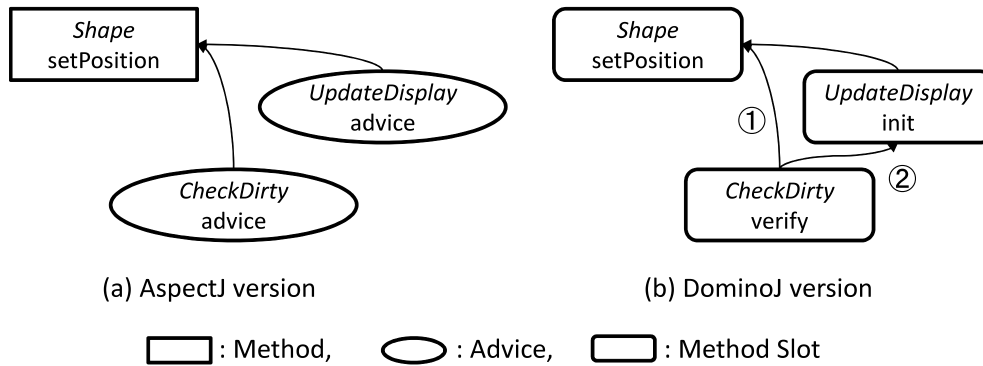

Figure 3.9: Adding another advice to the shape example in AspectJ and DominoJ

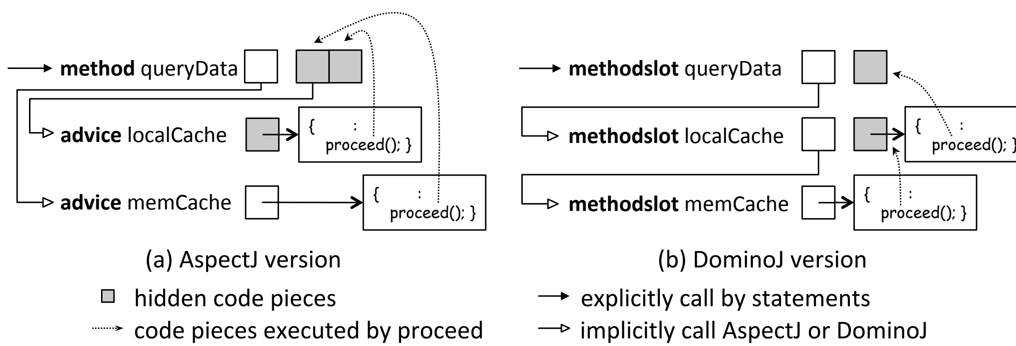

Figure 3.10: Calling proceed in AspectJ and DominoJ

Table 3.7: A summary of the significant characteristics of the two paradigms and the support in DominoJ

the Event-handler paradigm	DominoJ
implicit events	yes
dynamic binding	yes
event composition	yes

the Aspect paradigm	DominoJ
around advices	yes
the obliviousness	yes
inter-type declaration	no

3.4.7 Summary of the coverage

In the previous subsections we have discussed what a language must have for the event-handler paradigm and the aspect paradigm by comparing with EScala and AspectJ, respectively, from the viewpoint of constructs. In this subsection we summarize the significant characteristics of the two paradigms and discuss the support in DominoJ as shown in Table 3.7. In addition to being used for the event-handler paradigm and the aspect paradigm, DominoJ allows programmers to use both paradigms together.

For the event-handler paradigm, there are three significant properties: implicit events, dynamic binding, and event composition. DominoJ supports them all by method slots and only four operators. Rewriting a complex expression of event composition in EScala is also possible though it takes more lines. Introducing additional syntax may resolve the issue but it also complicates the model. As a result of regarding method slot calls as events, giving an event a different visibility from the method slots it depends on is not supported by DominoJ.

The aspect paradigm of AspectJ has three important features that cannot be provided by the event-handler paradigm: around advices, the obliviousness, and inter-type declaration. In DominoJ what the around advices in AspectJ does can be archived by assigning a closure calling another method slot using the = operator. DominoJ also supports the obliviousness in AspectJ by using the class construct as the aspect construct and attaching a method slot to a constructor of the target class. In the method slot attached to the constructor, programmers can further attach advices to the method slots at the target class. However, the inter-type declaration in AspectJ is not available in DominoJ. A possible solution is introducing a default method slot for undefined fields in a class like Smalltalk's `doesNotUnderstand` or what the `no-applicable-method` does in CLOS.

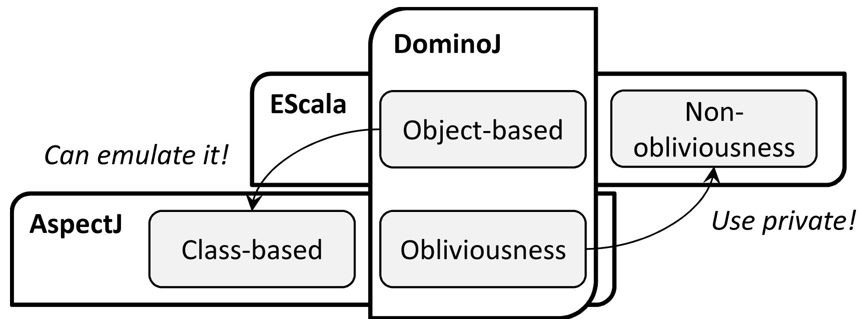

Figure 3.11: The design decision of EScala, AspectJ, and DominoJ

3.4.8 Event-handler vs. Aspect

Although the event-handler paradigm and the aspect paradigm are developed for resolving different issues, their implementation are almost the same, especially from the viewpoint of virtual machine. They both allow programmers to specify which code pieces should be executed after/before the execution of a method. The only difference is the model of specifying and executing the code pieces. First, the behavior of the event-handler paradigm used in EScala is object-based, while the behavior of the aspect paradigm is class-based. Second, in the aspect paradigm used in AspectJ the obliviousness is an important property, but in the event-handler paradigm the non-obliviousness is expected. Obviously, it is impossible to support the contradictory properties at the same time unless we give both constructs into one language. If we just put constructs for the two paradigms into one language, for example providing all syntax of EScala and AspectJ in a new language, it makes the code compilcated and programmers have to learn all of them; it is not what we want to do. Our goal is to make all available by a single construct and let programmers decide how to use it, so we have to choose between object-based behaviors and class-based behaviors, the obliviousness and the non-obliviousness.

The design decision of EScala, AspectJ, and DominoJ are shown in Figure 3.11. DominoJ chooses object-based behaviors and the obliviousness since we believe this design is most flexible—the class-based behaviors can be emulated by writing the bindings in the constructors and a method slot can be `private` if the obliviousness is not expected. In this sense, DominoJ can be regarded as either an object-based aspect-oriented language or an event mechanism. It does not matter how we call DominoJ since it is just a naming, but here we want to bring up the discussion on the similarities between the event-handler paradigm and the aspect paradigm.

3.5 Case study

To explore how method slots can be used in real-world applications, here three case studies are performed. The first one is rewriting the events example in C# tutorials [56] to show how to use method slots as events and handlers. The delegates examples in C# tutorials [55] are also rewritten to discuss the differences between method slots and delegates. The second one is analyzing the usage of the Observer pattern in the famous Java GUI framework, JHotDraw [39], and rewriting it in DominoJ. The third one is using DominoJ to rewrite the aspects in AspectTetris, a Tetris game written in AspectJ, and discuss the constructs used in the two versions.

3.5.1 The events and delegates examples in C# tutorials

In order to demonstrate how method slots can be used as events in practice, the example in the events tutorial of C#, a popular language supporting events, is rewritten. Listing 3.18 shows the C# code of the example in the tutorial. In this example the ArrayList is extended to support the Changed event, which is triggered when the elements in the list are changed. As shown in Line 4 of Listing 3.18 first a delegate type is declared, which defines the type of the handlers for the event. Then the event Changed is declared with the delegate type (Line 6), and a method OnChanged is declared for triggering the event. At the end of the methods Add and Clear, which modifies the elements in the list, OnChanged is called. Note that the method OnChanged is declared for not only checking if any handlers exist but also allowing the subclasses of ListWithChangedEvent to trigger the event. Here OnChanged is declared with protected to allow the subclasses to trigger Changed by calling it since in C# events can only be triggered by the class declaring the events.

The DominoJ verison of this example is shown in Listing 3.19. If the lines for importing classes and declaring the package it belongs are not taken into account, the total number of lines is almost the same as the C# version. An event in C# can be rewritten to a method slot without the default closure, and the line for declaring the delegate type is removed; the handlers must follows the type of the method slot or be generic method slots. However, the control of triggering an event in DominoJ is not as strict as in C#. On the other hand, in DominoJ there is no need to wrap a method in a delegate (Line 34 and Line 40 of Listing 3.18) or check the handlers before triggering an event (Line 8 of Listing 3.18). Such an example of using events in C# can be rewritten using method slots line by line.

Although delegates and events are quite different, here we want to rewrite

```
using System;
namespace MyCollections {
  using System.Collections;
  public delegate void ChangedEventHandler(object sender, EventArgs e);
  public class ListWithChangedEvent: ArrayList {
    public event ChangedEventHandler Changed;
    protected virtual void OnChanged(EventArgs e) {
      if (Changed != null)
        Changed(this, e);
    }
    public override int Add(object value) {
      int i = base.Add(value);
      OnChanged(EventArgs.Empty);
      return i;
    }
    public override void Clear() {
      base.Clear();
      OnChanged(EventArgs.Empty);
    }
    public override object this[int index] {
      set {
        base[index] = value;
        OnChanged(EventArgs.Empty);
      }
    }
  }
}
namespace TestEvents {
  using MyCollections;
  class EventListener {
    private ListWithChangedEvent List;
    public EventListener(ListWithChangedEvent list) {
      List = list;
      List.Changed += new ChangedEventHandler(ListChanged);
    }
    private void ListChanged(object sender, EventArgs e) {
      Console.WriteLine("This is called when the event fires.");
    }
    public void Detach() {
      List.Changed -= new ChangedEventHandler(ListChanged);
      List = null;
    }
  }
  class Test {
    public static void Main() {
      ListWithChangedEvent list = new ListWithChangedEvent();
      EventListener listener = new EventListener(list);
      list.Add("item 1");
      list.Clear();
      listener.Detach();
    }
  }
}
```

Listing 3.18: The example in C# events tutorial

```
1  package mycollections;
2  import java.util.ArrayList;
3  public class ListWithChangedEvent extends ArrayList {
4    public void changed(Object sender, EventArgs e);
5    protected void onChanged(EventArgs e) {
6      changed(this, e);
7    }
8    public boolean add(Object value) {
9      boolean b = super.add(value);
10     onChanged(EventArgs.Empty);
11     return b;
12   }
13   public void clear() {
14     super.clear();
15     onChanged(EventArgs.Empty);
16   }
17   public Object set(int index, Object value) {
18     Object o = super.set(index, value);
19     onChanged(EventArgs.Empty);
20     return o;
21   }
22 }
23
24 package testevents;
25 import mycollections.ListWithChangedEvent;
26 public class EventListener {
27   private ListWithChangedEvent list;
28   public EventListener(ListWithChangedEvent l) {
29     list = l;
30     list.changed += listChanged;
31   }
32   public void listChanged(Object sender, EventArgs e) {
33     System.out.println("This is called when the event fires.");
34   }
35   public void detach() {
36     list.changed -= listChanged;
37     list = null;
38   }
39 }
40
41 package testevents;
42 import mycollections.ListWithChangedEvent;
43 public class Test {
44   public static void main(String[] args) {
45     ListWithChangedEvent list = new ListWithChangedEvent();
46     EventListener listener = new EventListener(list);
47     list.add("item 1");
48     list.clear();
49     listener.detach();
50   }
51 }
```

Listing 3.19: The DominoJ version of the example in C# events tutorial

delegates by method slots to discuss their differences. The delegates in C# are similar to method slots from the viewpoint of their implementation, but the meaning of the delegates is more close to the function pointer in C or C++. Below the two examples in C# delegates tutorial are discussed in order. The first one is shown in Listing 3.20 and Listing 3.21. In this example the class BookDB maintains a database of books and exposes a method named ProcessPaperbackBooks, which allows its caller to give a delegate for processing all paperback books in the database (Line 26–31 of Listing 3.20). Then a delegate can be passed to ProcessPaperbackBooks (Line 23–24 and Line 26–27 of Listing 3.21) as a function pointer and dereferenced inside ProcessPaperbackBooks. Such a usage of function pointer is not possible in DominoJ. Instead, an event can be triggered at the timing of dereference the function pointer in order to execute the handler. Listing 3.22 and Listing 3.23 shows the DominoJ version of this example, where an event processPaperbackBookEvent must be declared and triggered inside processPaperbackBooks. Furthermore, the handlers must be bound to the event processPaperbackBookEvent before calling processPaperbackBooks (Line 26 and Line 28 of Listing 3.23) instead of giving the delegates as arguments.

The differences between delegates and method slots are even more clear by rewriting the second example in the delegates tutorial (Listing 3.24) to the DominoJ version (Listing 3.25). As shown in Listing 3.24 MyDelegate is a type and can be used to declare the variables a, b, c, and d. In DominoJ the declaration of a method slot is a kind of field rather than a type. That is the reason why we have to manually specify the types for a, b, c, and d to make them share the same type (Line 2–5 of Listing 3.25). On the other hand, there is no need to create instances as using delegates (Line 15–16 of Listing 3.24). The operators for delegates and method slots are also quite different. Although both the operators += in C# and DominoJ cause the execution of the one at right-hand side when the one at the left-hand side is called, the operator += in DominoJ creates a reference to the method slot at the right-hand side rather than copy the closures in the method slots at the right-hand side. For example, in Listing 3.24 if we change the value of b after Line 17:

```
b = a;
```

the result of calling c will not change since c has copied the delegates in a and b. The result of calling d will not be different, either. However, in DominoJ c holds the reference to a and b, and thus the result of calling c will be affected: the hello message will be printed twice. Note that Line 20 of Listing 3.24 can be regarded as the following lines in DominoJ since what c holds are the references to a and b:

```
using System;
namespace Bookstore {
  using System.Collections;
  public struct Book {
    public string Title;
    public string Author;
    public decimal Price;
    public bool Paperback;
    public Book(string title, string author,
                decimal price, bool paperBack) {
      Title = title;
      Author = author;
      Price = price;
      Paperback = paperBack;
    }
  }

  public delegate void ProcessBookDelegate(Book book);

  public class BookDB {
    ArrayList list = new ArrayList();
    public void AddBook(string title, string author,
                        decimal price, bool paperBack) {
      list.Add(new Book(title, author, price, paperBack));
    }
    public void ProcessPaperbackBooks(ProcessBookDelegate process) {
      foreach (Book b in list) {
        if(b.Paperback)
          process(b);
      }
    }
  }
}
```

Listing 3.20: The Bookstore namespace of the first example in C# delegates tutorial

```
namespace BookTestClient {
  using Bookstore;
  class PriceTotaller {
    int countBooks = 0;
    decimal priceBooks = 0.0m;
    internal void AddBookToTotal(Book book) {
      countBooks += 1;
      priceBooks += book.Price;
    }
    internal decimal AveragePrice() {
      return priceBooks / countBooks;
    }
  }

  class Test {
    static void PrintTitle(Book b) {
      Console.WriteLine("   {0}", b.Title);
    }
    static void Main() {
      BookDB bookDB = new BookDB();
      AddBooks(bookDB);
      Console.WriteLine("Paperback Book Titles:");
      bookDB.ProcessPaperbackBooks(
          new ProcessBookDelegate(PrintTitle));
      PriceTotaller totaller = new PriceTotaller();
      bookDB.ProcessPaperbackBooks(
          new ProcessBookDelegate(totaller.AddBookToTotal));
      Console.WriteLine("Average Paperback Book Price: ${0:#.##}",
          totaller.AveragePrice());
    }
    static void AddBooks(BookDB bookDB) {
      bookDB.AddBook("The C Programming Language",
          "Brian W. Kernighan and Dennis M. Ritchie", 19.95m, true);
      bookDB.AddBook("The Unicode Standard 2.0",
          "The Unicode Consortium", 39.95m, true);
      bookDB.AddBook("The MS-DOS Encyclopedia",
          "Ray Duncan", 129.95m, false);
      bookDB.AddBook("Dogbert's Clues for the Clueless",
          "Scott Adams", 12.00m, true);
    }
  }
}
```

Listing 3.21: The BookTestClient namespace of the first example in C# delegates tutorial

```
package bookstore;
public class Book {
  public String title;
  public String author;
  public float price;
  public boolean paperback;
  public Book(String title, String author,
              float price, boolean paperback) {
    this.title = title;
    this.author = author;
    this.price = price;
    this.paperback = paperback;
  }
}

package bookstore;
public class BookDB {
  ArrayList<Book> list = new ArrayList<Book>();
  public void addBook(String title, String author,
                      float price, boolean paperback) {
    list.add(new Book(title, author, price, paperback));
  }
  public void processPaperbackBooks() {
    for(int i=0; i<list.size(); i++) {
      Book b = list.get(i);
      if(b.paperback)
        processPaperbackBookEvent(b);
    }
  }
  public void processPaperbackBookEvent(Book b);
}
```

