**École Centrale de Nantes** **Université de Nantes** **École des Mines de Nantes**

ÉCOLE DOCTORALE STIM

« SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET MATHÉMATIQUES »

Année 2011



# Un modèle de programmation intégrant classes, événements et aspects

THÈSE DE DOCTORAT

Discipline: Informatique

Spécialité : Informatique

*Présentée*
*et soutenue publiquement par*

**Angel Rodrigo NÚÑEZ LÓPEZ**

*le 29 juin 2011, à l'École des Mines de Nantes,*
*devant le jury ci-dessous*

| | | | |
|---|---|---|---|
| Président | : | Colin DE LA HIGUERA, Professeur | Université de Nantes |
| Rapporteurs | : | Wolfgang DE MEUTER, Professeur | Vrije Universiteit Brussel |
| | | Jacques MALENFANT, Professeur | Université Pierre et Marie Curie |
| Examinateurs | : | Christophe DONY, Professeur | Université de Montpellier-II |
| | | Jean-Claude ROYER, Professeur | École des Mines de Nantes |
| | | Jacques NOYÉ, Maître-Assistant | École des Mines de Nantes |

Équipe d'accueil : ASCOLA - INRIA/EMN, LINA UMR CNRS 6241

Laboratoire : DÉPARTEMENT INFORMATIQUE DE L'ÉCOLE DES MINES DE NANTES
4, rue Alfred Kastler, BP 20 722 – 44 307 Nantes, CEDEX 3.

# Un modèle de programmation intégrant classes, événements et aspects

---

## *A Programming Model Integrating Classes, Events and Aspects*

**Angel Rodrigo Núñez López**

⋈

*favet neptunus eunti*

---

**Université de Nantes**

Angel Rodrigo Núñez López

***Un modèle de programmation intégrant classes, événements et aspects***

xii+202 p.

This document was edited with these-LINA v. 2.7 LaTeX2e class of the "Association of Young Researchers on Computer Science (IPGIN)" from the University of Nantes (available at : http://login.irin.sciences.univ-nantes.fr/). This LaTeX2e class is under the recommendations of the National Education Ministry of Undergraduate and Graduate Studies (circulaire n° 05-094 du 29 March 2005) of the University of Nantes and the Doctoral School of « Sciences et Technologies de l'Information et Mathématiques(ED-STIM) »

*Print : thesis.tex – 06/12/2011 – 12:42.*

*Last class review: these-LINA.cls,v 2.7 2006/09/12 17:18:53 mancheron Exp*

## Abstract

Object-Oriented Programming (OOP) has become the de facto programming paradigm. Event-Based Programming (EBP) and Aspect-Oriented Programming (AOP) complement OOP, covering some of its deficiencies when building complex software. Today's applications combine the three paradigms. However, OOP, EBP and AOP have not yet been properly integrated. Their underlying concepts are in general provided as distinct language constructs, whereas they are not completely orthogonal. This lack of integration and orthogonality complicates the development of software as it reduces its understandability, its composability and increases the required glue code.

This thesis proposes an integration of OOP, EBP and AOP leading to a simple and regular programming model. This model integrates the notions of class and aspect, the notions of event and join point, and the notions of piece of advice, method and event handler. It reduces the number of language constructs while keeping expressiveness and offering additional programming options.

We have designed and implemented two programming languages based on this model: EJava and ECaesarJ. EJava is an extension of Java implementing the model. We have validated the expressiveness of this language by implementing a well-known graphical editor, JHotDraw, reducing its glue code and improving its design. ECaesarJ is an extension of CaesarJ that combines our model with mixins and language support for state machines. This combination was shown to greatly facilitate the implementation of a smart home application, an industrial-strength case study that aims to coordinate different devices in a house and automatize their behaviors.



**Keywords:** object-oriented programming, event-based programming, aspect-oriented programming, programming languages



## Résumé

Le paradigme de la programmation par objets (PPO) est devenu le paradigme de programmation le plus utilisé. La programmation événementielle (PE) et la programmation par aspects (PPA) complètent la PPO en comblant certaines de ses lacunes lors de la construction de logiciels complexes. Les applications actuelles combinent ainsi les trois paradigmes. Toutefois, la POO, la PE et la POA ne sont pas encore bien intégrées. Leurs concepts sous-jacents sont en général fournis sous la forme de constructions syntaxiques spécifiques malgré leurs points communs. Ce manque d'intégration et d'orthogonalité complique les logiciels car il réduit leur compréhensibilité et leur composabilité, et augmente le code d'infrastructure.

Cette thèse propose une intégration de la PPO, de la PE et de la PPA conduisant à un modèle de programmation simple et régulier. Ce modèle intègre les notions de classe et d'aspect, les notions d'événement et de point de jonction, et les notions d'action, de méthode et de gestionnaire d'événements. Il réduit le nombre de constructions tout en gardant l'expressivité initiale et en offrant même des options de programmation supplémentaires.

Nous avons conçu et mis en œuvre deux langages de programmation basés sur ce modèle : EJava et ECaesarJ. EJava est une extension de Java implémentant le modèle. Nous avons validé l'expressivité de ce langage par la mise en œuvre d'un éditeur graphique bien connu, JHotDraw, en réduisant le code d'infrastructure nécessaire et en améliorant sa conception. ECaesarJ est une extension de CaesarJ qui combine notre modèle avec de la composition de *mixins* et un support linguistique des machines à états. Cette combinaison a grandement facilité la mise en œuvre d'une application de maison intelligente, une étude de cas d'origine industrielle dans le domaine de la domotique.



**Mots-clés :** programmation par objets, programmation événementielle, programmation par aspects, langages de programmation

*A mi familia*

# Acknowledgements

First of all, I am grateful to the European Project AMPLE, which, through ARMINES, financed the first half of my thesis. I am grateful to the Industry Minister of France, which, through the École des Mines the Nantes, financed the second half.

I am thankful to the École des Mines de Nantes, the institution that made this PhD work possible. In particular, I am thankful to the ASCOLA project for supporting my work and for bringing me the possibility to attend several conferences.

I would like to thank the people from Technische Universität Darmstadt, specially Vaidas Gasiūnas, for a precious collaboration which is reflected in this work.

I am really thankful to Jacques Noyé. He is the most important person in this work. I have to say that I learnt a lot from you Jacques, not only in the research context. Thank you for all these years of discussion and advice. Thank you also to my thesis advisor, Jean-Claude Royer.

I would like to express my gratitude to all the members of the jury for reviewing and commenting on this work.

I cannot finish these lines without admitting that all these years of research would not have been possible without the emotional support of my friends. Thank you very much Delphine, Salva, Cristian, Rodrigo, Ricardo, Stefanie, Anna, Ludo, Soizic, Lotte, Joost, Axelle, Sady and all my friends of AtlanMod.

Last but not least, this work is dedicated to my family. France is far from Chile. I know you miss me a lot all this time. However you have to know that part of my heart did not take the plane, that part was always with you.

# Table of Contents

## —*Appendixes*—

# Chapter 1

# Introduction

Research in programming languages is motivated by the need to provide proper means for building "good quality" software. The quality of a software is usually measured in terms of properties such as understandability, maintainability, reusability, and evolvability. Object-Oriented Programming (OOP) marked a point in history by introducing a straightforward and intuitive programming model promoting the encapsulation of code and state in objects. However, it did not scale well when building complex and large applications. New programming paradigms emerged trying to cover the deficiencies of OOP such as Event-Based Programming (EBP) and Aspect-Oriented Programming (AOP). EBP proposed a mechanism for minimizing the dependencies between the components of a software. AOP proposed a mechanism for modularizing the so-called *crosscutting concerns*. Due to the complexity of today's systems, it is not a surprise that today's software needs combinations of all the techniques introduced by these paradigms.

## Problem Statement

OOP, EBP and AOP have not been properly integrated. Their concepts are in general provided as distinct language constructs, whereas these constructs are not completely orthogonal. When these concepts are combined in an application, the lack of integration and orthogonality complicates the development of such an application because it reduces its composability and increases its glue code. The lack of orthogonality also reduces the understanding of the involved concepts, conflicting with the principles of programing-language design proposed by MacLennan [Mac95].

Now that OOP, EBP and AOP are better understood, researchers realize that these paradigms can be better integrated to each other for the sake of simpler and more regular languages. Although some first steps have been done in this direction [RS05, AGMO06], the integration is not complete yet.

## Thesis

We believe that OOP, EBP and AOP can be better integrated, leading to a simpler and more regular programming model. This model would facilitate the combination of the techniques proposed by the different paradigms in complex applications.

This dissertation presents a model that validates this thesis. Whereas EBP augments the object-oriented (OO) paradigm with the concepts of event, event handling and implicit invocation, AOP introduces a set of, in principle, new concepts: aspects, join points, point cuts and pieces of advice. Our model integrates the notions of class and aspect, the notions of event and join point, and the notions of piece of advice, method and event handler.

Figure 1.1 clarifies the context of our work. It illustrates, in italics, the different concepts proposed by the three programming paradigms that this dissertation takes into account. On

the sides of the triangle we indicate approaches that promotes, in some degree, integration of the concepts on the vertexes of each side. In the center of the triangle we would like to place approaches integrating concepts taken from the three paradigms. The empty triangle shows the fact that at the date of this dissertation no approach has been found to actually deal with this integration. The objective of our model is to fill this empty space.

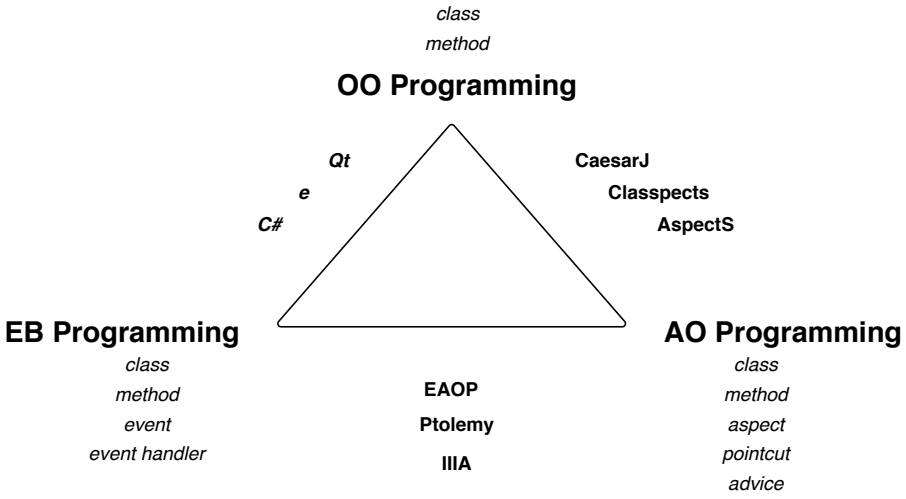


Figure 1.1 – Concepts

We have conceived two programming languages, EJAVA and ECAESARJ, implementing the proposed model. EJAVA is an extension to JAVA that validates the essence of our model. It has been used to improve a known application: JHotDraw [Eri11]. ECAESARJ combines the model with advanced OO techniques such as mixin composition. It has resulted in a suitable tool for implementing an industrial strength case study: the Smart Home case study [GNNM11].

## How to Read this Dissertation

This dissertation is structured as follows.

Chapter 2 gives the reader the necessary background in the areas of EBP and AOP (we assume that the reader already knows OOP). The last part of this chapter talks about an advanced OO technique: mixin composition.

Chapter 3 specifies our problem statement. We discuss about the opportunities of integration of the considered programming paradigms.

Chapter 4 presents the event-based (EB) side of our programming model. We introduce the notion of declarative event and present EJAVA, an extension to JAVA that concretizes the model.

Chapter 5 is about the integration of EBP and OOP. We present a unification of event triggering and method call and a unification of event handling and method execution. We show how these unifications improve the simplicity, the regularity and the orthogonality of our programming model.

Chapter 6 is about the integration of EBP and AOP. We show how the unifications presented in Chapter 5 enable AOP. In addition, we introduce a unification of events and pointcuts based on the notion of declarative event. Finally, we illustrate the support for

AOP of EJava.

Chapter 7 is about the integration of AOP and OOP. We eliminate the frontiers between classes and aspects and show how aspects can be used to implement functional concerns.

Chapter 8 presents an extension of our model with mixin composition and language support for state machines. We introduce ECaesarJ, the language that concretizes this extension, and show how it supports both stateful aspects and stateful objects in a seamless way.

Chapter 9 validates the usability of EJava and ECaesarJ in the implementation of real applications. We present the results of two experiences: the implementation of a graphical editor and the development of a home application.

Chapter 10 describes the main aspects of the implementation of EJava and ECaesarJ.

Chapter 11 concludes this dissertation with the limitations, the contributions and the perspectives of this work.

# Part I

## Context and Problem Statement

# Chapter 2

# State of the Art

A big part of the research on software engineering is about building "good quality" software. A software is good if it guarantees some properties such as modularity, extensibility, maintainability, efficiency, robustness, correctness, between others. The key to achieve many of these properties is a good separation of concerns (SoC), which follows the premise *divide and conquer* [Dij79]. Object-Oriented Programming (OOP) was proposed as a paradigm that structures programs in objects, which encapsulates data and behavior. OOP is the *by default* development paradigm in many domains. However, it presents some deficiencies for decoupling components and for modularizing crosscutting concerns. Event-Based Programming (EBP) and Aspect-Oriented Programming (AOP) are two paradigms that complement OOP with mechanisms to solve these deficiencies. In addition, pure OOP presents some problems of extensibility in large applications. Advanced OOP techniques have been developed to cover this problem.

This chapter is structured as follows. Section 2.1 and Section 2.2 present the necessary background on EBP and AOP, respectively. Section 2.3 describes *mixins* as an advanced OOP technique that improves the extensibility of OO programs. Section 2.4 concludes.

## 2.1 Event-Based Programming

The flow of an event-based program is determined by identifying *events*, *triggering* event notifications and *reacting* to these notifications. In an event-based program the integrated components communicate by means of these notifications [FMG02]. Components interested in certain events subscribe to them, so that upon occurrence of these events, proper notifications are automatically delivered to these components.

Event-based programming was created in order to alleviate the complexity of large reactive systems with two principal contributions:

1. **Simplification in the implementation of reactive applications**. Components of reactive systems, such as Graphical User Interfaces (GUI), need to be aligned with an environment that frequently changes. Events provide a notion of change, which is automatically distributed across the system. Instead of waiting on blocking operations for input, components can subscribe to events, which notify the availability of such an input. Events for reactive applications are usually asynchronous, enabling the management of the huge amount of events that reactive applications need to handle. Events are stored in event buffers and an event loop notifies components interested in these events, in an off-line mode.
2. **Reduction of the coupling between software components**. Event-based (EB) designs decouple the components of a system, offering a more stable design since the inter-dependencies are minimized. In EB systems, the components that are source of events are decoupled from the ones interested in them. Components that notify

events do not know at compile time what components are interested in these events, reducing the dependency between these components. This is not the case in the classical *Request/Reply* model of communication, in which there is a caller/callee dependency.

This dissertation is principally concerned with this last motivation. The remainder of this section goes deeper on how EBP reduces the coupling between software components. Section 2.1.1 defines the notion of *event*. Section 2.1.2 describes what constitutes an event-based design. Section 2.1.3 overviews the event support present in some representative state-of-the-art languages.

### 2.1.1 The Notion of Event

In a general sense an *event* denotes *something that happens at a given place and time.*∗ In the theory of relativity, events are the fundamental entities of observed physical reality. They are represented by single points in the space-time continuum. Associated to the concept of event is the concept of *causality* as the necessary relationship between one event, the *cause*, and another event, the *effect*, which is a direct consequence of the first.

#### 2.1.1.1 Events in the History of Computing

The concept of event has been used from the very beginning of computer science. With the advent of the digital computer in the 50's, digital simulation modeling became accepted as a technique for the analysis of engineering, business and behavioral systems [Sch78]. At that epoch, digital computers started to be used in the analysis of *discrete event systems* such as the Analysis of Air Traffic Control Systems, the Analysis of Large-Scale Military Operations, Job-Shop Scheduling, Competitive Market Analysis and Man-Machine Interface. An event was early defined as *an occurrence which may change the value of some data* [Par69] or simply as a *change of state* [ND66]. Many simulation languages included this notion of event, most notably Simula [ND66]. The Simula language was in principle designed to describe systems such that it is possible to regard its operation as consisting of a sequence of instantaneous events, each event being an active phase of a process. In all these simulation languages, events were conceived as the basic constituents of discrete event processes, in which *events may take place only at distinct "points" in time which are separated by periods in which no change is made in the state of the data* [Par69]. As an example, the acceptance of a passenger in the counter of an "airport departure" model corresponds to an event, which as a result makes the counter route the passenger either to the control office or to a fee collector [ND78].

The introduction of process algebras as algebraic/axiomatic approaches to formalize concurrent processes marked the 70's. Hoare published his influential paper [Hoa78] as a technical report in 1976. The paper presents the CSP (Communicating Sequential Processes) language. Events and primitive processes are the primitives of the language. Events represent communications or interactions. Importantly, CSP qualifies events as *indivisible* and *instantaneous*. They are represented by atomic names (e.g. `on`, `off`), compound names (e.g. `valve.open`, `valve.close`) or input/output events (e.g. `mouse?xy`, `screen!bitmap`). The work of Hoare is the basis of, without doubt, the most significant contribution in this area: the CCS (Calculus of Communicating Systems) language of Milner [Mil82]. Even if

∗. Definition taken from WordNet 3.0, Princeton University, 2006.

Milner does not use the word event in its presentation, his term *atomic action* represents the same concept.

Events became even more popular with the proliferation of reactive systems in the 80's. Active Database Systems [DHW94] introduced the Event Condition Action (ECA) pattern, to make conventional databases active: *the database system itself performs certain operations automatically in response to certain events occurring or certain conditions being satisfied* [WC96]. In other words, the ECA pattern permits the definition of rules, which trigger some action once an event happens and a given condition holds. The pattern made the role of events important: enabling the programming of reactions to them, i.e. the definition of the effect associated to the event.

Events in active databases are defined as data modifications such as the result of executing `insert`, `delete` or `update` operations in relational databases, or the creation, deletion or modification of objects in object-oriented database systems. They also include temporal events defined in terms of absolute time, repeated time or at periodical intervals. Interestingly, active databases also include support for application-defined events, e.g. high-temperature, user-login, etc. All these types of events are considered as primitive ones. Composite events are also supported as the combination of other events defined by using logical operators, sequences or temporal composition.

Starting in the 80's, events have been widely used in GUI design. In this area, events correspond to the input provided by humans in the interaction with machines. Mouse or keyboard input are examples of events used in GUIs.

#### 2.1.1.2 Events in Programs

As a summary, it is possible to provide the following definition:

**Definition 2.1.** *An event represents a change in the context of a computation that is reflected as a change in the state of such a computation. For simplicity it is possible to consider an event just as a change in the state of a computation.*

For example, the loan of a book is an event that is reflected as a change in a database. The physical click of a mouse is an event that is reflected as the reception of a signal from a mouse device or as the invocation of a method of an API.

From the previous definition any point in the execution of a program is an event since it changes the state of the execution. For example, the fact that the runtime has reached a point in which a certain method body has to be executed can be seen as an event, occurring at the beginning of the method body.

### 2.1.2 Event-Based Design

As discussed in the previous section, events are inherent to any computer program because they have been defined as a change in the state of its execution. However, not every program can be considered as following the principles of the EBP paradigm. The contribution of EBP is to structure the way specific events are identified in a program and the way the information about the occurrence of these events is transmitted to software components interested in these occurrences.

#### 2.1.2.1 Motivation

In the 80's, the Smalltalk-80 programming environment introduced the Model-View-Controller (MVC) triad for developing its user interface [KP88]. The main objective of MVC was to separate the concerns related to the view of its programming environment and the core functionality of the same. In order to effectively decouple views from models, the implementation of MVC needed a mechanism different from the traditional method invocation protocol used in contemporary programs. Thus, Smalltalk-80 adopts a different mechanism to decouple software components.

The different parts of the MVC triad are defined as follows:

> *Models are those components of the system application that actually do the work (simulation of the application domain). They are kept quite distinct from views, which display aspects of the models. Controllers are used to send messages to the model, and provide the interface between the model with its associated views and the interactive user inteface devices (e.g. keyboards, mouse).*
>
> Krassner and Pope [KP88]

As the MVC was conceived, views and controllers have just one model, but a model can have one or several views and controllers associated with it. A change in the model (initiated by a particular controller) needs to be reflected in all the views associated with it, while keeping independence between model code and view code:

> *To maximize data encapsulation and thus code reusability, views and controllers need to know about their model explicitly, but models should not know about their views and controllers.*
>
> Krassner and Pope [KP88]

```
class Figure {
  DrawingView view;
  void moveBy(int x, int y) {
    /* change figure position */
    view.repaint(this.getArea());
  }
}
```

Listing 2.1: Updating a view in a traditional program.

In a traditional programming scheme, model classes would need to keep fields referencing view classes as Listing 2.1 shows in JAVA. Model classes would need to be aware of view classes, not fulfilling the previous requirement.

The class `Figure` of Listing 2.1 belongs to the model of a figure editor, whereas the class `DrawingView` belongs to the view. The class `Figure` defines the method `moveBy`, which changes the position of the figure. The change needs to be reflected in the view. In the traditional programming style of Listing 2.1, the class `Figure` keeps an explicit reference to an instance of the class `DrawingView` and invokes the `repaint` method when the state of the model changes in order to update the view. The model class remains coupled to the view class. Programmers of the model cannot develop their classes without having some knowledge about the possible views of the application. In addition, inclusion of further views into the application requires updating the model code.

In Smalltalk-80, MVC decouples views and models by establishing a subscribe/notify protocol between them:

> *To manage change notification, the notion of objects as* dependents *was developed. Views and controllers of a model are registered in a list of dependents of the model, to be informed whenever some aspect of the model is changed. When a model has changed, a message is broadcast to notify all of its dependents about the change.*
>
> Krasner and Pope [KP88]

In the implementation of Smalltalk-80, model classes are subclasses of a class `Model`, which defines a list of dependents and provides methods to register new dependents or to unregister them. It also provides an implementation of a method `notify`, which broadcasts state changes to the registered dependents. Thanks to this design, programmers of the model classes do not need to be aware of views, which can be programmed and registered later on. This notion of dependents was further structured as a pattern by Gamma et al. [GHJV94] and it is the subject of the next section.

#### 2.1.2.2 Design Pattern *Observer*

The design pattern *Observer* [GHJV94] structures the notion of dependent objects introduced by Smalltalk-80. It organizes classes in observers of events and subjects, which trigger the events. Classes that are source of events extends the class `Subject`, which aggregates observers interested in the events by providing methods to register or unregister them. In languages such as JAVA, an observer is an object implementing the interface `Observer`. This interface includes the *callback* method `update`, which is called as a notification that the event has happened.

As an example, let us express the example of the previous section using the design pattern Observer in JAVA.

```
class Figure extends Subject {
  void moveBy(int x, int y) {
    /* change figure position */
    notifyObservers();
  }
}
```

Listing 2.2: Example of a subject in the design pattern Observer.

The code of Listing 2.1 can be expressed in a decoupled way by using the pattern Observer as Listing 2.2 shows. The class `Figure` is the source of an event (the one representing a change in the figure). As such it extends the class `Subject`, thus being equipped with infrastructure code for registering observers. Each time a change is produced (at the end of the execution of the method `moveBy`) it notifies all its registered observers by calling the method `notifyObservers`. Note that since JAVA does not support multiple inheritance, classes that already have a superclass need to include the infrastructure of `Subject` within their code.

The class `DrawingView` constitutes an observer of the class `Figure`, and therefore implements the method `update` of the interface `Observer` as Listing 2.3 shows.

```
class DrawingView extends Observer {
  void update(Subject subject) {
    repaint(((Figure)subject).getArea());
  }
}
```

Listing 2.3: Example of a JAVA observer in the design pattern Observer.

```
Figure figure = new Figure();
figure.registerObserver(new DrawingView());
```

Listing 2.4: Registration in the design pattern Observer.

The pattern Observer makes a clear distinction between source of events and observers. Afterwards, client code is in charge of properly connecting observers and subjects of events as illustrated in Listing 2.4.

The pattern Observer decouples model code (subjects) from view code (observers). In further versions of the application new observers can be implemented by providing a proper implementation of the interface `Observer` and by registering their interest in the proper events.

#### 2.1.2.3 JAVA Listeners

The pattern Observer makes it possible for a subject to announce a change without being coupled to observers. However, sometimes this information is not granular enough, usually when a subject can change in several ways. Observers need to find out how the subject has changed. For example, a change in a figure can be due to several factors: a change in size, a change in position, a change in color, etc. Finding out how the figure changed may be complicated and time-consuming. The JAVA Listeners [Sun97] offer a design that can be considered as a fine-grained version of the pattern Observer in JAVA.

The JAVA Listeners solution is as follows. First, a listener is an interface similar to the interface `Observer` of the pattern Observer. However, the interface may declare several callback methods representing different events. A callback method of a listener may accept, as an additional parameter, a reification of the event that produces the callback (a reification of the corresponding change). Second, several listener interfaces can be declared for several kinds of events. Third, a subject provides the required infrastructure for accepting several kinds of listeners and to notify the proper callback methods of the corresponding listeners at the proper places.

For example, the class `Component` of the graphical library AWT of JAVA defines methods for attaching listeners such as `MouseListener` or `KeyListener`. The interface `KeyListener` defines callback methods associated to keyboard events such as `keyPressed` or `keyReleased`. These methods receive as parameter an instance of `KeyEvent`, which provides information about the pressed/released key. For example, Listing 2.5 shows the use of `KeyListener`. A `Frame` instance as a subject provides the method `addKeyListener` in order to register a listener observing key events. The listener implemented in the class `MyKeyListener` prints the value of the key when the callback method `keyPressed` is invoked by the AWT framework.

Listing 2.6 shows how a figure can notify different kinds of events by using JAVA lis-

```
public class AClass {
  public static void main(String[] args){
    Frame f = new Frame("App");
    f.addKeyListener(new MyKeyListener());  ...
  }
}
class MyKeyListener implements KeyListener {
  public void keyPressed(KeyEvent e) {
    System.out.println(e.getKeyChar());
  }
}
```

Listing 2.5: Example of Java Listeners in AWT.

```
interface FigureListener {
  void figureMoved(FChangeEvent event);
  void figureColored(FChangeEvent event);
}
class Figure {
  List<FigureListener> figureListener;
  void addFigureListener(FigureListener l) { ... }
  void removeFigureListener(FigureListener l) { ... }

  void notifyMoveEvent(int x, int y) {... }
  void notifyColorEvent(Color color) {... }

  void moveBy(int x, int y) {
    /* change figure position */
    notifyMoveEvent(x,y);
  }
  void setColor(Color color) {
    /* change figure color */
    notifyColorEvent(color);
  }
}
```

Listing 2.6: Example of Java Listeners in a figure editor.

teners. It defines a listener interface `FigureListener` with two methods representing two types of events, a change in position or a change in color of a figure. The class `Figure` is explicitly equipped with the necessary infrastructure for notifying listeners. Indeed, in JAVA, when there are several kinds of listeners to be notified by a subject, it is not possible to extend a class such as the class `Subject` of the pattern Observer. The benefits of subclassing `Subject` is that it hides the notification infrastructure. This can be seen as a drawback of JAVA listeners. However, the drawback can also affect the pattern Observer in JAVA when a subject already has a superclass. Of course, this is not a problem in languages that support advanced class-composition mechanisms such as multiple inheritance.

#### 2.1.2.4 Publish/Subscribe Systems

At the end of the 80's, Birman et al. [BJ87] introduce the messaging paradigm Publish/Subscribe as a flexible communication model specially dedicated to distributed systems. Publish/Subscribe organizes components in publishers and subscribers. The notion of publisher and subscriber is equivalent to the notion of subject and observer in the pattern Observer. Unlike the latter, in publish/subscribe systems, subscribers do not register their interest in events directly with publishers:

> *Producers publish information on a software bus (an event manager) and consumers subscribe to the information they want to receive from that bus. This information is typically denoted by the term event and the act of delivering it by the term notification.*
>
> Eugster et al. [EFGK03]

This software bus is an event notification service, which acts as a mediator between both parties broadcasting published events to all the interested subscribers.

The key concepts introduced by publish/subscribe systems are the event notification service and the subscription mechanisms. The event notification service is, in general, implemented in a middleware connecting publishers and subscribers. The subscription mechanisms can be *topic-based*, *content-based* or *type-based*.

- *Topic-based subscription* [BJ87] is the earliest subscription scheme, which was based on the notion of *topics* or *subjects*. It was implemented in the context of *group communication* systems. *Subscribing to a topic T can be viewed as becoming a member of a group T, and publishing an event on topic T translates accordingly into broadcasting that event among the members of T* [EFGK03]. A topic is a name that acts as a keyword, which is given in the publication of an event.
- *Content-based subscription* [RW97] is a variant that improves on topics by providing a more dynamic event selection, which is based on properties of events such as attributes of data structures associated to the events or on some meta-data of events. Selection is made by using powerful filters defining specific constrains on the event content.
- *Type-based subscription* [EGD01, RL08, SPAK10] is the most recent subscription mechanism. Event types replace the name-based topic classification model by a scheme that filters events according to their type, enabling type safety at compile-time.

An important property of this kind of systems is *synchronization decoupling*. Publishers are not blocked while producing events. Subscribers can get asynchronously notified. This makes the Publish/Subscribe pattern well adapted to distributed environments.

#### 2.1.2.5 The Principle of EBP: Implicit Invocation

The pattern Observer, the Java Listeners and the systems Publish/Subscribe share a common pattern. In these approaches event sources are unaware of event destinations with registration and notification supported behind the hood. This architectural pattern is called *Implicit Invocation* [GN91] and can be considered as the essence of Event-Based Programming.