Listing 3.22: The DominoJ version of the first example in C# delegates tutorial (the bookstore package)

```
1  package booktestclient;
2  import bookstore.Book;
3  public class PriceTotaller {
4    int countBooks = 0;
5    float priceBooks = 0.0f;
6    public void addBookToTotal(Book book) {
7      countBooks += 1;
8      priceBooks += book.price;
9    }
10   public float averagePrice() {
11     return priceBooks/countBooks;
12   }
13 }
14
15 package booktestclient;
16 import bookstore.Book;
17 import bookstore.BookDB;
18 public class Test {
19   public static void printTitle(Book b) {
20     System.out.println("   " + b.title);
21   }
22   public static void main(String[] args) {
23     BookDB bookDB = new BookDB();
24     addBooks(bookDB);
25     System.out.println("Paperback Book Titles:");
26     bookDB.processPaperbackBookEvent += Test.printTitle;
27     PriceTotaller totaller = new PriceTotaller();
28     bookDB.processPaperbackBookEvent += totaller.addBookToTotal;
29     bookDB.processPaperbackBooks();
30     System.out.printf("Average Paperback Book Price: $%.2f\n",
31         totaller.averagePrice());
32   }
33   public static void addBooks(BookDB bookDB) {
34     bookDB.addBook("The C Programming Language",
35         "Brian W. Kernighan and Dennis M. Ritchie", 19.95f, true);
36     bookDB.addBook("The Unicode Standard 2.0",
37         "The Unicode Consortium", 39.95f, true);
38     bookDB.addBook("The MS-DOS Encyclopedia",
39         "Ray Duncan", 129.95f, false);
40     bookDB.addBook("Dogbert's Clues for the Clueless",
41         "Scott Adams", 12.00f, true);
42   }
43 }
```

Listing 3.23: The DominoJ version of the first example in C# delegates tutorial (the booktestclient package)

```
1  using System;
2  delegate void MyDelegate(string s);
3
4  class MyClass {
5    public static void Hello(string s) {
6      Console.WriteLine("  Hello, {0}!", s);
7    }
8
9    public static void Goodbye(string s) {
10     Console.WriteLine("  Goodbye, {0}!", s);
11   }
12
13   public static void Main() {
14     MyDelegate a, b, c, d;
15     a = new MyDelegate(Hello);
16     b = new MyDelegate(Goodbye);
17     c = a + b;
18     d = c - a;
19
20     Console.WriteLine("Invoking delegate a:");
21     a("A");
22     Console.WriteLine("Invoking delegate b:");
23     b("B");
24     Console.WriteLine("Invoking delegate c:");
25     c("C");
26     Console.WriteLine("Invoking delegate d:");
27     d("D");
28   }
29 }
```

Listing 3.24: The second example in C# delegates tutorial

```
MyClass.d += MyClass.a;
MyClass.d += MyClass.b;
MyClass.d -= MyClass.a;
```

so in the DominoJ version we simply assign `b` to `d` (Line 18 of Listing 3.25).

Rewriting the two C# tutorials shows that events can be easily translated to method slots, though the control of triggering events in C# and in DominoJ are different. Furthoremore, the similarities and the differences between method slots and C# delegates are also clarified.

3.5.2 The Observer pattern in JHotDraw

JHotDraw, a Java GUI framework for drawing technical and structured graphics, is well-known for the demonstration of using design patterns, especially the Observer pattern. One of its original developers is also the author of the GoF book. In this subsection we analyze how the Observer pattern

```
class MyClass {
  public static void a(String s);
  public static void b(String s);
  public static void c(String s);
  public static void d(String s);

  public static void hello(String s) {
    System.out.printf("  Hello, %s!\n", s);
  }

  public static void goodbye(String s) {
    System.out.printf("  Goodbye, %s!\n", s);
  }

  public static void main(String[] args) {
    MyClass.a =  MyClass.hello;
    MyClass.b =  MyClass.goodbye;
    MyClass.c += MyClass.a;
    MyClass.c += MyClass.b;
    MyClass.d =  MyClass.b;

    System.out.println("Invoking delegate a:");
    a("A");
    System.out.println("Invoking delegate b:");
    b("B");
    System.out.println("Invoking delegate c:");
    c("C");
    System.out.println("Invoking delegate d:");
    d("D");
  }
}
```

Listing 3.25: The DominoJ version of the second example in C# delegates tutorial

is used in JHotDraw to implement events, and rewrite it to the DominoJ version. The version of JHotDraw we analyzed is 6.0 beta 1.

Table 3.8 lists the 8 subject-observer pairs defined in JHotDraw, where the name of the subjects, the name of their observers, and the notification are summarized. Here the notification are the events in the event-handler paradigm, and the methods defined in the observers are the handlers. In JHotDraw the signatures of the handlers are defined in an interface named “listener”, for example CommandListener, which defines the following method signatures:

```
public void commandExecuted(EventObject e);
public void commandExecutable(EventObject e);
public void commandNotExecutable(EventObject e);
```

Any classes implementing CommandListener must define the body of the three methods. Every subject has to maintain a list of the observers and allow observers to register or unregister themselves. For example, in Command the two method signatures are defined:

```
public void addCommandListener(CommandListener l);
public void removeCommandListener(CommandListener l);
```

Note that in most cases the subjects and the observers are defined by interfaces rather than classes in order to allow the classes implementing it to inherit from other classes such as JFrame. Furthermore, a class could be both the observers of different subjects. For example, AbstractCommand is the subject of the **Command-CommandListener** pair, the observer of the DrawingEditor-ViewChangeListener pair, and the observer of the DrawingView-FigureSelectionListener pair at the same time.

In the DominoJ version the interfaces for the observers are preserved for making the method slots in all observers for a subject consistent, though method slots can be used without them. In a subject the list for holding its observers and the methods for adding and removing an observer are no longer necessary. Instead, using empty method slots to declare events. For example, in Command the two method signatures shown above are replaced with the follows:

```
public void commandExecutedEvent(EventObject e);
public void commandExecutableEvent(EventObject e);
public void commandNotExecutableEvent(EventObject e);
```

Table 3.8: The 8 subject-observer pairs defined in JHotDraw

Subject	Observer	Notification
Command	CommandListener	*the command is executed, executable, or not executable*
DesktopEventService	DesktopListener	*a DrawingView is added, removed, or selected*
Drawing	DrawingChangeListener	*the Drawing invalidates an area, changes its title, or requests update*
DrawingEditor	ViewChangeListener	*the active view has changed, a new view is created, or an existing view is to be destroyed*
DrawingView	FigureSelectionListener	*the figure selection is changed*
Figure	FigureChangeListener	*an area is invalidated, a figure is changed or removed, requesting to remove or update*
PaletteButton	PaletteListener	*the user selected or moved the mouse over the palette*
Tool	ToolListener	*the Tool is enabled, disabled, usable, unusable, activated, or deactivated*

For a notification there is no need to iterate the observers and explicitly call their methods; just triggering the events. Then observers can bind their handler method slots to the events. If the number of the events is large, the two methods for adding and removing an observer can be preserved for collecting the bindings at the subject side. For example, the addFigureChangeListener defined in the interface Figure can be preserved and let AbstractFigure bind the handlers to the events as follows:

```
public void addFigureChangeListener(FigureChangeListener l) {
  this.figureInvalidatedEvent    += l.figureInvalidated;
  this.figureChangedEvent        += l.figureChanged;
  this.figureRemovedEvent        += l.figureRemoved;
  this.figureRequestRemoveEvent += l.figureRequestRemove;
  this.figureRequestUpdateEvent += l.figureRequestUpdate;
}
```

As a result, which handlers to bind is determined at the subject side rather than at the observer side.

The classes and interfaces we modified for the 8 subject-observer pairs are grouped as shown in Table 3.9–3.10. The LOC column shows the difference of lines of code between the DominoJ version and the original version. For example, in Table 3.9 the +3 in the Command row and the -61 in the AbstractCommand row mean that the number of the lines of code in the DominoJ version increases 3 lines and decreases 61 lines, respectively. The — in the CommandListener row means that the code is not modified at all. Note that in Table 3.10 the 0 in the AbstractCommand row and the UndoableCommand row means that the number of lines of code are the same after rewriting to the DominoJ version. Although the DominoJ version needs more lines for defining events in the subject interfaces, the lines of code of the implementer classes are decreased since the code of observer management and notification are removed. As to the obvious changes of lines such as AbstractCommand and AbstractTool the main reason is that an inner class for dispatching events named EventDispatcher are removed.

The list of all the files modified in the DominoJ is shown in Table 3.11. The number of lines of code in Java, in DominoJ, and the difference between them are listed. The difference between the two versions for a file can also be got by adding up the difference caused by all subject-observer pairs that the class involves in as shown in Table 3.9–3.10. Note that in Table 3.11 the mark † means that the file is not modified for rewriting the Observer pattern, and the files marked with ⋆, DrawingEditor.java, DrawingView.java, Figure.java and DisposableResourceHolder.java, need additional lines to explicitly specify

Table 3.9: Rewriting the Observer pattern in JHotDraw by DominoJ

Class/Interface Name	Description	ΔLOC
Command	defines Command interface	+3
CommandListener	defines CommandListener interface	—
AbstractCommand	implements Command	-61
UndoableCommand	implements Command and CommandListener	-19
CommandMenu	implements CommandListener	+2
CTXCommandMenu	implements CommandListener	+2
DesktopEventService	defines DesktopEventService class	-6
DesktopListener	defines DesktopListener interface	—
DrawingEditor	define DrawingEditor interface	+1
ViewChangeListener	define ViewChangeListener interface	—
DrawApplet	implements DrawingEditor	-2
DrawApplication	implements DrawingEditor	-37
JavaDrawViewer	implements DrawingEditor	-2
AbstractCommand	implements ViewChangeListener	-12
AbstractTool	implements ViewChangeListener	-12
PaletteButton	defines PaletteButton class	+1
PaletteListener	defines PaletteListener interface	—
Tool	defines Tool interface	+6
ToolListener	defines ToolListener interface	—
AbstractTool	implements Tool	-79
UndoableTool	implements Tool and ToolListener	-19
ToolButton	implements ToolListener	+5

ΔLOC: the difference of LOC after rewriting.

Table 3.10: Rewriting the Observer pattern in JHotDraw by DominoJ (continued)

Class/Interface Name	Description	ΔLOC
Drawing	defines Drawing interface	-12
DrawingChangeListener	defines DrawingChangeListener interface	—
DrawingView	extends DrawingChangeListener	—
StandardDrawing	implements Drawing	-42
StandardDrawingView	implements DrawingView	+3
MiniMapView	uses DrawingChangeListener	+2
DrawingView	define DrawingView interface	-10
FigureSelectionListener	define FigureSelectionListener interface	—
StandardDrawingView	implements DrawingView	-31
NullDrawingView	implements DrawingView	-14
AbstractCommand	implements FigureSelectionListener	0
UndoableCommand	implements FigureSelectionListener	0
DrawingEditor	extends FigureSelectionListener	—
DrawApplication	implements DrawingEditor	+1
DrawApplet	implements DrawingEditor	+2
JavaDrawViewer	implements DrawingEditor	+1
Figure	defines Figure interface	+1
FigureChangeListener	defines FigureChangeListener interface	—
AbstractFigure	implements Figure	-7
AttributeFigure	extends AbstractFigure	—
ImageFigure	extends AttributeFigure	-2
Drawing	extends FigureChangeListener	—
CompositeFigure	extends AbstractFigure and implements FigureChangeListener	-8
DecoratorFigure	extends AbstractFigure and implements FigureChangeListener	-6
TextAreaFigure	extends AttributeFigure and implements FigureChangeListener	-4
TextFigure	extends AttributeFigure and implements FigureChangeListener	-2
GraphicalCompositeFigure	extends CompositeFigure	-4
StandardDrawing	extends CompositeFigure	-5

ΔLOC: the difference of LOC after rewriting.

Table 3.11: All the files modified in the DominoJ version of JHotDraw

Package/File	J	DJ	ΔLOC
org.jhotdraw.applet/DrawApplet.java	573	573	0
org.jhotdraw.application/DrawApplication.java	1,186	1,150	-36
org.jhotdraw.figures/AttributeFigure.java	250	250	—
org.jhotdraw.figures/ImageFigure.java	153	151	-2
org.jhotdraw.figures/TextFigure.java	509	507	-2
org.jhotdraw.framework/Drawing.java	317	305	-12
org.jhotdraw.framework/DrawingChangeListener.java	34	34	—
org.jhotdraw.framework/DrawingEditor.java	72	76	+4 ⋆
org.jhotdraw.framework/DrawingView.java	315	315	0 ⋆
org.jhotdraw.framework/Figure.java	340	344	+4 ⋆
org.jhotdraw.framework/FigureChangeListener.java	48	48	—
org.jhotdraw.framework/FigureSelectionListener.java	25	25	—
org.jhotdraw.framework/Tool.java	128	134	+6
org.jhotdraw.framework/ToolListener.java	27	27	—
org.jhotdraw.framework/ViewChangeListener.java	36	36	—
org.jhotdraw.samples/javadraw/JavaDrawViewer.java	123	122	-1
org.jhotdraw.standard/AbstractCommand.java	253	180	-73
org.jhotdraw.standard/AbstractFigure.java	482	475	-7
org.jhotdraw.standard/AbstractTool.java	395	304	-91
org.jhotdraw.standard/CompositeFigure.java	734	726	-8
org.jhotdraw.standard/DecoratorFigure.java	337	331	-6
org.jhotdraw.standard/StandardDrawing.java	227	180	-47
org.jhotdraw.standard/StandardDrawingView.java	1,183	1,155	-28
org.jhotdraw.standard/ToolButton.java	165	170	+5
org.jhotdraw.standard/NullDrawingView.java	430	416	-14
org.jhotdraw.util/Command.java	59	62	+3
org.jhotdraw.util/CommandListener.java	24	24	—
org.jhotdraw.util/CommandMenu.java	134	136	+2
org.jhotdraw.util/PaletteButton.java	148	149	+1
org.jhotdraw.util/PaletteListener.java	32	32	—
org.jhotdraw.util/UndoableCommand.java	134	115	-19
org.jhotdraw.util/UndoableTool.java	192	173	-19
org.jhotdraw.contrib/CTXCommandMenu.java	198	200	+2
org.jhotdraw.contrib/DesktopEventService.java	181	175	-6
org.jhotdraw.contrib/DesktopListener.java	22	22	—
org.jhotdraw.contrib/GraphicalCompositeFigure.java	355	351	-4
org.jhotdraw.contrib/html/DisposableResourceHolder.java†	121	124	+3 ⋆
org.jhotdraw.contrib/MiniMapView.java	249	251	+2
org.jhotdraw.contrib/TextAreaFigure.java	929	925	-4
the Observer pattern: 38 files (29 out of them are modified)	10,999	10,633	-366
All the source files: 484 files (30 out of them are modified)	71,721	71,374	-347

The ⋆ mark means that the file contains the modification for manually specifying methods by annotation.
The † mark means that the file does not have the modification for the Observer pattern.
J: *the Java version* DJ: *the DominoJ version*

that certain declarations are not method slots. For example, the method clone declared in the interface Figure must be explicitly marked as follows:

```
@MethodSlot(false)
public Object clone();
```

Without the annotation the DominoJ compiler will transform the declaration to a method slot and thus different from the clone method declared in the Object class. The clone in any classes implementing Figure will also be method slots rather than methods. In this case the result is unexpected since the clone in Figure is used to force its implementer classes to override the one in Object. The current version of the DominoJ compiler can automatically regard a method declaration as a method slot declaraion by traversing its superclasses and implemented interfaces. If a method-like declaration is inherited from the libraries that are not compiled by the DominoJ compiler, the method-like declaration is a method; otherwise, it is a method slot. However, the DominoJ compiler has no idea whether such a declaration, clone, is a method slot or not. Similarly, the method getPreferredSize in the interface DrawingView is defined for being compatible with Swing; such design intention is not available in the interface DrawingView.

To sum up, in general using method slots instead of implementing the Observer pattern can decrease the number of lines of code since the pattern code for maintaining the observer list and iterating the observers for sending the notification can be removed. The observer interface can be preserved if it is necessary to ensure the consistency among observers, and the number of lines of the subject interface might slightly increase due to the event declarations. There are 484 source files and 71,721 lines of code in JHotDraw. In the DominoJ version, 30 out of 484 files are modified and the number of lines of code becomes 71,374; 347 lines are reduced. If we only take the files involving in the 8 subject-server pairs into account, 29 out of 38 files are modified for the Observer pattern and the number of lines of code is reduced from 10,999 to 10,633; 366 lines are reduced in the DominoJ version.

3.5.3 The aspects in AspectTetris

To evaluate how DominoJ can be used for the aspect paradigm in practice, an AspectJ program, AspectTetris, is rewritten as a case study. AspectTetris is a Tetris game, which contains two parts: the main concern and the aspects. The author of the program first implemented the main concern in Java without thinking how to add the aspects later, and implemented the

aspects to add several functionalities. AspectTetris shows several typical usage of pointcuts and advices. In this subsection the aspects are rewritten to the DominoJ version.