> *The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events. Other components in the system can register an interest in an event by associating a procedure with the event. When the event is announced the system itself invokes all of the procedures that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of the procedures in other modules.*
>
> Garlan and Shaw [GS94]

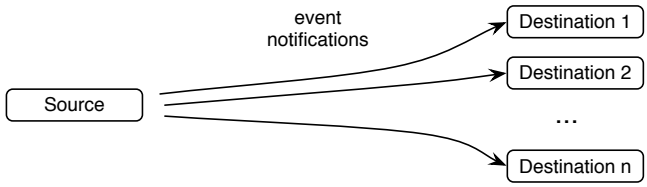


Figure 2.1 – Event-based design.

Figure 2.1 illustrates the implicit invocation architecture, which organizes components of a program in event sources and event destinations. When an event happens in a source, notifications are delivered to all the destinations interested in the event (see the arrows of the figure). The notification mechanism ensures that all destinations are informed of the event. A notification corresponds to a message informing that the event has occurred. What is fundamental in this design is the fact that sources of events do not know at compile time the potential destinations of their notifications (not in number, not in concrete type).

Let us provide some definitions:

**Definition 2.2.** *An event-based approach is an approach that* decouples *sources of events from destinations by adopting an Implicit Invocation scheme.*

**Definition 2.3.** *An event handler is code to be executed when an event occurs.*

**Definition 2.4.** *The notification of an event corresponds to a process in which all the event handlers registered with an event are invoked.*

**Definition 2.5.** *Event triggering or event announcement is an expression that produces the notifications of an event.*

#### 2.1.2.6 Inversion of Control

Event-based programming produces a phenomenon called *inversion of control*. Even if recent research has shown that it is possible to avoid it [HO09], inversion of control has been classically associated with EBP and permits us to better understand this paradigm.

Inversion of control is a phenomenon early identified in frameworks. In traditional programs, user code uses the functionality of libraries by directly calling its methods. In other words, the control goes from the application towards the libraries. In frameworks the control is inverted. Frameworks are configured with specific user code that is called from the framework. Literature associates inversion of control to the so-called *Hollywood principle*, which means "don't call us, we'll call you":

> *One important characteristic of a framework is that the methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code. The framework often plays the role of the main program in coordinating and sequencing application activity. This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the user tailor the generic algorithms defined in the framework for a particular application.*
>
> Johnson and Foote [JF88]

Like in frameworks, EBP produces inversion of control because the control resides within the event-driven infrastructure, rather than in the application code [Sam08]. Source of events are configured with user-defined handlers, which are invoked by the event-driven infrastructure when events arise.

We can illustrate the inversion of control phenomenon by comparing a *tree-based API* for parsing XML documents, such as DOM (Document Object Model)†, and an *event-based API*, such as SAX‡. For this, let us consider an XML file containing a list of contacts with their name and email address in tags with the following structure:

```
<contact name="Paty" email="panunez@chile.com" /> ...
```

A DOM library parses and maps an XML document into an internal tree structure. As an illustration, Listing 2.7 prints the emails of the contacts present in the file in JAVA. The tree acts as a database and the programmers use the API to navigate the resulting tree, *e.g.* to get all the elements associated to a tag by using the `getElementsByTagName` method.

```
domTree = domParser.parse(file);
nodes = domTree.getElementsByTagName("contact");

for(int i = 0; i < nodes.getLength(); i++) {
  Element element = (Element) nodes.item(i);
  System.out.println(element.getAttribute("email"));
}
```

Listing 2.7: Parsing XML documents using DOM.

The mechanism is quite different when using events. The SAX API reports parsing events (such as the start or the end of elements) directly to the applications. The application logic is implemented in event handlers, which receive these events and react accordingly. As an illustration, Listing 2.8 shows a SAX implementation of the previous example in

†. DOM website: http://www.w3.org/DOM/.
‡. SAX website: http://www.saxproject.org/.

```
saxParser.setContentHandler(new DefaultHandler() {
  public void startElement(String uri, String tagName,
                       String qName, Attributes attributes) {
    if(tagName.equals("contact"))
      System.out.println(attributes.getValue("email"));
  }
});

saxParser.parse(file);
```

Listing 2.8: Parsing XML documents using SAX.

JAVA. A first step consists of configuring the library by registering a proper handler with the SAX parser. The handler is programmed to handle the events generated during the parsing of the XML document. After configuration, the control is passed to the library by calling the `parse` method (see last line in the code), which notifies the respective events to the configured handler during the parsing process.

The inversion of control can be observed in the fact that in the DOM solution the execution is controlled by the user, who asks the library for information about the XML document. In the event-based solution, the control is inverted, it is the library that notifies the user code.

### 2.1.3 Language Support

Language support for events corresponds to the inclusion of constructs specially dedicated to EBP in a language. These constructs reduce the necessary infrastructure when using approaches such as the pattern Observer or the JAVA Listeners. In general, language support for events is provided in the form of two constructs: an event representing a state change, which in some approaches is just a signature, and an event handler, which in several approaches is just a method. This section shows the language support for events found in some mainstream languages and other approaches.

#### 2.1.3.1 QT Signals and Slots

QT is a cross-platform application and UI framework implemented in C++. Originally developed by Haavard Nord and Eirik Chambe-Eng (CEO and President of Trolltech, respectively) in 1991, it nowadays counts several versions, which have been used by enterprises such as Google Earth, KDE, Opera, Skype, among others. As a framework it provides a rich set of application building blocks, delivering all of the functionality needed to build advanced, cross-platform applications. In particular, QT provides event-based support as C++ extensions implemented using a special pre-processor, the Meta-Object Compiler.

Apart from standard method calls, QT objects can communicate using *signals* and *slots*. They are constructs defined as class members. A signal corresponds to an event. It is declared as a signature and can be triggered using the `emit` keyword. A slot is a method designed to handle signals, *i.e.* an event handler.

Listing 2.9 shows the QT implementation of the class `Figure` of a figure editor. In order to declare events and slots, classes need to extend the class `QObject` and include the macro `Q_OBJECT`, which is used by the pre-processor to instrument the class with the necessary infrastructure. Line 4 illustrates the declaration of a signal representing a change

```
1 class Figure: public QObject {
2   Q_OBJECT
3   signals:
4     void changed(Figure object);
5   public:
6     void moveBy(int x, int y);
7 }
8 void Figure::moveBy(int x, int y) {
9    /* change figure position */
10   emit changed(this);
11 }
```

Listing 2.9: Example of events in QT.

in the state of the figure. The signal is `emitted` in Line 10 passing the current object as a parameter.

```
class DrawingView: public QObject {
  Q_OBJECT
  public slots:
    void update(Figure source);
}
void DrawingView::update(Figure figure) {
   repaint(figure.getArea());
}
```

Listing 2.10: Example of event handlers in QT.

Classes can define slots as the implementation of the class `DrawingView` in Listing 2.10 shows. It defines the `update` method as a slot that receives a `Figure` instance as a parameter.

```
Figure figure; DrawingView view;
QObject::connect(&figure, SIGNAL(changed(Figure)),
                 &view, SLOT(update(Figure)));
```

Listing 2.11: Example of connections of signals and slots in QT.

The signals emitted by an object are *connected* to the slots defined in another object by using the method `connect` of the class `QObject`. The compiler verifies that the signature of the slots and the signals (the number of parameters and their type) coincide. Listing 2.11 connects the signal `changed` of an instance of `Figure` to the slot `update` of an instance of `DrawingView`.

Signals can only be used for event triggering inside the class or subclasses. Methods defined as slots can be used for handling signals and also for explicit invocation as standard methods. However a standard method cannot be used as a slot. Preplanning is needed to identify the methods that will serve as slots. As a framework, QT also includes event support in the form of JAVA-like Listeners equipped with a complex framework implementing an event loop.

Qt is a simple example of language support for implicit invocation. It clearly describes this pattern. First, an event is declared and triggered at sources. Second, destinations implement event handlers. Third, a connection between sources and destinations is made without sources being coupled to destinations.

#### 2.1.3.2 C# Events

```
1 delegate void ChangeHandler(Figure source);
2
3 class Figure {
4  public event ChangeHandler changed;
5
6   void moveBy(int x, int y) {
7     /* change figure position */
8     if(changed != null)
9       changed(this);
10  } ...
11 }
```

Listing 2.12: Example of events in C#.

Similarly to Qt, the language C# allows classes to explicitly define events and stay completely unaware of potential observers of these events. The language provides objects interested in these events with mechanisms to perform the corresponding subscriptions. As an example, Listing 2.12 shows an implementation of the class `Figure` in C#. The class `Figure` explicitly declares an event `changed` in line 4, which is triggered as a method call in the method `moveBy` in line 9. The event triggering produces an implicit notification of all objects interested in such an event, *i.e.* all objects that have previously subscribed to the event.

The notification mechanism of C# is based on the language construct `delegate`, which makes it possible to define a data-type name associated to a certain function type. For example, Line 1 of Listing 2.12 defines a delegate `ChangeHandler` associated to the functions returning `void` and receiving an object of type `Figure` as parameter. A delegate instance can be attached to a group of functions of the corresponding type. A delegate instance can be called. The call results in an invocation of all the attached functions. The construct `event` is used with a delegate and encapsulates an instance of such a delegate. An event can only be defined with delegates associated to `void` functions. In Listing 2.12, the definition of the event `changed` of Line 4 internally defines an instance of `ChangeHandler`. At runtime, the event `changed` (or more precisely the respective delegate) can be attached to any method of any class, as long as the method accepts an instance of `Figure` as a parameter and returns `void`. The call of line 9 serves as notification, in the form of a method call, to all the methods attached to `changed`.

A function is attached to an event (delegate) by using the operator `+=`. In EB parlance, the operator makes it possible to register a function as a handler of the event. In an inverse way, the C# operator `-=` can be used to unregister functions. For example, Listing 2.13 attaches the method `update` of a `DrawingView`'s instance to the event `changed` of a `Figure`'s instance. Note that the attached functions are, in this case, methods of particular objects.

A C# event defines not only a delegate but also a default implementation of the operators `+=` and `-=` (basically a forwarding to the delegate). This implementation can be

```
1 class Test {
2   public static void Main() {
3     Figure figure = new Figure();
4     DrawingView view = new DrawingView();
5     figure.changed += new ChangeHandler(view.update); ...
6   }
7 }
8 class DrawingView {
9   public void update(Figure figure) {
10    repaint(figure.getArea());
11  } ...
12 }
```

Listing 2.13: Registering to an event in C#.

overridden if an event is defined with a block of code providing a specific implementation. An event can be defined `abstract`, deferring the definition of such a block to subclasses. As a final remark, events cannot be triggered in subclasses. A special method triggering the event needs to be provided by the class defining the events. This method can then be used in subclasses to trigger the event.

#### 2.1.3.3 *e* Events

The language *e* [IEE08] is an object-oriented language designed to facilitate the verification of electronic designs. Language support for events is included as a way of specifying and verifying behavior over time.

> *All e temporal language features depend on the occurrence of events, which are used to synchronize activity within the simulation environment.*
>
> IEEE 1647 [IEE08]

Like in QT and C#, events in *e* are properties of objects. Interestingly, the events of *e* can be defined *declaratively* as composition of other events, with the lowest level relying on primitive events. The language provides several operators to combine events.

```
event sim_ready is change('top.ready') @sim;
event clk1 is rise('top.cpu_clk') @sim;
event clk2 is true(active == 1) @ clk1
event clk is ( clk2 or sim_ready ) @ sys.any
```

Listing 2.14: Examples of events in *e*.

Listing 2.14 shows some examples. The event `sim_ready` happens whenever the value of the simulator object `top.ready` changes. The construct `@sim` represents a callback from simulation. The event `clk1` represents a CPU clock tick and it is defined analogously as a raise in the value of the simulation object `top.cpu_clk`. The event `clk2` happens when `clk1` happens and the given condition evaluates to true. An expression `@`*event* represents a sampling event. The expression on the left of `@` is evaluated at the occurrence of the sampling event. Finally, the event `clk` happens when either `clk1` or `sim_ready` happen.

The event `sys.any` defines the finest granularity of time. This event happens whenever another event happens.

As shown in the examples, declarative events consist of a name and an expression, in *e* parlance, a *temporal expression*. Apart from the operators shown in the examples, the language provides other operators such as conjunction of events, negation, etc. Besides the declarative events, the language also makes it possible to define imperative events like in QT. Imperative events are triggered by using a keyword `emit`.

```
struct AClass {
  event sim_ready is change('top.ready') @sim;

  on sim_ready {
    transmit()
  }
}
```

Listing 2.15: Event handlers in *e*.

Reaction to events is programmed by attaching an event to a block of code as Listing 2.15 shows. In that example the block of code is executed whenever the event `sim_ready` happens. The code also shows that events and handlers are class members (`struct AClass`). In *e*, the handler of the event `sim_ready` defines a method `on_sim_ready()`. This makes it possible to override it in subclasses. However, in order to make it possible a handler can only be defined for events defined in the same class.

#### 2.1.3.4 Comparison

C# and QT implements imperative events. C# events are better integrated in the language than QT events (indeed QT is implemented as a pre-processor). First, in order to include events, a C# class does not need to extend a special class or to include a macro. Second, any C# method can be registered as a handler of an event. Third, the concept of delegate of C# makes it possible to associate a type name to a signature, which can be reused in the definition of several events, whereas in QT the signature has to be repeated in the definition of each event. However, the QT design is simpler. In particular, the mix event/delegate of C# is complicated for programmers.

The language *e* complements the design of events of languages like QT and C# with declarative events. However, the design is very focussed on verification of electronic devices, showing some deficiencies in other contexts. For example, *e* events do not support parameters.

### 2.1.4 Complex Event Processing

*Complex event processing (CEP) is a set of techniques and tools to help us understand and control event-driven information systems* [Luc01]. CEP focus on the complexity of dealing with a large amount of events which come from information systems such as cell phones, sensor networks, Internet, etc. CEP pays a special attention to the way events are *correlated* to form *complex events*, events that only happen if many other events happen. A CEP system monitors the *cloud of events* that results from running big applications and detects the occurrence of complex events by using techniques such as *pattern matching*.

Events are correlated in terms of *time*, *causality* and *aggregation*. A relationship of time defines a physical ordering of events by timestamping them. A relationship of causality defines a logical ordering where events are defined as caused by others. A relationship of aggregation defines complex (or composite) events, which aggregate several simpler events. *The timing of a complex event starts when the earliest activities of its member events start, and ends when the latest activities of its members end* [Luc01]. These relationships have mathematical properties. They are transitive and asymmetric forming a partial ordering, *i.e.* there can be events that are not ordered by the relationship.

The different approaches to CEP define pattern languages that make it possible to relate events and define complex events in terms of others. Events are related by using patterns that are matched against the cloud of events. In general, these patterns provide means to express the conjunction, the disjunction and the causality of events. The conjunction `A and B and C` matches a set of three events, `A`, `B` and `C`. The disjunction `A or B or C` matches any one of `A`, `B` or `C`. The causal relation `A` $\rightarrow$ `B` matches pairs of events `A`, `B` where `A` causes `B`. A pattern can also have a condition associated to be satisfied when matching events.

```
(Dollars ?X, ?Y; Account ?A)
    (Deposit(?X, ?A) → Withdraw(?Y, ?A)) where ?Y < ?X
```

Listing 2.16: Simple event pattern in Rapide.

As an example, Listing 2.16 shows a simple pattern written in the language Rapide-EPL [Luc97, Luc01], a strong typed language specially designed to support CEP. The pattern defines the variables `?X`, `?Y` and `?A`, which are bound during pattern matching. This pattern matches any pair of causally related `Deposit` and `Withdraw` events on the same account. The guard, defined by using the keyword `where`, restricts the matches to those pairs for which the amount withdrawn is less than the amount deposited.

The language Rapide-EPL provides several other relational operators such as the independent operator, the before operator, the union operator and the disjoint-union operator. For example, the before operator makes it possible to define an expression `A < B` that matches pairs of events `A`, `B` where `A` has an earlier timestamp than `B`.

```
(Network_Path ?Route; Real ?Load)
    Report(?Route, ?Load) where ?Load > 7 =>
    generate Notify(Manager_Console, "Warning", Name(?Route), ?Load, Time)
```

Listing 2.17: Simple event pattern rule in Rapide.

The approaches to CEP make it also possible to program reactions to the matching of these patterns following an Event Condition Action model [DHW94]. These reactions are defined as *event pattern rules*. For example, the language Rapide provides the symbol `=>` to define sequential rules (there is also an operator for parallel rules). Listing 2.17 defines a rule that generates an event `Notify` when the event `Report` matches with a certain binding for the variables `?Route` and `?Load` such that `?Load` is bigger than `7`. Note that this rule can also be seen as a way to create complex events.

The language Rapide-EPL has several other features for dealing with complex events in distributed settings. We have just shown the basic ones. A more recent language

EventJava [EJ09] provides support for CEP on top of Java. Listing 2.18 shows an example of an event pattern rule in EventJava. It consists of a trading example comparing `earningsReport` and `analystDowngrade` events in terms of their timestamps. It matches pairs of events where `analystDowngrade` occurs after `earningsReport`. The construction `when` makes it possible to test condition in the matching. If a stock has a negative earnings report (the actual earnings per share, `epsAct`, is less than the estimate `epsEst`), followed by an analyst downgrade to "Hold", then the algorithm recommends selling the stock.

```
class StockMonitor {
  Portfolio p;
  event earningsReport(String firm, float epsEst, float epsAct, String period),
        analystDowngrade(String firm1, String analyst, String from, String to)
  when (earningsReport < analystDowngrade && firm == firm1 &&
        epsAct < epsEst && to == "Hold") {
    p.RecommendSell(firm);
  }
}
```

Listing 2.18: Event pattern rule in EventJava.

There are several other CEP approaches [WDR06, ADGI08, BDG+07] with different features that help us better understand the cloud of events that results from running today's complex applications. We have shown the most important ideas to give a general flavor of CEP.

### 2.1.5 Summary

This section has described the basic elements of Event-Based Programming. Section 2.1.2 defined the essence of this paradigm: implicit invocation. The pattern Observer or the systems Publish/Subscribe provide a dedicated protocol and infrastructure, which support the implicit invocation of events (state changes) from sources to destinations. Section 2.1.3 described how some programming languages have included language support for EBP. They design events and event handlers as type-safe language constructs, whose use reduces the amount of glue code needed for implementing EB applications. C#, a representative language in this context, features imperative events (events explicitly triggered in application's code), event handlers as plain methods, and dynamic registration of objects interested in particular events. Interestingly, in order to facilitate the verification of electronic devices, the language *e* introduces declarative events: events defined in terms of others in a declarative way. Finally, Section 2.1.4 described advanced techniques to correlate events and to create complex ones. These techniques make it possible to better understand the cloud of events that results from complex applications and thus better adapt these applications.

## 2.2 Aspect-Oriented Programming

The previous section described the EBP paradigm. It showed how applications are decoupled by using the implicit invocation pattern. Destination components interested in the changes of source components get notifications when these changes arise, without sources being coupled to destinations. However, there are some problems that cannot be

solved by using EB or OO techniques: *preplanning*, *tangling* and *scattering*. As an example, the implementation of a figure editor using JAVA Listeners (see Listing 2.6) shows the needs for defining relevant events in the class `Figure` and triggering them within different methods of the class. This is also the case for the other approaches described in the previous section. If a subclass includes other methods that change the figure, proper notifications of the events have to be included, too. Preplanning refers to the necessity of planning in advance the events that destinations could need in the future use of the class. Scattering refers to the fact that the notifications of the event are present at many places in the code. Tangling refers to the fact that a method has to notify an event, whereas the notification is not part of its role. AOP provides a solution to these problems by modularizing the event definitions.

To illustrate the contribution of AOP the remainder considers an implementation of the figure editor in which figures do not know views interested in their changes and there is not explicit notification of events either. This section will show how the views can get notified of changes in the figures in a transparent way by using aspects.

### 2.2.1 Introduction

AOP is the convergence point of a group of technologies that emerged to address the limitations of OO technology in achieving SoC across more that one dimension. Starting in the late 90's, AOP has been the subject of wide research and nowadays it covers the whole software development process with the acronym AOSD (Aspect-Oriented Software Development).

At the end of the 90's, Kiczales et al. [KLM+97] observed that there were some design decisions that were difficult to cleanly capture in actual code due to the fact that they *cross-cut* the basic functionality of the system, they called them *aspects*. They observed that contemporary languages (either procedural, functional or object-oriented) did not offer mechanisms to cleanly modularize these design decisions. Whereas those languages were well-suited to implement a functional decomposition of a system, implementation of aspects, associated to non-functional concerns, was forced *to be scattered throughout the code, resulting in "tangled" code*, making code excessively difficult to develop and maintain. Kiczales et al. [KHH+01] introduced the ASPECTJ extension to JAVA, a language that permits the expression of crosscutting concerns in a modular way. AOP is nowadays an accepted paradigm and ASPECTJ is considered the AOP standard for JAVA.

In technical terms, the "traditional" premise is that a functional decomposition of a system is implemented using "standard" components (modules in a procedural or functional style, classes in an object-oriented style), producing what is called a *base application*. Aspects implement the crosscutting concerns by identifying in the base application places at which the code implementing the crosscutting concerns has to be invoked. Later on, at runtime, the aspects are *implicitly invoked* at the described points.

The understanding of an aspect-oriented program as consisting of a base application and aspects is considered as an *asymmetric view*. Hyper/J [TOHS99] is another approach to composing concerns that can be qualified as *symmetric*: there are no distinguished base concerns. A concern is composed of a set of plain JAVA classes, a *hyperSlice*, and concerns are composed together as specified by a *hyperModule*. A *hyperModule* specifies relationships between program points from the hyperSlices, *i.e.*, classes, class and instance members, defining how these program points should be composed using basic operations such as renaming and merging. Such an approach is by essence more regular than an asymmetric

approach. It does not mean it is simpler. An essential source of complexity is the fact that there is a considerable gap between the (static) program and its execution. For instance, in the absence of a notion of slice interface, looking at an individual slice does not give a clue of the role of the slice in the composition.

There are several approaches associated to AOSD at the different stages of the software development. This dissertation is mostly interested in approaches at the language level such as AspectJ, CaesarJ [AGMO06], AspectS [Hir02], and Classpects [RS05, RS09].

The remainder describes the different AOP concepts and mechanisms. These mechanisms are divided in two: behavioral aspects, which change the behavior of a base application, and structural aspects, which change its structure.

### 2.2.2 Behavioral Aspects

This section clarifies the concepts associated to AOP. It takes definitions from the book *Aspect-Oriented Software Development* [FECA05] and uses the language AspectJ (and sometimes others) to exemplify the concepts.

#### 2.2.2.1 Join-Point Model

In general, aspects are invoked on well-defined points in the program execution called *join points*. The possible kinds of join points are described in a *join-point model*. Method calls, method executions, access to object fields are typical examples of join points.

> **Definition 2.6.** *Join points are well-defined places in the structure or execution flow of a program where additional behavior can be attached. A join point model (the kinds of joint points allowed) provides the common frame of reference to enable the definition of the structure of aspects.*
>
> Filman et al. [FECA05]

#### 2.2.2.2 Pointcut Model

The join points that are relevant to an aspect are selected using predicates called *pointcuts*. In most aspect languages, most notably in AspectJ, a pointcut is an expression combining *a set of dedicated predicates that reason about the entire possible set of join points in a software application* [BCRD08]. In AspectJ these dedicated predicates are called *pointcut designators*. A pointcut uses these predicates to selects the subset of the application join points that are considered relevant to the aspect.

> **Definition 2.7.** *A pointcut is a relationship 'join point -> boolean', where the domain of the relationship is all possible join points.*
>
> van den Berg et al. [vdBCC05]

Listing 2.19 shows a pointcut written in AspectJ. It selects the executions of the method `moveBy` of `Figure` objects. The pointcut designator `execution` means the execution of a method and is used together with a pattern that identifies such a method. Similar constructs are provided by the AspectJ language to select method calls, field accesses, between others. In addition, a pointcut can obtain contextual information about the join

```
pointcut change(Figure figure):
        execution(* Figure.moveBy(..,..)) && this(figure)
```

Listing 2.19: ASPECTJ pointcut.

point using particular pointcut designators. For example, the `this` pointcut designator of the example is used to get the contextual object involved in the method execution (the one that is obtained when using `this` in the method body).

ASPECTJ provides a variety of pointcut designators, which make it possible to select precise points in the program execution. Most ASPECTJ pointcut designators can be tied to specific places in the code of an application. The pointcut designators `cflow` and `cflowbelow` are two exceptions. They make it possible to define pointcuts that depend on the runtime state of the program execution: its control flow. Besides the pointcut designator `cflow`, it is possible to say that ASPECTJ pointcuts are static because they cannot refer to dynamic context: variables or methods in the scope of the structure where they are defined. Even if ASPECTJ includes a pointcut designator `if`, which makes it possible to test conditions, this designator can only use static variables, static methods or the variables declared in the pointcut.

```
?jp matching
  reception(?jp, ?methodName, ?args),
  method(?methodName, moveBy),
  class(?class, Figure),
  inObject(?jp, ?figure)
```

Listing 2.20: CARMA pointcut.

Languages such as CAESARJ and ASPECTWERKZ [BV04] provide ASPECTJ-like pointcuts. A different model is adopted by languages such as CARMA [GB03] and ALPHA [OMB05], which define pointcuts using logical queries. Listing 2.20 expresses the previous pointcut in CARMA.

The pointcut designators in these languages correspond to logical rules. In this code, the rule `reception` selects the reception of a message, the rules `method` and `class` select static program structures, the rule `inObject` selects the contextual object of the join point. Differently to ASPECTJ-like pointcuts, logical pointcuts are not static. They permit the inclusion of contextual variables in the process of matching a join point. Indeed, Gybels et al. [GB03] present CARMA as a dynamic pointcut language.

#### 2.2.2.3 Advice

The aspect functionality is implemented in *pieces of advice*. In general, a piece of advice binds a pointcut and an action, the *advice body*, to be performed when the pointcut selects a join point. Pieces of advice also make it possible to define the location of the action: before, after, or instead of (around) the join-point execution.

> **Definition 2.8.** *Advice is the behavior to execute at a join point. For example, this might be the security code to do authentication and access control. Many aspect languages provide mechanisms to run advice before, after, instead of, or around join points of interest.*
>
> Filman et al. [FECA05]

```
after(Figure figure) : change(figure) {
  view.repaint(figure.getArea());
}
```

Listing 2.21: AspectJ after advice.

Listing 2.21 shows a piece of advice, written in AspectJ, which binds the pointcut of Listing 2.19 with code that is executed after the execution of a join point matched by `change`. The syntax of pieces of advice with before control is similar to the syntax used in Listing 2.21, but the keyword `before` is used instead of `after`. In such a case the piece of advice is executed before the join point.

```
void around(Figure figure) : change(figure) {
  if(!figure.isReadOnly())
    proceed(figure);
}
```

Listing 2.22: AspectJ around advice.

In particular, pieces of advice tagged as *around* are executed instead of the join point. The keyword `proceed` offers then the possibility of performing the original join-point execution inside the advice body. As an example, Listing 2.22 shows a piece of advice, written in AspectJ, which binds the pointcut of Listing 2.19 with code that executes the initial join point, through the use of `proceed`, only when the figure is not read-only.

```
1 /** @Around execution(* Figure.moveBy(..)) **/
2 void myMethod(JoinPoint joinPoint) {
3   Figure figure = (Figure) joinPoint.getThis();
4   if(!figure.isReadOnly())
5     joinPoint.proceed();
6 }
```

Listing 2.23: Advice in AspectWerkz.

Whereas in AspectJ the advice body corresponds to a Java block of code (with an extended syntax including the keyword `proceed` in case of around control), the AspectWerkz language defines a piece of advice as an annotated method. The annotation binds the method with some pointcut. Such a method receives an argument of type `JoinPoint` as parameter, which corresponds to the reification of the current join point and can be used to get its parameters or to proceed. Listing 2.23 shows the AspectWerkz version of the AspectJ piece of advice of Listing 2.22. In Line 3, a call to the method `getThis` on the `JoinPoint` object obtains the receiver of the call `moveBy`, whereas in line 5 the method `proceed` is called in order to execute the join point.

```
1 adviceFigureMovement
2   ^ AsAroundAdvice new
3      pointcut: ( AsJoinPointDescriptor
4                    targetClass: Figure targetSelector: #moveBy )
5      aroundBlock: [:receiver :arguments :aspect |
6                      receive isReadOnly ifFalse: [ aspect proceed ] ]
```

Listing 2.24: Advice in AspectS.

Like in Java, the AspectS [Hir02] extension to SmallTalk provides advice bodies as blocks. However, differently to Java blocks, SmallTalk blocks are first-class closures. Thus, advice blocks in AspectS can receive parameters such as the receiver of the message, the arguments passed in the call and a representation of the aspect (which can be used to proceed), while being more flexible than a method. Listing 2.24 implements in AspectS the AspectJ piece of advice of Listing 2.22. Note in Line 6 that `proceed` is a message sent to the third parameter received by the advice block.

#### 2.2.2.4 Aspect Definition and Instantiation

Pointcuts and pieces of advice are put together in a module called *aspect*. An aspect encapsulates the implementation of a concern.

> **Definition 2.9.** *An* aspect *is a modular unit designed to implement a concern. An aspect definition may contain some code (or advice [...]) and the instructions on where, when, and how to invoke it.*
>
> Filman et al. [FECA05]

Most aspect-oriented approaches conform to an asymmetric decomposition technique: aspects are defined as a special kind of modules separating crosscutting concerns. Aspects are equipped with pointcuts and pieces of advice and can share elements from the base language in which they have been embedded such as methods and fields. In languages such as AspectJ and AspectWerkz, for example, aspects are defined similarly to standard classes, but they follow particular instantiation strategies. Differently to classes, aspects cannot be explicitly instantiated. An aspect is implicitly instantiated by the language at load time, by default as a singleton. AspectJ also provides *per-object* implicit instantiation for aspects defined by using `perthis(`*pointcut*`)` or `pertarget(`*pointcut*`)`. These strategies instantiate one aspect per specified object. The pointcuts of an aspect instantiated for a particular object, using `perthis` or `pertarget`, can only select join points associated to the corresponding object, *i.e.* join points in which the result of the designator `this` or `target` is the corresponding object, respectively. AspectJ also provides strategies that instantiate an aspect for particular flows of execution, defined by using `percflow(`*pointcut*`)` or `percflowbelow(`*pointcut*`)`.

As an example, Listing 2.25 defines an AspectJ aspect that, given a view and a list of relevant figures, updates the view as soon as any of the relevant figures change.

AspectJ aspects can include standard class members, which are accessible in advice code, *i.e.* advice code is instance-specific. For example, Line 8 of Listing 2.25 uses the instance variable `relevantFigures` to test whether the figure bound by the pointcut is a relevant figure. This is, however, not the case for pointcuts, which are static. The pointcut

```
1 aspect Aspect {
2   DrawingView view; List relevantFigures;
3
4   pointcut change(Figure figure):
5            execution(* Figure.moveBy(..)) && this(figure)
6
7   after(Figure figure) : change(figure) {
8     if(relevantFigures.contains(figure))
9       view.repaint(figure.getArea());
10  }
11 }
```

Listing 2.25: ASPECTJ aspect.

of Line 4 observes the execution of all the figures of the editor, not only the relevant ones. Even if ASPECTJ makes it possible to use a pointcut designator `if` to test conditions when matching a join point, the condition cannot include instance-level context.

### 2.2.3 Structural Aspects

Structural aspects make it possible to change the static structure of a program. ASPECTJ provides a mechanism called *inter-type declarations*, also known as *introductions*, which makes it possible to change the structure of a base application by introducing members or ancestors to a target class or interface.

```
1 aspect SubjectObserverAspect {
2   declare parents: Figure extends Subject;
3   declare parents: DrawingView implements Observer;
4
5   public void DrawingView.update(Subject s) {
6     repaint(((Figure)s).getArea());
7   }
8
9   pointcut change(Subject s):
10    execution(void Figure.moveBy(..)) && this(s);
11
12  after(Subject s): change(s) {
13    for (Observer obs: s.observers)
14      obs.update(s);
15  }
16 }
```

Listing 2.26: ASPECTJ aspect using inter-type declarations.

For example, the ASPECTJ aspect of Listing 2.26 uses inter-type declarations for equipping the classes of the figure editor with the infrastructure of the pattern Observer. Lines 2 and 3 set `Subject` and `Observer` as ancestors of `Figure` and `DrawingView`, respectively. A figure can, in this way, keep a list of observers interested in its changes and a drawing view can register itself as an observer of a figure. By using inter-type declarations, Line 5 implements the method `update` (declared in the interface `Observer`) in the class `DrawingView`.

The pointcut `change` of the aspect is similar to the pointcut `change` of Listing 2.25. However differently to the aspect of Listing 2.25, upon a change in a figure, the piece of advice of the aspect of Listing 2.26 only notifies the observers registered for the figure.

```
interface Subject {
  void addObserver(Observer obs);
  void removeObserver(Observer obs);
}

abstract aspect SubjectObserverProtocol {
  Collection<Observer> Subject.observers = new Vector<Observer>();
  public void Subject.addObserver(Observer obs) {...}
  public void Subject.removeObserver(Observer obs) {...}

  abstract pointcut change(Subject s);

  after(Subject s): change(s) {
    for (Observer obs: s.observers)
      obs.update(s);
  }
}
```

Listing 2.27: ASPECTJ aspect using inter-type declarations.

JAVA does not support multiple inheritance so that setting `Subject` as a superclass of `Figure` is only possible because the latter has not been defined with a superclass. If a class already has a superclass, inter-type declarations can still be used for manually including all the members of `Subject` in such a class. An additional feature of ASPECTJ avoids this complexity. Inter-type declarations make it possible to associate an implementation to the members declared in an interface. Setting the interface as an ancestor of a class makes the associated implementation part of the class. For example, the aspect of Listing 2.27 uses inter-type declarations to associate an implementation to `Subject`, defined this time as an interface. Setting `Subject` as a super-interface of `Figure` in Listing 2.28 makes the associated implementation part of the class.

```
aspect SubjectObserverProtocolImpl extends SubjectObserverProtocol {
  declare parents: Figure implements Subject;
  declare parents: DrawingView implements Observer;

  public void DrawingView.update(Subject s) {
    repaint(((Figure)s).getArea());
  }

  pointcut change(Subject s):
    execution(void Figure.moveBy(..)) && this(s);
}
```

Listing 2.28: ASPECTJ aspect.

The aspect of Listing 2.27 modularizes the infrastructure of the pattern Observer, making it possible to apply the pattern to any application with a small amount of glue

code. The abstract pointcut `change` and the corresponding piece of advice encapsulates the subject-observer protocol. Listing 2.28 applies the pattern-Observer infrastructure to the figure editor (1) by setting `Subject` and `Observer` as ancestors of `Figure` and `DrawingView`, respectively, (2) by introducing an implementation of the method `update` in `DrawingView`, (3) by providing a concrete implementation of the pointcut `change`.

As seen in this section, the structural aspects of AspectJ complement the behavioral aspects. Structural aspects are used to add the fields supporting the infrastructure, behavioral aspects are used to capture the changes and trigger notification when they occur.

### 2.2.4 AOP Approaches

Besides the AspectJ-like approaches, some *modern* AOP approaches exist that somehow alter some of the elements of the classical view of AOP. On the one hand, some authors have observed similarities between the mechanisms of EBP and the mechanisms of AOP. Some of them propose a link between the implicit invocation mechanisms of both paradigms, in particular, a link between join points and events:

> *An AspectJ joinpoint with aspect code to be attached to it can easily be seen as an event-based system: when the event described by the joinpoint occurs, the specified code (essentially, a callback) is executed. Pointcuts are a means of describing higher-level events in terms of primitive ones.*
>
> Walker and Murphy. [WM01]

This section describes EAOP [DFS04] and Ptolemy [RL08] as two representative approaches taking these observations into account. On the other hand, languages like Ptolemy treat aspects as plain classes. There is no specific `aspect` construct as in AspectJ. This feature is shared by languages such as AspectWerkz [BV04], Spring [JHD+10], JBoss AOP [JBo10], Classpects [RS05] and CaesarJ [AGMO06]. These languages aim to break down the differences imposed by the asymmetric view of AOP. This section describes Classpects and CaesarJ as the most representative proposals in this regard.

#### 2.2.4.1 EAOP

Filman et al. [FH02] describe how AOP meets *the need to be able to respond to sequence of events*. AOP applications requires *to be able to quantify over anything that changes the data or program counter state of the abstract machine executing a given program*. Whereas the abstract interpreter is, in general, not completely accessible at the programming level, much of the program code (text or byte code) is accessible and most dynamic events are tied to places in such code.

Event-based AOP (EAOP) [DFS04] is an aspect-oriented (AO) model that considers join points as execution events occurring in an application. Aspect weaving is modeled as a monitor that *handles* these events by inserting instructions according to execution states. EAOP is richer than AspectJ-like approaches in that pointcuts describe sequences of events taken from the history of computation. An aspect describes the instructions to perform along specific event sequences. A big contribution of EAOP is a framework for detection and resolution of aspect interactions based on sequences of events.

The link between events and join points can also be observed in Concurrent EAOP (CEAOP) [DLBNS06], an extension of EAOP for concurrent settings. The approach proposes a translation from aspects into Finite State Processes [MK06]. An aspect is expressed in terms of actions and atomic events. These events coordinate several aspects together.

#### 2.2.4.2 PTOLEMY

PTOLEMY [RL08] is a language that combines the ideas of implicit invocation of EBP with ideas from AO languages. PTOLEMY is a simple OO language that includes event types. An event of a given type can be triggered at sources as an imperative event and quantified at destinations. *The language has classes, objects, inheritance and subtyping, but it does not have* `super`*, interfaces, exception handling, built-in value types, privacy modifies, or abstract methods.* Let us explain the main features of the language by means of an example.

```
1  Figure evtype Change {
2    Figure changedFig;
3  }
4  class Figure {
5    void moveBy() {
6      Figure changedFig = this;
7      event Change { /* code changing the figure */ this }
8    }
9  }
10 class Aspect extends Object {
11   DrawingView view; List relevantFigures
12   Aspect init() {
13     register(this)
14   }
15   Figure update(thunk Figure next, Figure changedFig) {
16     Figure res = invoke(next);
17     if(relevantFigures.contains(changedFig))
18       view.repaint(changedFig.getDrawingArea());
19     res
20   }
21   when Change do update
22 }
```

Listing 2.29: PTOLEMY example.

Listing 2.29 implements an aspect that updates a view when a group of figures of an editor change. Events are supported by means of event types. `Change` is an event type representing a change in a figure (Line 1). It declares a return type of type `Figure` and has as data the figure that changes. An occurrence of the event type is triggered with the keyword `event` and a block of code, hereafter referred to as the *event expression* or the *event's code*. The event expression is defined as returning an instance of the return type declared by the event type (Line 7). Its evaluation can be seen as defining an *explicit* join point. PTOLEMY makes it possible to replace the execution of the event expression by a call to a *handler method*. For example, the method `update` of the class `Aspect` (Line 15) can be used as a handler method for an event of type `Change`. The handler takes an *event closure* as its first argument, the data of the event type as second argument and returns

an instance of the return type declared by the event type. *An event closure contains code needed to run the applicable handlers and the original event's code.* The call to `invoke` runs the event closure (Line 16), which can be interpreted as raising the next handler or the event expression if there is no more handlers. After running the rest of the handlers and the event expression, the method repaints the view. The registration of the handler `update` to events of type `Change` is made in two steps. First, the class includes a statement `when`, which statically sets the method `update` as a handler of events of type `Change` (Line 21). Second, the instance of `Aspect` has to register itself as an observer of the events at runtime (Line 13).

Ptolemy combines the global quantification of AOP with the explicit triggering of EBP. Similarly to an AspectJ aspect, the observation of a Ptolemy aspect is global. In the previous example, an instance of `Aspect` observes all possible events of type `Change` occurring for all possible instances of `Figure`. Ptolemy unifies imperative events and join points. Whereas in EAOP a join point is seen as an implicit event occurring as a result of a computation and is defined declaratively, in Ptolemy a join point is an imperative event explicitly triggered as the execution of a block of code.

#### 2.2.4.3 CaesarJ

The language CaesarJ [AGMO06] is an extension to Java. An aspect is implemented as a class, which can include AspectJ pointcuts and pieces of advice. It uses the keyword `cclass` for classes supporting the specific CaesarJ features. The keyword `class` is reserved for pure Java classes for backwards compatibility. In addition, CaesarJ implements virtual classes and family class polymorphism, together with propagating mixin composition (Section 2.3.1 describes this feature).

```
1 cclass Aspect {
2   DrawingView view; List relevantFigures;
3
4   pointcut change(Figure figure):
5            execution(* Figure.moveBy(..)) && this(figure);
6
7   after(Figure figure) : change(figure) {
8     if(relevantFigures.contains(figure)) {
9       view.repaint(figure.getArea());
10    }
11  }
12 }
```

Listing 2.30: CaesarJ aspect.

As an example, Listing 2.30 implements an aspect that updates a view when a group of figures change. Note how the pointcuts and the piece of advice are similar to the ones implemented by the AspectJ aspect of Listing 2.25. Indeed, similarly to AspectJ, pieces of advice are dynamic since they can use instance-level state and pointcuts are static so that they observe the global occurrence of join points. The main difference is, however, the fact that a CaesarJ aspect is a class, and thus it can be explicitly instantiated. In Listing 2.30, the class `Aspect` can be explicitly instantiated for different views and for different groups of figures. When the pointcut `change` matches a join point, each aspect

instance will be notified and its corresponding view will be repainted if the figure that changes corresponds to a relevant figure.

#### 2.2.4.4 CLASSPECTS

CLASSPECTS [RS05] aspects are plain classes that have been equipped with pointcuts and pieces of advice similarly to CAESARJ aspects. A novelty of this language is the concept of instance-level advising and the fact that advice bodies are implemented as plain methods.

```
1  class Aspect {
2    DrawingView view;
3
4    Aspect(DrawingView view, List figures) {
5      this.view = view;
6      for(Object fig: figures)
7        addObject(fig);
8    }
9    void around execution(* Figure.moveBy(..)) && this(fig) && aroundptr(d):
10         onChanged(Figure fig, AroundADP d);
11
12   void onChanged(Figure figure, AroundADP d) {
13     view.repaint(figure.getArea());
14     d.innerInvoke(); /* equivalent to a proceed call */
15   }
16 }
```

Listing 2.31: CLASSPECTS aspect.

Listing 2.31 implements an aspect that updates a view when a group of figures change. The functionality of a piece of advice, *i.e.* its body, is a plain method in CLASSPECTS. A *join-point-method binding* construct ties together a pointcut and a method, such that when a join point is matched the method is executed (Line 9). Similarly to CAESARJ, the class can be instantiated for different views and groups of figures. However, in CAESARJ the aspect observes the join points associated to all the figures of the editor, not only the ones it is interested in. Consequently, the piece of advice has to test whether the figure is in its list of relevant figures before repainting the view as in Line 8 of Listing 2.30. The instance-level advising of CLASSPECTS makes it possible to associate an aspect with a group of relevant objects, by using the keyword `addObject`. As a result, the pointcuts of the aspect will only match join points in the context of the relevant group of objects avoiding additional tests in the pieces of advice. For example, Line 7 of Listing 2.31 associates the aspect with each figure passed as parameter in the constructor and therefore the piece of advice does not need to include any test.

#### 2.2.4.5 Comparison

**Implicit vs. explicit instantiation.** In CLASSPECTS, explicit instantiation is motivated by previous work of Sullivan et al. [SGC02], which has shown that the implicit instantiation scheme, as provided by AspectJ, does not properly cover the implementation of *behavioral relationships*. Indeed, ASPECTJ makes it easy to associate an aspect instance to a class

instance but does not make it possible to directly associate an aspect instance to two or more object instances (as in *Association aspects* [SMU+04]). In CaesarJ, explicit instantiation is key in managing dynamic deployment of aspect instances. CaesarJ significantly differs from Classpects with respect to inheritance in that it provides virtual classes and propagating mixin composition, which gives more structuring power. Also, as Section 2.3.2 will show, CaesarJ differs in that it supports structural aspects. However, the principle is the same in both cases: there is a uniformity of treatment between aspect classes and plain classes.

The different implementations of the figure editor shown in this chapter help to better clarify the differences between implicit instantiation and explicit instantiation. In a figure editor there are several views and figures, and each view is interested in a certain group of figures. Listing 2.25 shows an AspectJ aspect with a view and a group of figures as instance variables. Its piece of advice is executed when any figure of the editor changes. A proper condition is used in the advice body to select the relevant figure changes and notify the view when necessary. However, the aspect is a singleton. Consequently, it only makes it possible to communicate a single view with a single group of figures. One may think of using the *per-object* instantiation strategies of AspectJ, instantiating one aspect per view, and then setting a group of relevant figures for each instance at runtime. However, this does not work. An AspectJ aspect instantiated per object only applies its pointcuts to join points associated to the corresponding object. Clearly, the interesting join points are associated to figures and not to views. At the same time, instantiating an aspect per figure could be a solution for the considered example, but this solution is not scalable because the number of figures in an editor is in general much bigger (potentially unbounded) than the number of views. Explicit instantiation makes it possible to explicitly instantiate an aspect. As a result, several aspects exist at runtime, each aspect related to a particular view and a particular group of figures, without constraints such as the ones imposed by the per-object instantiation of AspectJ. This is the case of Ptolemy and CaesarJ, in which an aspect is a class, *i.e.* an *aspectual class* (see Listing 2.29 and Listing 2.30, respectively). Similarly to the AspectJ aspect of Listing 2.25, each aspectual class observes a change in any figure of the editor and the piece of advice filters the relevant figure changes and notify the corresponding view. Unlike in AspectJ, there are several aspectual-class instances. Explicit instantiation is also implemented in Classpects. Additionally, this language makes it possible to restrict the observation of an aspectual class to its relevant group of figures as Listing 2.31 shows. Thus, differently to Ptolemy or CaesarJ pieces of advice do not need to filter relevant figure changes.

Structural AspectJ aspects provides an acceptable solution with a singleton aspect as Section 2.2.3 shows. The pattern Observer is superimposed in the figure editor such that a figure keeps a list of observers. The implementation makes it possible to tie together arbitrary views and arbitrary groups of figures, and the aspect implements the communication protocol in an oblivious way. However, the solution is more verbose than using explicit instantiation.

**Aspect members.** With respect to pointcuts and advice, CaesarJ follows AspectJ. Classpects diverts from it when dealing with advice. As this section has shown, pieces of advice are anonymous. The lack of name results in the impossibility of selecting a specific piece of advice, which is an issue when implementing *higher-order concerns* [RS05], *i.e.*, aspects that advise aspects. Classpects solves this issue by replacing the advice body by a method call.

ASPECTJ localizes interactions with the join point in the advice body. This includes exposition of context information and actual execution of the join point in an around piece of advice via `proceed`. This is not possible in CLASSPECTS as the advice body is reduced to a method call. Instead, join point information, including a closure representing the join point, is accessed via specific primitive pointcuts. This data can then be freely used in methods. The result of all this is that the advice body, reduced to a method call, can now be easily selected by a pointcut `call`.

This leads the authors of CLASSPECTS to say that advice is eliminated in favor of methods. This is using a slightly different terminology from the one this chapter has used so far: in CLASSPECTS, a piece of advice is called a *binding declaration* and the corresponding advice body is called *advice*. This solution solves the issue of selecting advice bodies but the piece of advice itself remains anonymous and cannot be redefined.

### 2.2.5 Obliviousness, Quantification and Modularity

The term *obliviousness* in AOP refers to the fact that by looking at a piece of code in a base program, it is not possible to be aware of the action of aspects on such a piece of code. *Obliviousness is desirable because it allows greater separation of concerns in the system creation process - concerns can be separated not only in the structure of the system, but also in the head of the creators* [FECA05]. EBP, for example, does not enable complete obliviousness. An EB program explicitly trigger events. Consequently, it is possible to predict that some action may take place even though neither the observers not their actions are known.

Complete obliviousness is good because it avoids tangling. However, it requires powerful mechanisms of *quantification*. Quantification refers to the ability of a language to talk about precise execution points in the base program. Quantification is needed in AOP for inserting crosscutting functionality at the right places.

AOP is said to be a mix of quantification plus obliviousness [FF05]. It improves the modularity of software as it makes it possible to modularize crosscutting concerns. However, complete obliviousness does not make it possible to build truly modular systems. True modularity requires a contract between the interfaces of two components, so that it is not possible to modify the interface of one of them without the other being aware of such a change. In AOP, a base program can easily break the assumptions made by aspects, which is known as the pointcut fragility problem [KS04].

### 2.2.6 Summary

This section has described Aspect-Oriented Programming. This paradigm makes it possible to modularize crosscutting concerns by using implicit invocation, which is enabled through the use of a pointcut-advice model. By using pointcuts, aspects identify points in the execution of a program in an oblivious way, *i.e.* without the sources of these computations being aware of the aspect observations. Then, aspects attach dedicated pieces of code (pieces of advice) implementing the crosscutting concerns to these execution points. Most AO approaches are asymmetric because they make a separation between a base application and aspects. Some approaches such as CAESARJ and CLASSPECTS propose to eliminate aspects as dedicated language constructs in the favor of a more regular design. Other approaches such as PTOLEMY try to make AOP and EBP closer by modeling join points as special kinds of events. Languages such as ASPECTJ complements the behavioral

vision of AOP with structural aspects, which makes it possible to change the structure of an application, also in an oblivious way.

## 2.3 Advanced OOP

The modules in conventional programming languages are not sufficiently extensible: they do not provide mechanisms to extend definitions of existing classes and functions. An individual class can be extended by means of inheritance, but this leads to creating a new class rather than to refining the existing class, because the subclass does not automatically replace its superclass in its relationships with other classes.

This section describes how *mixin* technology can improve the modularization of functional concerns. Mixins, first introduced in the Flavors system [Moo86b] and CLOS [Kee89], are class definitions parametrized over the superclass. *By providing an abstraction mechanism for inheritance, mixins remove the dependency of the subclass on the superclass, enabling modular development of class hierarchies* [BPS99].

This section presents mixins as an advanced OOP technique, even if it can be "almost" considered as a symmetric AOP approach like HyperJ:

> *The scope of quantification is controlled by which classes inherit the mixin. That is, we can quantify over the descendants of some superclass for a given single method.[...] Except that class inheritance relationships are part of a class's definition, we would have an AOP system.*[FECA05]

For the sake of this dissertation, this section illustrates mixin technology as supported by the language CaesarJ.

### 2.3.1 Virtual Classes and Propagating Mixin Composition

Mixins are implemented in CaesarJ in the form of *virtual classes*. Composition of mixins is enabled by *propagating mixin composition*. The concept of a virtual class stems from the Beta programming language [MMP89] and was further developed in gbeta [Ern99a], which introduced propagating mixin composition [Ern99b] and a type-safe family polymorphism [Ern01]. CaesarJ integrates these concepts to Java. It has different method overriding semantics than Beta, and supports abstract methods and classes.

*Virtual classes* are late-bound inner classes. Like virtual methods, they can be refined in subclasses of the enclosing class. They can also be considered as members of the objects of the enclosing class. Objects with class members are called *family objects*; their members are instances of their virtual class members. Analogously, the enclosing classes are referred as *family classes*. A refinement of a virtual class, also known as a *further binding*, implicitly inherits from the class it refines. In the further binding we can add new methods, fields and inheritance relationships as well as override the inherited methods. In each family class all references to a virtual class are always bound to its most specific refinement. Virtual classes enable application of class inheritance on a larger scale: Inheritance between two family classes is in principle inheritance between the groups of classes constituting these families.

Large-scale multiple inheritance is enabled by *propagating mixin composition*. Any class in CaesarJ, including family classes and virtual classes, can be used as a mixin. They are parameterized over their super parameter, *i.e.*, rather than extending a concrete superclass,

they extend a super parameter, which is not bound at their definition time but at composition time. Mixin composition in CaesarJ is a form of multiple inheritance, which is based on linearization of the inheritance graph. Propagating mixin composition means that the composition propagates into virtual classes: all inherited declarations of virtual classes with the same name are automatically composed using the same composition semantics.

#### 2.3.1.1 Modularization in Java

As an example, let us consider the modular development of a figure editor. A first component implements the basic functionality of figures and a drawing view. A second component extends the basic editor with drawing functionality.

```
1  interface IFigure {
2    void moveBy(int x, int y);
3  }
4
5  abstract class Figure
6           implements IFigure {
7    void moveBy(int x, int y) {
8      ...
9    }
10   ...
11 }
12
13 class Circle extends Figure {
14   void setRadius(double r) {
15     ...
16   } ...
17 }
18
19 class DrawingView {
20   Collection<IFigure> figures;
21   void add(IFigure f) {
22     figures.add(f);
23   } ...
24 }
```

Listing 2.32: Basic figure editor in Java.

Figure 2.32 implements the basic figure editor in Java. The interface `IFigure` declares the methods of any figure in the editor. The class `Figure` represents the superclass of any "primitive" figure.§ It implements the basic methods of figures such as `moveBy`. The class `Circle` implements a particular type of figure: a circle. The class `DrawingView` implements the aggregation of several figures.

```
1  interface IDrawableFigure
2            extends IFigure {
3    void draw(Graphics g);
4  }
5
6  abstract class DrawableFigure
7        extends Figure
8        implements IDrawableFigure { }
9
10 class DrawableCircle
11       extends DrawableFigure {
12   void draw(Graphics g) {
13     ...
14   }
15   void setRadius(double r) {
16     ...
17   } ...
18 }
19
20 class DrawableDrawingView
21       extends DrawingView {
22
23   void repaint() {
24     for(IFigure fig: figures) {
25       ((IDrawableFigure)fig).draw(g);
26     }
27   } ...
28 }
```

Listing 2.33: Extension of the basic figure editor with drawing functionality in Java.

§. The editor can have composite figures, but they are not relevant to this section.

Figure 2.33 implements the extension of the basic editor with drawing functionality. It provides a new interface `IDrawableFigure`, which extends `IFigure` with the method `draw`. The abstract class `DrawableFigure` extends `Figure` with the abstract method `draw` of `IDrawableFigure`. All the subclasses of `Figure` implemented in the basic editor need to be redefined in order to implement the new method. For example, the class `DrawableCircle` implements the redefinition of `Circle`. It extends `DrawableFigure` and provides a proper implementation of `draw`. Since JAVA does not support multiple inheritance, `DrawableCircle` cannot inherit from `Circle` and the methods such as `setRadius` have to be manually copied in the new class. An extension of `DrawingView` is also provided. It implements a method `repaint` by invoking the method `draw` for each aggregated figure.

The implementation of the figure editor in JAVA does not support type-safe extension with new functionality. The design of the extension presented in Figure 2.33 is not type-safe. For example, the type system does not prevent the programmer from aggregating an instance of `Circle` to an instance of `DrawableDrawingView`. As a result, a type cast is required at line 25 to call the newly introduced method `draw`. This type cast would cause a runtime error if an object with an inappropriate type is set.

#### 2.3.1.2 Modularization in CAESARJ

```
1  cclass Editor {
2    abstract cclass IFigure {
3      abstract void moveBy(int x, int y);
4      ...
5    }
6
7    abstract cclass Figure
8             extends IFigure {
9      void moveBy(int x, int y) {
10       ...
11     }
12     ...
13   }
14   cclass Circle extends Figure {
15     void setRadius(double r) {
16       ...
17     } ...
18   }
19
20   cclass DrawingView {
21     Collection<IFigure> figures;
22     void add(IFigure f) {
23       figures.add(f);
24     } ...
25   }
26 }
```

Listing 2.34: Basic figure editor in CAESARJ.

Figure 2.34 shows the implementation of the basic figure editor in CAESARJ. Declaring the classes with the `cclass` keyword denotes that they have the special CAESARJ semantics¶, i.e., the inner classes implementing the figures and the drawing view are virtual classes. The figure interface is declared as an abstract virtual class.‖ The class `Editor` is a family class.

Since the implementation of the figures and the drawing view are declared as virtual classes they can be *refined* in the subclasses of the family class. For example, Figure 2.35 shows the implementation of the extension with drawing functionality. The class `IFigure` is extended to include the method `draw`. The virtual class `Circle` is also refined in order to implement `draw`. The class `DrawingView` is extended with the method `repaint`.

¶. The classes declared with the `class` keyword in CAESARJ preserve the standard JAVA semantics.

‖. Actually, here is no special syntax for virtual interfaces in CAESARJ. However, this is not a limitation as compared to Java, because CAESARJ supports multiple inheritance for classes.

```
1  cclass DrawableEditor extends Editor {
2    abstract cclass IFigure {
3      abstract void draw(Graphics g);
4    }
5
6    cclass Circle {
7      void draw(Graphics g) {
8        ...
9      }
10   }
11   cclass DrawingView {
12
13     void repaint() {
14       for(IFigure fig: figures) {
15         fig.draw(g);
16     }
17   }
18 }
```

Listing 2.35: Extension of the basic figure editor with drawing functionality in CaesarJ.

Virtual classes implicitly inherit from their predecessor versions. The virtual classes `IFigure`, `Circle` and `DrawingView` of `DrawableEditor` implicitly inherit from the virtual classes `IFigure`, `Circle` and `DrawingView` of `Editor`, respectively. The superclasses are implicitly inherited too, but rebound to their new implementations. For example, in `DrawableEditor` the class `Figure` is not redeclared, but it is still implicitly inherited; the super of the implicitly inherited version is bound to the refined version of `IFigure` in `DrawableEditor`, hence, implicitly inheriting the method `draw`.

Virtual classes are also automatically rebound in type references, which enables a type-safe access to the methods introduced in extensions. For example, at line 15 of Figure 2.35 no type cast is needed to draw a figure. The collection `figures` is declared in `Editor` with elements of type `IFigure`, but in the context of `DrawableFigure`, the reference to `IFigure` is rebound to the refined version of the class, which introduces the method `draw`. The type system also ensures that only the figures of `this` family can be aggregated to the `figures`.

#### 2.3.1.3 Comparison

The type system of CaesarJ enables type-safe extension of figures. Moreover, type-safety is achieved without any sophisticated type declarations or any other code overhead.

```
1  cclass ObservableEditor
2        extends Editor {
3
4    abstract cclass Figure
5           extends Subject {
6      void moveBy(int x, int y) {
7        super.moveBy(x,y);
8        notify();
9      }
10     ...
11   }
12   cclass Circle {
13     void setRadius(double r) {
14       super.setRadius(r);
15       notify();
16     } ...
17   }
18   cclass DrawingView
19         extends Observer {
20     void update(Subject s) {
21       ((IFigure)s).draw(g);
22     }
23   }
24 }
```

Listing 2.36: Extension of the basic figure editor with pattern Observer in CaesarJ.

Since Java does not support multiple inheritance, different Java extensions of the figure editor cannot be composed. This makes it impossible to reuse independent pieces of behavior in different compositions. For example, Figure 2.36 is an extension incorporating

the pattern Observer to the figure editor in CaesarJ. It declares the class `Figure` as a subject and the class `DrawingView` as an observer. The drawable behavior of Figure 2.35 and the observable behavior of Figure 2.36 are independent extensions of the behavior of the figure editor. A figure editor that support both drawable figures and observable figures would need both of them. Composition of such extensions is possible in languages supporting multiple inheritance, but requires a lot of additional code. Extensions of the figure editor are composable in CaesarJ. The previous extensions of the figure editor can be composed using propagating mixin composition. This is done by simply declaring a class `CompleteEditor` that inherits from both `DrawableEditor` and `ObservableEditor`:

```
cclass CompleteEditor extends DrawableEditor & ObservableEditor { }
```

`CompleteEditor` inherits all virtual classes of `DrawableEditor` and `ObservableEditor`, and the virtual classes with the same name are again composed by the same multiple inheritance semantics. This means that `CompleteEditor` implicitly contains virtual classes `IFigure`, `Figure`, `Circle` and `DrawingView`. The compiler checks whether the composition is consistent, otherwise the developer must provide missing method implementations or resolve conflicting method implementations.

Composition of orthogonal behavioral extensions is fully automatic. In case of inconsistencies, only the code for resolving these inconsistencies must be provided; the remaining functionality is still composed automatically.