The 8 aspects defined in AspectTetris are summarized as shown in Table 3.12. The first one, TestAspect, is a typical aspect showing a log message. The second one, DesignCheck, is used to check if the specific methods and the constructor are called from a proper class or package. The purpose of the two aspects in the Aspects.Gui package, GameInfo and Menu, is to extend the GUI of the game. The two aspects in the Aspects.Highscore, Counter and Levels, count the deleted lines and add the concept of levels to the game. As to the two aspects in the Aspects.Logic package, NewBlocks and NextBlock, they extend the types of blocks and show the next block, respectively.

The 8 aspects except DesignCheck can be rewritten by DominoJ. The aspect DesignCheck uses the pointcut within, and there is no corresponding construct in DominoJ. The result of rewriting AspectTetris to the DominoJ version is shown in Table 3.13, where the corresponding constructs and the lines of code are listed.

Generally, the AspectJ code can be rewritten by the following steps. First rewriting replace the aspect declaration with the class declaration. Taking the example of TestAspect, which attaches an advice for printing a log message before the method loadImage in the class TetrisImages. The declaration of TestAspect looks like:

```
public aspect TestAspect {
    :
}
```

It can be rewritten by replacing aspect with class:

```
public class TestAspect {
    :
}
```

The next step is moving the body of advices into method slots. The before advice for the pointcut logPoint in TestAspect:

```
before(String fileName) : logPoint(fileName) {
  System.out.println(thisJoinPoint.getSignature()
                                   + ", " + fileName);
}
```

In the DominoJ version it is moved to a method slot:

Table 3.12: The aspects in AspectTetris

Aspect Name	Package Name	Description
TestAspect	Aspects	show a log message when loading the image files for the background and color blocks.
DesignCheck	Aspects.Development	show a warning message if an object instance of the class BlockPanel is created neither inside Gui package nor inside Aspects package. show a warning message if the methods declared in the class AspectTetris are called neither inside AspectTetris itself nor Aspects package.
GameInfo	Aspects.Gui	add an information panel at the left-hand side for showing the information added by the aspects Counter, Levels, and NextBlock.
Menu	Aspects.Gui	add a menu "File", which contains "New Game", "Pause", and "Exit" items.
Counter	Aspects.Highscore	count and show the deleted lines on the information panel.
Levels	Aspects.Highscore	increase the fall speed when the number of the deleted lines reaches the next level. show the current level on the information panel.
NewBlocks	Aspects.Logic	add two new types of blocks to the game.
NextBlock	Aspects.Logic	show the next block on the information panel.

Table 3.13: Rewriting the aspects in AspectTetris to the DominoJ version

Aspect Name	Language Constructs		LOC		Note
	AJ	DJ	AJ	DJ	
TestAspect	aspect call args before thisJoinPoint	^= constructor $predecessor $retval	17	22	In DominoJ $retval needs to be returned.
DesignCheck	aspect call within	N/A	16	N/A	within is not available in DominoJ.
GameInfo	aspect execution before thisJoinPoint	^= constructor $predecessor	33	33	
Menu	aspect execution target after before thisJoinPoint	^= += constructor $predecessor	65	67	
Counter	aspect call execution after before	^= += constructor $predecessor	60	68	
Levels	aspect call execution target args set after before	^= += constructor $predecessor	60	64	set is not available in DominoJ, so select the only one method slot manually.
NewBlocks	aspect call args get around proceed	+= = constructor proceed	101	102	+= is used to bind method slots in the constructor. get is not available in DominoJ, so a getter must be declared.
NextBlock	aspect call execution after around	+= = constructor	47	52	

AJ: *the AspectJ version* DJ: *the DominoJ version*

```
public static Image logPoint(String fileName) {
  System.out.println($predecessor + ", " + fileName);
  return $retval;
}
```

Here the method slot is given the name of the pointcut, logPoint, to make it easy to understand. Here the usage of the reflection in AspectJ, thisJoinPoint, must be replaced since the reflection is not supported in DominoJ. In this example we simply print out the object instead of the signature of method slot. Note that in DominoJ the return value must be always returned. If there is no need to change the return value, just simply returning $retval. As to the parameters, in AspectTetris fileName comes from the pointcut declaration logPoint:

```
pointcut logPoint(String fileName)
  : call(* TetrisImages.loadImage(String))
    && args(fileName);
```

This example is easier to describe in DominoJ since the argument passing is not complicated; the signatures of loadImage and logPoint are the same:

```
public static void init() {
  ((TetrisImages)$predecessor).loadImage
                                ^= TestAspect.logPoint;
}
```

The operator ^= works as attaching a before advice to the join point. The binding is put into a static method slot for being bound to the constructor:

```
static { TetrisImages.constructor ^= TestAspect.init; }
```

It means that the method slot init must be called before the constructor when an object instance of TetrisImages is created. Here the operator ^= is used rather than the operator += since loadImage is called in the constructor and thus logPoint must be bound to loadImage before executing the constructor. Note that in this example there is no need to create different aspects for different TetrisImages object instances, so we simply use the modifier static for logPoint and do not create any object instances of TestAspect.

Similarly, the other aspects except DesignCheck can also be rewritten by the steps discussed above. The aspects using complicated pointcuts such as within or cflow cannot be rewritten. As shown in Table 3.13 the DominoJ version use less constructors but the number of lines of code slightly increases;

it needs an additional static initializer for binding to the constructor. This table also shows the mapping between advices and operators: after and +=, before and ^=, around and =. Furthermore, in AspectTetris the difference of using call or execution can be ignored. The proceed calls are the same, but target/args is replaced with $predecessor. On the other hand, calling getThis() on thisJoinPoint can be replaced with $predecessor, but other calls such as getSignature() cannot be rewritten. Since the field pointcuts in AspectJ are not available in DominoJ, getter and setter method slots must be declared rather than using get/set directly. For example, the pointcut get is used in the aspect NewBlocks and an around advice is attached to it:

```
pointcut numberOfTypes()
  : get(static int Blocks.NUMBEROFTYPES);

int around() : numberOfTypes() {
  return 9;
}
```

When Blocks.NUMBEROFTYPES is read, the around advice is executed and 9 is returned. In the DominoJ version a getter method slot for the field Blocks.NUMBEROFTYPES must be declared inside Blocks and replacing all read access with the getter call.

```
public class Blocks {
    :
  public static int getNumberOfTypes() {
    return NUMBEROFTYPES;
  }
}
```

Then binding the method slot that returns 9 to the getter method in NewBlocks:

```
public class NewBlocks {
  static { Blocks.constructor += NewBlocks.init; }
  public static void init(Object[] args) {
      :
    Blocks.getNumberOfTypes = NewBlocks.getNumberOfTypes;
  }
  public static int getNumberOfTypes() {
    return 9;
  }
```

```
    :
}
```

For the pointcut set a setter method slot can also be declared and replacing all write access with the setter call. However, in the aspect Levels there is only one method slot, Counter.addLines, writing the field Counter.totalLines. In this case it is possible to manually select addLines and bind the method slot as the advice:

```
Counter.addLines += Levels.deleteLines;
```

where the method slot Levels.deleteLines is rewritten from the advice attached to the pointcut set.

This case study shows a general usage of aspects in practice but not the complicated usage in enterprise software frameworks since the pointcuts for control flows are not used. However, AspectTetris demonstrates a typical usage of aspects that extend the functionalities of a program without modifying the code of the original version, and rewriting AspectTetris shows that DominoJ satisfies the needs in most cases.

3.6 Related work

The delegation introduced by C# [54] allows programmers to declare an event, define its delegate type, and bind a corresponding action to the event. Event composition is also supported by adding a delegate to two or more events. Although the delegate interface hides the executor from the caller, implicit events are not supported. The event must be triggered manually when the change happens. However, C# is able to emulate DominoJ using an unusual programming style: declaring an additional event for every method and always triggering the event rather than the method. From the point of view, a delegate is very similar to a method slot except the operator += in C# copies the handlers in the event but not creates a reference to the event. However, as in EScala, events and methods are still separate language constructs. Supporting by only one construct means that programmers do not need to decide between using such an unusual style or a normal style at the design stage whether newer modules might regard those methods as events or not. Furthermore, it is annoying that event fields and methods in C# cannot share the same name. Another disadvantage is that we have to ensure that there is at least one delegate for the event before triggering it. Otherwise it will raise an exception. This is not reasonable from the

viewpoint of the event mechanism since it just means no one handles the event. In DominoJ no handlers for an event does not raise an exception and the one that triggers an event on a method slot is unaware of handlers.

There are a number of research activities on the integration of OOP and the aspect paradigm. Those research use a single dispatch mechanism to unify OOP and the aspect paradigm and reveal that the integration makes the model clearer, reusable, and compositable. Delegation-based AOP [35, 75] elegantly supports the core mechanisms in OOP and the aspect paradigm by regarding join points as loci of late binding. The model proposed in [36] provides dedicated abstractions to express various object composition techniques such as inheritance, delegation, and aspects. The difference is that DominoJ integrates the event-handler paradigm and the aspect paradigm based on OOP. Another difference is that we propose a new language construct rather than a machine or language model, which makes it compatible with existing object-oriented languages such as Java. Other work such as FRED [68], composition filters [5], predicate dispatching [25], and GluonJ [10] can also be regarded as such integration work.

The method combination in Flavors [8] and CLOS [20] makes related methods easy to combine but not override. By default the combined method in Flavors first calls the before methods in the order that flavors are combined, following by the first primary method, then the after methods in the reverse order. The return value of the combined method is supplied by the primary method, while the return values of the before and after methods are ignored. Similarly, CLOS provides a standard method combination for generic functions. For a generic function call, all applicable methods are sorted before execution in the order the most specific one is first. Besides the primary, before, and after methods, CLOS provides the around methods and call-next-method for the primary and around methods. From the viewpoint of method combination, the default closure of a method slot looks like a primary method that can be dynamically added to other method slots as a before or after method, and even as an around method by assigning to the target method slot then using proceed as call-next-method. It is also easier to express the method combination as a hierarchy in DominoJ.

With regard to the event mechanism, several research activities are devoted to event declaration. Ptolemy [70] is a language with quantified and typed events, which allows a class to register handlers for events, and also allows a handler to be registered for a set of events declaratively. It has the ability to treat the execution of any expression as an event. The event model in Ptolemy solves the problems in implicit invocation languages and aspect-oriented languages. EventJava [27] extends Java to support event-based distributed programming by introducing the event method, which are

a special kind of asynchronous method. Event methods can specify constraints and define the reaction in themselves. They can be invoked by an unicast or broadcast way. Events satisfying the predicate in event method headers are consumed by a reaction. Context-aware applications can be accommodated easily by the mechanism. Both the two research make events clear and expressive, but they do not support implicit events, which is one of the most significant properties as an event mechanism, whereas DominoJ supports it. Moreover, all events in their model are class-based, so that events for a specified object have to be filtered in the handlers. The binding in DominoJ is object-based, so it can describe the interaction between objects more properly.

On the other hand, several research support the event-handler paradigm upon the aspect paradigm. ECaesarJ [62] introduces events into aspect-oriented languages for context-handling. The events can be triggered explicitly by method calls or defined by pointcuts implicitly. EventCJ [40] is a context-oriented programming language that enables controlling layer activation modularly by introducing events. By declaring events, we can specify when and which instance layer is activated. It also provides layer transition rules to activate or deactivate layers according to events. EventCJ makes it possible to declaratively specify layer transitions in a separate manner. Comparing with DominoJ, using events in the two languages may beak modular reasoning since their event models rely on the pointcut-advice model. Furthermore, events are introduced as a separate construct from methods.

Flapjax [53] proposes a reactive model for Web applications by introducing behaviors and the event streams. Flapjax lets clients use the event-handler paradigm by setting data flows. The handlers for an event can be registered in an implicit way. However, unlike other event mechanisms, it requires programmers to use a slightly different event paradigm. The behavior of DominoJ is more similar to the typical event mechanism while it has the basic ability for the aspect paradigm as well.

Fickle [22] enables re-classification for objects at runtime. Programmers can define several state classes for a root class, create an object at a certain state, and change the membership of the object according to its state dynamically. With re-classification, repeatedly creating new objects between similar classes for an existing object can be avoided. Both Fickle and DominoJ allow to change the class membership of an object at runtime, so other objects holding the identity of the object can be unaware of the changes. The difference is that Fickle focuses on the changes between states while DominoJ focuses on the effect of calling specified methods. Fickle provides better structural ability such as declaring new fields in state classes. However, if the relation between states is not flat and cannot be separated clearly, program-

mers still have to maintain the same code between state classes. The common code to only part of states can be gathered up into one class in DominoJ. Furthermore, DominoJ is easier to use for the event-handler paradigm.

The lambda expressions [66] will be introduced in Java 8 as a new feature to support programming in a multicore environment. With the new expression, declaring anonymous classes for containing handlers can be eliminated. The lambda expression of Java 8 is a different construct from methods but method slots can be regarded as a superset of methods.

3.7 Summary

In this chapter we showed how to use a single generic construct to support multiple paradigms. The implementations of OOP, the event-handler paradigm, and the aspect paradigm can be supported by the single generic construct named *method slots*. We successfully made it possible to support more than two paradigms by generic constructs. The idea of *method slots* comes from the methods in JavaScript and the similarity among the method calls in OOP, the handler calls in the event-handler paradigm, and the advice calls in the aspect paradigm. *Method slots* are simple and flexible.

How *method slots* support the three paradigms was discussed in detail. We presented a Java-based language named *DominoJ* to show how *method slots* can be introduced in an OO language. For the support of OOP, the GoF design patterns were rewritten by *DominoJ* to show that *method slots* can replace methods and provide better modularity. For the support of the event-handler paradigm and the aspect paradigm, we analyzed their dedicated constructs and discussed the coverage of expressive ability of *method slots*. Although the expression of *method slots* is not as rich as other languages, it is much simpler and able to express most functionality in the two paradigms.

We also showed the feasibility by implementing a prototype compiler and running preliminary microbenchmarks and the DaCapo benchmark suite. How *method slots* can be used in practice was shown by case studies.

Chapter 4

An Extension for Supporting the Reactive Paradigm

As we presented in the previous chapter, *method slots* are very simple and flexible. In this chapter we demonstrate the process of extending *method slots* to support one more paradigm—the reactive paradigm, which recently attracts a lot of attentions from OOP community. First we find that the reactive paradigm is very similar to one of the paradigms that *method slots* already support: the event-handler paradigm. By discussing the reactive paradigm from the viewpoint of the event-handler paradigm, we can understand what the major difference is and what existing event mechanisms lack of. Then we propose a very small extension to *method slots*, which only adds a new operator to satisfy the requirement for emulating the reactive paradigm by the event-handler paradigm.

4.1 Introduction

Reactive programs attract a lot of interest over the past few years. For example, an application developed for mobile devices must respond to users' input and update the status according to hardware signals. Such reactive programs need smooth and immediate response, thus polling style is not preferred. A number of frameworks and research activities have shown that

the event-handler paradigm is a promising approach to implementing reactive programs with OO design. However, using an event-based language such as C# [54] to implement reactive programs needs a lot of explicit bindings and triggers due to the lack of event composition. Even using a language supporting event composition such as EScala [32] or DominoJ [93] we still need to enumerate the lower-level events for composing a higher-level event.

On the other hand, in another approach to implementing reactive programs, functional-reactive programming (the reactive paradigm) [24, 61, 88, 89, 90, 53], the propagation of changes is implicit. The behaviors in functional-reactive languages are time-varying values, which cause the re-evaluation of all expressions reading them when their values change. Maybe we can regard the reactive paradigm as a kind of the event-handler paradigm: an event is pre-defined for every behavior and all expressions reading a behavior are the handlers bound to the pre-defined event for that behavior. However, unlike the event-handler paradigm, the reactive paradigm does not fit OO design [74]. This observation led us to develop a mechanism to automatically select events for a higher-level event as what the predicate pointcuts in the aspect paradigm do. The predicate pointcuts such as cflow in AspectJ select join points for a pointcut according to the given rule. A predicate-based definition of higher-level events could help to reduce the explicit bindings for event composition and make it easier to implement reactive programs in the event-handler paradigm.

Our contributions presented in this chapter are two-fold. First, we propose a new kind of predicate-based definition to select events for a higher-level event by inference, present its semantics with an implementation, and give a comparison between existing definitions. As far as we know, such an inference for predicates has not been introduced for defining higher-level events yet since existing predicates are used to filter events, for example the one in EventJava [27]. Second, we demonstrate how the definition can be used to write reactive programs and compare it with the reactive paradigm in examples.

4.2 Emulating the reactive paradigm by the event-handler paradigm

This section clarifies the differences between the reactive paradigm and the event-handler paradigm, and points out the lack of inference rules for event composition in current event mechanisms. We first use the reactive paradigm and the event-handler paradigm to implement a typical example of reactive

programs, the spreadsheet program, to show the similarities between the two paradigms. Then we discuss the difference between the two implementations and reveal the possibility of emulating the reactive paradigm by the event-handler paradigm. We can regard the reactive paradigm as a kind of event-handler paradigm, the event composition of which is automatic. In other words, the events for a higher-level event are automatically selected by inference in the reactive paradigm. Then we discuss the event composition in existing event mechanisms to show that current event mechanisms lack such an inference rule for event composition.