### 2.3.2 CaesarJ Mixins vs. Structural Aspects

```
cclass ObservableEditor2 extends Editor {
  abstract cclass Figure extends Subject { }

  cclass DrawingView extends Observer {
    void update(Subject s) {
      ((IFigure)s).draw(g);
    }
  }

  pointcut change(Subject s):
    execution(void Figure.moveBy(..)) && this(s);

  after(Subject s): change(s) {
    for (Observer obs: s.observers)
      obs.update(s);
  }
}
```

Listing 2.37: Extension of the basic figure editor with pattern Observer in CaesarJ.

The structural aspects of AspectJ make it possible to introduce new members or new ancestors to classes or interfaces. The concept of virtual-class refinement of CaesarJ makes it possible to achieve the same in the context of a family class. For example, the family class of Figure 2.37 plays the same role as the AspectJ aspect of Listing 2.26. Both introduce `Subject` as an ancestor of `Figure`, `Observer` as an ancestor of `DrawingView` and

an implementation of the method `update` in `DrawingView`. In addition, both include the same pointcut and the same piece of advice.

```
cclass SubjectObserverProtocol {
  abstract cclass Subject { ... }
  abstract cclass Observer { ... }

  abstract pointcut change(Subject s);

  after(Subject s): change(s) {
    for (Observer obs: s.observers)
      obs.update(s);
  }
}
```

Listing 2.38: Extension of the basic figure editor with pattern Observer in CaesarJ.

The AspectJ modularization of the pattern-Observer infrastructure presented in Listing 2.27 can be programmed using family classes as in Figure 2.38. The family class includes `Subject` and `Observer` as virtual classes, the abstract pointcut `change`, and the corresponding piece of advice. Differently to the AspectJ implementation, `Subject` is a normal class. It is not necessary to define it as an interface and include verbose code to add members to such an interface as in Listing 2.27. Indeed, CaesarJ supports multiple inheritance. Figure 2.39 implements in CaesarJ the analogous of the AspectJ aspect of Listing 2.28. It introduces `Subject` and `Observer` as ancestors of `Figure` and `DrawingView`, respectively, and implements the abstract pointcut `change`.

```
cclass ObservableEditor extends Editor & SubjectObserverProtocol {
  abstract cclass Figure extends Subject { }

  cclass DrawingView extends Observer {
    void update(Subject s) {
      ((Figure)s).draw(g);
    }
  }

  pointcut change(Subject s):
           execution(void Figure.moveBy(..)) && this(s);
}
```

Listing 2.39: Extension of the basic figure editor with pattern Observer in CaesarJ.

The mixins of CaesarJ are expressive enough to achieve the same results as when using inter-type declarations, in the context of a family class. However, there are some differences between both mechanisms. On the one hand, a benefit of inter-type declarations is that it refines classes by keeping their same names. In CaesarJ, if a program outside the family class needs to use a refined virtual class, it would need to include the name of the family class that introduced the refinement. The only way to avoid this problem is by defining the program inside the refining family class because in such a context the class references are rebound for the refined virtual classes. On the other hand, CaesarJ

mixins implement multiple inheritance with the associated propagating mixin composition mechanism. These mechanisms include much more flexibility by making it possible to compose different extensions, detect and solve conflicts. As a simple example, if the class `DrawingView` would already have a method `update`, the AspectJ introduction of `update` in `DrawingView` of Listing 2.28 would produce a compile-time error. The CaesarJ solution of Figure 2.39 overrides the method, as expected.

## 2.4 Conclusion

Section 2.1 has described the EBP paradigm. It has shown how applications are decoupled by using the implicit invocation pattern. Destination components interested in the changes of source components get notifications when these changes arise, without coupling sources to destinations. EBP relies on the explicit triggering of events at sources. Section 2.2 has described some problems that cannot be solved by using EBP or OOP: *preplanning*, *tangling* and *scattering*. AOP proposed a solution to these problems by making it possible to define *implicit* events in a modular way with pointcuts. Differently to EB events, AO events are defined at destinations. Section 2.3 has described the use of advanced OOP techniques to solve modularization problems that cannot be solved with *pure* OOP. In particular, the section has described *mixins* as implemented by the language CaesarJ.

AOP, EBP and (advanced) OOP cover different parts of the design space. Whereas AOP is useful when the obliviousness of sources is important, EBP is useful when getting modular sources is more important. Whereas AOP is often used for modularizing non-functional concerns, advanced OOP techniques, such as mixins, are often used for modularizing functional ones. The intersection of the design spaces these paradigms deal with is not empty. This suggests that at some point some combination of the techniques of the different paradigms may be worthwhile. The next chapter analyzes how the different paradigms are integrated together and then presents the problem statement of this dissertation.

# Chapter 3

# Problem Statement

The design of programming languages described by MacLennan [Mac95] puts forward the principles of orthogonality, simplicity and regularity. The orthogonality principle dictates that independent functions of a language should be controlled by independent mechanisms. For example, in class-based OO languages classes are orthogonal to methods as their functions do not overlap. The simplicity principle states that a language should be as simple as possible. There should be a minimum number of concepts with simple rules for their combination. *The small number of built-in data types and operations in a language like PROLOG* [Mac95] is an example of simplicity. One can also think about the design of Smalltalk in which there are only objects and messages. Note that the simplicity principle has to be balanced with providing the necessary abstractions to describe explicitly the intention of programmers. The regularity principle corresponds to a coherent and systematic use of the rules governing the syntax and semantics of these concepts. *The uniform treatment of all data types as predicates and terms is an example of regularity in PROLOG* [Mac95]. One can also think about the design of Smalltalk that treats everything as an object.

The proliferation of programming paradigms requires that languages provide appropriate mechanisms for them. This is a source of complexity in contradiction to the above principles. The weak integration of different programming paradigms in an application increases its complexity. For example, the intensive use of listeners when programming EB functionality in languages like Java increases considerably the size of programs and complicates their understandability. The situation gets worse when this is used in combination with aspects, which in general require a complete new set of concepts.

This situation calls for a better integration of paradigms such as EBP, AOP and OOP in order to provide simpler and more regular languages including all the necessary concepts. This corresponds to the general problem statement of this dissertation, refined in the remainder of this section.

This chapter is structured as follows. Section 3.1 presents the problems of integration between EBP and OOP, mainly related to the lack of regularity of the event construct and the lack of orthogonality of events and methods. Section 3.2 presents the problems of integration between EBP and AOP, mainly related to the lack of orthogonality of advising and event handling. Section 3.3 presents the problems of integration between AOP and OOP, mainly related to the lack of orthogonality of aspects and classes. Finally, Section 3.4 concludes.

## 3.1 EBP and OOP

The classical way to program EB applications with mainstream OO languages such as Java is by using callbacks or listeners. Event handlers are encapsulated as objects conforming to some known interface, which is invoked as notification of the occurrence

of an event. Some mainstream OO languages such as C# have included special language support for events in order to reduce the amount of necessary infrastructure when using callbacks. Events are defined as instance members, are polymorphic and can be used to emit event occurrences, which produce the automatic notification of the registered handlers. Interestingly, events are also designed as instance members in OO languages such as *e* [IEE08] and SystemVerilog [IEE09], focussed on the verification of hardware. However, in none of the above-mentioned languages the event construct is seamlessly integrated with OOP.

### 3.1.1 Regularity of Events

```
delegate void ChangeHandler(Figure source);
class Figure {
  public virtual event ChangeHandler changed;
  void moveBy(int x, int y) { ...
    changed(this);
  } ...
}
class Circle: Figure {
  void setRadius(double r) { ...
    changed(this);
  } ...
}
```

Listing 3.1: C# events cannot be triggered in subclasses.

```
class Circle: Figure {
  override event ChangeHandler changed;
  void setRadius(double r) { ...
    changed(this);
  } ...
}
class Program {
  static void main(string[] args) {
    Circle f = new Circle();
    f.changed += myHandler;
    f.moveBy(10, 50);
  } ...
}
```

Listing 3.2: Events in C# do not support inheritance.

A C# event can be declared abstract. At first sight this makes one think that C# events support inheritance like methods. However, this is not the case. Events cannot be directly triggered in subclasses. For example, in Listing 3.1 the statement that triggers the event `changed` in the method `setRadius` of the class `Circle` produces a compile error because the event is not visible in subclasses of the class `Figure`. Declaring the event public and virtual does not change the situation. The compile error can be avoided by overriding the event in the subclass as shown in Listing 3.2. However, even if the event can

now be triggered in the subclass, it is not associated to the same delegate as the event of the superclass. Both event declarations introduce different private delegates. Consequently, Listing 3.2 entails a runtime error when calling the method `moveBy` in the method `main` because in the private extent of the class `Figure`, the event `changed` is null (it does not have any registered handler). This situation introduces a regularity problem in the language as the rules that govern the inheritance of events are different from the ones governing the inheritance of other instance members.

The situation gets improved in the language *e*. Events can be declared abstract and are subject to inheritance. However, they cannot be refined in subclasses like methods. A subclass can just redefine an event without a mechanism to refer to the definition provided in the superclass like the one available for methods.

### 3.1.2 Orthogonality of Events and Methods

The method invocation and the event handling mechanisms of OOP and EBP are similar. Whereas a method is executed as a result of a method call, an event handler is executed as a result of an event occurrence. Interestingly, in languages such as C# there is no syntactic difference between an event triggering and a method call. The similarity is made more evident in approaches that use message passing like the actor model and Smalltalk. In the actor model the actors of a program (*i.e.* its objects) communicate by message passing. Interestingly, in the actor model [HBS73, Hew10] *events represent the receiving of a message by an actor* [Gre75]. When a message is received, or in other words, when an event is detected at the interface of an actor, a handling process is triggered that enqueues the message if the actor is busy, or that executes a set of actions (aka a method body), otherwise.

The above observations make it possible to conclude that event triggering and method call are not completely orthogonal mechanisms. This conflicts with the principles of MacLennan [Mac95] because C#-like languages provide them as different language constructs.

## 3.2 EBP and AOP

EBP and AOP improve separation of concerns by reducing the coupling between software components. Both use the same mechanism: implicit invocation. In EB systems, an event handler programmed in a component is implicitly executed on the occurrence of an event triggered in another component without keeping an explicit dependency between both components. Similarly, in AO systems, the behavior of an aspect is implicitly invoked in the implementation of other components. The implementers of these other components can be largely unaware of crosscutting concerns. In spite of the fact that both EBP and AOP share the same mechanism, there are no languages offering a seamless integration of event handling and aspect advising. Ptolemy [RL08] has made some interesting steps in this direction. However, it presents some deficiencies, which are described in Section 3.2.2.

### 3.2.1 Orthogonality of Advising and Event Handling

Events and join points are similar concepts. This is made explicit in EAOP, which models join points as events. The differences between the definition of events in EB languages and the definition of join points in AO languages is mainly a matter of *obliviousness*. Whereas, in languages such as AspectJ [KHH+01] join points are declaratively defined

at destinations by using pointcuts (see Listing 2.25), in mainstream EB languages such as C# the occurrences of an event can only be defined by imperatively enumerating them at their source (see Listing 2.12).

Event handlers and pieces of advice play the same role: to react to the occurrence of an event or a join point, respectively. The similarity is explicit by looking at languages like C# and AspectWerkz [BV04], which implement event handlers and pieces of advice as methods, respectively.

Events and join points are similar concepts and the role of event handlers and pieces of advice is analogous. Similar mechanisms provided by different constructs conflicts with the orthogonality and simplicity principles proposed by MacLennan [Mac95].

### 3.2.2 Incomplete Integration Efforts

Ptolemy provides some flavor of EBP and AOP using the same language constructs as Section 2.2.4.2 describes. A join point is an event like in EAOP. However the support for AOP is very limited. Events (join points) cannot be defined declaratively at destination side as in AspectJ. As Listing 2.29 shows, the aspect can only express that it is interested in an event type (the event type `Change`) by using the keyword `when`, but it cannot express when or where the event occurs. The event has still to be imperatively triggered as in C#. Consequently, it is a strong statement to say that Ptolemy is an AOP language or that it integrates both EBP and AOP. The language is mostly an EB language with some quantification capabilities taken from AOP.

### 3.2.3 Advantages of an Integration

Apart from solving the orthogonality problem previously exposed, a better integration of EBP and AOP entails a list of additional advantages.

**Modular events.** In mainstream EB languages the statement that notifies the occurrence of an event is manually written in the code. This produces a phenomenon of tangling because the event notification usually is not part of the logic of the application. At the same time, when the state change represented by an event may happen at multiple places in code, the statement notifying the event becomes scattered over the class or even over multiple classes. For example, Listing 3.1 shows the necessity of triggering an event in different methods of a class and at different places in the class hierarchy. Triggering an event `changed` does not correspond to the role of a method `moveBy` or a method `setRadius`. Compared to the imperative triggering of an event (at different places in code), the declarative definition of a join point has the advantage that it identifies in one place (the pointcut) the different occurrences of the join point (in a modular way). For example, the pointcut of Listing 3.3 identifies in a compact way that a change is produced by the execution of two methods.

```
pointcut change(Figure figure):
        ( execution(* Figure.moveBy(..,..)) ||
          execution(* Circle.setRadius(..))) && this(figure)
```

Listing 3.3: AspectJ pointcut.

The manual event notification mechanism of EB applications requires programmers to preplan the possible events that clients of source classes will need, in order to insert the statements that trigger these events at the corresponding places. If in further uses of these classes, new events are needed, the classes need to be changed in order to insert the respective events. Something analogous happens if the information passed as event reification is incomplete with respect to further requirements. The preplanning is due to the fact that in EB applications events are identified at sources. The declarative definition of join points does not suffer from this issue because join points are defined from the point of view of the observers (aspects), avoiding preplanning at source-side code.

A better integration of EBP and AOP would enable a modular definition of events.

**Coarse-grained join points.** Join points or the data related to the join points are sometimes insufficient to define useful events. Such a selection depends on the expressiveness of the join-point model and the pointcut model of the AO language. In approaches such as AspectS [Hir02], which only consider join points as consisting in method executions, the availability of events depends on the granularity of decomposition into methods. Thus, an oblivious design cannot guarantee availability of all necessary events as join points. The events that do not naturally correspond to the boundaries of methods must still be modeled as events that are explicitly triggered at the appropriate locations within the methods of the class. Languages such as AspectJ improve this situation as they provide a richer pointcut model in which designators can select, besides the method-execution join points, changes or accesses to instances variables. However, they still make it impossible to access local dynamic state or evaluation of control structures such as `if` statements. Events depending on these structures cannot be modeled. A better integration of EBP and AOP would enable definition of aspects advising coarse-grained join points by explicitly triggering them in code as imperative events as in Ptolemy.

As an example, consider the necessity of defining a "general" pointcut `move` representing a movement of any figure. Consider Listing 3.4 with a method `myTransform`. The intention of the programer of this method is that the figure moves twice depending on some conditions (the comments point out the two movements). There is no direct way in AspectJ to identify the two movements in the method `myTransform` by using a pointcut. If the pointcut `move` is defined as a disjunction of a change in the variable `x` and a change in the variable `y`, then there would be three movements detected. Approaches such as EAOP [DFS04] make it possible to define a sequence of a change in `x` and a change in `y` as a single movement event. However, the second movement consists in a single variable change. Setting the pointcut as the disjunction of a sequence of two variable changes and a single variable change makes it impossible to determine whether a variable change is part of a sequence or a single movement. In some cases, it could still be possible to define a dedicated pointcut that considers all the possible combinations for a given method. However, the pointcut would become strongly dependent on the implementation structure of the method. Any change in the code could break down such a pointcut. Triggering `move` as an imperative event at the proper places would facilitate the task and in some case provide a more stable design.

**Event handling is better aligned with OOP.** In a typical OO design, each piece of functionality is defined from the perspective of a certain object using the methods, attributes and relationships of the object. An object does not have a complete knowledge of the world, only of its own state and the interfaces of directly referenced objects. In a proper

```
class SomeFigure extends Figure {
  void myTransform() {
    ...
    /* first movement */
    this.positionX = 10;
    this.positionY = 50;
    ...
    /* second movement */
    this.positionY = 10;
    ...
  }
}
```

Listing 3.4: Need of coarse-grained join points.

OO design all functionality is defined from a local perspective, while global functionality is avoided as much as possible. Approaches such as C# make it possible to express interest in events of specific objects by means of explicit registration using operators such as `+=` as Listing 2.13 shows. A view is interested in the events of the model that it observes rather than all instances of the model class. This is not the case with AOP. AO design is focused on separating crosscutting concerns often involving different parts of an application. Therefore, AO design tends to view the application as a whole and modularize its different concerns into aspects that cut across the object-based design. Consequently, the pointcut language of AspectJ is designed to quantify over the static structure of an application: the members of a class, the classes of the application, inheritance relationships between classes. Although the pointcut language supports dynamic conditions in pointcuts, these conditions are executed in static context and can use only static variables and methods. For example, in Listing 2.25 it is not possible to define the event `change` in terms of the relevant figures. The aspect observes the changes associated to all the figures of the editor. As described in Section 2.2.4.5, the implicit instantiation strategies of AspectJ such as `perthis` cannot be used either.

## 3.3 AOP and OOP

AOP features an *asymmetric* composition of concerns, distinguishing classes, implementing *base* concerns, from *aspects*, implementing crosscutting concerns. In its most successful incarnation, AspectJ, aspects look very much like classes but contain two new member types: pointcuts and advice. A pointcut is a predicate selecting the execution points of interest, also called *join points*. A piece of *advice* specifies which action should be taken when a join point is selected by its associated pointcut. Although an aspect looks very much like a class, the rules governing its static and dynamic semantics are different. In particular, it can be extended as a class but in a constrained way and it cannot be explicitly instantiated. This adds quite a lot of complexity to standard OOP, conflicting with the principles of simplicity, regularity and orthogonality of MacLennan [Mac95]. On the one hand, there are a number of new constructs. On the other hand, the basic principles for working with these constructs (instantiation, inheritance) are not regular. The corresponding design choices can be partly explained by the will to better capture the intension of the programmer and avoid the issues encountered when using reflection and metaobject

protocols [KdRB91], a more general and integrated approach, which unfortunately makes it much too easy to break base code. They can be also partly explained by performance consideration.

### 3.3.1 Orthogonality and Regularity of Aspects and Classes

Kiczales has made an effort to make AspectJ aspects different from classes in order to provide dedicated support for crosscutting concerns. The four main characteristics that make AspectJ aspects different from plain classes are: implicit instantiation, limited inheritance, specific members (pointcuts, advice), and inter-type declarations. These differences, however, do not make aspects orthogonal to classes. Indeed an aspect is very similar to a class. For example aspect inheritance is just a restricted form of class inheritance and aspect instantiation creates standard objects. This situation conflicts with the principles of orthogonality and simplicity of MacLennan [Mac95]. Understanding aspects requires an additional effort compared to understanding classes, whereas they both implement similar mechanisms. Finally, the design of AspectJ aspects to resemble classes is due to an effort for providing a seamless extension to Java. The design of pointcut and advice is however not regular regarding the members of classes. For example, pointcuts are "almost static" and pieces of advice cannot be overridden.

The main reason for this design is that traditional AO design focuses on separation of crosscutting concerns often involving different parts of an application. Therefore, AO design tends to view the application as a whole and modularize its different concerns into aspects that cut across the object-based design.

**Implicit Instantiation.** In AspectJ, an aspect is implicitely instantiated. By default, it is a singleton, instantiated at load time. It can still be associated a null constructor. Additional declarations, like `perthis(`*pointcut*`)` and `percflow(`*pointcut*`)`, make it possible to implicitly create aspect instances and associate them to various entities of the base program encountered in the selected join points, in the case of `perthis` and `percflow`, to each `this` object and each control flow, respectively.

It is often said that, because of implicit instantiation, aspects are not first class. However, aspect instances can still be returned by different variants of the static method `aspectOf`, depending on whether the aspect is a singleton or not, and then handled as any plain class instance.

**Aspect extension.** Aspects can extend aspects in the same way as classes can extend classes except that aspects may only extend abstract aspects. Of course classes cannot extend aspects.*

**Pointcuts and advice.** Behavioral aspects include two new members: (named) pointcuts and advice. A pointcut is almost a full-fledged static member of both classes and aspects. It can be abstract, can have its visibility defined through the usual modifiers, can be redefined, but it cannot be overloaded. A piece of advice can only be present in an aspect. It binds a pointcut and the action, the *advice body* to be performed when the pointcut selects a join point, and defines the location of the action, before, after, or around

*. In the early days of AspectJ, there was a period of time when a class could extend an aspect, but only plain Java members would be inherited.

the join point. Like methods it has access to instance state and can be inherited by subaspects, however, unlike methods, it is anonymous and therefore cannot be redefined. This irregularity is an issue when implementing *higher-order concerns* [RS05], *i.e.*, aspects that advise aspects. Indeed, the lack of name results in the impossibility of selecting a specific piece of advice. This is also an issue when different pieces of advice of the same aspect must have different precedence rules [MW08].

In ASPECTJ, the composition of an aspect is very similar to the composition of a class but aspects differ essentially in that they include pieces of advice. Aspects are instantiated implicitly whereas classes are instantiated explicitly. Rules governing aspect instantiation are quite constrained. Moreover, the new members introduced by aspects are not pure instance members. Pointcuts are static and cannot be overloaded. Pieces of advice are anonymous and therefore cannot be redefined.

### 3.3.2 Incomplete Integration Efforts

The need of distinguishing between classes and aspects, with special instantiation and inheritance rules for aspects, is arguable. The orthogonality, simplicity and regularity principles suggest to unify classes and aspects by providing a single construct, let us call it a *class*, comprising, on top of the usual members, both pointcuts and pieces of advice. Using a single construct should not be a syntactic trick associated to program analyses that would distinguish *aspect classes* from *plain classes*. Identical instantiation and inheritance rules should apply. This is not a new approach. It has already been followed by a number of frameworks and languages, including ASPECTWERKZ [BV04], Spring [JHD+10], JBoss AOP [JBo10], CLASSPECTS [RS05] and CAESARJ [AGMO06]. Section 2.2.4 described CLASSPECTS and CAESARJ as the most representative proposals. As described there, both merge aspects and classes in a single class construct, providing explicit aspect instantiation and uniform class and aspect extension rules. Explicit instantiation covers the implementation of *behavioral relationships*, which is not properly covered with the implicit instantiation scheme, as provided by ASPECTJ. With respect to pointcuts and advice, CAESARJ follows ASPECTJ. CLASSPECTS diverts from it when dealing with advice. It replaces the advice body by a method call making it possible to implement *higher-order concerns* [RS05].

However, the integration is still not complete. Integration of AOP and OOP has mainly two faces: aspects as classes and full-fledged *aspectual class members*. Approaches such as CAESARJ and CLASSPECTS do well with treating aspects as plain classes. The treatment of pointcuts and advice as full-fledged class members is, however, not completely achieved. Pointcuts in CAESARJ are static. In CLASSPECTS the join-point selection of a pointcut is instance-specific. However, as Section 2.2.4.4 showed, pointcuts are embedded in a join-point-method binding construct. Consequently, they are not real class members. For example, they cannot be inherited nor reused. Regarding pieces of advice, they are in general not named as standard class members are, limiting their use. In addition, join point selection is still based on the static structure of programs rather than implementing an OO collaboration between classes.

## 3.4 Conclusion

This chapter has shown that the OOP, the EBP and the AOP paradigms are not well integrated together, and that this lack of integration conflicts with the programming-language

principles proposed by MacLennan [Mac95]. This integration problem is the motivation of this dissertation. The next chapters present an approach that provides a better integration of these paradigms. The cornerstone of the integration is an EB programming model with declarative events, smoothly extended to support AOP and naturally well integrated with OOP. The next chapter introduces this programming model showing the needs for declarative means for expressing events.

# Part II

# Contribution

# Chapter 4

# A Model of Declarative and Polymorphic Events

The previous chapter described the problem statement of this dissertation: the lack of integration of OOP, EBP and AOP. This chapter presents the main elements of the programming model that we propose to solve this problem. This chapter describes the EB side of this model. We introduce a novel notion of *declarative event*. This notion improves existing EB models such as the one of C#. Section 4.1 motivates the contribution of this chapter. Section 4.2 reviews the imperative composition of events in languages such as C#. Section 4.3 introduces the programming model with a special focus on declarative events. Section 4.4 presents EJava, a programming language that materializes the programming model. Section 4.5 discusses related work. Finally, Section 4.6 concludes.

## 4.1 Motivation

Events have been introduced as language constructs in mainstream OO languages such as C# as a way to facilitate the implementation of reactive applications. However, the design of events in these languages has failed in providing appropriate means for expressing events as composition of other events. Composite events are often needed in large applications. In layered applications, for example, upper layers can define events in terms of events provided in lower layers. It is natural for programmers to think of composite events as logical combinations of events. A declarative expression such as `someoneEntered = aliceEntered || bobEntered` describes well the intention of a programmer: the event indicating that someone entered a place is either the event that Alice entered the place or the event that Bob entered the place. Unfortunately, in C#-like languages there is a gap between the way composite events are written and the intention of programmers. Composite events are expressed as imperative events and cannot be defined using a declarative expression.

The contribution of this chapter is twofold. First, it introduces the programming model that further chapters use for integrating the different programming paradigms that this dissertation takes into account. Second, it describes the EB side of this model, including, besides imperative events, *declarative events*. This kind of events are defined as composition of other events in a declarative way, better expressing the intention of programmers.

The next section shows how events are imperatively composed in languages such as C#.

## 4.2 Imperative Composition of Events

The events of an application can be related to each other. For example, an event indicating that night is falling can be related to an event indicating that it is getting dark or to an event indicating that it is getting late. Analogously, the getting-dark event can be related to an event indicating a severe change in the intensity of light when it is still daytime. In languages such as C#, the relation between events is expressed imperatively. The occurrence of an event is triggered as a result of the occurrence of another. Unfortunately, such an imperative composition does not express the intention of a programmer in a direct way.

```
1  delegate void ChangedHandler(Figure source);
2
3  abstract class Figure { ...
4    event ChangedHandler changed(Figure source);
5    abstract void onChanged(Figure source) {
6      changed(source);
7    }
8  }
9  abstract class AbstractFigure extends Figure { ...
10   void moveBy(int x, int y) { ...
11     onChanged(this);
12   }
13 }
14 class Circle extends AbstractFigure { ... }
```

Listing 4.1: Event definition and event triggering in C#.

As an example, Listing 4.1 illustrates the imperative composition of events in the implementation of a simple figure editor in C#. The abstract class `Figure` is on top of a basic hierarchy of figures. The event `changed` represents a change in a figure (line 4), observed by the views of an editor in order to update themselves when the figure changes. The event `changed` declares a parameter of type `Figure` representing the figure that has to be repainted. This is useful for composite figures, which can indicate the child that is the source of the change, avoiding to repaint the complete composite figure. For other figures, the parameter is most of the times the figure defining the event. The class `Figure`, apart from defining the event `changed`, defines the method `onChanged` for triggering the event in subclasses (line 5). The class `AbstractFigure` (lines 9-13) is the superclass of figures such as the class `Circle`. It triggers the event `changed` at the end of methods such as `moveBy` (line 11), which change the state of the figure. It passes the figure itself as a parameter of the event.

A composite figure serves for grouping figures. It changes when any of its children change, so that its event `changed` depends on the event `changed` of its children. Listing 4.2 shows how the occurrence of an event is related to the occurrence of other events in C#. The composite event is still imperative. A dedicated handler has to be included that is registered with the composed events. The handler is in charge of imperatively triggering the composite event when the composed events happen.

A drawing is a composite figure. The view of a drawing defines an event `invalidated` meaning that the view does not correctly reflect the state of the drawing. When this event happens the figure that changed has to be repainted. The occurrence of the event is related

```
class CompositeFigure extends Figure { ...
  Figure fig1, fig2;

  CompositeFigure() {
    fig1.changed += changedHandler;
    f1g2.changed += changedHandler;
  }
  void changedHandler(Figure source) {
    onChanged(source);
  }
}
class Drawing extends CompositeFigure { ... }
```

Listing 4.2: An event defined in terms of other events in C#.

```
1  class DrawingView { ...
2    event ChangeHandler invalidated(Figure changedFig);
3    Drawing drawing;
4    boolean isVisible;
5
6    DrawingView() {
7      drawing.changed += changedHandler;
8      invalidated += invalidatedHandler;
9    }
10   void changedHandler(Figure changedFig) {
11     if(isVisible)
12        invalidated(changedFig);
13   }
14   void invalidatedHandler(Figure changedFig) {
15     repaint(changedFig.getDrawingArea());
16   }
17 }
```

Listing 4.3: An event defined in terms of another event and a condition in C#.

to the occurrence of the event `changed` of its drawing when the view is visible. Listing 4.3 shows the implementation of the drawing view and how an event can be related to the refinement of an event with a condition in C#. Note how the condition is tested in the handler before triggering the composite event (line 11).

The way events are related in these kinds of languages presents the problem that the relationships between the events are not explicit.

By looking at the previous examples, it is possible to observe that the way two events are related in C# follows a common pattern. First, the first event is defined as an imperative event of the class (like the event `invalidated` in line 2 of Listing 4.3). Second, at some place in the class (for example in the constructor) a handler is registered with the second event (like with the event `changed` of the drawing in line 8 of Listing 4.3). Third, the registered handler is included as a member of the class. Its body triggers the first event depending on an optional condition (like the condition `isVisible` in line 11 of Listing 4.3). An event can be related to two or more events by repeating the same steps as for the event `changed` in Listing 4.2. Even if the relationships between events follow a certain pattern, the pattern is frequently embedded in other code complicating its recognition. The different relationships between events need to be determined by analyzing the code of the application. This analysis can be a complicated task. The lack of explicitness may result in a misunderstanding of applications, complicating their maintainability and extensibility.

A solution to this problem is the inclusion of declarative means to relate events. Declarative events would be explicit and compact, expressing in an effective way the intention of programmers. In addition, they would enable optimizations. The next section presents our EB model supporting declarative events.

## 4.3 A Programming Model with Declarative Events

This section introduces an EB programming model that includes *declarative events* besides imperative events. Declarative events can be defined in terms of other events in a declarative way, thus making explicit the relationships between the events of an application. The understanding, the maintenance and the evolution of the application get improved.

In order to illustrate the design decisions of our programming model, this section considers a simple, typed, and class-based OO language like MiniMAO$_0$ [CL05]. This section describes the support for events of our programming model on top of this language. In order to be as general as possible, the different elements of our model are illustrated in abstract syntax.

Listing 4.4 shows the abstract syntax of a MiniMAO-like language. The terminals $t$, $f$, $m$ and $v$ range on class names, field names, method names, and variable names, respectively. A class consists of a name, the name of its superclass and zero or more members. Class members are fields or methods. A field consists of a type and a name. A method consists of a return type, a name, zero or more formal parameters (*FormalParam*), and zero or more expressions (*Expr*). A method returns the value that results from the evaluation of the last expression.* An expression can be an expression *New* used to instantiate a class, a method call *Call*, a field access *FieldAccess*, a field assignment *FieldAssign*, a type casting *Cast*, the object *This*, the object *Null* or a variable $v$ declared in the signature of a method.

∗. This makes it also possible to model languages with statements. The list of expressions can be seen as equivalent to a list of statements with the last statement being a return.

```
Class       ::= t  t Member*
Member      ::= Field | Method

Field       ::= t  f

Method      ::= t m FormalParam* Expr*
FormalParam ::= t  v
Expr        ::= New | Call | FieldAccess | FieldAssign | Cast | This | Null | v

New         ::= t
Call        ::= Expr m Expr*
FieldAccess ::= Expr f
FieldAssign ::= FieldAccess Expr
Cast        ::= t Expr
```

Listing 4.4: Abstract syntax of a simple typed class-based OO language.

### 4.3.1 Events and Event Handlers as Properties of Objects

We equip the language with new class members: events and event handlers (see Listing 4.5). The terminal $e$ ranges on event names. An event can be declared as abstract or defined as either imperative or declarative (non-terminals *AbsEvent*, *ImpEvent* and *DecEvent*, respectively). Both abstract events and imperative events consist of a name and zero or more formal parameters. An abstract event defers its definition to subclasses. An imperative event is used to explicitly trigger an event occurrence similarly to the events of C#. Declarative events consist of a name, zero or more formal parameters and an event expression (*EventExpr*). A declarative event can combine other events, by means of its event expression, similarly to the declarative events of the language $e$ [IEE08]. A declarative event is a composite event (unless it simply aliases a primitive event), whereas an imperative event is a primitive one.

An event expression can be an access to an event defined elsewhere (*EventAccess*), a disjunction of two event expressions (*Disjunction*), a conjunction of two event expressions (*Conjunction*), the refinement of an event expression with a condition (*Refinement*), an event expression with an explicit parameter binding (*Binding*), an event expression with local formal variables (*LocalVar*) and a quantification on the events of a list of objects (*Quantification*). These expressions are detailed in Section 4.3.3.

An event handler (non-terminal *Handler*) is defined by indicating the signature of an event (*i.e.* its name and its list of formal parameters) and zero or more expressions implementing the body of the event handler. These expressions are either the standard expressions of the body of a method† or the expressions *Trigger* or *Next*, used to trigger an event and to compose event handlers, respectively (*Trigger* and *Next* are detailed in Sections 4.3.2 and 4.3.4, respectively). The event handler is defined as bound to the corresponding event, so that it is executed when the event occurs. In order to provide event handlers with an identity, an event handler can only be bound to an event defined in the same class or in a superclass.‡ Thus, the identity of an event handler corresponds to the

†. In Java a method can have a return statement. A priori an event handler does not return so the return statement it not part of an event handler.

‡. In order to handle an external event, the class has to define a (local) declarative event in terms of the external event.

```
Member         ::= Field | Method | AbsEvent | ImpEvent | DecEvent | Handler

AbsEvent       ::= e FormalParam*
ImpEvent       ::= e FormalParam*
DecEvent       ::= e FormalParam* EventExpr

EventExpr      ::= EventAccess | Disjunction | Conjunction |
                   Refinement  | Binding     | LocalVar     | Quantification

EventAccess    ::= Expr e Expr*
Disjunction    ::= EventExpr EventExpr
Conjunction    ::= EventExpr EventExpr
Refinement     ::= EventExpr Condition
Condition      ::= Expr
Binding        ::= EventExpr Bind
Bind           ::= v Expr
LocalVar       ::= FormalParam* EventExpr
Quantification ::= Expr e Expr*

Handler        ::= e FormalParam* HandExpr*
HandExpr       ::= Expr | Trigger | Next
Trigger        ::= Expr e Expr*
```

Listing 4.5: Extension of the syntax of Listing 4.4 with the elements of our model.

signature of the corresponding event. Note that in Listing 4.5 the name of an event handler is the name of an event. As a consequence, a class can have pairs (event, handler) sharing the same signature. There is no ambiguity since events and event handlers are used in different contexts.

```
1 Class(Automator, Object, [
2
3   Field(LightSensor, sensor)
4
5   DecEvent(nighttime ,[] ,
6       EventAccess(FieldAccess(sensor), gettingDark, [])) ,
7
8   Handler(nighttime, [] , close-shutters)
9 ])
```

Listing 4.6: Simple example of a declarative event and an event handler.

In order to give a first feeling of the model, consider Listing 4.6. It illustrates, by using the abstract syntax of the language, a class equipped with a field, a declarative event and an event handler attached to the event. Listing 4.6 and the rest of the listings of this section use the ATerm syntax formalism [VdBdJKO00], which makes it possible to represent an abstract syntax tree. The name of a non-terminal is written in italics and uppercase, and its constituents are put in a tuple. Terminals are typewritten and lists of elements are put between square brackets. Variables representing folded syntax trees are put in italics and lowercase.

The class `Automator` of Listing 4.6 defines a declarative event `nighttime` in line 5, indicating that night is falling, and an event handler `nighttime` in line 8, handling the event `nighttime`. The event `nighttime` is defined as an access to the event `gettingDark` of the field `sensor`. This field, defined in line 3, represents a light sensor. The event describes in a declarative way that night is falling as soon as the light sensor detects that it is getting dark (this is, of course, an approximation). The event is used by the class to close the shutters of a house by using the event handler `nighttime`. The folded implementation of the event handler is represented by the variable *close-shutters*

Events are properties of objects. The external access to an event has to be qualified with an expression returning the object the event belongs to. This means that events are late-bound since their concrete value depends on the evaluation of the corresponding expression at runtime. Internal access to an event is qualified with *This*. Events are accesses in the definition of declarative events (see non-terminal *EventAccess*) and when triggering imperative events (see non-terminal *Trigger*). In our model any object can trigger an imperative event of another object. This is compatible with the design of EB applications such as JHotDraw [Eri11], in which, as an answer to a command, the current tool applies several transformations to a figure and then triggers the imperative event `changed` of the figure to indicate that the figure has changed.

Event handlers are also properties of objects. However, an event handler is only accessed explicitly in the class hierarchy for inheritance purposes.

The fact that both events and handlers are object properties also implies that they can use instance-level context in their definition such as references to fields or methods of the object they belong to. In addition, events and event handlers are subject to inheritance like fields and methods. Inheritance is described in Chapter 5.

### 4.3.2 Imperative Events and Runtime Events

As previously said, our model enables external triggering: any object can trigger an imperative event of another object. *Occurrences* of imperative events are *triggered* to point out changes in the state of an application. We use the term *event occurrence* for a runtime event, whereas we use the term *event* for the corresponding language construct. When we say that an imperative event is triggered, we actually mean that an occurrence of the imperative event is triggered. An imperative event is triggered at runtime by providing each declared parameter with a value. The execution of a program produces a sequence of runtime events indicating several state changes.

As an example, Listing 4.7 shows how the C# implementation of Listing 4.1 can be implemented using the abstract syntax of the language. The event `changed` of a figure is declared abstract in line 2. Indeed, in our model abstract events make sense because subclasses may provide either an imperative definition or a declarative definition of the event. The class `AbstractFigure` defines the event as imperative in line 6 and uses it to trigger an occurrence inside the method `moveBy` in line 10. It provides the object *This* as a parameter.

### 4.3.3 Declarative Events

A declarative event is an *event selector*, a function that selects the runtime events that notify the state change that the event represents. An imperative event is also an event selector, but it selects its own occurrences. We refer to the occurrences of an event for referring to the runtime events that the event selects. We say that an event occurs when

```
1 Class(Figure, Object, [
2   AbsEvent(changed, [FormalParam(Figure, source)])
3 ])
4
5 Class(AbstractFigure, Figure, [
6   ImpEvent(changed, [FormalParam(Figure, source)]),
7
8   Method(void, moveBy, fparams, [
9     implementation,
10    Trigger(changed, [This])
11  ])
12 ])
```

Listing 4.7: Definition of an imperative event and example of triggering in the model.

the event has successfully selected a runtime event. For example, if a figure defines an imperative event `changed`, triggered in proper places for indicating a change in the figure state, the declarative event `changed` of a composite figure can be defined such that it selects the occurrences of the imperative event `changed` of its children. Since these occurrences indicate that some child figure has changed, they also indicate in a proper way that the composite figure has changed.

A declarative event is defined in terms of other events by using an event expression. An event expression is also an event selector, so that a declarative event selects the runtime events that its event expression selects. The different event expressions are described as follows:

**Event access.** An event access (non-terminal *EventAccess*) makes it possible to access an event of an object. It is a *primitive* event expression that selects the runtime events that the accessed event selects. For example, the event `nighttime` of Listing 4.6 is defined as an access to the event `gettingdark` of a sensor. The event `nighttime` selects the same runtime events that the event `gettingdark`, so that `nighttime` occurs when `gettingdark` occurs.

**Disjunction.** An event expression may consist of the disjunction of two event expressions (non-terminal *Disjunction*). It selects the union of the runtime events that the two event expressions select.

```
1 Class(CompositeFigure, Figure, [
2   Field(Figure, fig1),
3   Field(Figure, fig2),
4
5   DecEvent(changed, [FormalParam(Figure, source)],
6       Disjunction(
7           EventAccess(FieldAccess(fig1), changed, [source]),
8           EventAccess(FieldAccess(fig2), changed, [source])))
9 ])
```

Listing 4.8: An event defined in terms of other two events in the model.

As an example, Listing 4.8 shows how the C# code of Listing 4.2 can be expressed in our model. The event `changed` of a composite figure is defined as a disjunction of the events `changed` of its children. Thus, whenever the event `changed` of a child occurs, the event `changed` of the composite figure also occurs.

**Refinement.** An event expression can represent the refinement of another event expression with a condition (non-terminal *Refinement*). It selects the runtime events selected by the given event expression occurring when the condition holds.

```
1  Class(DrawingView, Object, [
2    Field(Figure, drawing),
3    Field(Boolean, isVisible),
4
5    DecEvent(invalidated, [FormalParam(Figure, changedFig)],
6        Refinement(
7            EventAccess(FieldAccess(drawing), changed, [changedFig]),
8            Condition(FieldAccess(isVisible)))),
9
10   Handler(invalidated, [FormalParam(Figure, changedFig)], [
11     implementation,
12     Next
13   ])
14 ])
```

Listing 4.9: An event defined in terms of another event and a condition in the model.

As an example, Listing 4.9 shows how the C# code of Listing 4.3 can be expressed in our model. The event `invalidated` of a drawing view is defined in line 5 as a refinement of the event `changed` of a drawing with the condition `isVisible`. Thus, whenever the event `changed` of a drawing occurs, the event `invalidated` also occurs when the view is visible.

The examples so far show how declarative events make the programming intention more explicit than the same examples implemented in C#. This explicitness is even more evident in Section 4.4, which shows these examples in concrete syntax.

**Parameters.** As previously seen, the signature of an event can declare parameters. An imperative event is triggered at runtime by providing each declared parameter with a value. These values can be captured and bound to the parameters of declarative events, concretely in event accesses, and propagated to other declarative events. An event handler attached to an event can use the bound parameters to perform its actions. For example, line 10 of Listing 4.7 triggers the event `changed` of a figure by providing the object *This* as parameter. This object is captured and bound to a parameter `source` when accessing the events of the children of a composite figure in the declarative event `changed` of line 5 of Listing 4.8. The propagation of the parameter continues until reaching the event handler `invalidated` of Listing 4.9, which uses the bound parameter to repaint the area of the corresponding figure.

An event binds values for its parameters when it selects a runtime event. Parameters can play a role in the process of selection as they can be used in conditions of the refinement of an event expression (non-terminal *Refinement* of Listing 4.5). The parameters of a declarative event are implicitly bound when accessing other events. In addition, the event

expression *Binding* of Listing 4.5 makes it possible to explicitly bind a parameter with a custom value, *e.g.* with the result of calling a function.

In general, the conditions of an event expression can use the parameters bound by declarative events and any field or method of the contextual object. In addition, our model makes it possible to declare local event variables (non-terminal *LocalVar* of Listing 4.5). These variables are local to an event expression and can be bound and used in conditions inside the event expression. They are not visible outside.

```
1 DecEvent(newRedCars, [FormalParam(List, redCars)],
2     Refinement(
3         LocalVar(
4             [FormalParam(List, cars)],
5             Binding(
6                 EventAccess(This, newCars, [cars])),
7             Bind(redCars, Call(This, filterRedCars, [cars])))),
8         Condition(Call(redCars, isNotEmpty, []))))
```

Listing 4.10: Example of explicit binding and local event variables.

Listing 4.10 shows an example of an explicit binding and local event variables. It defines a declarative event `newRedCars` in terms of an imperative event `newCars`. The event `newCars` represents the availability of a list of new cars and the event `newRedCars` the availability of a list of new red cars. The event `newRedCars` is defined by accessing `newCars` in line 6. The list of cars provided by `newCars` is bound to a local variable `cars` defined in line 4. An explicit binding is used in line 7 to bind the variable `redCars` with the result of filtering the red cars from `cars`. The event `newRedCars` occurs when the event `newCars` occurs and the list of red cars is not empty, which is tested in line 8.

**Conjunction.** In our model two or more events can happen simultaneously as they can select the same runtime event. Thus, it makes sense to provide, apart from disjunction, conjunction of events. The conjunction of two event expressions (non-terminal *Conjunction*) selects the runtime events that the two expressions select.

```
1 DecEvent(newHotCars, [FormalParam(List, redCars), FormalParam(List, cheapCars)],
2     Conjunction(
3         EventAccess(This, newRedCars, [redCars])),
4         EventAccess(This, newCheapCars, [cheapCars]))
5
6 DecEvent(newHotCars, [FormalParam(List, cars)],
7     LocalVar(
8         [FormalParam(List, redCars), FormalParam(List, cheapCars)],
9         Binding(
10             EventAccess(This, newHotCars, [redCars, cheapCars]),
11             Bind(cars, Call(This, intersect, [redCars, cheapCars])))))
```

Listing 4.11: Example of conjunction of events.

As an example, Listing 4.11 defines a declarative event `newHotCars`, in line 1, as a conjunction of two events `newRedCars` and `newCheapCars`. The event `newRedCars` is the one presented earlier, representing the availability of a list of new red cars. The event

`newCheapCars` represents the availability of a list of new cheap cars and it is defined in terms of an imperative event `newCars` in a analogous way as `newRedCars`. When the event `newCars` happens, both `newRedCars` and `newCheapCars` can happen simultaneously, each one filtering the proper list of cars. The event `newHotCars`, defined as a conjunction of these events, represents the availability of new cars where some are red and the other are cheap. The event defines two parameters that capture each kind of cars. The event `newHotCars` of line 6 uses the event `newHotCars` of line 1 to provide a single list of cars, which are both red and cheap.

In the previous example, the event `newHotCars` could have been defined directly in terms of the event `newCars` without requiring a conjunction of other events. However, we illustrate here how we can reuse existing events.

**Quantification.** So far we have considered objects related by one-to-one relationships, but an object may also be related to a collection of other objects and need to globally observe occurrences of a specific event defined by all these objects. Our model enables a form of existential quantification on a list of objects (non-terminal *Quantification* of Listing 4.5). An event can be defined as a disjunction of computed event expressions all related to the same event name by using an expression of the form *Quantification(list, eventName, params)*, where *list* is an expression evaluating to a list of objects, *eventName* an event defined by these objects and *params* a list of formal parameters.

*DecEvent*(`changed`, [*FormalParam*(`Figure`, `source`)],
*Quantification*(*FieldAccess*(*This*, `figures`), `changed`, [`source`]))

Listing 4.12: Example of quantification on the events of a list of objects.

For example, Listing 4.12 shows how a change of a figure, which is a composition of a list of figures (variable `figures`), can be described in terms of the events observed on its children. It defines that the event `changed` of the composite figure occurs when the event `changed` of some composed figure occurs.

### 4.3.4 Informal Operational Semantics

An event evaluates to a selector of runtime events. On a runtime event all the selectors of the objects of the application are used to select the runtime event. The event handlers bound to the ones that select the runtime event are then executed.§.

An event selector is a function that receives a runtime event as a parameter and returns a tuple with values for its declared parameters when it selects the occurrence or $\epsilon$, otherwise. Let us show what these selectors look like in Scheme.

We consider a runtime of the the form `(list` *obj e tuple*`)`, where *obj* is an object, *e* is an event name and *tuple* is a tuple of values. An imperative event evaluates to a selector that select its own occurrences. It can be defined in Scheme by the function `imp-event` shown below, where `this` is a function that returns the object that owns the imperative event:

The definition of a declarative event evaluates to a selector that depends on its event expression. An event expression also evaluates to a selector of runtime events. However, this

§. This is optimized in the concrete implementation.

```
(define (imp-event event-name)
  (lambda (rt-event)
    (if (and (eqv? (car rt-event) (this)) (eqv? (cadr rt-event) event-name))
        (cddr rt-event)
        'epsilon)))
```

selector receives as parameters not only a runtime event but also an environment with pairs name-value representing variable bindings, where the variables are either the parameters declared by the event or local event variables. These bindings can be used in conditions. The selector returns an environment with the new parameter bindings computed inside the expression. The selector of a declarative event calls the selector of its event expression and from the obtained environment calculates the corresponding tuple. The definition of a declarative event looks as follows in Scheme, where `var-names` are the list of parameters of the event and `s` is the selector that resulted from evaluating the event expression:

```
(define (dec-event var-names s)
  (lambda (rt-event)
    (let ((env (s rt-event '())))
      (if (eqv? env 'epsilon)
          'epsilon
          (map (lambda (x) (cadr (assq x env))) var-names)))))
```

The returned selector calls `s` passing the runtime event and an empty environment as parameters to obtain a new environment with variable bindings. It creates a proper tuple of parameters values from the obtained environment and the list of parameters of the event. The selector `s` evaluates to one of the functions described in the remainder:

- An event access of the form *expr.eventName(varName1,...,varNameN)* evaluates to a selector which, when applied to a runtime event, applies to it the selector of the event *eventName* of the object that result from evaluating the expression *expr*. We obtain as a result a tuple of parameter values `'(`*value1 ... valueN*`)`. By using such a tuple the selector creates and returns an environment of the form `'((`*varName1 value1*`) ... (`*varNameN valueN*`))`. The definition of an event access looks as follows in Scheme, where `delayed-expr` is a function that evaluates to an object, `event-name` is the name of the accessed event and `var-names` the list of parameters of the event:

  ```
  (define (event-access delayed-expr event-name var-names)
    (lambda (rt-event env)
      (letrec ((s (get-member event-name (delayed-expr)))
               (tuple (s rt-event)))
        (if (eqv? 'epsilon tuple) 'epsilon (map list var-names tuple)))))
  ```

  The function `delayed-expr` is a function that "wraps" the expression *expr* that returns the object that belongs the accessed event. It is of the form `(lambda()`*expr*`)`, delaying the evaluation of *expr*. This highlights the late-bound nature of our events. The expression is evaluated in the process of selection of a runtime event.
- The definition of the disjunction of two event expressions looks as follows in Scheme, where `s1` and `s2` are the selectors that result from evaluating the composed event expressions:

```
(define (disjunction s1 s2)
  (lambda (rt-event env)
    (let ((env1 (s1 rt-event env)))
      (if (eqv? 'epsilon env1) (s2 rt-event env) env1))))
```

The selector is such that when applied it returns the result of applying `s1` when the obtained environment is not $\epsilon$, or the result of applying `s2`, otherwise.

– The definition of the refinement of an event expression with a condition looks as follows in Scheme, where `s` is the selector that results from evaluating the refined event expression, and `condition` represents the condition:

```
(define (refinement s condition)
  (lambda (rt-event env)
    (let ((new-env (s rt-event env)))
      (if (eqv? 'epsilon new-env)
          'epsilon
          (if (eval condition (union new-env env)) new-env 'epsilon)))))
```

The selector is such that when applied, it applies `s` to obtain an environment with new parameter bindings. With the union of the obtained environment and the original environment `env` it evaluates the condition. The environment with the new parameter bindings is returned if the condition holds. We use a function `eval` that evaluates an expression considering a given environment. The function `union` returns the union of two lists.

– The definition of the conjunction of two event expressions looks as follows in Scheme, where `s1` and `s2` are the selectors that result from evaluating the composed event expressions:

```
(define (conjunction s1 s2)
  (lambda (rt-event env)
    (let ((new-env1  (s1 rt-event env)))
      (if (eqv? 'epsilon new-env1)
          'epsilon
          (let ((new-env2  (s2 rt-event (union new-env1 env))))
            (if (eqv? 'epsilon new-env2)
                'epsilon
                (union new-env1 new-env2)))))))
```

The selector applies `s1` with the original environment and then `s2` with the union of the original environment and the one obtained from `s1`. Thus, the second expression can use the variables bound by the first event expression. Finally, the returned environment is formed by the union of the two newly obtained environments.

– The definition of an explicit binding looks as follows in Scheme, where `s` is the selector that results from evaluating the corresponding expression, `var-name` is the name of a variable and `var-value` the value to bind.

```
(define (binding s var-name var-value)
  (lambda (rt-event env)
    (let ((new-env (s rt-event (union env (list var-name var-value)))))
      (if (eqv? 'epsilon new-env)
          'epsilon
          (union new-env (list var-name var-value))))))
```

Note that the new binding is used in the application of `s`. Then, the selector returns the environment obtained from `s` extended with the new parameter binding.

– The definition of an expression with local event variables looks as follows in Scheme, where `s` is the selector that results from evaluating the given expression and `var-names` the list of local variables:

```
(define (local-var s var-names)
  (lambda (rt-event env)
    (let ((new-env (s rt-event env)))
      (if (eqv? 'epsilon new-env) 'epsilon (rm-vars new-env var-names)))))
```

The function returns the environment obtained from applying `s`. However, it removes the bindings corresponding to local variables by using the function `rm-local`. Thus they cannot be used by enclosing selectors as they are local.

– The definition of an quantification on a list of objects looks as follows in Scheme, where `delayed-expr` evaluates to a list of objects, `event-name` is the name of event to access and `var-names` is the list parameters of the event:

```
(define (some delayed-expr event-name var-names)
  (lambda (rt-event env)
    (fold (lambda (tail obj)
            (if (eqv? tail 'epsilon)
                ((event-access (lambda() obj)
                               event-name
                               var-names) rt-event env)
                tail))
          (delayed-expr))))
```

Similarly to simple event access, the function `delayed-expr` delays to selection time the evaluation of the expression that returns the required list of objects.

**Composition of event handlers.** The handling of a runtime event consists of executing a list of event handlers. These handlers are the ones bound to the events that selected the runtime event. Each one is executed with values for its parameters taken from the tuple of parameter values that resulted from the application of the corresponding event selector.

The execution of event handlers is *nested* by using the event expression *Next*. The execution is such that the first handler of the list can use *Next* to nest the execution of the second handler. The second handler can use *Next* to nest the execution of the third handler, and so on. When the last handler executes *Next*, the control comes back just after the triggering of the event. Expressed in other terms, event handlers can control the rest of the process of handling. If an event handler does not execute *Next*, the remaining handlers are not executed.

In order to illustrate the nesting of event handlers, consider Listing 4.13. It defines an inspector of figures with a handler for the event `changed` of a figure. Thus, two handlers can be registered for the occurrences of the event `changed` of the same figure: the one of a drawing and the one of a figure inspector. For example, consider that at runtime there is a view (`view`) of a drawing (`draw`), which consists of a single figure (`fig`). In addition, there is a figure inspector (`inspector`) associated to the figure. Figure 4.1 shows what happens when the event `changed` of the figure is triggered. The new runtime event is detected by the language runtime, which evaluates all the events of the application in order to determine the ones that occur. The gray arrows point out the process of evaluation. As a result the language runtime determines that the event `invalidated` of the view (defined in Listing 4.9) occurs and also the event `changed` of the inspector. Thus, the event handler `invalidated` of the drawing and the event handler `changed` of the inspector have to be

```
1  Class(FigInspector, Object, [
2    Field(Figure, figure),
3
4    DecEvent(changed, [FormalParam(Figure, fig)],
5         EventAccess(FieldAccess(figure), changed, [fig])),
6
7    Handler(changed, [FormalParam(Figure, fig)], [
8      implementation,
9      Next
10   ])
11 ])
```

Listing 4.13: A second handler attached to the change of a figure.

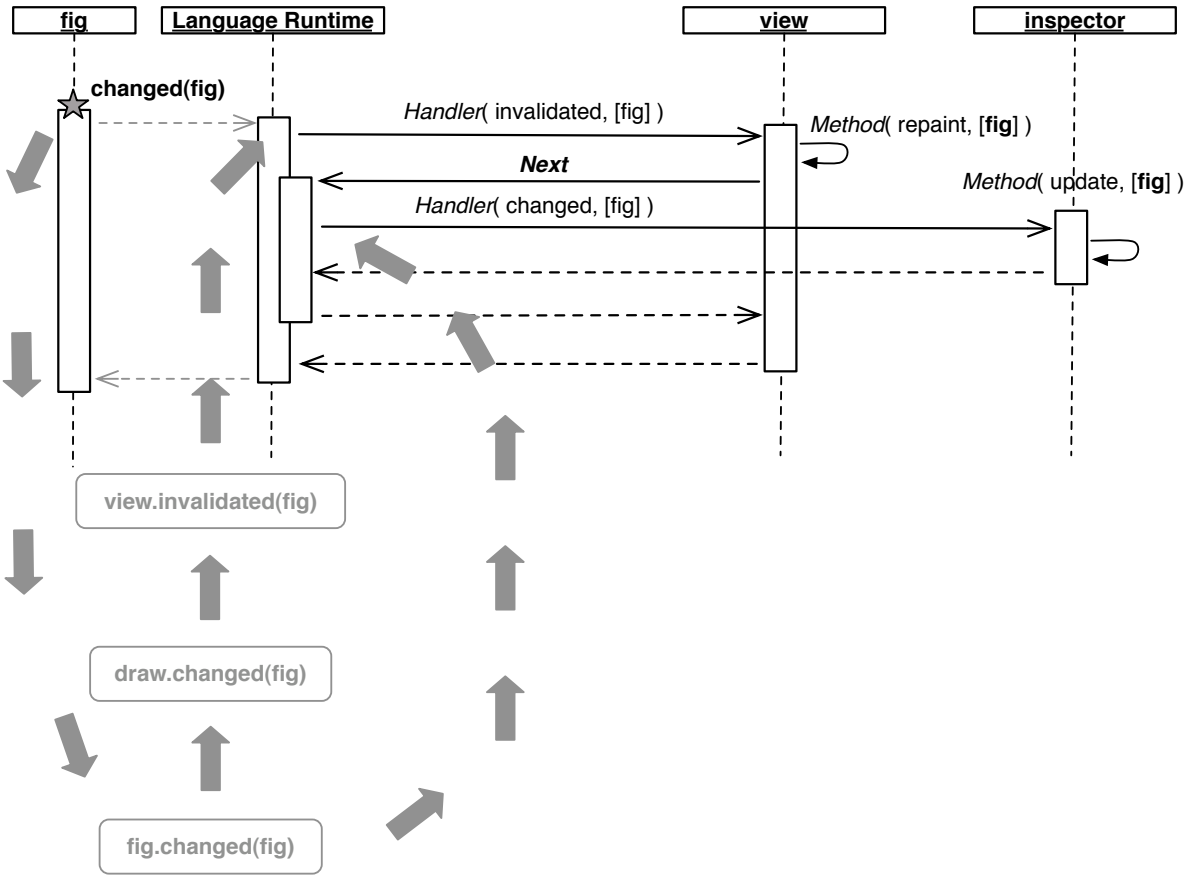


Figure 4.1 – The runtime model when a figure changes in an editor.

executed together. The handler `invalidated` is first executed (the order of event handlers is described below). The call to *Next*, after repainting the figure, nests the execution of the handler `changed`.

**Ordering of event handlers.** The order of execution of event handlers is as follows. Given a declarative event *declarativeEvent* and given an event *accessedEvent* accessed in the event expression that defines *declarativeEvent*, an event handler directly bound to *declarativeEvent* is executed before an event handler directly bound to *accessedEvent*. When this statement is not enough to determine the order between two event handlers, the order of their executions is arbitrary. A consequence of this design is that an event handler directly bound to an imperative event is always the last handler to be executed when this event is triggered.

```
1  Class(A, Object, [
2    ImpEvent(evt, []),
3    Handler(evt, [], [ Next ])
4  ])
5
6  Class(B, Object, [
7    Field(A, a),
8    DecEvent(evt1, [], EventAccess(FieldAccess(a), evt, [])),
9    Handler(evt1, [], [ Next ])
10 ])
```

Listing 4.14: Ordering of event handlers. The event handler `evt1` is executed before the event handler `evt`.

As an example, in Listing 4.14 the event handler `evt1` of an instance of `B` is executed before the execution of the event `evt` of its instance variable `a`. The order between the event handlers `evt1` of two instances of `B` is arbitrary.

### 4.3.5 Conclusion

By looking at the runtime execution of the figure editor we can see how the triggering of the event `changed` of a figure *directly* produces the execution of the handler that repaints the figure in the view. When events are composed imperatively like in C#, the same effect requires to trigger each involved composite event in a method. The model of this section improves on this situation. No intermediate method execution is needed. The model also provides a compact way to express event composition.

The next section makes even more evident how composition of events is compact and expresses in a direct way the intention of programmers.