4.2.1 An example of the reactive paradigm

A typical example of reactive programs is the spreadsheet program. All data and expressions are stored in cells and sheets, and formulas are also provided for complex calculation. Figure 4.1 shows an example of spreadsheets, where B1 and C1 are given constant values: 2 and 1 respectively, and A1 are given an expression "B1 + C1". Whenever B1 or C1 is changed, A1 will be updated automatically. Implementing such reactive programs is very easy in a language supporting the reactive paradigm. For example, Flapjax [53] is such a language based on JavaScript and designed for Web applications, which gives the behaviors and event streams to model reactive programs. The behavior is a time-varying value, and its change can cause all expressions reading it to be updated. Listing 4.1 shows the spreadsheet example, which is composed of HTML tags and the Flapjax code. In Line 2–11 we first use the tags table and input to draw a sheet which has three cells named A1, B1, and C1, and then assign the values to the cells in the function loader() as shown in Line 16–19. Here the functions extractValueB and insertValueB are the ones provided by the Flapjax library for getting and setting the value of a behavior to an HTML element. In Line 16–17 the values of the cells B1 and C1 are extracted as behaviors and stored in variables b1 and c1, respectively. Line 18–19 calculate the result of "b1 + c1", assign it to another variable a1, and insert the behavior a1 to the cell A1. Note that Line 18 is a typical expression in the reactive paradigm, where a new behavior can be got by simply adding the behaviors b1 and c1. It looks like calculating the sum of constant values, but the meaning is different—the value of a1 is always equal to the sum of b1 and c1. However, the reactive paradigm does not fit OO design due to the lack of the support to complicated structures and mutable objects; the reactive paradigm is originally introduced in functional programming. For example, if b1 and c1 are objects of non-primitive types, creating a new object for a1 every time rather than incrementally updating the values inside

A1	=B1+C1		
	A	B	C
1	3	2	1
2			

Figure 4.1: An example of spreadsheet

the object makes it inefficient [74]. This makes it difficult to integrate the reactive paradigm with OOP.

4.2.2 The event-handler paradigm can do the same thing

The event-handler paradigm can also be used to implement reactive programs. Programmers may define noteworthy changes as events and bind proper handlers for them. When the changes happen, the handler will be executed automatically. To implement the spreadsheet program by the event-handler paradigm, we can prepare two events for the changes of the fields b1 and c1, move the assignment "a1 = b1 + c1" to a method, and bind the method as a handler to the two events. Here we demonstrate how to use DominoJ presented in Chapter 3, which replaces the methods in Java with method slots and gives the assignment operators for method slots, to implement the spreadsheet example as shown in Listing 4.2 (The source code of IntCell is shown in Listing 4.3). Note that those method-like declarations are method slot declarations, and the operator += used in Line 13–15 means that the body of the method slot at the right-hand side should be executed after the left one is called. In other words, the calls to b1.setValue and c1.setValue are two events, this.changed is a higher-level event composed of the two events, and this.update is the handler for this.changed. Note that unlike other event-based languages events and method calls (*a.k.a.* triggers) are different entities, in DominoJ all method slot calls are automatically regarded as events. Since this.changed is an empty method slot (no default closure is associated), a call to this.changed makes nothing happen except a successive call to the handler. When the values of b1 or c1 is changed, the statement in the method body of this.update will be executed again.

```
<body onload="loader()">
    <table width=200>
        <tr><th>A1</th>
            <th>B1</th>
            <th>C1</th>
        </tr>
        <tr><th><input id="A1" size=2 value="0" /></th>
            <th><input id="B1" size=2 value="0" /></th>
            <th><input id="C1" size=2 value="0" /></th>
        </tr>
    </table>
</body>

<script type="text/flapjax">
function loader() {
    var b1 = extractValueB("B1");
    var c1 = extractValueB("C1");
    var a1 = b1 + c1;
    insertValueB(a1, "A1", "value");
}
</script>
```

Listing 4.1: Using Flapjax to implement the spreadsheet example

```
public class PlusSheet extends Sheet {
    private IntCell a1=null, b1=null, c1=null;
    public PlusSheet() {
        super(2, 3);
        setHeaders("A1", "B1", "C1");
        a1 = new IntCell(0);
        b1 = new IntCell(0);
        c1 = new IntCell(0);
        add(a1);
        add(b1);
        add(c1);
        pack();
        b1.setValue += this.changed;
        c1.setValue += this.changed;
        this.changed += this.update;
    }
    public void changed(int v);
    public void update(int v) {
        a1.setValue(b1.getValue() + c1.getValue());
    }
    public static void main(String[] args) {
        PlusSheet p = new PlusSheet();
        p.show();
    }
}
```

Listing 4.2: Using DominoJ to implement the spreadsheet example

```
1  public class IntCell {
2    private int value = 0;
3    public void setValue(int v) {
4      this.value = v;
5    }
6    public int getValue() {
7      return this.value;
8    }
9  }
```

Listing 4.3: The source code of IntCell

4.2.3 The major difference between the two paradigms

Such an example of reactive programs implemented by the event-handler paradigm works well but is not satisfying. The explicit bindings for event composition cannot be avoided. In this example the definition of the higher-level event this.changed clearly shows the design intention that it abstracts all events that affect the output of this.update, but which events it is composed of have to be manually specified. In other words, we have to check the body of update to find out the two events b1.setValue and c1.setValue by ourselves and explicitly specify them in the bindings. It means that we cannot write such explicit bindings without the knowledge about the implementation of update. If the implementation of update is modified, we have to carefully check the bindings for this.changed. Unfortunately, the bindings might be placed far from the implementation of update and not easy to maintain.

The reactive paradigm can be regarded as a kind of the event-handler paradigm without the explicit bindings for event composition. A handler for re-evaluating expressions is bound to some sort of higher-level events, which are automatically composed by inference through the behaviors. Programmers do not have to enumerate the events behind a higher-level event and bind them to the higher-level event individually. Furthermore, there is no need to bind the handler to the higher-level event. Instead, a handler containing the expression is pre-defined, and an inference rule is implicitly given to automatically select the events for composing the higher-level event.

4.2.4 The definitions of higher-level events in existing event mechanisms

As far as we know, there are three types of definitions of higher-level events in existing event mechanisms: enumeration-based definition, pattern-based definition, and predicate-based definition. The three types of definitions have

	which events are selected is	use runtime info?	languages	preferred when
enumeration-based	*explicit*	*no*	*AspectJ, EScala, DominoJ, ...*	*the number of events is small*
pattern-based	*implicit*	*no*	*AspectJ, Ptolemy, ...*	*the naming of events is well-formed*
predicate-based	*implicit*	*yes*	*AspectJ, ...*	*the event composition is complicated*

Table 4.1: A comparison among the three types of definitions

different merits and demerits, thus are preferred in different cases. Table 4.1 is a comparison among the three types of definitions. Note that here we classify the pointcuts in AspectJ into the predicate-based definition since we can use AspectJ as an event mechanism. As we discussed in Section 3.2 and Section 3.4.8, the implementations of the event-handler paradigm and the aspect paradigm are very similar, though the two paradigms are developed to resolve different issues. Pointcut declarations are similar to event declarations, and thus the join points selected by a pointcut declaration can be regarded as the events selected by an event declaration.

Which events are selected in the enumeration-based definition is explicit, while it is implicit in the pattern-based definition and the predicate-based definition. The enumeration-based definition and the pattern-based definition do not use runtime information, but the predicate-based definition uses. Here we only discuss the three pure definitions, but a hybrid definition is also possible, for example use patterns in a predicate-based definition.

Enumeration-based definition

The enumeration-based definition is the most primitive one for defining a higher-level event, so that it is usually supported in an event mechanism which supports higher-level events. To define a higher-level event programmers have to enumerate all lower-level events that the higher-level event depends on. The enumeration pointcuts in AspectJ [45], the event composition in EScala [32] and plain DominoJ can be classified as the enumeration-based definition. For example, suppose that we have a Point class in Java:

```
public class Point {
```

```
  private int x, y;
  public int getX() { return x; }
  public int getY() { return y; }
  public void setX(int nx) { x = nx; }
  public void setY(int ny) { y = ny; }
}
```

In AspectJ we can define a higher-level event (named a pointcut) moved in an aspect, which depends on setX and setY:

```
public aspect Update {
    :
  pointcut moved():
    execution(void Point.setX(int))
    || execution(void Point.setY(int));
}
```

Similarly, using EScala we can define moved inside the Point class:

```
public class Point {
    :
  event moved[Unit] = after(setX) || after(setY);
}
```

Note that here we do not discuss the difference of the design philosophy between languages such as object-based or class-based, more declarative or more dynamic; we focus on how they select events for composing a higher-level event. In DominoJ we can declare an empty method slot moved inside the Point class:

```
public class Point {
    :
  public void moved(int v);
}
```

and let it be triggered by setX and setY:

```
public Point() {
    :
  this.setX += this.moved;
  this.setY += this.moved;
}
```

Which events are selected in such an enumeration-based definition is explicit and easy to control. However, modifying it is error-prone. Suppose that we are going to port the Point class from 2D to 3D and thus a new method (or method slot in DominoJ) is necessary: setZ. For every higher-level event that depends on setX and setY we have to check if setZ should be added. The enumeration-based definition is preferred when the dependency is not complicated and the number of lower-level events is not large.

Pattern-based definition

The examples of pattern-based definition include the wildcard expression for pointcuts in AspectJ and the quantified, typed events in Ptolemy [70]. It is a solution to avoid enumerating lower-level events by using patterns to match the name of lower-level events. Type checking is also used along with lexical matching for improving the safety of selection. For example, we can rewrite the above example to the pattern-based definition in AspectJ:

```
public aspect Update {
    :
  pointcut moved():
    execution(void Point.set*(int));
}
```

This definition is shorter than the enumeration version of AspectJ. Furthermore, by using a wildcard the new method setZ mentioned above can be implicitly included. However, lexical matching is not always good to show the design intention. If we define another method setColor which also takes an integer as the argument, setColor will be unexpectedly included in moved. In Ptolemy we can declare a higher-level event (event type) using the keyword evtype first:

```
int evtype Moved { ... }
```

and use the keyword event to define primitive events with the event type Moved in methods:

```
public class Point {
    :
  public void setX(int nx) {
    x = nx;
    event Moved { ... }
  }
}
```

To bind a handler, for example print, to the higher-level event Moved:

```
public class Update {
    :
  public void print { ... }
  when Moved do print
}
```

The event selection in Ptolemy might be classified as the pattern-based definition since the primitive events are selected by name and type. In the pattern-based definition, which events are selected is implicit since they are not explicitly listed. It is preferred when there are a number of similar methods and the naming scheme of the methods is well-formed.

Predicate-based definition

The predicate-based definition allows programmers to use a set of constraints as a predicate for the definition of a higher-level event. Unlike the enumeration-based definition and the pattern-based definition, the predicate usually needs runtime information to select events. The event method introduced in EventJava [27] is another example of predicate-based definition, but it is used for filtering primitive events. In AspectJ several predicate pointcuts such as cflow, within, and args are given to use with other pointcuts. The predicate pointcuts in AspectJ cannot be used alone since they only filter selected join points rather than select join points directly. In general the design intention of predicate-based definitions is more clear since the predicate is a description of the common ground among lower-level events rather than the names of lower-level events. This is the reason why we decide to introduce a kind of predicate-based definition rather than a pattern-based definition to DominoJ.

The reactive paradigm can be regarded as an interesting example of the predicate-based definition if we regard the reactive paradigm as an event mechanism. The reactive paradigm provides a kind of predicate-based definition with an inference rule. In other words, in the reactive paradigm there is a implicit predicate used to select events for composing the higher-level event, the description of the predicate is automatically defined by inference. In the reactive paradigm a higher-level event is pre-defined for all variables, the definition of which is an implicit predicate that all write access to a variable are selected as lower-level events. Furthermore, all expressions that read the variable are the implicit handlers for the pre-defined higher-level event. When a variable is written in an expression, all the expressions that read the variable are automatically re-evaluated.

4.2.5 The lack of inference rules

To emulate the reactive paradigm by the event-handler paradigm, we need a new kind of predicate-based definition which can work as the inference rule in the reactive paradigm. The event composition in the reactive paradigm can be classified into the predicate-based definition if we regard the reactive paradigm as a kind of the event-handler paradigm. However, the predicate-based definition in the reactive paradigm is more implicit and automatic: the event composition is done by inference rather than an explicit predicate specified by programmers. For example, we can assume that there are two lower-level events selected to compose a higher-level event for the behavior `a1` by inference in Listing 4.1—the two lower-level events are the changes of the behaviors `b1` and `c1`. In Listing 4.2 the higher-level event is `this.changed` and the two lower-level events `b1.setValue` and `c1.setValue`. Indeed the event composition in this example is not complicated since the higher-level event only consists of two lower-level events, but it could be be very complicated if the behaviors `b1` and `c1` depend on other behaviors. If we can apply the inference rule in the reactive paradigm to the event-handler paradigm, we can make the event composition automatic and implicit. Such an inference rule makes it possible to emulate the reactive paradigm by only binding the handler to the higher-level event.

4.3 ReactiveDominoJ

We propose the *inference-based definition*, a new kind of predicate-based definition of higher-level events for the event-handler paradigm, to automatically select events. With the *inference-based definition* programmers may use the event-handler paradigm with the idea inside the reactive paradigm and reuse existing OO libraries. As far as we know, such an *inference-based definition* of higher-level events has not been introduced into the event-handler paradigm yet.

4.3.1 Using DominoJ for the event-handler paradigm

In this subsection we quickly go through how to use DominoJ for the event-handler paradigm. As we discussed in Section 3.4.5 a method slot can be used as both an event and a handler in the event-handler paradigm. The call to a method slot is an event, and other method slots can be wrapped in closures and added as a handler to the array of that method slot. The assignment operator += provided by DominoJ takes two method slots, creates a closure

for calling the right one, and then appends to the end of the left one's array. As shown in Line 13 of Listing 4.2, a closure calling this.changed will be wrapped in a closure as shown below:

```
void (int v) -> {
  this.changed(v);
}
```

and appended to b1.setValue; for all possible combinations of overloaded setValue and changed. When b1.setValue is called, the default closure of b1.setValue is executed and then this.changed is called with the same arguments. Other assignment operators provided by DominoJ include the operator ^= for inserting the created closure at the beginning of the left one's array, the operator −= for removing such closures from the left one's array, and the operator = for removing all closures and adding the created closure to the left one's array. Note that the method slots at both sides of the assignment operator must share the same type.

4.3.2 The braces operator for the inference-based definition

To provide a new kind of predicate-based definition, the inference-based definition, we propose an extention of DominoJ, which is named ReactiveDominoJ, for describing reactive programs. A new operator, the braces operator { }, is introduced in ReactiveDominoJ to support the inference-based definition of higher-level events for method slots. For a method slot, the braces operator selects all method slots on the owner object that might affect its output through the access to the fields on the owner object. For example, the braces operator on the method slot update in Listing 4.2

```
{p.update}
```

selects the method slots b1.setValue and c1.setValue. Using the braces operator with the assignment operators supported by plain DominoJ can add or remove a closure to multiple method slots at once. For example, the statement:

```
{p.update} += p.update(0);
```

adds a closure that calls p.update(0) to both b1.setValue and c1.setValue.

ReactiveDominoJ makes the binding syntax in DominoJ more generic from the viewpoint of the event-handler paradigm by providing the braces operator, which takes only a method slot and can only be used at the left-hand side of the assignment operators supported by plain DominoJ. The syntax of using the assignment operator such as += operator in DominoJ is extended as shown below:

⟨*event*⟩ ⟨*assignment_operator*⟩ ⟨*handler*⟩;

where the ⟨*assignment_operator*⟩ can be =, ^=, +=, or −= as in DominoJ, and the ⟨*event*⟩ is a method slot with or without the braces operator { }. As in DominoJ, if ⟨*event*⟩ does not include the braces { }, ⟨*handler*⟩ is the name of a method slot like this:

```
b1.setValue += this.update;
```

Here no arguments are necessary for this.update since the arguments for calling b1.setValue are given to this.update. If a method slot with the braces is given to ⟨*event*⟩, a closure wrapping the method slot at the right-hand side is added to the method slots selected by the braces operator. For example,

```
{b1.getValue} += this.update(0);
```

a closure calling this.update is appended to the method slots selected by {b1.getValue}. Note that the argument 0 is passed by the closure to this.update when the closure calls it. The arguments to b1.getValue are not passed.