## 4.4 A Concrete Syntax with EJAVA

The language EJAVA provides the model presented so far with a concrete syntax and a concrete implementation. The semantics of this concrete language corresponds to the semantics of the extended model previously described in Section 4.3 (except some slight modifications).

```
abstract? class JavaId ( extends JavaId )? { Member* }          -> Class
Event | Handler | JavaFieldDec | JavaMethodDec                  -> Member

AbsEvent | ImpEvent | DecEvent                                  -> Event
abstract event EventHead ;                                      -> AbsEvent
event EventHead ;                                               -> ImpEvent
event EventHead = EventExpr ;                                   -> DecEvent

JavaId ( { JavaFormalParam , }* ) JavaThrows?                   -> EventHead

EventAccess | Disjunction | Conjunction |
Refinement  | Binding     | LocalVar     | Quantification |
( EventExpr )                                                   -> EventExpr

( JavaExpr . )? JavaId ( { ExprOrType , }* )                    -> EventAccess
EventExpr || EventExpr                                          -> Disjunction
EventExpr && EventExpr                                          -> Conjunction
EventExpr && JavaExpr                                           -> Refinement
EventExpr with JavaId = JavaExpr                                -> Binding
( { JavaFormalParam , }* ) EventExpr                            -> LocalVar
some ( ( JavaId : )? JavaExpr ) . JavaId ( { JavaExpr , }* )    -> Quantification

JavaExpr | JavaType                                             -> ExprOrType

EventHead => JavaBlock[[HCtx]]                                  -> Handler
```

Listing 4.15: EJAVA syntax.

```
next                                -> JavaBlockStm[[HCtx]]
JavaReturn[[HCtx]]                  -> JavaBlockStm[[HCtx]] {reject}
```

Listing 4.16: EJAVA syntax of JAVA block statements in the context of an event handler.

EJAVA is built on top of JAVA. We skip visibility annotations for the sake of simplicity. Listing 4.15 shows the syntax of EJAVA, a concretization of the abstract syntax presented in Listing 4.5. This concrete syntax definition, as well as the other concrete definitions shown in this dissertation, are based on the syntax definition formalism SDF [Vis97b] (close to EBNF) and its implementation with scanner-less generalized-LR parsing (SGLR) [Vis97a, vdBSVV02]. A SDF production $s_1...s_n$ -> $s_0$ defines that an instance of non-terminal $s_0$ can be produced by concatenating elements from symbols $s_1...s_n$, in that order. SDF provides notation for optional (*?*) and iterated (**,+*) non-terminals. The notation *{s lit}** represents a list of *s* separated by *lit*.

EJAVA uses different symbols in a JAVA-like fashion: parameters are put between parentheses, the body of handlers is enclosed between brackets and its statements separated with semicolon. In addition, thanks to SDF, the syntax of JAVA is reused: all the non-terminals and terminals prefixed with *Java* in Listing 4.15. For example, the fields and the methods of EJAVA have the JAVA syntax.

Similarly to approaches such as C# or *e* [IEE08], EJAVA events are defined as full-fledged instance members by using the keyword `event`. Abstract events are qualified by `abstract`. Concrete syntax is provided for all the abstract event expressions introduced in Section 4.3. JAVA expressions are used to access events of external objects, bind parameters or test conditions. The disjunction of event expressions is expressed by using the operator `||`. The conjunction of event expressions and the refinement of an event expression use the operator `&&`. Parameter binding uses the keyword `with`. Local event variables are put between parentheses. In addition, a new event expression *Quantification* represents a quantification over the events of a list of objects as described in Section 4.4.3.

The syntax of event triggering is not included because it is the same as the syntax of method call.

Event handlers are defined with the signature of an event, the symbol `=>` and a body. A handler body is similar to a method body. Indeed, it is defined in Listing 4.15 as a JAVA block by using the non-terminal *JavaBlock*[[*HCtx*]]. This non-terminal includes a *mixin parameter*, a SDF facility which makes it possible to reuse the original syntax of the non-terminal for a given context. The parameter is propagated within the non-terminal, making it possible to add new productions or remove existing ones. In our case *JavaBlock* includes the context of an event handler *HCtx*. This makes it possible to include the productions of Listing 4.16. A JAVA block statement in the context of an event handler (non-terminal *JavaBlockStm*[[*HCtx*]]) can include a `next` statement implementing the *Next* expression of the model. A return statement in the context of an event handler (non-terminal *JavaReturn*[[*HCtx*]]) is removed as a valid production by using the "reject" annotation.

### 4.4.1 Examples of Basic Features

Listing 4.17 shows the EJAVA implementation of the basic figure editor of Section 4.3. The class `Figure` being the superclass of any figure of the editor declares the `abstract` event `changed` (line 2). The class `AbstractFigure` provides the event `changed` with an imperative definition (line 6) and uses it for explicit triggering (line 8). The class `CompositeFigure` defines the event `changed` as a declarative event (line 16). The event occurs when the event `changed` of some of the composed figures occurs. The event expresses in a proper way the fact that if one of the composed figures changes, the composite figure also changes. The parameter of the event represents the primitive figure that has to be repainted. Event

```
1  abstract class Figure { ...
2    abstract event changed(Figure source);
3  }
4
5  abstract class AbstractFigure extends Figure { ...
6    event changed(Figure source);
7    void moveBy(int x, int y) { ...
8      changed(this);
9    }
10 }
11
12 class Circle extends AbstractFigure { ... }
13
14 class CompositeFigure extends Figure { ...
15   Figure fig1, fig2;
16   event changed(Figure source) =
17     fig1.changed(source) || fig2.changed(source);
18 }
19
20 class Drawing extends CompositeFigure { ... }
21
22 abstract class DrawingView { ...
23   Drawing drawing;
24   event invalidated(Figure changedFig) = drawing.changed(changedFig);
25
26   invalidated(Figure changedFig) => {
27     repaint(changedFig.getDrawingArea());
28     next;
29   }
30 }
```

Listing 4.17: EJava example of abstract events (line 2), imperative events (line 6), declarative events (line 16 and 24), event handlers (line 26) and event triggering (line 8).

handlers are also defined as full-fledged instance members. The class `DrawingView` implements a view of a drawing in the editor. It defines an event `invalidated` (line 24) and an event handler bound to this event (line 26). The event `invalidated` represents a change into a situation in which the view does not reflect anymore the state of the drawing because a figure has changed. The event handler obtains the figure passed as a parameter and repaints the drawing area of the figure. The handler is named as the event it handles. Consequently, the class `DrawingView` has two members with the same name: the event `invalidated` and the handler `invalidated`. No ambiguity is introduced since events and handlers are accessed in different contexts.

### 4.4.2 Local Event Variables and Anonymous Parameters

```
1 class ConnectionFigure extends Figure { ...
2   Figure fig1, fig2;
3   event changed(Figure source) =
4     ( (Figure f) fig1.changed(f)
5     ||  fig2.changed(Figure)
6     )
7     with source = this;
8 }
```

Listing 4.18: Connector between two figures in EJAVA.

As explained in the model, local event variables allow a projection on the parameters of events. For example, Listing 4.18 shows the implementation of a class representing a connector between figures. Being a subclass of `Figure`, the class `ConnectionFigure` is equipped with an event `changed`. Line 4 shows how the definition of the event uses a local variable for capturing the parameter bound by the event `changed` of the first figure. A local event variable makes it possible to locally declare and bind a variable. The variable can be then used in conditions if needed. However, the local variable of line 4 is not further used. EJAVA includes *anonymous* parameters (non-terminal *ExprOrType* of Listing 4.15), which make it possible to indicate types in the place of parameters of event accesses without the necessity of declaring local variables. An anonymous parameter is syntactic sugar for the declaration of a local variable, without a further use of it. In the example, an anonymous parameter is used for the event `changed` of the second figure in line 5. The effect is the same as using a local variable like in line 4.

### 4.4.3 Quantification

Quantification on a collection of objects is defined in EJAVA with an event expression of the form **some**(*collection*).*eventName*(*params*), where *collection* is an expression evaluating to a collection of objects, and *eventName* an event defined by these objects (non-terminal *Quantification* of Listing 4.15).

For example, a change of a figure, which is a composition of a collection of figures, can be described in terms of the events observed on its children. The construct `some` is useful in this case. Listing 4.19 defines that the event `changed` of the composite figure occurs when the event `changed` of some composed figure occurs. The semantics of the construct `some` is a disjunction for an unbounded amount of objects. The construct `some` can, in addition,

```
abstract class CompositeFigure extends Figure {
  Collection<Figure> figures;
  event changed(Figure source) = some(figures).changed(source);
  ...
}
```

Listing 4.19: Example of use of event quantifier `some`.

```
event changed(Figure source) = some(source: figures).changed(Figure);
```

Listing 4.20: Use of event quantifier `some` capturing the object whose event occurs.

obtain the first object whose event occurs. For example, Listing 4.20 shows another version of the event `changed` binding its parameter with the first figure whose event selects an event occurrence.

### 4.4.4 Exception Handling

Dealing with exceptions in EB applications is a challenging task [PH07]. This dissertation does not go deep into this subject. We have adopted a JAVA-like treatment by using *checked exceptions*. Event handlers can throw exceptions like methods by using the `throw` keyword. An event handler that throws an exception (which is not a runtime exception) has to declare in its definition the type of these exceptions with a `throws` statement. In the absence of asynchronous events, event triggering and event handling of a given event occurrence belong to the same flow of control. As a result, events also need to be declared as throwing exceptions, which can be captured as a result of triggering an event.

This treatment of exceptions requires: (1) all related events to be declared as throwing the same types of exceptions, and (2) event handlers to be declared as throwing the same types of exceptions as their bound events modulo subtyping.

```
class B {
  A a;
  event evt3() throws E1, E2, E3 = a.evt1() || a.evt2();

  evt3() throws E1, E2, E3  => {
    if(condition)
      next;
    else
      throw new E3();
  }
}
```

Listing 4.21: Example of exceptions in EJAVA.

The example of Listing 4.21 justifies this design. An event `evt3` is defined as a disjunction of two imperative events `evt1` and `evt2`. Consider that there are handlers bound to `evt1` that can throw exceptions of type `E1` and handlers bound to `evt2` that can throw exceptions of type `E2`. A handler `evt3` is defined that can call `next` or throw and exception of type `E3`. Since `next` can nest a handler registered with `evt1` or a handler registered with

evt2, the handler evt3 has to be defined as potentially throwing exceptions of the types E1, E2 and E3. Then we say that the event evt3 can throw the same kinds of exceptions. At the same time, since the handler evt3 is a part of the process of handling the events evt1 and evt2, these events have to be defined as potentially triggering the same kinds of exceptions, E1, E2 and E3.

The problem with checked exceptions is that event sources need to preplan the type of exceptions that handlers can throw. If a handler needs to throw an exception of a type that was not taken into account, it is necessary to modify the header of its bound event and all the related events, together with the header of all the handlers bound to all the modified events.

There are some discussions about whether *checked exceptions* could be replaced by *unchecked exceptions* as in C# and other languages [Bru07]. Unchecked exception would be more suited in our case as they do not require events to declare the types of exceptions that they can throw. This is still possible in EJAVA by throwing runtime exceptions.

### 4.4.5 Deployment

```
class A {
  A a; String id;

  event bar();
  event externalBar() = a.bar();

  void foo() { bar(); }

  bar() => {
    print(id + ".bar");
  }
  externalBar() => {
    print(id + ".extBar"); next;
  }
}
```

```
class Main {
  public void main(String[] a) {
    A a1 = new A(), a2 = new A();
    a1.id = "a1"; a1.a = a2;
    a2.id = "a2"; a2.a = a1;

    //prints "a1.bar"
    a1.foo();

    deploy(a2);

    //prints "a2.extBar" "a1.bar"
    a1.foo();
  }
}
```

Listing 4.22: Example of use of deploy.

We can classify the events of an object in two categories: internal events and external events. An internal event is either an imperative event or a declarative event formed as compositions of only internal events. The other events are external. An external event is related to events of external objects. In order for an event handler that is bound to an external event to be executed on the occurrence of this external event, the object needs somehow to be registered with the corresponding external objects. If this is the case, when the object is not explicitly referred by other objects, it cannot be garbage collected immediately. This may produce unexpected results as the object, still alive, can continue to be notified of occurrences of external events. As a design decision, when an object is instantiated in EJAVA, it is only able to be notified of the occurrences of internal events (and execute the corresponding handlers), so that the object can be garbage collected. The object is able to be notified of occurrences of external events (and execute the corresponding handlers) only after *deployment*. Deployment is achieved by using a special construct deploy. The language also includes a construct undeploy that *undeploys* an object, reverting to its

original state. Listing 4.22 shows an example of use of `deploy`.

## 4.5 Related Work

The model presented in this chapter has similarities with the model of the language *e* [IEE08]. Both models provide imperative events and declarative events and treat events and event occurrences in a similar way. In *e* like in EJava a handler can only be bound to an event declared in the class. However, the similarities in the models of both languages are for a part a coincidence. The decisions taken in these models meet different needs. In the language *e* declarative events make it possible to define temporal situations, most of these events involving callbacks from simulated devices or clocks. Events do not define parameters and there is a lot of special purpose operators. The model presented in this section makes it possible to improve the design of EB applications and specially to better decouple software components. For the sake of encapsulation the model provides parametrized events, which make it possible to execute event handlers for different parameter bindings. We provide a regular treatment of event and event handler enabling inheritance as the next chapter shows. In addition, several design decisions in the model of EJava have as a justification to make it easier to integrate different programming paradigms as further chapters describe.

The declarative events introduced in this chapter are related to the complex events of approaches to Complex Event Processing (CEP) such as the language Rapide-EPL [Luc97, Luc01]. In CEP, complex events are defined by using patterns of events. These patterns use operators to relate events in a declarative way, similarly to our event expressions. Like in our model, these operators include conjunction and disjunction of events. They also include other operators that are specific to CEP. Despite the similarities of the operators, our declarative events are different from the complex events of CEP. Our events provide different views of a common runtime event, whereas in CEP a complex event relates different runtime events. For example, in CEP the conjunction of two events expresses that two occurrences of these events have happened in the past. This is not the case in our model, where the conjunction of two events defines another view of the same occurrence.

## 4.6 Conclusion

This chapter has presented a programming model that integrates imperatively triggered events with a notion of declarative events. We have presented EJava, a language that materializes the model. By using some examples we have illustrated the convenience of including declarative means to define events in EB programs.

The model presented in this chapter has been validated in previous works [NN08, NnNG09]. There we have used declarative events to design context-oriented applications. A context, describing a situation such as “it is night” or “there is somebody in the house”, is modeled as an object with two events, `in` and `out`. The event `in` activates the context. The event `out` deactivates the context. Our model makes it possible to define these events in terms of other events of the application in a declarative way. The composability of our events facilitates the composition of contexts, expressing complex situations in terms of simpler ones. A composite context is translated into a composition of the `in` and `out` events of the composed contexts. Event handlers are used to adapt an application at the beginning and at the end of a context.

In the following chapter we use the programming model and its concretization in the

EJAVA language to present the integration of the different programming paradigms put forward by this dissertation.

# Chapter 5

# Integrating EBP and OOP

The previous chapter presented the idea of declarative events. It illustrated how an EB language complementing imperative events and declarative events provides a good degree of flexibility and simplicity for defining EB behavior. This chapter looks at the presented programming model from the point of view of OOP, discussing how EBP and OOP can be better integrated. Section 5.1 motivates the contribution of this chapter. Section 5.2 revisits the programming model of Section 4.3 from the point of view of OOP and extends it with the unification of event triggering and method call and the unification of event handling and method execution. Section 5.3 describes the EJava features that implement the modeling elements of this chapter. Section 5.4 discusses related work. Finally, Section 5.5 concludes.

## 5.1 Motivation

EBP is born as a paradigm on top of OOP. For example, Smalltalk-80, the pattern Observer or the Java Listeners consider sources and destinations of events as objects. A destination subscribes to events happening in the context of the objects it explicitly knows. Languages such as C# have increased the integration between both paradigms by including events as language constructs in the core of OO languages. They offer events as instance members like methods or fields. However, a good integration implies a regular and orthogonal treatment of the new constructs. As Section 3.1 described, this is not the case in these languages:

- C# events do not support inheritance as fields or methods. An event declared in a class cannot be directly triggered in a subclass. A dedicated method has to be included in the class in order to trigger the event in subclasses. This results in a problem of regularity as events are not treated as other class members.
- The event-triggering/event-handling mechanism of EBP is very similar to the traditional method-call/method-execution mechanism of OOP. An event handler responds to the occurrence of an event as a method responds to a method call. This results in a problem of orthogonality as two similar mechanisms are provided by different language constructs.

The question that arises then is whether we can design regular and orthogonal language constructs to implement both explicit and implicit invocation. We answer to this question with a programming model that considers method calls as events and method bodies as event handlers. The next section extends our programming model with this idea. It reconsiders message passing from the point of view of EBP obtaining an orthogonal and simple design, which unifies event handling and method call. The resulting EB model is more regular and better integrated with OOP.

## 5.2 Programming Model

This section describes the principles of integrating EBP and OOP. Basically, we revisit and extend the programming model of Section 4.3, elaborating on the orthogonality of event handling and the regularity of events and event handlers.

### 5.2.1 Unification of Event Handling and Method Execution

```
Class(Callee, superclass, [ Method(T, m, [], methodBody) ])
```

Listing 5.1: A method `m` defined in a class `Callee`.

The metaphor of OOP is that objects communicate by interchanging messages. An object sends a message to another one, and the latter reacts by looking for a method and executing its body. For example, the method `m` of the class `Callee` of Listing 5.1 defines a piece of code *methodBody* to be executed as a response to a call to `m`.

Event handlers and methods play a similar role. Whereas a handler is executed as a response to the occurrence of an event, a method (body) is executed as a response to a method call. What about providing a uniform treatment of both?

#### 5.2.1.1 Method Calls as Imperative Events

```
Class(Callee, superclass, [
  ImpEvent(T, m, []),
  Handler(T, m, [], methodBody)
])
```

Listing 5.2: De-sugaring of the method `m` of the class `Callee` of Listing 5.1.

Our model provides a uniform treatment of event handling and method execution by modeling method calls as triggering imperative events.

The standard OO syntax of methods is provided as a shortcut for defining both an imperative event and an event handler registered with such an event. The event handler contains the method body. More concretely, the class definition of Listing 5.1 is de-sugared in Listing 5.2. Thus, calling a method of an object is equivalent to triggering the corresponding imperative event. The handler containing the method body is executed in the process of handling the triggered event. In other words, the method body handles the corresponding method call.

Note that the event handler of Listing 5.2 is equipped with a return type. Indeed, the unification requires some modifications to our abstract language of Section 4.3. Listing 5.3 shows the reformulated abstract syntax*. The main modification corresponds to equipping events and event handlers with a return type. The syntax also incorporates abstract methods and abstract event handlers, together with *super calls*. As a result, the abstract syntax of both methods and event handlers is the same. The remainder describes these extensions.

*. The non-terminals that are not defined correspond to the non-terminals of Listing 4.5.

*Class* ::= *t t Member*∗
*Member* ::= *Field* | *AbsEvent* | *ImpEvent* | *DecEvent*
| *AbsHandler* | *Handler* | *AbsMethod* | *Method*

*AbsEvent* ::= *t e FormalParam*∗
*ImpEvent* ::= *t e FormalParam*∗
*DecEvent* ::= *t e FormalParam*∗ *EventExpr*

*EventExpr* ::= *Super* | ... *EventExpr* of Listing 4.5
*Super* ::= *e Expr*∗

*AbsHandler* ::= *t e FormalParam*∗
*Handler* ::= *t e FormalParam*∗ *Expr*∗
*Expr* ::= *Super* | *Next* | ... *Expr* of Listing 4.5

*AbsMethod* ::= *t e FormalParam*∗
*Method* ::= *t e FormalParam*∗ *Expr*∗

Listing 5.3: Extension of the abstract syntax of Listing 4.5.

#### 5.2.1.2 Events with Return Type

As Listing 5.3 shows, an event handler is defined with a return type. It returns the value that results from evaluating the last expression of its body†. An event is also defined with a return type, indicating that the triggering of an imperative event (or a method call) returns the value returned by the first executed handler. An event handler can compose its return value with the value returned by the rest of the event handlers. This value can be obtained by using the expression *Next*. The execution of *Next* returns the value returned by the next handler in the list of handlers to be executed.

```
1 Class(Agenda, Object, [
2   ImpEvent(List, meeting, [FormalParam(Date, date)]),
3   Handler(List, meeting, [FormalParam (Date, date)], [ New(List) ])
4 ])
5
6 Class(Employee, AbstractEmployee, [
7   Field(agenda),
8
9   DecEvent(List, meeting, [FormalParam(Date, date)],
10    EventAccess(FieldAccess(agenda), meeting, [date])),
11
12  Handler(List, meeting, [FormalParam(Date, date)],
13          [Call(Next, add, [This])])
14 ])
```

Listing 5.4: Utility of events with a return value in the model.

As an example, Listing 5.4 implements an agenda with an event `meeting` representing the planning of a meeting at a certain date. Several employees can observe this event and

†. In JAVA a handler defined with a return value includes a return statement.

point out their attendance. The event `meeting` returns the list of employees that can attend the meeting. The handler of an employee is supposed to indicate whether the employee can attend the meeting. The conditional has been omitted for simplicity. In line 13, the handler uses the expression *Next* to obtain the list returned by the other handlers and add its owner object in the list. The method `add` of a list returns the resulting list.

It is important to remember the fact that an event handler directly registered with an imperative event is always the last handler to be executed when the event is triggered (see Section 4.3.4). In particular, this is the case of the event handlers implicitly defined by using method syntax.

In line 3 of Listing 5.4 the class `Agenda` defines an event handler that returns a preliminary empty list. The fact that this handler is the last handler to be executed ensures that the precedent handlers will obtain a valid initial list when calling *Next*. The event handler of line 3 is a *default event handler*. Note that the event `meeting` and the event handler `meeting` could have been defined by using a method.

The reader may wonder what happens if there is no handler registered with an imperative event defined with a return type, in particular, what happens if a value is expected when triggering the event. Since the hypothetical language considered by our model is a "pure-object" language, it is possible to impose that the return value is the object *Null*. This can be valid for languages such as Smalltalk. However, in a language such as JAVA this could be a problem, for example, when the event is defined as returning a built-in primitive type such as `int`. In a language like JAVA it would be necessary to impose that a class defining an imperative event with a return type (distinct to `void`) has to provide a default event handler as in line 3 of Listing 5.4. In other words, in these cases we should always be able to use a method syntax.

#### 5.2.1.3 Abstract Event Handlers

An event handler can be declared abstract, thus deferring its implementation to subclasses. An abstract handler obliges concrete subclasses to handle an event. This can be useful for understanding in an abstract way the relationships between event sources and event destinations.

```
1 Class(AbstractEmployee, Object, [
2   Field(agenda),
3
4   DecEvent(List, meeting, [FormalParam(Date, date)],
5     EventAccess(FieldAccess(agenda), meeting, [date])),
6
7   AbsHandler(List, meeting, [FormalParam(Date, date)])
8 ])
```

Listing 5.5: Abstract handler.

For example, Listing 5.5 defines an abstract class representing employees. It defines an abstract handler in line 7 for the event `meeting` defined in line 4. The event refers to the event `meeting` of an agenda and the event handler indicates in a proper way that an employee handles this event.

An abstract method is a shortcut for declaring an imperative event and an abstract handler registered with the event. Thus, whereas in OO languages an abstract method

requires subclasses to provide an implementation, the abstract handlers play the same role.

#### 5.2.1.4 Super Calls

Events and event handlers are provided with *super calls* (non-terminal *Super* of Listing 5.3). In the context of an event, a super call is considered as an event expression that makes it possible to access the definition of an event of the super class. Its semantics is similar to the semantics of a standard event access. The super call can be composed with other event expressions in a standard fashion.

```
Class(Employee2, Employee, [
  Field(Boolean, isAvailable),

  DecEvent(List, meeting, [FormalParam(Date, date)],
    Refinement(Super(meeting, [date]), FieldAccess(isAvailable))
])
```

Listing 5.6: Declarative-event definition using a super call.

For example, Listing 5.6 shows a subclass of the class `Employee` of Listing 5.4 that redefines the event `meeting` by attaching a condition to the event defined in the super class indicating whether the employee is available.

An event handler can also include *super calls* indicating the name of an event handler defined in the super class. The code of the event handler of the superclass can then be executed. However, this is only possible when the definition in the superclass does not include *Next* statements.

### 5.2.2 Events as Regular Instance Members

Methods are the *de facto* class members in the OO paradigm. The following three principles, included in our model, ensure a regular treatment of events and event handlers regarding to the treatment of methods in OO languages:

1. **Design of events and event handlers as polymorphic properties of objects**. This has basically three consequences. First, the access to an event (event handler) has to be qualified with an expression returning the object the event (event handler) belongs to. Second, it is possible to refer to an event (event handler) without knowledge of the concrete value of the event (event handler), which depends on the runtime object returned by the expression qualifying the event (event handler). Third, an event (event handler) definition can include instance level context such as references to members of the object the event (event handler) belongs to.
2. **Support for inheritance of events and event handlers.** The events or event handlers of a class are inherited by subclasses. Similarly to methods, subclasses can override the events or event handlers defined in the superclass. When overriding an event (event handler) a *super* expression is available to refer to the definition of an event (event handler) provided in the superclass.
3. **Abstract events and event handlers.** An event or an event handler can be declared *abstract*, deferring its definition to subclasses. The importance of abstract events

is that the reactive behavior of applications can be completely programmed without any knowledge about the way events are concretely defined. At a later stage, different versions of a piece of software can be easily instantiated by providing different event definitions adapted to each situation without modifying the already programmed reactive behavior. The importance of an abstract event handler is that it obliges a concrete subclass to implement an event. This makes it easier to understand the abstract relationships between objects.

### 5.2.3 Conclusion

As a conclusion, the events and the event handlers of our model are orthogonal and regular. Method-call/method-execution is eliminated as a different mechanism in the favor of a single and orthogonal mechanism: event-triggering/event-handling. Classes only have three kinds of members: fields, events and event handlers. Methods are included as syntactic sugar for the definition of events and event handlers. Finally, the treatment of events and event handlers is regular regarding to the treatment of methods in OO languages. The next section shows in concrete syntax the elements discussed in this section.

## 5.3 OO Features of EJAVA

This section presents an extension of EJAVA by incorporating the object-oriented features previously described, such as inheritance of events and the unification of event handling and method execution.

### 5.3.1 Features

```
JavaResultType? JavaId ( { JavaFormalParam , }* ) JavaThrows?        -> EventHead

EventHead => JavaBlock                                               -> Handler

EventAccess  | Disjunction  | Conjunction  | Refinement  |
Binding      | LocalVar     | Quantification | Super                -> EventExpr

super . JavaId ( { JavaFormalParam , }* )                            -> Super
next                                                                 -> JavaExpr
```

Listing 5.7: Extension of the EJAVA syntax of Listing 4.15 unifying methods and events.

The EJAVA syntax presented in the previous chapter (Listing 4.15) is extended, in Listing 5.7, in order to cover the elements presented in the previous section. First, events and event handlers are equipped with an optional return type (see *EventHead*). Second, *Super* is incorporated as an event expression. Third, the keyword `next` is not anymore a JAVA statement but rather a JAVA expression similarly to a method call. The syntax of the body of methods and the body of event handlers are now equivalent (a *JavaBlock*).

#### 5.3.1.1 Abstract Events

Abstract events are declared in EJAVA by using the keyword `abstract` as the syntax of Listing 5.7 shows. Similarly to methods, an abstract event must be defined in a subclass in

order to be used. Note that we have already used abstract events in the previous chapter. The class `Figure` of Listing 4.17 defines an abstract event `changed` representing a change in a figure such as a movement, a change in its size, etc. The event `changed`, similarly to the abstract methods of the class, has no definition. Definitions of abstract members are postponed to a separate stage in which specific settings can provide the needed details to implement them. The previous chapter showed the utility of defining the abstract event `changed`. Classes such as `DrawingView` (see Listing 4.17) use this event to program its behavior independently of the particular definitions made in concrete figures.

#### 5.3.1.2 Inheritance of Events

The semantics of event inheritance in EJava is analogous to the semantics of method inheritance. The inherited events can be overriden by binding them to new event expressions. The definitions of the event inherited from the subclass can be accessed through the `super` reference.

```
class ConnectionFigure2 extends ConnectionFigure {
  event childChanged() = super.changed();
  event connectionChanged();
  event changed() = connectionChanged();

  childChanged() => {
    next;
    /* recalculate area */
    connectionChanged();
  }

  Area area;
  Area getDrawingArea() {
    return area;
  }
}
```

Listing 5.8: A figure that connects figures with a line.

As an example, the class `ConnectionFigure2` of Listing 5.8 extends the class `ConnectionFigure` of Listing 4.18 and overrides the event `changed` defined there. As a reminder, the class `ConnectionFigure` defines the event `changed` in terms of the events `changed` of the connected figures. It does so by making the assumption that its method `getDrawingArea`, used by the class `DrawingView` of Listing 4.17, recalculates the shape of the connection figure. If the connected figures change, the connection will be correctly drawn. Contrary to `ConnectionFigure`, the method `getDrawingArea` of `ConnectionFigure2` does not recalculate the shape of the connection. The class keeps the area in an instance variable and the method `getDrawingArea` just returns this variable. Consequently, as soon as a connected figure changes, the connection shape has to be recalculated *in situ* and assigned to the instance variable. The class includes an event `childChanged` representing a change in a connected figure. Its definition is made in terms of the event `changed` provided in the superclass by using the keyword `super`. As a reaction to `childChanged`, the class recalculates the area and triggers an imperative event `connectionChanged` representing a change in the connection. The event `changed` of

`ConnectionFigure2` is redefined in terms of the event `connectionChanged`, representing in this way a change in the shape of the connection.

#### 5.3.1.3 Unification of Method Bodies and Event Handlers

A method in EJAVA is syntactic sugar for the definition of an imperative event with the signature of the method and an event handler binding the method body to the event.

```
abstract class AbstractCompositeFigure
      extends CompositeFigure
{
  void draw(Graphics2D g) {
    /* method body */
  }
}
```

```
abstract class AbstractCompositeFigure
      extends CompositeFigure
{
  event draw(Graphics2D g);
  draw(Graphics2D g) => {
    /* method body */
  }
}
```

Listing 5.9: Method syntax on the left and de-sugared syntax on the right.

As an example, the left side of Listing 5.9 shows the class `AbstractCompositeFigure` defining the method `draw`. This class definition is equivalent to the one on the right side. The latter defines an imperative event `draw` and an event handler `draw` containing the method body. Once the event `draw` occurs the method body is executed together with the code of all the rest of the handlers registered with the event. As already explained, a method call is equivalent to trigger the respective imperative event and the associated method execution is equivalent to handling the event.

#### 5.3.1.4 Abstract Event Handlers

```
abstract class Figure {
  abstract void draw(Graphics2D g);
  ...
}
```

```
abstract class Figure {
  event draw(Graphics2D g);
  abstract draw(Graphics2D g) =>;
}
```

Listing 5.10: Abstract method syntax and de-sugared abstract method syntax.

EJAVA makes it possible to define abstract event handlers as defined in the syntax of Listing 5.7. An example of an abstract handler is the definition of an abstract method, which is equivalent to the definition of an imperative event and an abstract handler bound to it. For example, the left of Listing 5.10 shows a class `Figure` with an abstract method `draw` and the right shows its de-sugared version.

#### 5.3.1.5 Inheritance of Event Handlers

Due to the unification of method bodies and event handlers, inheritance of event handlers becomes equivalent to inheritance of methods. Event handlers have an identity given by the name of the event they handle. This makes it possible to override them in subclasses as shown in Listing 5.11. The class `LabeledLineConnectionFigure` overrides the handler `transform` defined in its superclass. The handler `transform` is indeed a handler bound to

```
1 class LabeledConnectionFigure extends ConnectionFigure {
2   transform(AffineTransform tx) => {
3     super.transform(tx)
4     /* rest of the implementation */
5   }
6   ...
7 }
```

Listing 5.11: Inheritance of event handlers.

the imperative event `transform` defined in the hierarchy of figure. This design makes the class have two members with the same name. No ambiguity is produced because these members are accessed in different contexts. For example, the statement `super.transform(tx)` of Line 3 nests the execution of the handler `transform` defined in the super class and does not correspond to an event triggering. The same statement without the `super` keyword would be interpreted as a triggering of the event `transform`.

The unification of event handling and method execution in a common notion of message passing permits a more flexible definition of events. New events can be declaratively defined as composition of events defined with method syntax.

### 5.3.2 Example

```
abstract class Agenda {
  abstract List meeting(Date date);
}
abstract class Employee {
  abstract boolean isAvailable(Date d);
}
```

Listing 5.12: Abstract structure of an agenda.

EJAVA can be used to implement an agenda with the structure given in Listing 5.12. The abstract class `Agenda` declares an abstract method `meeting`, which is invoked for scheduling a meeting at a given date. It returns a list with the employees that can attend the meeting. The abstract class `Employee` declares an abstract method `isAvailable` indicating the availability of the employee at a given date.

EJAVA makes it possible to implement these interfaces by using either explicit invocation or implicit invocation, both in a straightforward way. As an example, Listing 5.13 shows an implementation with explicit invocation. The class `ExplicitAgenda` keeps a list of employees. Its implementation of `meeting` iterates on this list checking the availability of each employee and adding the available ones to the resulting list. The class `ExplicitEmployee` extends `Employee` with a basic implementation of the method `isAvailable`.

Listing 5.14 shows an implementation of this example using implicit invocation. The class `ImplicitAgenda` is "lazy". It implements `meeting` by returning an empty list. It is the role of each employee to manifest its interest in a meeting. The class `ImplicitEmployee` observes the imperative event `meeting` of its instance variable `agenda`. Such an event has been implicitly defined by the method `meeting`. The class `ImplicitEmployee` defines a declarative event `meeting` in terms of the event `meeting` of `agenda` refined with the

```
class ExplicitAgenda implements Agenda {
  List<Employee> employees;

  List meeting(Date date) {
    List result = new ArrayList();
    for(Employee e: employees) {
      if(e.isAvailable(date))
        result.add(e);
    }
    return result;
  }
}

class ExplicitEmployee extends Employee {
  boolean isAvailable(Date d) { ... }
}
```

Listing 5.13: Implementation of the agenda of Listing 5.12 with explicit invocation.

```
class ImplicitAgenda implements Agenda {
  List meeting(Date date) {
    return new ArrayList();
  }
}

class ImplicitEmployee extends ExplicitEmployee {
  Agenda agenda;
  event List meeting() = (Date d) agenda.meeting(d) && isAvailable(d);

  List meeting() => {
     List list = next;
     list.add(this);
     return list;
  }
}
```

Listing 5.14: Utility of events with a return value in the model.

condition `isAvailable`. A handler attached to this event adds the employee to the returned list.

The example points out how the unification of event handling and method execution increases the expressiveness of a language such as Java. A standard class such as `Agenda` can be implemented either by using explicit invocation in a standard fashion or by using implicit invocation facilitated by the previous unification. In addition, the example shows the regularity of events and event handlers as they are used as any instance member.

## 5.4 Related Work

In the actor model [HBS73, Hew10] the sending and the receipt of a message are decoupled, so that *events represent the receiving of a message by an actor* [Gre75]. When several actors *concurrently* send several messages to an actor, the order in which these events will be received by the actor is undetermined. The actor captures the message receipts as events happening at different points in time and can choose when and how to react to them. Our model, instead of modeling message receipts as events, models message sends as events. The reason is that calling a method is an imperative action, which can be easily seen as imperatively triggering an event. Indeed, since we do not deal yet with concurrency or distributed systems, choosing between attaching a method body to a method sending or a method receipt is analogous. As future work we plan to provide a finer model including message receipts as events.

Our approach is related to approaches that, like C#, Qt and *e* [IEE08], include events as class members. In these approaches, methods can implement event handlers, however, there is no unification of event handling and method execution as we propose.

## 5.5 Conclusion

This chapter has presented an integration of EBP and OOP based on a unification of event handling and method execution. This permits us to use EJava for both EBP and OOP in a regular way. In the next chapter we will take a different point of view. We will show how the notion of event can be used to program aspects and how our programming model contributes to a better integration of EBP and AOP.

# CHAPTER 6

# Integrating EBP and AOP

The previous chapter described how EBP and OOP can be better integrated by unifying events and methods. This chapter describes the integration of EBP and AOP. Section 6.1 motivates the need for a better integration of EBP and AOP. Section 6.2 presents the elements of our programming model integrating these paradigms. Section 6.3 materializes this model by showing how classical AO examples can be programmed in EJAVA. Section 6.4 discusses related work. Finally, Section 6.5 concludes.

## 6.1 Motivation

As argued in Section 3.2, the main difference between EBP and AOP is a matter of *obliviousness* and *modularity*. EB programs explicitly trigger events at the source side causing the notification of the respective event handlers at the destination side. Sources need to pre-plan the events that destinations may need and insert the event calls at the proper places in the code. Obliviousness is desirable because it allows greater separation of concerns [FECA05]. For instance, in most of the examples of the previous chapters the figures of the figure editor define an event `changed`. However, it can be that at some point in time the destinations require finer-grained events such as `moved` or `resized`. If it is the case, the sources need to be refactored in order to fulfill the new requirements. AO programs provide the destinations with the flexibility of defining the events that they need. This is transparent for the sources, which are unaware of the computations that trigger these events. However, as described in Section 2.2.5, AOP sacrifices modularity for obliviousness: it basically works on the static structure of programs and it is invasive, so that it may not respect the inter-object interfaces of sources. Besides this difference, the mechanisms are conceptually similar as both implement some form of implicit invocation. Providing similar mechanisms with different language constructs conflicts with the orthogonality principle of programming-language design [Mac95].

What about providing a more general model including both the EB mechanism and the AO mechanism? Some existing work goes in this direction. Approaches such as EAOP consider join points as events, which may be part of the computation history of a program. More technically, aspects describe actions to perform along certain sequences of events. However, EAOP [DFS04] is an AO model and does not enable source-side event triggering. Recently, PTOLEMY [RL08] has proposed a model, which integrates EB ideas and AO ideas. However, PTOLEMY is more a new EB language including AO features such as quantification than a language featuring a general approach in which both AOP and EBP are symmetrically enabled. In particular, source-side obliviousness cannot be achieved in PTOLEMY.

Based on the previous remarks this chapter reconsiders AOP from the point of view of EBP. As a result, we get a programming model providing a good integration of both worlds, enabling both styles of programming in a symmetric way. A big part of this integration is a product of the unification of event handling and method execution already presented in Chapter 5. We show how this unification turns EJAVA into a language for separation of crosscutting concerns comparable to languages such as ASPECTJ [KHH+01].

## 6.2 Programming Model

EBP and AOP are in general not well-integrated because the mechanisms of event handling and advising are treated in an orthogonal way. The key to the integration of EBP and AOP proposed by our model is the unification of these mechanisms. We show how this unification is possible thanks to the unification of event handling and method execution introduced in the previous chapter.

### 6.2.1 Unification of Event Handling and Advising

As explained in Section 5.2.1, a method is syntactic sugar for the definition of an imperative event and an event handler registered with such an event. This event handler contains the method body. Consequently, pieces of advice can be simply modeled as event handlers handling the occurrences of the same imperative event. The semantics of the model ensures that the event handler introduced by a method is the last event handler to be executed when the event is triggered (see Section 4.3.4). Thus, the control will be always given to the pieces of advice before it is given to the method body. The pieces of advice can control, in this way, the execution of the method body in the same way a standard piece of advice controls its join point. Let us describe this in more detail.

```
aspect Aspect {
  pointcut m() = call(void Callee.m());

  void around(): m() {
    /* before code */ proceed(); /* after code */
  }
}
```

Listing 6.1: An ASPECTJ aspect advising a method `m` of a class `Callee`.

**Advising in ASPECTJ.** Given a class `Callee` with a method `m`, the ASPECTJ aspect of Listing 6.1 defines an *around* piece of advice whose execution *replaces* the call to the method `m` of such a class. The keyword `proceed` nests the execution of the join point in the advice body if needed. In other words, an around piece of advice takes the control of the execution of the join point, preventing the base program from executing it when `proceed` is not invoked.

If several around pieces of advice have been defined for a call to `m`, then all these pieces of advice are composed using `proceed`. When the first piece of advice is executed, a call to `proceed` nests the execution of the next piece of advice in its body. The next piece of advice does the same, and so on. When the last piece of advice is executed, a call to `proceed`

nests the execution of the join point. If some piece of advice does not call `proceed`, then the rest of the pieces of advice are not executed nor the join point.

```
Class(Aspect, Object, [
  Field(Callee, callee),

  DecEvent(void, m, EventAccess(FieldAccess(callee), m)),

  Handler(void, m, [beforeCode, Next , afterCode])
])
```

Listing 6.2: Modeling an aspect with an event and an event handler.

**Advising in EJAVA.** Section 5.2.1 showed that in our model the method `m` of a class `Callee` (Listing 5.1), is just syntactic sugar for the definition of an imperative event representing a call to `m`, and an event handler registered with that event (Listing 5.2). A class in our model, on behalf of a design-level aspect, can observe the event of a method call and implement a piece of advice as an event handler. The ASPECTJ aspect of Listing 6.1 can be expressed in our model as shown in Listing 6.2. The aspect is modeled as a class with a field `callee`. The pointcut is modeled as a declarative event `m` accessing the event `m` of `callee` (the method call). The corresponding piece of advice is programmed as an event handler bound to the event `m`. As a result, two event handlers are bound to the same event: the one of Listing 5.2 containing the method body and the one of Listing 6.2 containing the advice body. The effect of the ASPECTJ aspect of Listing 6.1 is obtained because the handler programmed in the aspect is executed first, so that it controls the execution of the method body. Indeed, Section 4.3.4 describes that an event handler directly registered with an imperative event is always the last handler to be executed when the event occurs. The expression *Next* in the event handler of the aspect nests the execution of the next event handler, *i.e.* the method body, in the same way a call to `proceed` nests the execution of the method body in ASPECTJ.

### 6.2.2 After Events and Point-in-time Model

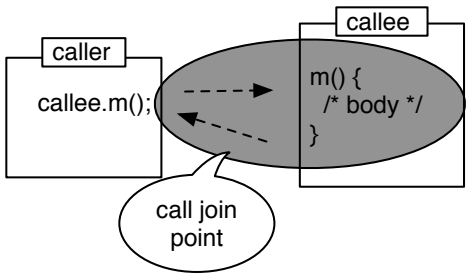


Figure 6.1 – ASPECTJ point-in-region model.

Our model is compatible with the join-point model of Masuhara et al. [MEY06], *i.e.* join points (events) are atomic points in execution time. The Masuhara's model is finer-grained than the join-point model of ASPECTJ, which considers a join point as a region. For example, this region includes the method-body execution in the case of a method-call

join point. Figure 6.1 illustrates the region that a method-call join point represents in ASPECTJ for a method `m`.

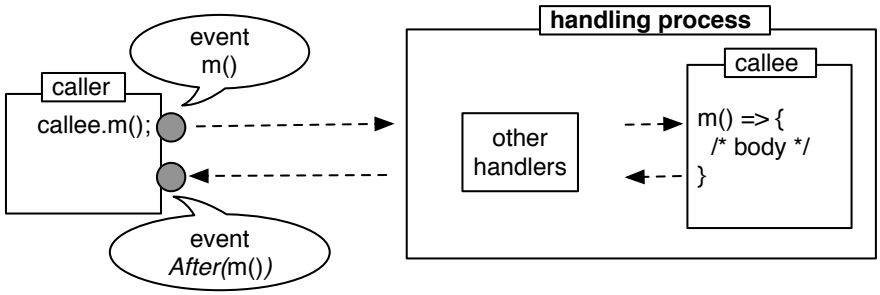


Figure 6.2 – EJAVA point-in-time model.

In our model, a method-call corresponds to an atomic event: the triggering of the corresponding imperative event. Figure 6.2 illustrates this event as the small gray circle on top. This atomic event is similar to the call join point of the model of Masuhara et al. [MEY06]. As a reaction to this event, a process of handling is started. All the handlers are executed, where the last one corresponds to the method body.

*EventExpr* ::= *After* | ... *EventExpr* of Listing 5.3
*After* ::= *Expr e Expr* v?*

Listing 6.3: Abstract syntax of after events.

Note in Figure 6.2 a second gray circle at the bottom of the previous one. This circle represents an event *After(*`m`*)* that is triggered once the event `m` has been handled. This event is called an *after event* and is equivalent to the reception join point of the model of Masuhara et al. [MEY06]. Our model introduces after events implicitly to be able to notify that an event has been handled. For every event definition, an associated after event is made available. This after event can be accessed in the definition of other events by using an event expression with the abstract syntax of Listing 6.3. The non-terminal *After* is similar to an event access. In addition, it can include an optional variable that makes it possible to bind the value returned by the process of handling. After events are useful when the handling of an event introduces changes and we need to react to such changes. For example, an after event can be used to define the event `changed` of a figure as triggered after the handling of events such as `moveBy`. Section 6.3.2 will show the use of after events.

Let us now see the unification of EBP and AOP in concrete syntax.

## 6.3 AOP Support of EJAVA

This section shows how standard OO programs can be enhanced with aspects in EJAVA. These aspects are classes defining event handlers for method-call events, *i.e.* imperative events implicitly defined with method syntax.

### 6.3.1 Simple AO Examples

Given a program including the class `AbstractCompositeFigure` of Listing 5.9, Listing 6.4 implements a simple aspect that traces the executions of the method `draw` of

```
1 class FigureTracing {
2   Figure figure;
3   event drawPct(Graphics2D g) = figure.draw(g);
4
5   drawPct(Graphics2D g) => {
6     next;
7     log("Method␣'draw'␣has␣been␣executed");
8   }
9 }
```

Listing 6.4: Aspect saving a message after a figure is drawn. The call to `next` nests the execution of the method body, which is one of the next event handlers.

instances of such a class in EJAVA. The aspect observes the event that represents that the message `draw` has been sent to the figure and defines a handler that saves a proper logging message. The message is correctly saved after the body of method `draw` is executed. Indeed such a method body just corresponds to an event handler that is registered for the call to `draw` as explained in Chapter 5. The call to `next` of line 6 makes it possible to execute all the remaining event handlers, including the one with the method body. The class `FigureTracing` plays the role of an aspect, its event `drawPct` plays the role of a pointcut and its event handler `drawPct` plays the role of a piece of advice.

```
class BackupAspect {
  File file;
  event savePct() = file.save();

  savePct() => {
    backup(file); next;
  }
}
```

Listing 6.5: Aspect that backups a file before it is saved.

Listing 6.5 shows another typical aspect which makes backups of a file before the file is saved. The file is an instance of a class `File`, which defines a method `save`. The aspect defines an event `savePct` representing the sending of a message `save` to a `file`. It also defines a proper event handler, which performs the `backup` and then calls `next` in order to nest the execution of the remaining handlers, in particular, the one with the body of the method `save` of the class `File`.

Finally, Listing 6.6 shows an example of an aspect that measures the time elapsed in the execution of a command.

### 6.3.2 After Events

EJAVA supports the after events introduced in Section 6.2.2. Every event definition implicitly introduces the definition of an after event. After events are accessed by using an event expression with the syntax of Listing 6.7. Line 2 shows its canonical form. An event expression with the form `after`(*expr.evt*(*params*), *var*) accesses the after event implicitly defined for the event *expr.evt*(*params*). The second argument *var*, which is optional, binds

```
class CommandTimingAspect {
  Command command;
  event execute() = command.execute();

  execute() => {
    long t = time(); next;
    saveElapsedTime(time() - t);
  } ...
}
```

Listing 6.6: Aspect that times the execution of a command.

```
1 After | ... EventExpr of Listing 5.7          -> EventExpr
2 after ( MemberAccess ( , JavaId )? )          -> After
3 after ( EventExpr ( , JavaId )? )             -> After
4 ( JavaExpr . )? JavaId ( { ExprOrType , }* )  -> MemberAccess
```

Listing 6.7: EJAVA syntax including before and after events.

the value returned by the handling of the previous event. Line 3 introduces syntactic sugar that makes it possible to inline the definition of a declarative event inside `after`.

```
abstract class AbstractFigure extends Figure {
  event aboutToChange(Figure source) =
    (moveBy(int,int) || resize(int)) with source = this;

  event changed(Figure source) = after(aboutToChange(source));
  ...
}
```

Listing 6.8: Declarative definition of the event `changed` of primitive figures.

As an example, after events can be used to improve the figure editor of Chapter 4. Such a chapter illustrated the flexibility of our model to implement the event `changed` of composite figures in a declarative way. However, the event `changed` of primitive figures (instances of subclasses of `AbstractFigure`) was still defined as an imperative event manually triggered at the end of the methods that change the figures. This manual triggering crosscuts all the methods that change the figures and introduces tangling. The after events of EJAVA make it possible to express the event `changed` of a primitive figure in a declarative way in terms of the calls to the methods that change the figure. See Listing 6.8. The event `aboutToChange` is defined as the disjunction of `moveBy` and `resize`, where `moveBy` and `resize` are methods that change the figure. The event `changed` is defined as an access to the after event implicitly defined for `aboutToChange`. In other words, `changed` is programmed to occur after the handling of the event `aboutToChange` and consequently after the figure changes. Listing 6.9 shows an equivalent version with the event expression defining `aboutToChange` in-lined inside the `after` construct.

With this extension the event `changed` of a primitive figure can be defined in a declarative way. As a normal event, `changed` can still be refined in subclasses as in Listing 6.10.

```
abstract class AbstractFigure extends Figure {
  event changed(Figure source) =
    after((moveBy(int,int) || resize(int)) with source = this);
  ...
}
```

Listing 6.9: Alternative declarative definition of the event `changed` of primitive figures.

```
class Circle extends AbstractFigure {
  event changed(Figure source) =
    super.changed(source) || after(setRadius(int) with source = this);
  ...
}
```

Listing 6.10: Inheritance of the event `changed` of primitive figures.

### 6.3.3 Telecom Example

Let us now show the implementation in EJAVA of an example taken from the distribution of ASPECTJ. It corresponds to a Telecom Simulation, a small application that simulates a telephony system in which customers make, accept, merge and hang-up both local and long distance calls.* In the example, two aspects are provided. First, a timing aspect is concerned with timing the connections and keeping the total connection time per customer. Second, the information captured by the timing aspect is later on used by a billing aspect in order to charge customers for their calls.

The telecom simulation comprises, between others, the classes `Connection` and `Customer`. Simple calls are made between one customer (the caller) and another (the receiver), a `Connection` object connects them. Conference calls between more than two customers will involve more than one connection. A customer may be involved in many calls at a time.

**Timing aspect.** Listing 6.11 shows the implementation of the timing aspect in EJAVA. The aspect is instantiated with the list of customers to monitor (line 5). The role of the aspect is to record the total connection time of each customer (method `getConnectionTime` of line 6). The event `call` of line 8 represents the instant after the call to the method `call` of a customer is handled. The variable `ret` captures the returned `Call` object. The `call` handler, in line 12, captures the connection created by the call to be stored in a list with the connections of the monitored customers (variable `connections` of line 3). The event `completed` of line 9 represents the instant after the call to the method `complete` of some connection has been handled. It indicates that a connection has been established. The handler bound to this event, in line 17, obtains the timer for the given connection and starts it. The event `dropped` of line 10 is defined in an analogous way. On the occurrence of this event, the timer associated to the respective connection is stopped and the connection times of both the caller and the receiver of the connection are updated (line 22).

Similarly to the ASPECTJ solution, the class `Timing` of Listing 6.11 modularizes a crosscutting concern. The classes of the application model (the classes `Connection` and

*. http://eclipse.org/aspectj/doc/released/progguide/examples-production.html

```
1  class Timing {
2    List<Customer> customers;
3    List<Connection> connections;
4
5    Timing(List<Customer> customers) { ... }
6    Long getConnectionTime(Customer customer) { ... }
7
8    event call(Call ret) = after(some(customers()).call(Customer), ret);
9    event completed(Connection c) = after(some(c: connections()).complete());
10   event dropped(Connection c) = after(some(c: connections()).drop());
11
12   call(Call call) => {
13     storeConnection(call.getInitialConnection());
14     next;
15   }
16
17   completed(Connection conn) => {
18     getTimer(conn).start();
19     next;
20   }
21
22   dropped(Connection conn) => {
23     getTimer(conn).stop();
24     Long time = getTimer(conn).getTime();
25
26     updateConnectionTime(conn.getCaller(), time);
27     updateConnectionTime(conn.getReceiver(), time);
28     next;
29   }
30   ...
31 }
```

Listing 6.11: Timing aspect in EJAVA.

Customer among others) are unaware of the timing concern. They implicitly trigger events when calling the methods of the model. These events are observed by the timing aspect in a transparent way. Unlike the AspectJ solution, the timing aspect implemented in EJava is instantiated for a group of customers. The AspectJ solution, however, works on the static structure of the application. This difference is discussed Chapter 7.

```
1  class Billing {
2    Timing timing;
3
4    Billing(List<Customer> customers) {
5      timing = new Timing(customers);
6    }
7    Long getCharge(Customer customer) { ... }
8
9    event billing(Connection conn) = after(timing.completed(conn));
10
11   billing(Connection conn) => {
12     Long time = timing.getConnectionTimer(conn).getTime();
13     long cost = callRate(conn) * time.longValue();
14     updateCharge(conn.getCaller(), cost);
15     next;
16   } ...
17 }
```

Listing 6.12: Billing aspect in EJava.

**Billing aspect.** Listing 6.12 implements the billing aspect of the Telecom application in EJava. This aspect uses the timing aspect of Listing 6.11 in order to calculate the cost of connection of a group of customers (line 2). The aspect defines an event `billing` in line 9, representing the instant when a timing has been performed. Actually, the event is defined as an alias to the after event of the event `completed` of the timing aspect. The handler `billing` of line 11 obtains the time of a connection using the timing aspect and calculates the cost. The cost is attached to the caller of the connection.

The implementation of the billing concern in EJava shows again how the unification of event handling and method execution makes it possible to implement aspects. This implementation highlights the fact that aspects can be related to each other because they are regular classes: the billing aspect instantiates a timing aspect. This is discussed in more detail in the next chapter.

## 6.4 Related Work

EJava is closely related to approaches that combine ideas of imperative events and AOP such as Ptolemy [RL08] and Implicit Invocation with Implicit Announcement (IIIA) [SPAK10]. Ptolemy and IIIA occupy different points in the design space. First, unlike EJava and Ptolemy, IIIA keeps classes and aspects distinct, but, like EJava, provides both imperative and implicit events (defined through AspectJ-like pointcuts), whereas Ptolemy does not provide implicit events. Second, and more importantly, Ptolemy and IIIA event handlers/advice react to events characterized by their (event) type. Using

event types fully decouples event sources and sinks while providing robust interfaces between both ends. But it also means that events are "application-level" events. Restricting observation to specific sources requires either to set up a filtering mechanism on the observer/sink side, with possible performance issues if too many useless events are emitted, or to fall back on the standard pattern Observer. In line with the OO philosophy, we have rather chosen the opposite route by making it easy and efficient to observe selected events from selected objects, handling the pattern Observer behind the scene.

In addition to these crucial differences, there are some other noticeable ones. First, working with event types, PTOLEMY and IIIA are more limited with respect to composing event abstractions. In PTOLEMY, the only composition operator is a disjunction operator, which makes it possible to associate a single event handler to a disjunction of event types. In IIIA, event types can be organized in an inheritance hierarchy defining their subtyping relationship and implemented by ASPECTJ-like polymorphic pointcuts (different pointcuts are attached to different classes but produce events of the same type). The definition of an event type can be seen as a disjunction of its class-local definitions. This has to be contrasted with EJAVA, which provides a rich set of composition operators, on *event names* rather than event types, which makes it possible to create new event names.

The declarative events of EJAVA are similar to the (declarative) ASPECTJ-like pointcuts, where we use event names instead of pointcut designators. A declarative event selects and event occurrence, which is equivalent to a join point. An event expression supports disjunction and conjunction and can be refined with conditions. However, the conditions of EJAVA are dynamic as they can include instance-level context, contrasting with the static nature of ASPECTJ-like pointcuts. In addition, ASPECTJ pointcut designators such as `this` are not longer needed, because EJAVA events observe join points associated to already known sources. Since EJAVA aims at keeping the modularity of sources, EJAVA pointcuts only act on the public interface of sources: their method executions. In this regard, EJAVA is very related to other AO language that work on the interface of objects, such as ASPECTS [Hir02]. The join-point model of EJAVA is closer to the join-point-in-time model of Masuhara than the point-in-region model of ASPECTJ as explained in Section 6.2.2. Even if, for the time being, we only support two of the four join-point types proposed by Masuhara, EJAVA makes it possible to cover the ASPECTJ `call` join points and implement scenarios that require these kinds of join points. Support for a richer join-point model is left as a perspective.

Finally, EJAVA uses the same construct to deal with both method bodies and pieces of advice: an event handler. This makes the treatment of advice similar to languages such as ASPECTWERKZ [Bon04], which implements pieces of advice by using methods.

## 6.5 Conclusion

This chapter has shown how EBP and AOP can be better integrated. We have done so by describing the support of our programming model for AOP. In previous chapters we presented EJAVA as an AO language well integrated with OOP. The same integration has been the basis to present EJAVA as an AO language. An advantage of the integration is that it can promote the adoption of AOP. In general, people from the OO world get more comfortable with concepts such as events and event handlers, maybe as a the result of the fact that the notion of event was born together with the notion of object. Programmers that are not comfortable with AOP can use the language as an EB and OO tool and ignore the concepts tied to AOP. The integration may permit an adoption of AOP at a later stage

without modifying already existing code and without much effort.

# Chapter 7

# Integrating AOP and OOP

The previous chapters have shown how our programming model improves the integration of EBP and OOP, and the integration of EBP and AOP. This chapter discusses the last side of the triangle. It describes how our model contributes to the integration of AOP and OOP. Section 7.1 motivates the need for integrating both paradigms. Section 7.2 revisits our programming model, already presented in the previous chapters, showing how it contributes to a better integration of AOP and OOP. Section 7.3 illustrates the expressiveness of EJava by showing its ability to offer new programming possibilities besides the support for standard AOP idioms. Section 7.4 discusses related work. Finally, Section 7.5 concludes.

## 7.1 Motivation

AOP emerged as a complement to OOP making it possible to locally capture, down to the implementation, *crosscutting* concerns that would otherwise result in scattered and tangled code [KLM+97]. AOP features an *asymmetric* composition of concerns, which distinguishes classes, implementing *base* concerns, from *aspects*, implementing crosscutting concerns. In its most successful incarnation, AspectJ [KHH+01], aspects look very much like classes but contain two new member types, *pointcuts* and *advice*. Although an aspect looks very much like a class, the rules governing its static and dynamic semantics are different. In particular, it can be extended as a class but in a constrained way and it cannot be explicitly instantiated. This adds quite a lot of complexity to standard OOP. On the one hand, there are a number of new constructs. On the other hand, the basic principles for working with these constructs (instantiation, inheritance) are not regular. As a whole, they have resulted in a very effective tool for practitioners as well as a concrete token against which researchers could test new ideas. However, now that the concepts at stake are better understood, one may wonder whether this complexity, and especially this lack of regularity, could not be lowered to the benefit of a wider adoption of AOP, making it easier to implement AOP in new languages, as well as to learn and use AOP. Such a position has, for instance, been put forward by the authors of Classpects [RS05, RS09].

The next section describes how our model reconsiders the basic ingredients of AOP at the light of the previous remarks and slightly alter them in order to get both regularity and symmetry while keeping the possibility of defining behavioral aspects with a direct dynamic interpretation.

## 7.2 Programming Model

### 7.2.1 Principles

The starting point of our design, shared with CLASSPECTS, and many other language designs, are the principles of simplicity, orthogonality and regularity put forward by MacLennan [Mac95]. The four main characteristics that make ASPECTJ's aspects different from plain classes are: implicit instantiation, limited inheritance, specific members (pointcuts, advice), and intertype declarations. In Section 3.3 we discussed these characteristics at the light of the above-mentioned principles. Let us now explain how and why our programming model diverge from ASPECTJ. Our treatment of the first two characteristics is not novel but is still worth discussing in order to make the overall logic of the design clear.