The semantics of event selection by the braces operator is described in Listing 4.4, which is pseudocode for showing the steps of selecting events. First, the braces operator takes a method slot M on the object O, the actual type of which is A. Suppose that the set of all the super classes of A including A is $\overline{S_A}$. The default closure D, which is the one declared in the last class overriding M in $\overline{S_A}$, is obtained as well. Then the braces operator investigates D and finds all the fields $\overline{f}$ on the owner object that may be read within the body of D. $\overline{f}$ includes the fields read only in a conditionally executed block, such as an else block. Note that only the fields declared in A or the super classes of A are included. In other words, only the fields on "this" object are included and other variables such as parameters, local variables, and the fields on other objects are not taken into account. The braces operator then selects the method slots $\overline{m}$ that might write any of $\overline{f}$ and are declared in A or the super classes of A, *i.e.* $\overline{m}$ must belong to the owner object O. Furthermore, the braces operator repeats all the above steps for every

```
1   procedure select_events(O, M) {
2     A = the actual type of class of O;
3     S_A = All the super classes of A including A;
4     D = the default closure of M declared in the class
5           which is the last one overrides M in S_A;
6     f = the fields which are read in D
7           AND declared in any of S_A;
8     m = the method slots which write any of f
9           AND are declared in any of S_A;
10    M' = the method slots called in D
11          AND belong to the objects held in the fields of O;
12    return m ∪ select_events(O_M', M');
13  }
```

Listing 4.4: The semantics of event selection

method slot *M'* that is called from *D* and belongs to an object held in the fields of *O*. Note that the search refers to the actual type of *O* so the dynamic method dispatch in OOP is correctly traced. For the performance concern during the investigation for method slots only the default closure of the method slot is investigated. Investigating all the closures in the method slot might be a more powerful implementation since the closures added by the assignment operators at runtime can also be investigated. However, such an implementation needs to either do all the investigation at runtime or disable the separate compilation as we will discuss later (Section 4.3.3).

In the semantics of event selection the method slot M can be considered as the reader of the fields on the same object and the method slots $\overline{m}$ can be considered as the writers of those fields. The relation between the reader method slot M and the writer method slots $\overline{m}$ can be regarded as a kind of getter-setter relation through one or more fields on the same object. All possibly read or written fields in the default closure are clues to select method slots. The inference-based definition does not care about the lexical relation and enable N-to-N relation between getters and setters. We assume that fields are only accessed inside an object and thus the inference fits OO design.

The extended DominoJ, ReactiveDominoJ, can be used to write reactive programs. The spreadsheet example discussed in Section 4.2 can be rewritten as shown in Listing 4.5. There are two differences from the DominoJ version (Listing 4.2). First, the explicit bindings for the event composition of this.changed are eliminated. Second, the higher-level event this.changed is omitted as well, and the handler is bound to {this.update} instead of this.changed. The meaning of {this.update} in Listing 4.5 is the same as this.changed in Listing 4.2, which is composed of all events that affect the

```
public class PlusSheet extends Sheet {
    private IntCell a1=null, b1=null, c1=null;
    public PlusSheet() {
        super(2, 3);
        setHeaders("A1", "B1", "C1");
        a1 = new IntCell(0);
        b1 = new IntCell(0);
        c1 = new IntCell(0);
        add(a1);
        add(b1);
        add(c1);
        pack();
        {this.update} += this.update();
    }
    public void update() {
        a1.setValue(b1.getValue() + c1.getValue());
    }
    public static void main(String[] args) {
        PlusSheet p = new PlusSheet();
        p.show();
    }
}
```

Listing 4.5: Using ReactiveDominoJ to implement the spreadsheet example

output of this.update. Note that in the ReactiveDominoJ version we remove the parameter of update since it is not necessary.

To explain the relation among events and handlers, here we use a graph to describe their dependencies: a node is an event or a handler, and an edge is a dependency relationship such as a binding. The dependency graph for this.changed in the DominoJ version is shown in Figure 4.2, which means that a higher-level event changed depends on two events b1.setValue and c1.setValue, and the handler is bound to changed. In the DominoJ version we know which events the higher-level event changed should depend on since we have the knowledge about the default closure of update and the getter-setter relation between getValue and setValue. We have to investigate the implementation of the default closure of update and enumerate all the events behind this.changed manually. Such a definition of higher-level events is enumerated-based. The investigation is not easy since the dependency behind the default clousre of a method slot might change for future maintenance or extension. In the ReactiveDominoJ version the definition of such a higher-level event is inference-based. We give the method slot update with the braces operator as a hint, and let ReactiveDominoJ select the events by inference. ReactiveDominoJ investigates recursively and selects all the events update depends on according to the clues such as Figure 4.3 and Figure 4.4. After the binding

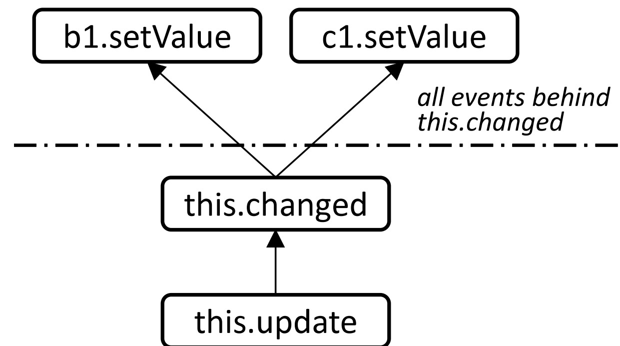

Figure 4.2: The DominoJ version needs the explicit bindings for the event composition of this.changed

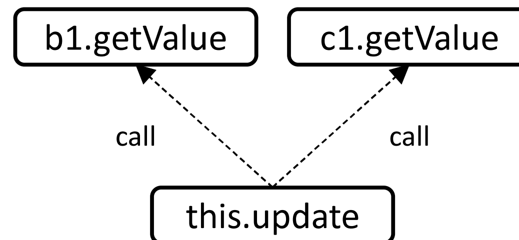

Figure 4.3: According to the default closure of update, this.update depends on b1.getValue and c1.getValue

in Line 13 of Listing 4.5 is performed, the dependency graph will be the one shown in Figure 4.5, where the dashed line is completed by ReactiveDominoJ. In other words, such a definition of higher-level events is inference-based since the actual bindings are completed by inference.

Some readers might notice that in the ReactiveDominoJ version (Listing 4.5) it is also possible to write the following statement in the body of main rather than the binding in Line 13:

```
{p.update} += p.update();
```

The braces operator improves the encapsulation of a class since this manner allows to bind a handler to the events inside the object without directly touching the fields in that object. The fields in the class can be observed through getter methods and the observers can be unaware of the implementation of the class. In this example the fields of PlusSheet, a1, b1 and c1, and

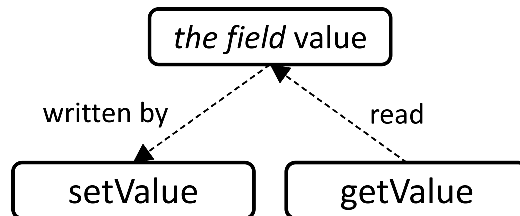

Figure 4.4: According to Listing 4.3, getValue depends on setValue

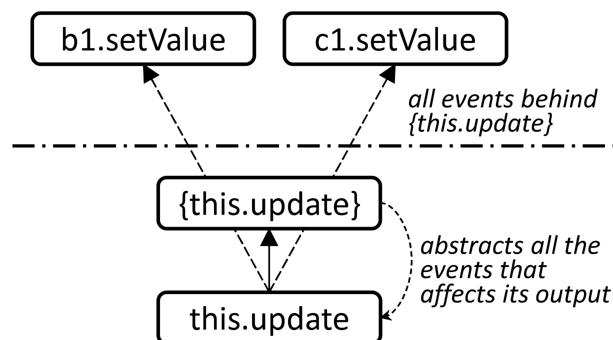

Figure 4.5: The actual bindings are completed by inference

the field in IntCell, value, do not have to be exposed by declaring as public while they still can be observed outside.

Listing 4.6 is an example of the event-handler paradigm in DominoJ for demonstrating how the braces operator selects method slots on different objects. This example implements a class Meter, which automatically calculates and prints out the distance between two given Shape objects in the Cartesian plane when either the name or the coordinates of the two objects are changed. Note that a Shape object can be placed on another Shape object by giving the parent object as the second argument to its constructor, and its coordinates will be relative to its parent object rather than the origin. In Shape class and Meter class we define several events and bind handlers to them as shown in Figure 4.6 and Figure 4.7, respectively. Now we give two Shape objects s1 and s2, to the Meter object m, where the parent of s2 is s3, the dependency is a little complicated as shown in Figure 4.8. Whenever the name or the coordinates of s1, s2, and s3 are modified, m will re-calculate the distance between s1 and s2 and print out a string. The ReactiveDominoJ version of this example is shown in Listing 4.7, where all events and explicit bindings for event composition are replaced with the binding to {this.print} (Line 39). The braces operator selects writer method slots by tracing the reader method slots, so programmers do not have to enumerate all the writer method slots and bind the handler to them manually.

4.3.3 The implementation of the reactive extension

We implemented a ReactiveDominoJ compiler[1] to demonstrate the feasibility of the inference-based definition of higher-level events. As what the DominoJ compiler does, the ReactiveDominoJ compiler transforms ReactiveDominoJ code into plain Java code and then compiles into Java bytecode. To improve the performance of selecting method slots in the Java-based language, ReactiveDominoJ, the selection is splitted into compile time and runtime. At compile time the compiler generates helper methods for collecting the method slots and binding a closure to them. Then at runtime the corresponding helper method is called according to the actual type of the owner object to collect method slots and bind the given closure to them. Note that a method slot is owned by every instance. It is not shared among the instances of the same class.

Taking the example of Listing 4.5, the following helper method is gener-

[1]The prototype compiler of ReactiveDominoJ is available from the project webpage: `http://www.csg.ci.i.u-tokyo.ac.jp/projects/rdominoj/`, which is an extension to the DominoJ compiler and is also built on top of JastAddJ [23].

```
1  public class Shape {
2    private Shape parent; private String name;
3    public String getName() {
4      if(parent == null)    return name;
5      return parent.getName() + ":" + name;
6    }
7    public void setName(String n) { name = n; }
8    public void renamed(String s);
9    private int x = 0, y = 0;
10   public void setX(int nx) { x = nx; }
11   public void moveX(int dx) { x = x + dx; }
12   public int getX() {
13     if(parent == null)    return x;
14     return parent.getX() + x;
15   }
16   public void setY(int ny) { y = ny; }
17   public void moveY(int dy) { y = y + dy; }
18   public int getY() {
19     if(parent == null)    return y;
20     return parent.getY() + y;
21   }
22   public void moved(int v);
23   public Shape(String n, Shape p) {
24     parent = p; name = n;
25     setName += renamed;
26     setX += moved; moveX += moved;
27     setY += moved; moveY += moved;
28     if(parent != null) {
29       parent.moved += moved;
30       parent.renamed += renamed;
31     }
32   }
33 }
34 public class Meter {
35   private Shape from, to;
36   public double distance() {
37     double x2 = Math.pow(to.getX()-from.getX(), 2);
38     double y2 = Math.pow(to.getY()-from.getY(), 2);
39     return Math.sqrt(x2 + y2);
40   }
41   public void changed(Object[] args);
42   public void print(Object[] args) {
43     System.out.println("the distance between "
44                        + from.getName() + " and " + to.getName()
45                        + " is " + distance() + ".");
46   }
47   public Meter(Shape a, Shape b) {
48     from = a; to = b;
49     from.renamed += changed; from.moved += changed;
50     to.renamed += changed; to.moved += changed;
51     changed += print;
52   }
53 }
```

Listing 4.6: The shape and meter example in DominoJ

```
1  public class Shape {
2    private Shape parent; private String name;
3    public String getName() {
4      if(parent == null)    return name;
5      return parent.getName() + ":" + name;
6    }
7    public void setName(String n) { name = n; }
8    private int x = 0, y = 0;
9    public void setX(int nx) { x = nx; }
10   public void moveX(int dx) { x = x + dx; }
11   public int getX() {
12     if(parent == null)    return x;
13     return parent.getX() + x;
14   }
15   public void setY(int ny) { y = ny; }
16   public void moveY(int dy) { y = y + dy; }
17   public int getY() {
18     if(parent == null)    return y;
19     return parent.getY() + y;
20   }
21   public Shape(String n, Shape p) {
22     parent = p; name = n;
23   }
24 }
25 public class Meter {
26   private Shape from, to;
27   public double distance() {
28     double x2 = Math.pow(to.getX()-from.getX(), 2);
29     double y2 = Math.pow(to.getY()-from.getY(), 2);
30     return Math.sqrt(x2 + y2);
31   }
32   public void print() {
33     System.out.println("the distance between "
34                        + from.getName() + " and " + to.getName()
35                        + " is " + distance() + ".");
36   }
37   public Meter(Shape a, Shape b) {
38     from = a; to = b;
39     {this.print} += this.print();
40   }
41 }
```

Listing 4.7: The shape and meter example in ReactiveDominoJ

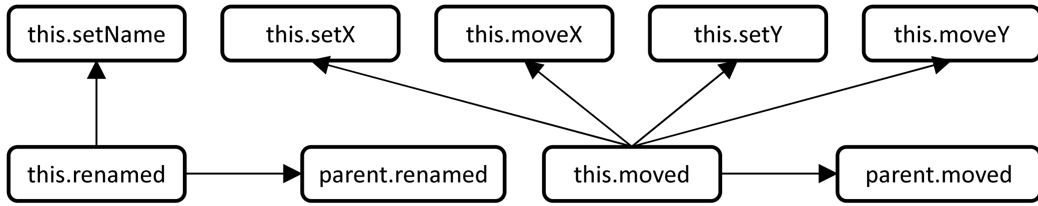

Figure 4.6: The dependency graph for Shape

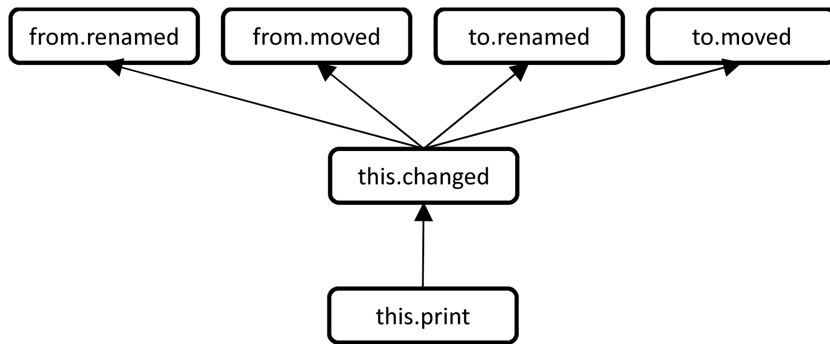

Figure 4.7: The dependency graph for Meter

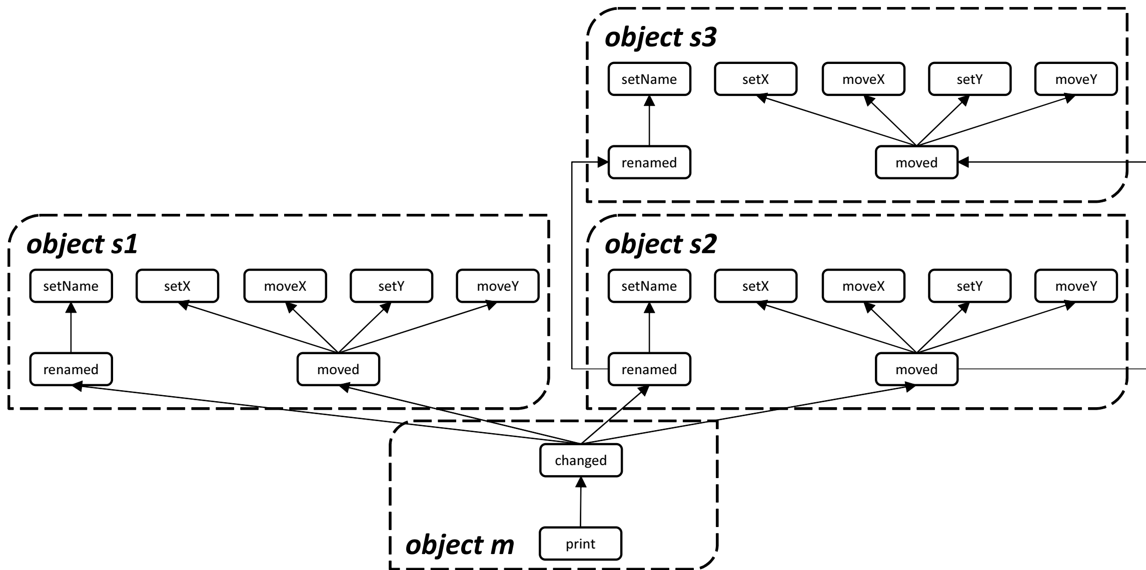

Figure 4.8: The dependency between the objects m, s1, s2, and s3

ated for using the braces operator with the += operator on the method slot update:

```
public void update$helper$after(Closure c) {
  if(a1 != null)  a1.setValue$helper$after(c);
  if(b1 != null)  b1.getValue$helper$after(c);
  if(c1 != null)  c1.getValue$helper$after(c);
}
```

No method slots are selected in this helper method since the fields a1, b1, and c1, which are read by update, are not written by any method slots in the owner object. However, the method slots a1.setValue, b1.getValue, and c1.getValue are called in update, so that the given closure c is passed to the helper methods for them recursively. The body of the helper method setValue$helper$after is empty since no fields are read in setValue of IntCell as shown in Listing 4.3. On the other hand, the helper method getValue$helper$after looks like this:

```
public void getValue$helper$after(Closure c) {
  setValue$after(c);
}
```

the method setValue$after(c) is a method for appending a closure to the array of the method slot setValue, which has been generated in the DominoJ compiler. Note that the helper methods setValue$helper$after and getValue$helper$after are declared in IntCell. After generating the helper methods the compiler transforms the following binding:

```
{this.update} += this.update();
```

into the call to the helper method:

```
this.update$helper$after(c);
```

where c is an anonymous class which extends the Closure interface of DominoJ and calls this.update(). Therefore, at runtime the helper method update$helper$after is called for collecting all method slots and binding the closure c to them. Note that for all subclasses of PlusSheet that override update and all subclasses of IntCell that override getValue and setValue, the compiler generates helper methods for them individually, so that the corresponding helper method can be called on the object according to the actual type of the object; the dynamic method dispatch is preserved. Note that in order to simplify the implementation the ReactiveDominoJ compiler generates helper

```
1  procedure transform(O, M, C) {
2    A = the apparent type of class of O;
3    procedure generate_helper(A, M, C) {
4      S_A‾ = All the super classes of A including A;
5      D = the default closure of M declared in A;
6      f‾ = the fields which are read in D
7             AND declared in any of S_A‾;
8      m‾ = the method slots which write any of f‾
9             AND are declared in any of S_A‾;
10     M'‾ = the method slots called in D
11            AND belong to the objects held in the fields of O;
12     H = method(C) {
13              add C to the array of m‾;
14              execute H'_M(C)‾;
15          };
16     declare H in A;
17   }
18   ∀ subclasses A'‾ of A do generate_helper(A', M, C);
19   replace "{O.M} += C" with "O.H(C)";
20 }
```

Listing 4.8: The event selection in the ReactiveDominoJ compiler

methods for every method slot in every class. Furthermore, the calls to a method slot that is not compiled by ReactiveDominoJ are not recursively investigated.

The pseudocode of how the ReactiveDominoJ compiler generates helper methods is shown in Listing 4.8. What the procedure generate_helper does is the same as the procedure select_events shown in Listing 4.4 except the compiler generates helper methods for all subclasses of the apparent type of the given object[2]. The procedure generate_helper adds (suppose that += operator is used) the given closure to collected method slots immediately rather than returns the set of collected method slots. Therefore, the proper helper method can be selected by the dynamic method dispatch in OOP. As a consequence, if a method slot is both selected in different helper methods, it is difficult to avoid binding to the same method slot twice.

The limitation of current implementation The limitation of this implementation includes endless propagation loops and the changes of objects. Unlike a spreadsheet program, programmers need to be careful with the propagation

[2] Current implementation has a defect on selecting writer method slots through the fields declared in the super classes. A method slot that is declared in the subclass and writes a protected field declared in the super class is not selected by the braces operator. We have the plan to fix this implementation issue in the near future.

loop. In most spreadsheet programs such a propagation loop are usually resolved by evaluating all cells in a default order: left-to-right or top-to-bottom, but programming in languages is different. In a ReactiveDominoJ program, if a field is both read and written in the default closure of a method slot, using the braces operator on that method slot will refer to itself. When programmers bind the method slot itself as a handler, the method slot will trigger itself recursively. For example, suppose there is a method slot calc using three fields this.x, this.y, and this.z:

```
public void calc() {
  this.x = this.y;
  this.y = this.z;
}
```

where the field this.y is both read and written. If programmers bind calc to the braces operator on itself:

```
{this.calc} += this.calc();
```

every time the field this.y is set, the method slot calc will be executed, and then set this.y again. In this case, using local variables instead of fields or splitting calc into two method slots might be possible work-arounds. Another example of endless propagation loop is the circular reference. If there are two method slots as shown below:

```
public void syncXtoY() {
  this.x = this.y;
}
public void syncYtoX() {
  this.y = this.x;
}
```

Binding one of them to the braces operator on itself is safe, but if programmers bind both of them as follows:

```
{this.syncXtoY} += this.syncXtoY();
{this.syncYtoX} += this.syncYtoX();
```

The two statements lead to an endless propagation loop as shown in Figure 4.9: when the field this.y is set, the method slot this.syncXtoY will be executed and the field this.x will be set; when the field this.x is set, the method slot this.syncYtoX will be executed and the field this.y will be set. It

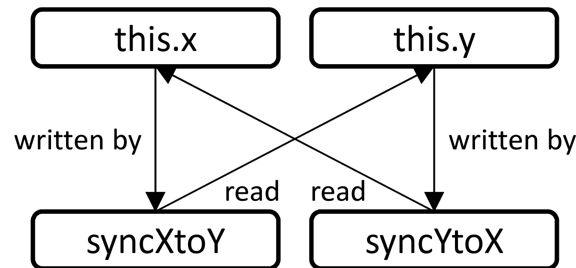

Figure 4.9: An endless propagation loop

is possible to give some compilation warnings against the possibility of propagation loops like what a modern spreadsheet program does. However, the loops in a program cannot be exactly determined at compile time. The endless propagation loop might only happen under certain conditions at runtime since the semantics in ReactiveDominoJ does not care about the conditional branches. Like the recursive call, programmers may explicitly use a propagation loop and then stop it by unbinding the handler when the conditions are satisfied at runtime.

The second limitation of the implementation is the changes of objects. The object referred to by a field might be changed after the binding is performed. In that case, the handler is still bound to the previous object rather than the new one. Such usage is not recommended since it is hard to trace. For example, suppose that there is a setter method slot setB1 for b1 in Listing 4.5 and at first the object held by the field b1 is O_b. Then the method slots selected by the braces operator in Line 13 of Listing 4.5 include the setB1 on p and the setValue on O_b. When the object held by b1 is changed to another object $O_{b'}$ by calling the setB1 on p, the handler update() will be executed and thus has a chance to check if the object held by b1 is the same. However, the handler is still bound to the setValue on O_b rather than the one on $O_{b'}$. A possible work-around is to prepare two additional handlers: one for unbinding the handler update() from O_b before setB1 is called, and one for rebinding the handler update() to $O_{b'}$ after setB1 is called; either by the compiler or by programmers themselves.

A more powerful implementation As we explained in Section 4.3.2 the semantics of the event selection in current version of ReactiveDominoJ only take the default closure into account for the performance concern. In this implementation, only the default closure is investigated for a method slot given to the braces operator. It is possible to investigate all the closures added to

the given method slot at runtime if we implement reflection API for method slots. However, in such an implementation all the selection must be done at runtime and the performance will not be good. Another solution is to generate helper methods for all the method slots that might be added the given method slot. For example, if we use the braces operator on the method slot this.update as follows:

```
{this.update} += this.refresh;
```

and then append a closure calling another method slot logger.debug to this.update somewhere in the program:

```
this.update += logger.debug;
```

It might be possible to find out all the method slots like logger.debug, generate helper methods for them, and prepare a switch for calling every helper method in the event selection for {this.update}. If the above statement is executed, switching on the call to the helper method for logger.debug. Then it is possible to selects the events selected by logger.debug when selecting the events for this.update. In other words, not only the default closure but also the closures calling other method slots in the given method slot are investigated for selecting the events. As a trade-off the whole program must be compiled together to find out all the method slots that might be added to the method slot given to the braces operator; separate compilation is not available. Such an implementation is much powerful since after using the assignment to add the closure calling a method slot, we do not have to use the braces operator for that method slot again as follows:

```
this.update += logger.debug;
  :
{this.update} += this.refresh();
{logger.debug} += this.refresh();
```

Instead, we can:

```
this.update += logger.debug;
  :
{this.update} += this.refresh();
```

In the latter one we do not have to apply the braces operator again for this.refresh. Some readers might notice that investigating all closures in a

method slot can further improve the modularity when using the braces operators along with the event-handler paradigm or the aspect paradigm, though it might be a little slower and disable the separate compilation. In this thesis we only implement the simple one that investigates the default closure for showing such an extension to method slots is possible. Implementing such a powerful one that investigates all the closures in the method slot is included in our future work.

4.4 Discussion

4.4.1 Preliminary microbenchmarks

In order to measure the overheads of selecting events by the braces operator at runtime, we run two preliminary microbenchmarks. The first one is to compare the performance of binding and unbinding a handler to a set of method slots in DominoJ and in ReactiveDominoJ. The second one is to measure the cost of tracing a number of objects before the method slot is selected. We first define a simple class that has a reader method slot and a writer method slot:

```
public class MyInt {
  private int value = 0;
  public int get() { return value;}
  public void set(int v) { value = v; }
}
```

In DominoJ an observer is defined as follows, which binds and unbinds a handler to the writer method slot repeatedly and calculates the average time.

```
public class Observer {
  public void print(int v) {
    System.out.println("print");
  }
  public static void test() {
    MyInt m = new MyInt();
    Observer o = new Observer();
    int N = 1000000;
    long start = System.nanoTime();
    for(int i=0; i<N; i++) {
        m.set += o.print;
```

```
      m.set -= o.print;
    }
    long stop = System.nanoTime();
    System.out.println((stop-start)/N);
  }
  public static void main(String[] args) { test(); }
}
```

Then we increase the number of writer method slots from 1 to 9 by adding new writer method slots to MyInt such as add and lshift. In DominoJ the binding to the writer method slots are manually enumerated, while in ReactiveDominoJ it is always the same:

```
{m.get} += o.print();
{m.get} -= o.print();
```

Note that the handler print in the ReactiveDominoJ version takes no parameter. We run the preliminary microbenchmark on the JVM of OpenJDK 1.7.0_25 and Intel Core i7 (2.67GHz, 4 cores) with 8GB memory. The average time of binding and unbinding a set of method slots in DominoJ and ReactiveDominoJ is shown in Figure 4.10. The overheads of binding and unbinding a method slot in DominoJ and in ReactiveDominoJ are 226ns and 421ns, respectively. The difference is due to when the closure for calling the handler is created. In ReactiveDominoJ the closure must be generated just before appending to the array for the method slot at the left-hand side due to the evaluation of the arguments given to the handler. In DominoJ the closure is prepared in advance and held by a field on the owner object of the handler. Furthermore, in ReactiveDominoJ the closure is passed to the helper method for the reader method, get in this experiment, and then appended to the selected method slots, while in DominoJ the closure is appended to the selected method slots directly.

In the second experiment, MyInt has only one writer method set and it is changed to have a super class Base:

```
public class Base {
  private Base real = null;
  public int get() { return real.get();}
  public Base(Base m) { real = m; }
}
```

Then we modified the body of test so that m will not directly refer to a MyInt object but indirectly through several Base objects and thus the selection by the braces operator will trace a chain of Base objects.

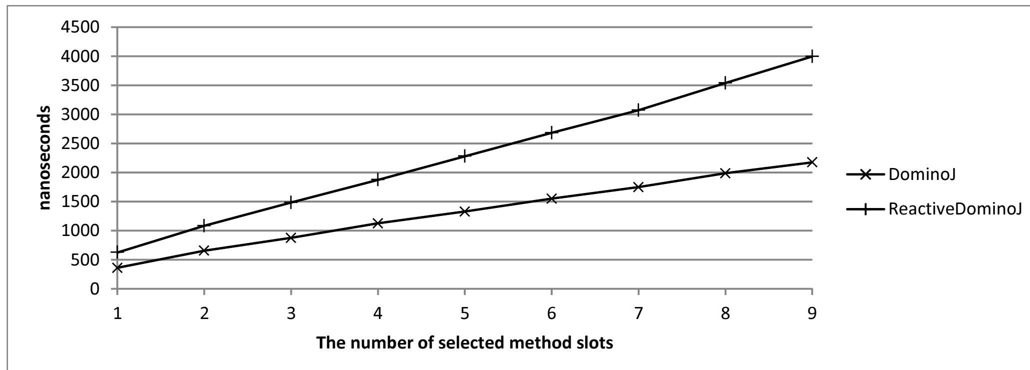

Figure 4.10: The average time of selecting a set of method slots in DominoJ and in ReactiveDominoJ

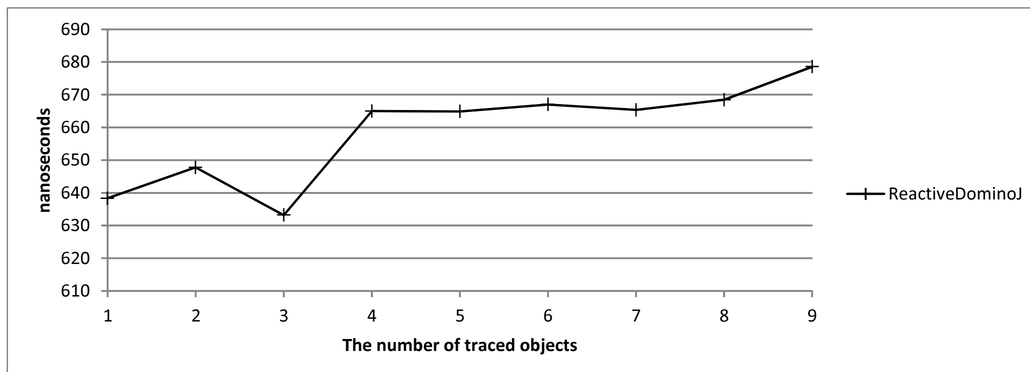

Figure 4.11: The average time of selecting a method slot through a number of objects in ReactiveDominoJ

```
MyInt m3 = new MyInt();
Base  m2 = new Base(m3);
Base  m  = new Base(m2);
```

In this example the number of traced objects is 3. As shown in Figure 4.11 the overheads of tracing are small. Here we only show the data in ReactiveDominoJ since such selection is not possible in DominoJ.

4.4.2 Comparing with DominoJ

Using the braces operator to select events can avoid binding to all events explicitly. The merits of inference-based definitions include writing the depen-

dency among events only once, improving the encapsulation, and simplifying the event propagation.

Write the dependency once The dependency among events in a typical event mechanism are usually described twice. As shown in Listing 4.2, once for calculating values (Line 19) and once for making it automatic (Line 13–14). In other words, for each dependency programmers have to maintain it in two places. In ReactiveDominoJ the bindings for all dependencies in a method slot are reduced to a binding to the higher-level event, so that the number of lines of source code can be smaller. The maintainability can also be improved since we do not have to ensure the consistency between the binding and the calculation. The binding to a method slot with the braces operator is always the same no matter how the calculation is modified.

Improve the encapsulation The inference mechanism in ReactiveDominoJ improves the encapsulation of a class. In Listing 4.5 the fields are hidden from the clients of the PlusSheet object, while the clients can bind a handler to the method slots that might write the fields through the public method slots using the braces operator. In other words, the fields do not have to be public and the clients can be oblivious to the implementation details of PlusSheet.

Simplify the event propagation It is also possible to simplify the event propagation since the handler is directly bound to the selected method slots through the fields behind the higher-level event. Figure 4.12 shows the difference on how to select the method slots from.setX and from.moveX in Listing 4.6 and Listing 4.7. Instead of manually constructing the dependency one by one (solid lines) the braces operator in ReactiveDominoJ automatically selects the method slots by tracing the calls and the fields it eventually depends on (dashed lines). A binding to {this.print} will be interpreted as the bindings to from.setX and from.moveX. On the other hand, in the DominoJ version the bindings behind changed have to be manually constructed and the value changes of the fields are propagated through the events from.moved and this.changed. In other words, the event propagation in a typical event mechanism is reduced to one step while the abstraction is preserved.

4.4.3 Comparing with AspectJ

The inference-based definition supported by the braces operator in ReactiveDominoJ is also a kind of predicate-based definition. However, the meaning of the braces operator is differnt from the predicate pointcuts in AspectJ.

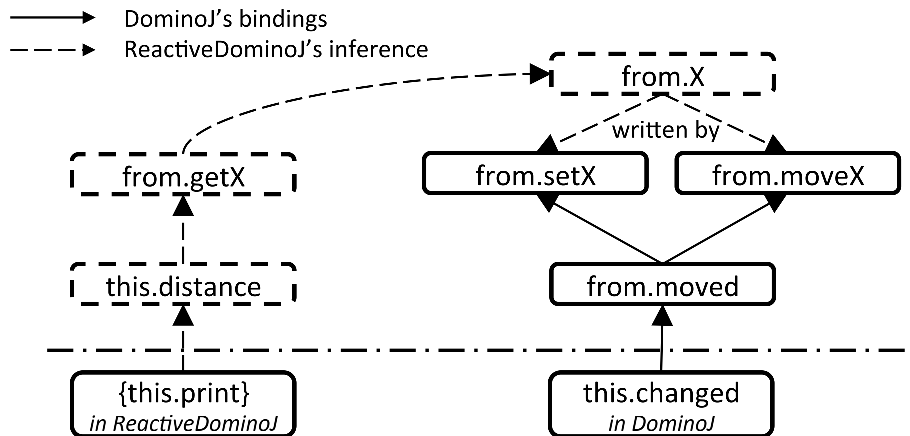

Figure 4.12: The braces operator infers the fields it eventually depends on and selects their writer method slots

We can regard event composition as merging multiple sources to a sink; the sources are lower-level events and the sink is the higher-level event. In that sense, the predicate pointcuts in AspectJ is used to describe the sources while the braces operator in ReactiveDominoJ takes a sink as the hint to infer the sources. For example, a statement in the reactive paradigm that prints the values of two behaviors x and y (we simply assume that they are variables in an Point object due to the difference between functional-reactive languages and OO languages) can be rewritten using AspectJ as shown in Listing 4.9. The write access to the fields x and y are selected by set pointcut. Then predicate pointcuts such as target and within can be used to further select the join points. The ReactiveDominoJ version is shown in Listing 4.10. We can move advice body to a method slot and use the braces operator to bind the method slot itself as the handler. The inference-based definition in ReactiveDominoJ is very implicit and needs a method slot body for inferring which events to select.

4.4.4 For the reactive paradigm

Behaviors The most important difference between the reactive paradigm and the event-handler paradigm is that the reactive paradigm supports behaviors, which are used to express data flows without any bindings for updating values. The behaviors work well in functional programming languages since they are given to a function as the input and affect the output. When the

```
1  public aspect Update {
2      :
3    pointcut moved(Point p):
4      (set(int Point.x) || set(int Point.y))
5      && target(p);
6    after(Point p): moved(p) {
7      System.out.println("x=" + p.getX() + ", y=" + p.getY());
8    }
9  }
```

Listing 4.9: Selecting events for printing two fields in AspectJ

```
1  public class Update {
2    Point p;
3      :
4    public Update() {
5        :
6      {this.print} += this.print();
7    }
8    public void print() {
9      System.out.println("x=" + p.getX() + ", y=" + p.getY());
10   }
11 }
```

Listing 4.10: Selecting events for printing two fields in ReactiveDominoJ

values of the behaviors change, any expressions using the behaviors will be re-evaluated and the output will be changed. We can regard the behaviors as a way to select the expressions for re-evaluation. Similarly, the inference-based definition in ReactiveDominoJ selects the expressions for re-evaluation according to the states of objects. For example, suppose that we have the following assignment in a functional-reactive language:

```
a = b + c;
```

where a, b, and c are behaviors. The value of a will be assigned again when the value of b or c is changed. This means that this assignment is re-evaluated when the values of the behaviors it reads are changed. In other words, in the reactive paradigm a behavior selects the expressions reading it and re-evaluates the selected expressions when its value changes. In ReactiveDominoJ, a field read in the default closure of a method slot can be regarded as a behavior if we apply that method slot to the braces operator for defing a higher-level event. For example, there are three fields, a, b, and c, in the following method slot:

```
public void updateA() {
  a = b + c;
}
```

then the following binding:

```
{this.updateA} += this.updateA();
```

specifies that setting a value to any of the fields read in the default closure of updateA will cause the evaluation of the default closure of updateA again; even the value is not changed. Similar to what the reactive paradigm does, the braces operator selects that expression and re-evaluates it when the values of the fields it reads change.

However, the effective scope of the "behaviors" in ReactiveDominoJ is different from the behaviors in the reactive paradigm. In a functional-reactive language a behavior is effective in the whole program. It means that all expressions reading the behavior are selected and involved in the re-evaluation caused by the value change of that behavior. In ReactiveDominoJ only the expressions in the default closure of the method slot bound to itself by the braces operator are selected and re-evaluated for the value changes of the read fields. In the example updateA shown above, even we bind the method slot this.updateA to {this.updateA}, expressions using b or c out of this.updateA are not re-evaluated when the value of b or c is changed. This design takes

efforts to control when a field should be a "behavior". However, it also makes the design intention simple and clear. Currently all functional-reactive languages demonstrate their usage in functional programming with small pieces of code. Applying the behaviors in the reactive paradigm to imperative programming languages such as Java might make a program complicated since fields are not local variables and side effects are massively used. Unlike the spreadsheet programs, it is hard to define an update order for all expressions using the same behavior. Arbitrarily updating all expressions using the same field might cause unnecessary propagation or race condition and then out of control. In ReactiveDominoJ the "behaviors" must be explicitly specified and only effective in a limited scope.

Snapshot and lifting Although ReactiveDominoJ needs a binding to let the fields work as "behaviors", the transformation between behaviors and constant values can be avoided. Several functional-reactive languages such as Fran [24] replace constant values with behaviors directly, and other functional-reactive languages such as Flapjax introduce behaviors as a different type of variables from constant values. In both the two styles of functional-reactive languages, the transformation, in other words getting a constant value (snapshot) of a behavior and lifting a constant value to a behavior, are necessary. On the other hand, in ReactiveDominoJ all fields hold constant values and work like "behaviors" only when we specify the binding. The transformation between behaviors and constant values is not necessary.

The Time behavior A notable behavior in the reactive paradigm is the time behavior. In Fran the time behavior is given by a keyword and directly supported by the language, while in Flapjax it is given by a class and supported by the library. The time behavior is a first-class object and can be given to other functions for selecting the expressions in the function for re-evaluation. In the current version of the ReactiveDominoJ compiler, the time behavior is not directly available. It might be possible to introduce an inference rule that selects all expressions involving the call to getting the system time. However, the time behavior is not absolutely necessary in Java. Programmers may use the timer and the task as what they do with a typical event mechanism though the code is more verbose. For example, giving the time behavior to a function printCurrentTime in Flapjax:

```
printCurrentTime(timerB(100));
```

where timerB is the time behavior provided by the Flapjax library, which holds the value of current time in milliseconds and takes an argument as the

interval of updating its value, also in milliseconds. In ReactiveDominoJ, we can use Timer and TimerTask in the Java library to do the same thing:

```
TimerTask task = new TimerTask() {
  public void run() {
    printCurrentTime(System.currentTimeMillis());
  }
};
new Timer().schedule(task, 0, 100);
```

Both of them cause the execution of printCurrentTime function with the current time every 100 milliseconds.

Event streams Another important concept in the reactive paradigm is the event stream. Event streams can be naturally represented by a sequence of events from the same source. In Flapjax we can use the library function mergeE to combine two event streams such as moveE and dropE, which refer to mouse movement and mouse dropping respectively, to a new event stream outputE, which fires when an event from either of them occurs. In DominoJ and ReactiveDominoJ, merging the two event streams can be described by the OR composition of two events moved and dropped:

```
moved += output;
dropped += output;
```

here we use += operator twice to compose the two events. Whenever moved or dropped is triggered, the composed event output will be triggered.

Loops and states In the reactive paradigm a behavior is not allowed to refer to itself for imperatively stepping its state, for example:

```
c = c + 1
```

If c is a behavior, the meaning is weird. However, holding a state and stepping forward is still necessary to a program, for example implementing a counter. In Elm automatons can hold a state and produce the next state according to the current state. In Elm an automaton with state can be created as follows:

```
count = state 0 (\\_ c -> c+1)
```

The state has an initial value 0, and steps forward when the input is given. Similarly, in Flapjax event streams can be used to hold snapshots for behaviors. For example, counting the number of events that occur in Flapjax:

```
mouseMoveE.collectE(0, function(_, count) {
                         return count+1; });
```

Then the variable count will be increased according its previous value when its input is given, i.e. the event stream mouseMoveE happens. The event streams in the reactive paradigm are mapped to the events in DominoJ/ReactiveDominoJ, i.e. the calls to method slots, and the above example can be rewritten as:

```
// in a certain class
int state_collect = 0;
int collect() { this.state_collect += 1; }
// suppose we have a system library s
s.mouseMove += this.count;
```

Then every time the call to s.mouseMove happens, this.count is called to step the state forward.

Re-evaluation There is an important difference between ReactiveDominoJ and functional-reactive languages: which parts of an expression should be re-evaluated when a behavior in the expression changes its value. For example in Flapjax:

```
x = a + b;
y = c + d;
z = x + y;
```

According to the semantics of Flapjax y should be not evaluated again when x is updated. However, in ReactiveDominoJ all expressions in the method slot are evaluated again. Suppose that the above code are described in a method slot update, then the braces operator can be used to make the fields look like “behaviors”:

```
{this.update} += this.update();
```

In other words, the unit of re-evaluation in the reactive paradigm is a behavior while the one in ReactiveDominoJ is the whole block of the closure specified at the right-hand side of the operator. Using fields and getters/setters could be a work-around for this drawback. For example, the above code can be modified as follows:

```
// in a certain class
int a=0, b=0, c=0, d=0;
int x = 0;
int getX() { return x; }
void setX() { x = a + b; }
int y = 0;
int getY() { return y;}
void setY() { y = c + d; }
int z = 0;
int update() { z = getX() + getY(); }

// in a certain method
{this.update} += this.update();
```

where this.setX and this.setY will be selected by the braces operator. Although the body of this.getY will still be evaluated again when this.setX is called, it simply returns the value of y rather than re-calculate c + d. This work-around might be helpful to avoid re-evaluation. However, it is not a good solution for all cases since it forces programmers to break an expressions into several method slots.

4.5 Related work

Like other research on complex event processing (CEP) [47, 26, 14], EventJava [27] focuses on the events in distributed systems. The event methods proposed by EventJava allow programmers to define an event with constraints on event attributes through predicates. The event defined by an event method may be triggered for unicast or broadcast. When the event is triggered, the statements in the body of that event method will be executed or not according to the predicates. Complex events are also supported by declaring multiple event method headers in an event method. However, event methods are some sort of primitive events since they are directly triggered by calls rather than other events. Furthermore, the body of the event method is the only handler for the event. On the other hand, the inference-based definition of higher-level events in ReactiveDominoJ can be composed of either higher-level events or primitive events and may be bound to other events. Another difference is that the inference-based definition in ReactiveDominoJ is more implicit than the predicates in EventJava.

Functional-reactive languages such as Fran [24] make the expressions in a reactive program more declarative and look like data flows. Several proposals

such as FrTime [11] bring the concept of the reactive paradigm to different languages and discuss the evaluation and semantics. However, most of them are based on functional programming while the languages that we want to integrate into such as Java are imperative style. This differences between functional programming and imperative programming make it hard to introduce the reactive paradigm to OO languages and use with existing OO libraries. There is also a question that it can be used to design a complex system since behaviors are primitive types and the results of functions are immutable. The inference-based definition in ReactiveDominoJ is an extension to an event mechanism and thus fully compatible with OO design.

On the other hand, several research activities [53, 63, 73, 13] are devoted to bring the reactive paradigm to OO languages. Flapjax [53], a functional-reactive language designed for Web applications, brings behaviors and event streams to JavaScript. Flapjax supports behaviors as special variables at library level, and refines events as infinite streams of discrete events. The model of the reactive paradigm is gracefully introduced to JavaScript for Web UI. The code scattering of the initialization and updates for the data in Ajax applications can be eliminated and the control flows are more clear. However, obtaining and giving the behaviors heavily rely on the library. Furthermore, it is also a question that Flapjax can help complex OO design since all the behaviors are supplied by the built-in library. In ReactiveDominoJ it is possible to use fields as "behaviors" in a specified scope. Frappé [13], an implementation of the reactive paradigm for Java, allows to use a Java Beans component as a source or a sink for event streams. Frappé makes it possible to use the reactive paradigm with Java Beans by integrating the event/behavior model in the reactive paradigm with the event/property model in Java Beans. However, the support of Frappé is at library level rather than language level, and limited to Java Beans. As a result, writing a program with Frappé cannot avoid mixing the style of the reactive paradigm with imperative style. Furthermore, the transformation between behaviors and constant values is necessary as well.

Data-flow languages [87, 9, 6, 4] such as Lucid [87] give good abstraction for passing time-varying values between components. A flow can be established by connecting components, and the input and output of a component is described as signals, which are similar to the behaviors in the reactive paradigm. An important difference from the reactive paradigm is that the time is not an explicit value. The concept of data flows is close to hardware design and thus several languages such as Verilog and VHDL are proposed for specific domains. Furthermore, some data-flow languages such as LabVIEW/G [46] are designed with a graphical developement environment. However, using data-flow languages to write a complicated program

is not as easy as using OO languages. SuperGlue [52] proposes a language to assemble components, where the signals represent the states in a component. The key difference between SuperGlue from other data-flow languages is that the signals are typed as in OO languages. SuperGlue elegantly eliminates the glue code for connecting components in the model-view-controller architecture but imperative languages such as Java are still necessary inside a component.

Other related work includes the libraries developed for reactive support in existing OO languages. Scala.React [49] is a library-based reactive data-flow extention to Scala. In [48] the drawbacks of using the Observer pattern [30] are discussed and the concept of reactors is proposed with the behaviors and event streams in the reactive paradigm. The Reactive Extensions for .NET [59] is a library for writing asynchronous and event-based programs with LINQ. It makes the search in LINQ asynchronous by using IObservable for collections, so that the UI thread may subscribe and return immediately rather than wait for the result. There are also other implementations on reactive collections, which automatically update their clients such as queries and UI components when their elements are modified. These frameworks greatly help programming in existing OO languages but what they focus is the support for the collections and different from the event approach in ReactiveDominoJ.

The inference based on the value changes of fields in ReactiveDominoJ might remind readers of the field pointcuts in AspectJ [77]. Furthermore, the Observer pattern can be modularized by aspects as discussed in [34]. AspectJ provides get and set pointcuts for picking up access to specified fields as join points. The pointcuts for fields are very powerful since they can filter fields by a pattern and used with other pointcuts. Attaching an advice to a set pointcut for a field can be regarded as binding a handler to the value change of that field. However, selecting the access to the fields on a specific object in AspectJ is not easy since AspectJ is designed for resolving crosscutting concerns. Furthermore, with the field pointcuts in AspectJ we still have to enumerate all fields we need for a handler by ourselves.

4.6 Summary

We demonstrated how to add a small extension to *method slots* for supporting one more paradigm. The process of extending *method slots* can be summaried as follows. First, considering the paradigm we want to support from the viewpoint of a paradigm that has been supported by *method slots*. Second, finding out the major difference between the two paradigms. Third, analyzing

why the existing solutions cannot do the same thing in the paradigm we want to support. Then finally we can know which operators we should add for extending *method slots* to support that paradigm.

In this chapter we extended *method slots* to support the reactive paradigm by proposing only one more operator. We first compared the reactive paradigm with the event-handler paradigm and pointed out the major difference between them is that the event composition in the reactive paradigm is much implicit and automatic. Why existing event mechanisms cannot do the same thing as the reactive paradigm can be due to the lack of inference rules for event composition. To overcome this limitation we proposed a small extension named *ReactiveDominoJ*, which adds the braces operator to automatically select events for a higher-level event by inference. The semantics of such a new kind of predicate-based definition for event composition, *inference-based definition*, was presented along with code examples and a feasible implementation. The limitations of using *ReactiveDominoJ* for the reactive paradigm were discussed as well.

Chapter 5

Conclusions

This thesis successfully advanced the approach of generic constructs to cover four paradigms. In existing research activities devoted to this approach the number of supported paradigms is limited to only two. This thesis also showed the possibility of supporting OOP, the event-handler paradigm, the aspect paradigm, and the reactive paradigm by only a single language construct.

How to support the implementation of programming paradigms is an important issue in language design since programming paradigms are so important. Better paradigm support can greatly improve the quality of a program such as code modularity, which also implies that the code is easy to maintain, resue, and migrate. This thesis took the approach of generic constructs to suport the implementation of paradigms since this approach can provide better support for multiple paradigms in a language while keep the language design simple. The theis pointed out the insufficiency of the state of the arts of this approach is that the number of supported paradigms is quite limited due to the inflexibility of existing generic constructs. To overcome the problem this thesis presented a new generic construct that is very simple and flexible enough to support more than two paradigms.

To develop a flexible generic construct that can support more than two paradigms, this thesis first focused on three paradigms: OOP, the event-handler paradigm, and the aspect paradigm. They are very important and widely used in the real world, and there are several similarities among their

implementations. Although the event-handler paradigm and the aspect paradigm are developed for resolving different issues, they all implicitly lead to the execution of several pieces of code according to a certain rule. An event implicitly leads the execution of handlers bound to it according to the event declaration, and a join point implicitly leads to the execution of advices attached to it according to the pointcut declaration. On the other hand, methods must be explicitly called and contain only one piece of code. However, no one could notice the similarities and develop a flexible generic construct for the three paradigms based on the common ground among their implementations before this thesis. This observation led us to develop a new language construct that can lead to the execution of multiple pieces of code in either an explicit way or an implicit way.

This thesis extended the methods in prototype-based OO languages such as JavaScript to a new generic construct, *method slots*, which is an object's property and can hold more than one function closure. *Method slots* can be used as methods, events, and advices. The assignment operators for *method slots* can manipulate the closures in a *method slot* to lead the execution of other *method slots*, and thus can be shared among OOP, the event-handler paradigm, and the aspect paradigm. This thesis also discussed the differences between the event-handler paradigm and the aspect paradigm to clarify the essentials of them, for example how the code pieces are selected. The coverage of expressive ability of *method slots* was also presented and the result showed that most functionalities can be covered. For OOP the GoF design patterns were rewritten using *method slots* to show that *method slots* can replace the methods and provide better modularity. The feasibility of *method slots* was shown by the compiler implementation, and the overheads were measured by running benchmarks. How *method slots* can be used in practice was showed by case studies as well.

To augment the flexibility of *method slots*, this thesis further demonstrated how to extend *method slots* to support more paradigms by taking the example of the reactive paradigm. The extension to *method slots* is very small and gives only one more operator. In this thesis the process of extending *method slots* was explained in detail. In order to find out how to support the reactive paradigm by *method slots*, first we compared the similarities between the reactive paradigm and the most similar one in the three supported paradigms: the event-handler paradigm. Then we analyzed what the most important difference between them is, and pointed out existing event mechanisms lack inference rules for event composition. To bring the inference rule to the event-handler paradigm, we proposed a new kind of predicate-based definition for event composition, *inference-based definition*, to automatically select lower-level events for a higher-level event by inference as what the reac-

tive paradigm does. This small extension to *method slots* was also presented with a feasible compiler implementation and microbenchmarks. The limitations of emulating the reactive paradigm by the event-handler paradigm using the *inference-based definition* were also discussed.

Contributions

The contributions of this thesis are summarized as follows:

- This thesis showed the possibility of supporting more than two paradigms by generic constructs. Although the generic constructs proposed in existing research activities are not sufficient to support more than two paradigms, the new generic construct proposed in this thesis successfully supports the implementations of four paradigms. We showed that the new generic construct, *method slots*, is flexible and easy to extend for different paradigms.

- The language support proposed by this thesis covers the four paradigms: OOP, the event-handler paradigm, the aspect paradigm, and the reactive paradigm; no one covers all of them so far as we know. The four paradigms are very important paradigms in both academia and industry, and attract a lot of interest, especially from OOP community. This thesis made it possible to support the four paradigms by only a single language construct. An OO language can simply replace methods with *method slots* to support the four paradigms.

- This thesis revealed the similarities among the implementations of the four paradigms, which were not recognized before this thesis since these paradigms are developed for different purposes and their constructs look quite different. This thesis helped to clarify the understanding of the four paradigms from the viewpoint of implementation and pointed out the differences among the four paradigms. The code examples written in different languages in this thesis not only showed the coverage of expressive ability of *method slots* but also clarified the essentials of the four paradigms.

Future Directions

Possible future directions of this thesis include:

Supporting more and more paradigms Since method slots are flexible and easy to extend, we believe that it is possible to support more paradigms by small extensions. How the support to a paradigm can be added has been shown in Chapter 4. On the other hand, how method slots can be used along with other generic constructs might be worth studying as well.

Metaprogramming for method slots Allowing to view and modify the status of method slots might be helpful in programming. For example, providing a reflection library for method slots in DominoJ. In the current version of DominoJ the manipulation of method slots is limited to add or remove a closure. Allowing to get the information of a method slot such as its type and the number of the closures in it could help programmers to debug. Making it possible to search and invoke a method slot by given properties could also exploit the capability of method slots.

Method slots in parallel computing Introducing method slots in parallel computing might be interesting. Currently the closures in a method slot are always executed in order since the execution order is important to the paradigms discussed in this thesis. If we consider using method slots in parallel computing, the clousres in a method slot could be executed on different cores at the same time for optimizing the performance. The closures in a method slot could be all the concurrent tasks for an event, which are scheduled and executed according to the hardware conditions for getting the maximum throughput.

Method slots in a prototype-based OO language This thesis showed how to use method slots along with the inheritance in a class-based OO language such as Java. Implementing method slots in a prototype-based OO language such as JavaScript could be an execise. The semantics could be further simplified since no classes and the closures are usually supported directly. On the other hand, the potential capability of method slots that was not demonstrated by DominoJ might be found.

Further improving the DominoJ/ReactiveDominoJ compiler Although the overheads of current version of the DominoJ/ReactiveDominoJ compiler are not large, we are trying to make the performance better. Several ideas on the static analysis might be applied to avoid dynamically adding the closures if we disable the separate compilation. For example, copying the body of the method slots at the right-hand side of the assignment operators to the body of the ones at the left-hand side instead of really creating closure-like objects

at runtime. Investigating all the closures rather than only the default closure in a method slot for the braces operator (as we discussed in Section 4.3.3) is also an interesting topic.

A source-to-bytecode compiler for DominoJ The current version of the DominoJ compiler is a source-to-source compiler, which is fully compatible with plain Java while cannot directly benefit from Java bytecode. Directly transforming DominoJ source code into Java bytecode might improve the performance; the output is not limited to Java specifications and new JVM instructions for non-Java languages such as invokedynamic could also be utilized.

Bibliography

[1] Apple Inc. Cocoa Frameworks. `https://developer.apple.com/technologies/mac/cocoa.html`.

[2] Apple Inc. Objective-C. `https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html`.

[3] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. Transactions on aspect-oriented software development i. chapter An Overview of CaesarJ, pages 135–173. Springer-Verlag, Berlin, Heidelberg, 2006.

[4] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of Computer Programming*, 16(2):103 – 149, 1991.

[5] Lodewijk Bergmans and Mehmet Aksit. Composing crosscutting concerns using composition filters. *Commun. ACM*, 44(10):51–57, October 2001.

[6] Gerard Berry, Georges Gonthier, Ard Berry Georges Gonthier, and Place Sophie Laltte. The Esterel synchronous programming language: Design, semantics, implementation, 1992.

[7] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel,

A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA'06, pages 169–190, New York, NY, USA, October 2006. ACM Press.

[8] Howard I. Cannon. Flavors: A non-hierarchical approach to object-oriented programming. Symbolics Inc., 1982.

[9] P. Caspi, D. Pilaud, N. Halbwachs, and J. A. Plaice. LUSTRE: a declarative language for real-time programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL'87, pages 178–188, New York, NY, USA, 1987. ACM.

[10] Shigeru Chiba, Atsushi Igarashi, and Salikh Zakirov. Mostly modular compilation of crosscutting concerns by contextual predicate dispatch. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA'10, pages 539–554, New York, NY, USA, 2010. ACM.

[11] Gregory H. Cooper and Shriram Krishnamurthi. Embedding dynamic dataflow in a call-by-value language. In *In European Symposium on Programming*, ESOP'06, pages 294–308, 2006.

[12] Pascal Costanza and Robert Hirschfeld. Language constructs for context-oriented programming: An overview of ContextL. In *Proceedings of the 2005 Symposium on Dynamic Languages*, DLS'05, pages 1–10, New York, NY, USA, 2005. ACM.

[13] Antony Courtney. Frappé: Functional reactive programming in Java. In *In Proceedings of Symposium on Practical Aspects of Declarative Languages. ACM*, PADL'01, pages 29–44. Springer-Verlag, 2001.

[14] Gianpaolo Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Comput. Surv.*, 44(3):15:1–15:62, June 2012.

[15] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for GUIs. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, PLDI'13, pages 411–422, New York, NY, USA, 2013. ACM.

[16] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured Programming*. Academic Press Ltd., London, UK, UK, 1972.

[17] Ole-Johan Dahl. *SIMULA 67 Common Base Language, (Norwegian Computing Center. Publication)*. 1968.

[18] Ole-Johan Dahl and Kristen Nygaard. SIMULA: An ALGOL-based simulation language. *Commun. ACM*, 9(9):671–678, September 1966.

[19] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, and Wolfgang De Meuter. Ambient-oriented programming in AmbientTalk. In *In Proceedings of the 20th European Conference on Object-oriented Programming*, ECOOP'06, pages 230–254. Springer, 2006.

[20] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An overview. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP'87, pages 151–170, London, UK, UK, 1987. Springer-Verlag.

[21] Digia Plc. The Qt Project.
`http://qt-project.org/`.

[22] Sophia Drossopoulou, Ferruccio Damiani, Dezani Dezani-Ciancaglini, and Paola Giannini. Fickle: Dynamic object re-classification. In *15 European Conference on Object-Oriented Programming*, ECOOP'01, pages 130–149, June 2001.

[23] Torbjörn Ekman and Görel Hedin. The JastAdd extensible Java compiler. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA'07, pages 1–18, New York, NY, USA, 2007. ACM.

[24] Conal Elliott and Paul Hudak. Functional reactive animation. In *Proceedings of the second ACM SIGPLAN international conference on Functional programming*, ICFP'97, pages 263–273, New York, NY, USA, 1997. ACM.

[25] Michael Ernst, Craig Kaplan, and Craid Chambers. Predicate dispatching: A unified theory of dispatch. In *Proceedings of the 12nd European Conference on Object-Oriented Programming*, ECOOP'98, pages 186–211. Springer-Verlag, 1998.

[26] Opher Etzion and Peter Niblett. *Event Processing in Action*. Manning Publications Co., Greenwich, CT, USA, 1st edition, 2010.

[27] Patrick Th. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In *Proceedings of the 23rd European conference on Object-Oriented Programming*, ECOOP'09, pages 570–594, 2009.

[28] Gustav Evertsson. Tetris in AspectJ. `http://www.guzzzt.com/coding/aspecttetris.shtml`.

[29] Robert W. Floyd. The paradigms of programming. *Commun. ACM*, 22(8):455–460, August 1979.

[30] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[31] D. Garlan, S. Jha, D. Notkin, and J. Dingel. Reasoning about implicit invocation. In *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT'98/FSE-6, pages 209–221, New York, NY, USA, 1998. ACM.

[32] Vaidas Gasiunas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD'11, pages 227–240, New York, NY, USA, 2011. ACM.

[33] Adele Goldberg and David Robson. *Smalltalk-80: the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[34] Jan Hannemann and Gregor Kiczales. Design pattern implementation in Java and AspectJ. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA'02, pages 161–173, New York, NY, USA, 2002. ACM.

[35] Michael Haupt and Hans Schippers. A machine model for aspect-oriented programming. In *Proceedings of the 21st European conference on Object-Oriented Programming*, ECOOP'07, pages 501–524, Berlin, Heidelberg, 2007. Springer-Verlag.

[36] Wilke Havinga, Lodewijk Bergmans, and Mehmet 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*, AOSD'10, pages 145–156, New York, NY, USA, 2010. ACM.

[37] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology, March-April 2008, ETH Zurich*, 7(3):125–151, 2008.

[38] Adrian Holzer, Lukasz Ziarek, K.R. Jayaram, and Patrick Eugster. Putting events in context: aspects for event-based distributed programming. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD'11, pages 241–252, New York, NY, USA, 2011. ACM.

[39] JHotDraw as Open-Source Project.
`http://www.jhotdraw.org/`.

[40] Tetsuo Kamina, Tomoyuki Aotani, and Hidehiko Masuhara. EventCJ: a context-oriented programming language with declarative event-based context transition. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD'11, pages 253–264, New York, NY, USA, 2011. ACM.

[41] Christian Kästner, Sven Apel, and Klaus Ostermann. The road to feature modularity? In *Proceedings of the 15th International Software Product Line Conference, Volume 2*, SPLC'11, pages 5:1–5:8, New York, NY, USA, 2011. ACM.

[42] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP'01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[43] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the 11st European Conference on Object-Oriented Programming*, ECOOP'97. Springer-Verlag, 1997.

[44] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th international conference on Software engineering*, ICSE'05, pages 49–58, New York, NY, USA, 2005. ACM.

[45] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *Proceedings of the 19th European conference on Object-Oriented Programming*, ECOOP'05, pages 195–213, Berlin, Heidelberg, 2005. Springer-Verlag.

[46] LabVIEW System Design Software.
http://www.ni.com/labview/.

[47] David C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[48] Ingo Maier and Martin Odersky. Deprecating the Observer Pattern with Scala.react. Technical report, 2012.

[49] Ingo Maier and Martin Odersky. Higher-order reactive programming with incremental lists. In *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, pages 707–731, Berlin, Heidelberg, 2013. Springer-Verlag.

[50] Alessandro Margara and Guido Salvaneschi. Ways to react: Comparing reactive languages and complex event processing. In *Workshop on Reactivity, Events and Modularity 2013*, REM'13, 2013.

[51] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A fine-grained join point model for more reusable aspects. In *The Fourth ASIAN Symposium on Programming Languages and Systems*, APLAS'06, pages 131–147, 2006.

[52] Sean McDirmid and Wilson C. Hsieh. SuperGlue: component programming with object-oriented signals. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 206–229, Berlin, Heidelberg, 2006. Springer-Verlag.

[53] Leo A. Meyerovich, Arjun Guha, Jacob Baskin, Gregory H. Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: a programming language for Ajax applications. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA'09, pages 1–20, New York, NY, USA, 2009. ACM.

[54] Microsoft Corporation. C# language specification.

[55] Microsoft Corporation. C# Tutorials: Delegates Tutorial.
http://msdn.microsoft.com/en-us/library/aa288459(v=vs.71).aspx.

[56] Microsoft Corporation. C# Tutorials: Events Tutorial.
http://msdn.microsoft.com/en-us/library/aa645739(v=vs.71).aspx.

[57] Microsoft Corporation. Messages and Message Queues. http://msdn.microsoft.com/en-us/library/ms632590(VS.85).aspx.

[58] Microsoft Corporation. Microsoft Foundation Class. http://msdn.microsoft.com/en-us/library/hh967573.aspx.

[59] Microsoft Corporation. Reactive Extensions for .NET (Rx). http://msdn.microsoft.com/en-us/data/gg577609.

[60] Microsoft Corporation. Windows API. http://msdn.microsoft.com/en-us/library/cc433218.aspx.

[61] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Haskell'02, pages 51–64, New York, NY, USA, 2002. ACM.

[62] Angel Núñez, Jacques Noyé, and Vaidas Gasiūnas. Declarative definition of contexts with polymorphic events. In *International Workshop on Context-Oriented Programming*, COP'09, pages 2:1–2:6, New York, NY, USA, 2009. ACM.

[63] Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, and Ted Kaehler. KScript and KSWorld: A time-aware and mostly declarative language and interactive GUI framework. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! '13, pages 117–134, New York, NY, USA, 2013. ACM.

[64] Oracle Corporation. http://www.oracle.com/technetwork/java/index.html.

[65] Oracle Corporation. Abstract Window Toolkit. http://docs.oracle.com/javase/jp/7/api/java/awt/package-summary.html.

[66] Oracle Corporation. OpenJDK: Project Lambda. http://openjdk.java.net/projects/lambda/.

[67] Oracle Corporation. Swing. http://docs.oracle.com/javase/jp/7/api/javax/swing/package-summary.html.

[68] Doug Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st international conference on Aspect-oriented software development*, AOSD'02, pages 56–64, New York, NY, USA, 2002. ACM.

[69] Christian Prehofer. Feature-oriented programming: A fresh look at objects. pages 419–443. Springer, 1997.

[70] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In *Proceedings of the 22nd European conference on Object-Oriented Programming*, ECOOP'08, pages 155–179, 2008.

[71] Red Hat. JBoss Application Server.
`http://www.jboss.org/jbossas`.

[72] Hidehiko Masuhara Robert Hirschfeld and Atsushi Igarashi. L – context-oriented programming with only layers. In *5th International Workshop on Context-Oriented Programming*, COP'13. ACM, 2013.

[73] Guido Salvaneschi, Gerold Hintz, and Mira Mezini. REScala: Bridging Between Object-oriented and Functional Style in Reactive Applications. In *Modularity '14 - 13th International Conference on Modularity - 2014*. ACM Press, April 2014.

[74] Guido Salvaneschi and Mira Mezini. Reactive behavior in object-oriented applications: an analysis and a research roadmap. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD'13, pages 37–48, New York, NY, USA, 2013. ACM.

[75] Hans Schippers, Dirk Janssens, Michael Haupt, and Robert Hirschfeld. Delegation-based semantics for modularizing crosscutting concerns. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA'08, pages 525–542, New York, NY, USA, 2008. ACM.

[76] Randall B. Smith and David Ungar. Programming as an experience: The inspiration for Self. In *Proceedings of the 9th European Conference on Object-Oriented Programming*, ECOOP'95, pages 303–330, London, UK, UK, 1995. Springer-Verlag.

[77] The AspectJ Project.
`http://www.eclipse.org/aspectj/`.

[78] The Boost Project. Boost.Signals.
`http://www.boost.org/libs/signals/`.

[79] The C++ Standards Committee.
`http://www.open-std.org/jtc1/sc22/wg21/`.

[80] The Eclipse Foundation. SWT: The Standard Widget Toolkit.
`http://www.eclipse.org/swt/`.

[81] The GTK+ Team. The GTK+ Project.
`http://www.gtk.org/`.

[82] The Qt Project. Signals & Slots.
`http://qt-project.org/doc/signalsandslots`.

[83] The SAX project. Simple API for XML.
`http://www.saxproject.org/`.

[84] The Self project.
`http://selflanguage.org/`.

[85] The X.Org project. Xlib in X Window System.
`http://www.x.org/`.

[86] VMware. Spring Framework.
`http://projects.spring.io/spring-framework/`.

[87] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[88] Zhanyong Wan and Paul Hudak. Functional reactive programming from first principles. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI'00, pages 242–252, New York, NY, USA, 2000. ACM.

[89] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *Proceedings of the sixth ACM SIGPLAN international conference on Functional programming*, ICFP'01, pages 146–156, New York, NY, USA, 2001. ACM.

[90] Zhanyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. In *Proceedings of the 4th International Symposium on Practical Aspects of Declarative Languages*, PADL'02, pages 155–172, London, UK, UK, 2002. Springer-Verlag.

[91] Jennifer Widom and S. J. Finkelstein. Set-oriented production rules in relational database systems. In *Proc. of the Int'l Conf. on Management of Data*, SIGMOD'90, pages 259–270. ACM Press, 1990.

[92] YungYu Zhuang and Shigeru Chiba. Applying DominoJ to GoF Design Patterns. Technical report, Department of Mathematical and Computing Sciences, Graduate School of Information Science and Engineering, Tokyo Institute of Technology, 2011.

[93] YungYu Zhuang and Shigeru Chiba. Method slots: supporting methods, events, and advices by a single language construct. In *Proceedings of the 12th annual international conference on Aspect-oriented software development*, AOSD'13, pages 197–208, New York, NY, USA, 2013. ACM.