In CAESARJ and CLASSPECTS, aspects are implemented as classes. The same instantiation and inheritance rules are used. However, the rules governing pointcuts and advice are still specific. Pointcuts are class members and pieces of advice are anonymous. What about promoting pointcuts and advice as full-fledged instance members?

**Pointcuts** The basic advantage of promoting pointcuts as instance members is that they can then refer to dynamic information (instance state) and can therefore be more selective. When defined in a different class than its associated piece of advice, this means that the pointcut can select join points based on the state of the advised instance (on the state of the subject, using the vocabulary of the observer pattern), beyond the information accessible through the join point. When defined in the same class, the pointcut can select join points based on a combination of the dynamic state of the aspect (or observer) and the dynamic state of the advised instance available through the join point.

The use of dynamic pointcuts makes it possible to locate the tests on the dynamic state of the aspect indifferently in the pointcut or in the piece of advice, depending on the logic of the test, which can either be seen as part of join point selection or part of the advice body. This makes a difference in terms of reuse as well as when aspects of aspects are used (pointcuts are not advised whereas advice bodies are). It also changes the picture when considering the use of quantification without obliviousness. In such a case, the pointcut is part of the interface between the advised class and the aspect. Such an interface should not reveal the details of the implementation of the advised class. Thus, it may be impossible to put the dynamic tests in the advice body, as the latter may not have access to the relevant dynamic information.

In our model, pointcuts are dynamic. They are implemented as declarative events, which, as described in Chapter 5, are full-fledged instance members and can then refer to dynamic information.

**Advice** In ASPECTJ and CAESARJ, a piece of advice has access, within the advice body, to instance state (in CLASSPECTS, this is true of the *advising method*, *i.e.*, the method body of the method called in the piece of advice). However, this piece of advice cannot be redefined in a subclass (in CLASSPECTS, the *advising method* can be redefined, not the existing binding between a pointcut and a method). Making it possible to redefine a piece of advice requires to be able to identify it.

A piece of advice can be identified by the name of the pointcut it advises as in Listing 7.1. Identifying a piece of advice necessarily excludes the possibility of using a syntax with an anonymous pointcut as Listing 7.2 shows. When excluding anonymous pointcuts,

```
pointcut detect(): execution(Sensor.detect()) { ... }
before(): detect() { ... }
```

Listing 7.1: A piece of advice in ASPECTJ with its (named) pointcut.

```
before(): execution(Sensor.detect()) { ... }
```

Listing 7.2: A piece of advice in ASPECTJ with an anonymous pointcut.

the ASPECTJ syntax can be simplified so that only the control and the pointcut are necessary as in Listing 7.3.

The design of Listing 7.3 coincides with the design of our event handlers. In our model, pieces of advice implemented as event handlers have an identity given by the event they handle, thus enabling inheritance.

**Redefinition, overloading and interfaces** In our model, both advice and pointcuts can be redefined using `super` and overloaded. When redefining a piece of advice, `super` within the definition of the pointcut refers to the definition of the pointcut in the superclass and `super` within the definition of the advice body refers to the definition of the advice body in the superclass.

In the case of pointcuts, overloading refers to the possibility of having pointcuts with the same name and different signature. In the case of advice, overloading refers to the possibility of having pieces of advice bound to pointcuts with the same name and different signature. For example, EJAVA makes it possible to write the class of Listing 7.4. Both events `changed`, modeling pointcuts, are different as their signatures are different. Similarly the event handlers, modeling pieces of advice, are different as they are bound to events with different signatures.

Finally, as full-fledged instance members, in our model both pointcuts and advice may appear in interfaces as standard class members.

The next section describes the advantages of the integration of AOP and OOP enabled by our programming model.

### 7.2.2 Advantages

As previously described, our model reconsiders the basic ingredients of AOP in order to get both regularity and symmetry while keeping the possibility of defining behavioral aspects with a direct dynamic interpretation. Two key design decisions make this possible: (1) pieces of advice and pointcuts are made full-fledged instance members, (2) method call is seen as *explicitly* triggering imperative events. As a result, we get a model that is simpler and more regular than traditional AOP, while being more expressive.

**Simplicity and regularity** There is no need to distinguish between classes and aspects with special instantiation and inheritance rules for aspects. Methods and pieces of advice can be seen as different disguises for the same concept, that we refer to as *event handlers*, whereas pointcuts can be seen as *event selectors* – we will refer to them as events as they are a way to denote the events they select. Pointcuts/events and pieces of advice/event handlers can be inherited and redefined, possibly referring to their `super` definition.

```
pointcut detect(): execution(Sensor.detect()) { ... }
before detect() { ... }
```

Listing 7.3: A simple piece of advice.

```
class Aspect {
  List<Figure> figures;

  event changed() = some(figures).moveBy(int, int);
  event changed(Figure source) = some(source: figures).moveBy(int, int);

  changed() => { ... }
  changed(Figure source) => { ... }
}
```

Listing 7.4: Overloading of events and event handlers.

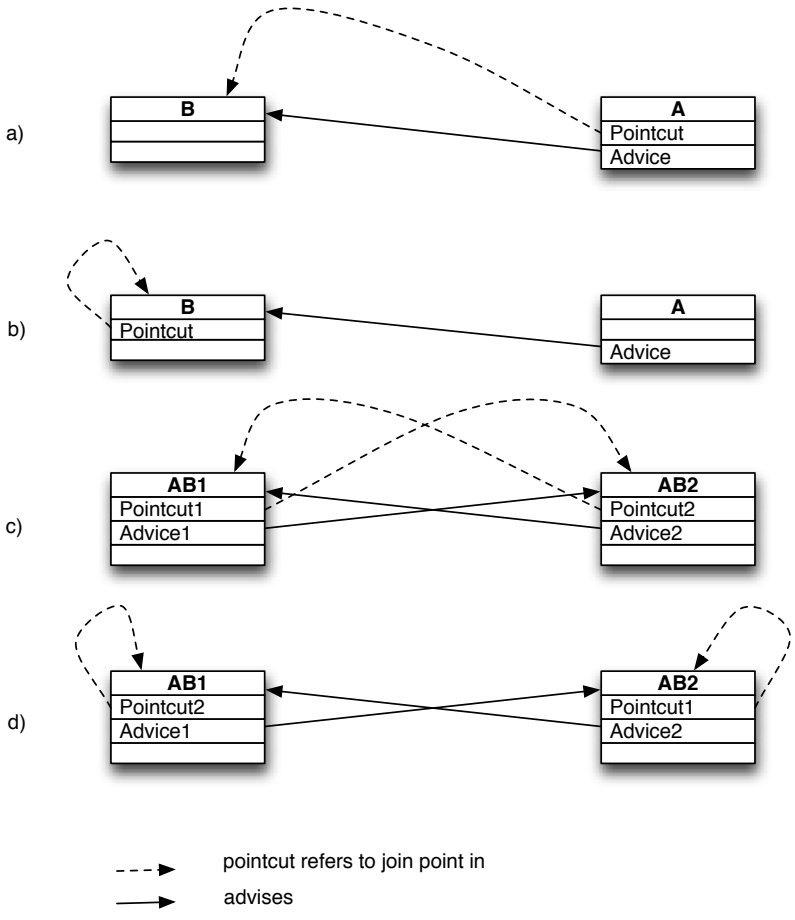


Figure 7.1 – Various localizations of pointcuts and advice.

**Expressiveness**

- As a class member, a pointcut can refer to other class members and therefore to dynamic values depending on its owner object. This makes it possible to program context-aware applications where the context can be flexibly captured in pointcuts and include the context of the observee, as usual, but also the context of the observer.
- As illustrated in Figure 7.1a, it is still possible to program à la AspectJ, maintaining both obliviousness and the asymmetry of advising, by defining base classes and aspect classes that encapsulate all the necessary pointcuts and their associated pieces of advice. But it is also possible to define the pointcuts in the base classes as illustrated in Figure. 7.1b, which makes compositions more robust to independent evolution as advocated by Aldrich [Ald05]. The trade-off between robustness and obliviousness is not imposed on the user but left as a choice depending on the needs of the application. Finally, it is possible to program in a symmetrical way, using what could be called *aspectual components*, which can both advise some other components and be advised by them as illustrated in Figure 7.1c and Figure 7.1d. Here again, this may assume obliviousness (this is already available in proposal merging classes and aspects, such as CaesarJ and Classpects).
- The merging of aspects and classes, together with the definition of pointcuts and advice as full-fledged class members, improves on the reusability of aspects by making it possible to use the standard rules of OOP.

The next section shows how the elements contributing to the integration of AOP and OOP have been made part of EJava.

## 7.3 EJava Integrating AOP and OOP

In EJava, there are three main features contributing to the integration of AOP and OOP. First, pointcuts and pieces of advice, implemented as events and event handlers, have been incorporated in the language as full-fledged instance members. Thus, aspect instances have access to more information about the context when selecting join points and can therefore be more selective. Furthermore, the different instances of the same "aspectual class" still share the same general behavior when selecting join points, but this behavior is parametrized by their particular dynamic context. Second, an EJava aspect is a plain class. It implements a crosscutting concern by defining proper events and handlers on available events of the base application (usually defined using method syntax). Third, as full-fledged instance members they can be redefined in subclasses. Redefinition of pieces of advice is possible because an event handler has an identity given by the event it handles.

This design reconciles the AOP world and the OOP world. Aspects are now standard classes, which can be freely instantiated and which interact with other objects by respecting the OO principles. But these objects are also special: they modularize crosscutting concerns in collaboration with the objects defined in their interfaces.

Let us now see how the design of EJava makes it possible to program aspects better aligned with OOP and how it enables the implementation of symmetric aspects.

### 7.3.1 Flexible Asymmetric Aspects

The integration of AOP and OOP of EJava enables the implementation of asymmetric aspects which are more flexible than the asymmetric aspects of languages such as AspectJ. With flexibility we refer to the possibility of freely instantiate them and the integration

of the OO features discussed in the previous sections. This section describes two examples that use this flexibility.

#### 7.3.1.1 Revisiting the Telecom Example

As described in Section 6.3.3, the standard distribution of ASPECTJ includes a simple Telecom application in its list of examples. A crosscutting concern dealing with the timing of the connections of customers is implemented in ASPECTJ in the `Timing` aspect. This implementation shows clearly the static nature of ASPECTJ aspects. The timing aspect is implemented in such a way that it calculates the connection time of all the customers of the application by quantifying on the static structure of the class `Connection`. The same is done for the billing aspect, which obtains a reference to the timing aspect by invocation of the static ASPECTJ construct `aspectOf`. The static nature of ASPECTJ does not make it possible to directly instantiate different billing and timing aspects for different groups of customers when needed. This could be needed for example if a business strategy requires to apply different billing schemes to different groups of customers.

EJAVA promotes aspect instances as objects, so that they interact with other objects in a standard OO fashion. An EJAVA aspect, as a plain class, can be instantiated and associated to a group of objects. The dynamic pointcuts, as events, make it possible to observe events related to these objects. This is observed in the EJAVA implementation of the timing aspect of Listing 6.11. Instead of operating on the static structure of the program, the aspect observes events associated to the group of customers passed as parameters in its constructor. In this sense, by being more aligned with OOP, EJAVA is less invasive than ASPECTJ, respecting encapsulation. The freedom of aspect instantiation is observed in the implementation of the timing aspect of Listing 6.12, which instantiates a timing aspect for the group of objects the billing is interested in. The timing aspect is part of the dynamic information of the billing aspect, and its use in the definition of the `billing` event highlights the benefits of dynamic pointcuts modeled as events.

#### 7.3.1.2 Pattern *Mediator* in EJAVA

Let us now see how EJAVA makes it possible to implement the pattern *Mediator* [GHJV94] in a truly modular way. We illustrate this by using an example taken from the presentation of CLASSPECTS [RS09]. We have chosen this example because it has already served to illustrate the benefits of the unification of classes and aspects. We illustrate the expressiveness of EJAVA to program such an example.

The example consists of a system integrating two types of components: sensors and cameras. A camera can be related to several sensors and a sensor to several cameras. When some sensor detects a signal each related camera takes a picture. Once the picture taken, the camera resets the sensor so that it can sense again.

```
class Sensor {
  void movementDetect() { ... }
  void temperatureDetect() { ... }
  void reset() { ... }
}
```

```
class Camera {
  void click() { ... }
}
```

Listing 7.5: Classes implementing sensors and cameras.

The CLASSPECTS solution, presented in [RS09], illustrates the convenience of using the

pattern *Mediator* [GHJV94]. Such a solution considers the classes `Sensor` and `Camera` of Listing 7.5, implementing the sensors and the cameras, respectively.

```
class Mediator {
  Sensor s; Camera c;

  Mediator(Sensor s, Camera c) {
    this.s = s; this.c = c;
    addObject(s); addObject(c);
  }

  after(): execution(void Sensor.movementDetect()) ||
           execution(void Sensor.temperatureDetect()): onDetected();
  after(): execution(void Camera.click()): onClicked();

  onDetected() { c.click(); }
  onClicked()  { s.reset(); }
}
```

Listing 7.6: Implementation of a mediator between a sensor and a camera in CLASSPECTS.

The class `Mediator` of Listing 7.6 modularizes the relationship between a camera and a sensor in CLASSPECTS. The class is instantiated for a sensor and a camera. It captures the execution of the methods `movementDetect` or `temperatureDetect` of the sensor and makes the camera take a picture. Analogously, it captures the executions of the method `click` of the camera and resets the sensor. The design using a mediator is an example of a programming style à la ASPECTJ, maintaining both obliviousness and the asymmetry of advising. The cameras and sensors can be seen as the base program, whereas the mediators can be seen as the aspect instances. The benefits of a unification of classes and aspects is made evident since these mediators can be flexibly instantiated to represent each individual relationship between several sensors and cameras.

```
class Mediator {
  Sensor s; Camera c;

  Mediator(Sensor s, Camera c) {
    this.s = s; this.c = c;
  }

  event onDetected() = after(s.movementDetect() || s.temperatureDetect());
  event onClicked()  = after(c.click());

  onDetect() => { c.click(); next; }
  onClick()  => { s.reset(); next; }
}
```

Listing 7.7: Implementation of a mediator between a sensor and a camera in EJAVA.

EJAVA shares with CLASSPECTS the unification of classes and aspects, which makes it possible to program flexible asymmetric aspects such as the mediator. Listing 7.7 shows the implementation of the mediator in EJAVA. Like the CLASSPECTS mediator, the EJAVA

mediator is instantiated for a sensor and a camera, it observes the events of the sensor in order to make the camera take a picture, and it observes the events of the camera in order to reset the sensor. However, EJAVA is more flexible than CLASSPECTS. First, it makes it possible to define finer-grained events. Second, it allows a truly modular design by defining events at the source side. The remainder describes these features.

**Fine-grained events** An event is defined in EJAVA by using a single construct. In CLASSPECTS, the definition of an event is made in two steps. First, an ASPECTJ-like pointcut describes the join points to be observed in a base program. Second, the `addObject` construct makes it possible to restrict the observation to a given set of objects. For example, an instance of the class `Mediator` of Listing 7.6 observes the execution of methods such as `movementDetect` and `click` restricted to a particular sensor and camera. The difference in the way events are defined in EJAVA and CLASSPECTS has an impact on the expressiveness of these languages to discriminate between events of different objects.

```
class Mediator {
  Sensor s1, s2; Long time;

  event onDetect1() = after(s1.movementDetect());
  event onDetect2() = after(s2.movementDetect());

  onDetect1() => {
    time = currentTime();
    next;
  }

  onDetect2() => {
    report(currentTime() - time);
    s1.reset();
    s2.reset();
    next;
  }
}
```

Listing 7.8: Double detection implemented in EJAVA.

In CLASSPECTS it is not easy to discriminate between the events of an object and the events of another when these objects are of the same type. Consider as an example the EJAVA code of Listing 7.8. It implements a class that calculates the time taken for a particle to go from one point to another, assuming that at each point there is a sensor. The class is instantiated with two sensors, one for each point. When the first sensor detects the particle the instance calculates an initial time. When the second sensor detects the particle the class calculates the final time. The class defines an event for the detection of each sensor and its corresponding event handler. This example cannot be implemented in CLASSPECTS in a direct way. The reason is that CLASSPECTS can only differentiate events in terms of the static structure of the program that generates the join point. Since in this example the join point is generated by the same method `movementDetected`, CLASSPECTS cannot define two different events.

**A design à la Open Modules.** In the example of the sensors and the cameras, an important feature of the presented design is the obliviousness of camera and sensor code regarding their mutual integration. Integration code, which has been shown to be a crosscuting concern, has been moved out of camera and sensor code. However, the presented solution is *fragile* because it does not properly reflect the relations between the base program and the mediator. The base program is not aware of possible violations of the assumptions made by the mediator. For example, the class `Mediator` of Listing 7.7 assumes that sensor detection is implemented by the methods `movementDetect` or `temperatureDetect`. A programmer can independently evolve the sensing code without being aware of a possible violation of this assumption.

```
class Sensor {  ...
  /* provided event */
  event onDetected() = after(movementDetect() || temperatureDetect());
}

class Camera {  ...
  /* provided event */
  event onClicked() = after(click());
}

class Mediator {
  Sensor s; Camera c;

  event onDetected() = s.onDetected();/* required event */
  event onClicked()  = c.onClick();/* required event */

  onDetected() => { c.click(); next; }
  onClicked()  => { s.reset(); next; }
}
```

Listing 7.9: Asymmetric solution à la Open Modules with events defined at the base-program side (`Sensor` and `Camera` classes).

As described in Section 2.2.5, complete obliviousness contradicts truly modularity. Aldrich [Ald05] has proposed the idea of Open Modules as a truly modular design by sacrificing complete obliviousness. The idea is that base program provides the pointcuts that the aspects will use. Thus, the semantics of these pointcuts is maintained by the base program along its evolution. These pointcuts belong now to the interface of the base program, permitting both parts to evolve independently while preserving this interface. Listing 7.9 shows how EJAVA is expressive enough to provide a solution à la Open Modules. Note how the `onDetected` and `onClicked` events are now provided by the `Sensor` and `Camera` classes, respectively. The `Mediator` object can observe these events in order to implement their handlers.

Definition of pointcuts at the base-program side is already statically possible in ASPECTJ. The novelty of EJAVA is the fact that events are full-fledged instance members contrasting with the static nature of the ASPECTJ pointcuts. Note how the `Mediator` object observes the `onDetected` event on the proper `Sensor` object defined as part of its interface. This would not be possible in ASPECTJ, which would need to quantify on all possible `Sensor` instances and include an additional conditional statement in the advice to

select the correct one.

### 7.3.2 Symmetric Aspects

Traditional AOP introduces aspects as asymmetric entities. Aspects do not belong to the same level as the classes implementing an application. Aspects see the application as a base program, which they observe and modify. The unification of aspects and classes enables a symmetrical style of programming in which a component can be provided with aspectual behavior. This symmetry makes no distinction between aspects and base program.

A simple example of this symmetry is the class `ConnectionFigure` of Listing 4.18. An instance of this class observes the changes in the figures that it connects. The `changed` event is equivalent to a pointcut. The instances of `ConnectionFigure` define a reaction to these events so that they can be seen as aspects. On the other hand, it is difficult to consider the class `ConnectionFigure` as an aspect and see the figures as a base program because the class itself is a subclass of `Figure`. Thus, `ConnectionFigure` plays both the role of an aspect and of a base program at the same time.

## 7.4 Related Work

AOP has been designed in such a way that an aspect instance has a global vision of all the join points occurring in the extent of its *deployment scope*. Several languages propose different strategies to define such an scope. For example, the scope of an AspectJ-like aspect is by default the whole application, so it applies its pointcuts to all the occurring join points. The *per-object* AspectJ deployment strategy refines the scope of an aspect instance, such that it applies its pointcuts only to the join points that directly occur in the context of the object it is deployed on. CaesarJ shares the default strategy adopted by AspectJ and also the *per-object* strategies. Additionally, it introduces *per-thread* deployment and dynamic deployment in a block of code. Classpects extends the *per-object* strategies by making it possible to the set the scope of an aspect instance to a group of objects instead of a single object. The authors of Classpects talk about *instance-level advising* because an instance can set and modify its group of objects at runtime. Tanter [Tan08] provides a study on the scope of aspects and proposes a flexible and general model of scoping strategies. However, if an aspect has several pointcuts, it is, in general, not possible to associate a particular scope to each individual pointcut of an aspect. In traditional AO approaches the same scope is shared by all the pointcuts of an aspect. EJava differs from other approaches in that each individual pointcut (event) is able to indicate the scope on which it operates, *i.e.* each pointcut observes the event occurrences associated to specific objects. For example, the scope of a pointcut defined with the event expression `some(figures).changed()` is the group of objects `figures`. It observes the occurrences associated to this group and selects the event `changed`. Some approximation could be achieved by coding and passing a specific *join-point filter* parameter in the scoping strategies of Tanter [Tan08]. However, this makes the definition of the relevant events less explicit, because pointcuts do not contain the complete information about the events to be selected and must be analyzed in combination with the deployment instructions of the aspect.

## 7.5 Conclusion

This chapter has discussed the benefits of knocking down the barriers that separate aspects from classes. The unification of aspects and classes reconciles AOP with the OO world because a design-level aspect implemented as a class respects the frontiers between objects. However, this approach is not expressive enough to implement complete obliviousness, which requires powerful quantification mechanisms, often working on the static structure of a program such as the ones of AspectJ. Instead, we replace complete obliviousness for true modularity based on the ideas of Open Modules [Ald05]. Furthermore, the distinction between aspects and base program can be dissolved.

So far we have discussed the integration of the three paradigms considered by this dissertation: OOP, EBP and AOP. The next chapter combines our programming model with advances OO techniques such as mixins. This combination will enable, between other applications, the definition of structural aspects.

# Chapter 8

# Mixins, Implicit Invocation and Stateful Behavior

So far we have presented a model of implicit invocation on top of an object-oriented language. EJava has been provided as an implementation of the model on top of Java. Since standard OO languages have exhibited problems with decomposing large applications and enabling transparent extensions, this chapter investigates the applicability of the model to a language with support for advanced decomposition features such as virtual classes and propagating mixin composition. As a motivation this chapter studies the implementation of state-dependent behavior using state machines. It shows how advanced decomposition features together with implicit invocation as provided by the model makes it possible to program extensible state-dependent event handling. In addition, the unification of event handling and advising makes it possible to implement extensible stateful aspects in a transparent way.

The chapter is structured as follows. Section 8.1 motivates the combination of advanced decomposition features and implicit invocation. Section 8.2 revisits propagating mixin composition and compares the implementation of extensible state machines in both Java and CaesarJ. Section 8.3 introduces ECaesarJ, an application of the model of implicit invocation of this dissertation to the CaesarJ language. It presents how ECaesarJ supports extensible state machines. The work on ECaesarJ has been conducted in the context of the European project AMPLE [AMP10, GNNM11]. Section 8.4 discusses related work. Finally, Section 8.5 concludes.

## 8.1 Motivation

In simple scenarios, modularization of functional requirements can be achieved by using the object-oriented paradigm. However, as argued in Section 2.3, functional requirements are often not aligned to the structure of software objects and operations. In these cases, advanced structural decomposition mechanisms like virtual classes make it possible to split definitions of objects and operations and distribute them into multiple modules.

Independently of the decomposition technique used, modularization of requirements also requires modularization of the dynamic structure of the program, i.e. the structure of its control flow. The conventional computational model is based on decomposing control flow into procedures (or functions), where each procedure modularizes a segment of a control flow. In such a structure a caller is modularized from the callee, but not the other way around. It is difficult to align the structure of control flow to requirements, because the direction of functional dependencies between requirements does not always correspond

to the direction of the control flow: the functionality of a requirement may need to be called within the control flow of the implementation of another requirement, whereas the latter (the caller) is logically independent of the former (the callee). In order to derive a modularization of the structure according to such dependencies, some form of inversion of control and implicit invocation is required.

We illustrate the convenience of a combination of advanced decomposition features and implicit invocation by implementing extensible state machines. The latter are usually implemented through the design pattern *State* [GHJV94]. Unfortunately, this pattern is verbose and does not properly support extension and composition of state machines and state-dependent behavior. Although a state can be made extensible by using factory methods and generics, as demonstrated by Chin and Millstein [CM08], such a solution introduces even more glue code, and does not support composition of extensions.

The contributions of this chapter are as follows. First, we illustrate the utility of advanced decomposition mechanisms and investigate to which extent some of the problems of the design pattern State can be alleviated in a language such as CaesarJ [AGMO06]. Second, we motivate the benefits of incorporating event support in CaesarJ. Whereas EJava extended Java with the implicit-invocation model of this dissertation, we introduce ECaesarJ, an extension of CaesarJ with the same model. As a result, virtual classes enable type-safe extension of classes that represent object states with handlers for new events. Propagating mixin-composition improves the composability of extensions. Third, we propose language support for (a) state machines, (b) their extension and composition, and (c) hierarchical states. The new language features provide a lot of flexibility for fine-grained decomposition of behavior: hierarchical states support hierarchical refinement of object behavior with respect to its life cycle, while inheritance enables refinement with respect to the object type.

## 8.2 Extensible State Machines

### 8.2.1 Example

This section presents an example involving state-dependent behavior and extensions thereof. The example is from the domain of building automation. Its purpose is to automate the control of various devices in a building. Specifically, we consider the need of implementing a *shutter control* encapsulating the interaction with a *shutter device*. We compare an implementation in Java and an implementation using virtual classes in CaesarJ.

A *base* shutter control encapsulates the basic interaction with the physical device. The shutter control can receive *aperture commands* in terms of percent of aperture and it translates them into low-level device commands. Two extensions are proposed on top of the base shutter control. A *stuck* shutter control informs the users of the control whether the physical device has got stuck. A *stop* shutter control is available whenever the physical device supports stop commands. It allows a user to send a *stop command* to stop the device and to override the behavior of the aperture command while a device is moving.

The behaviors of the base, stuck and stop shutter controls are described by the finite state machines of Figure 8.1, whereby the operator `=>` denotes the relationship between an event and the respective *action*. The behavior of the base shutter control (Figure 8.1a) introduces two possible states: `Idle` and `Moving`. When a shutter control is in the state `Idle`, it can receive a `setAperture` command, which initiates the movement of the physical device and makes the object transit into the state `Moving`. The shutter control returns into

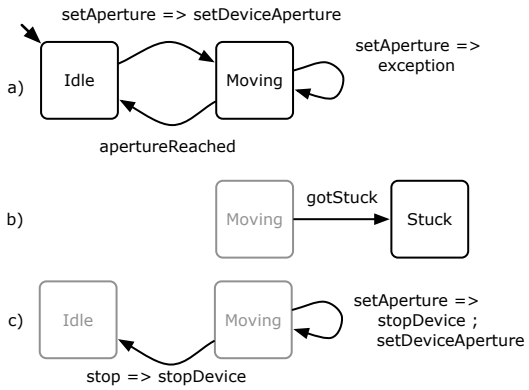


Figure 8.1 – Behaviors of a shutter control.

the state `Idle` once an event `apertureReached` is triggered by the physical device. When an aperture command is received in the state `Moving`, an exception is triggered indicating an unsuccessful command. The behavior of the stuck shutter control (Figure 8.1b) adds a new state `Stuck` and the corresponding transition from the state `Moving` triggered by an event `gotStuck`. Finally, the behavior of the stop shutter control (Figure 8.1c) *refines* the base behavior to permit the execution of an aperture command while the device is moving. The device is stopped before being forwarded the new aperture command. In addition, a stop command is provided, which makes it possible to stop the physical device and causes a transition from the state `Moving` into the state `Idle`.

### 8.2.2 Implementation in JAVA

Listing 8.1 shows the implementation of the base shutter control using the design pattern *State* [GHJV94] in JAVA. The class `BaseShutter` contains an inner interface `State`, which declares the messages that a shutter control object can receive. The two inner classes `Idle` and `Moving` represent the two possible states of the base shutter control by implementing `State`. The current state of the shutter control (variable `currentState`) is an object of type `State`. All state-dependent methods of the shutter control are forwarded to the current state. A transition to a particular state is achieved by instantiating the corresponding state class and assigning the instance to `currentState`. The state classes are not instantiated directly, but over the corresponding factory methods (lines 16 and 21) to enable the extension of the state-dependent behavior for subclasses of `BaseShutter`. The class `BaseShutter` is defined as a *listener* of the physical device. The interface `BaseDevListener` declares the method `apertureReached`, which is invoked by the device on the occurrence of the corresponding event.

Listing 8.2 shows the extension of `BaseShutter` with a *stop behavior*. The extension provides a new message to stop the device and permits sending a new aperture command to a moving device. In order to enable different handling of the stop message in different states, the state interface must be extended with a new method (line 18). The classes `Idle` (line 20) and `Moving` (line 25) must be extended accordingly. The new `Moving` class also overrides the implementation of `setAperture` to stop the device and send the new aperture to it. The factory methods are overridden to instantiate the new implementations of `Idle` and `Moving`. An implementation for the stuck shutter control can be provided in an analogous way.

```
1  interface BaseDevListener {
2    void apertureReached();
3  }
4
5  class BaseShutter
6        implements BaseDevListener {
7    State currentState = newIdle();
8
9    void setAperture(int ap) {
10     currentState.setAperture(ap);
11   }
12
13   void apertureReached() {
14     currentState.apertureReached();
15   }
16
17   void transit(State state) {
18     currentState = state;
19   };
20
21   Moving newMoving() {
22     return new Moving();
23   }
24
25   Idle newIdle() {
26     return new Idle();
27   }
28   interface State {
29     void setAperture(int ap);
30     void apertureReached();
31   }
32
33   class Idle implements State {
34     void setAperture(int ap) {
35       /* sends device aperture */
36       transit(newMoving());
37     }
38
39     void apertureReached() {}
40   }
41
42   class Moving implements State {
43     void setAperture(int ap) {
44       throw new Exception();
45     }
46
47     void apertureReached() {
48       transit(newIdle());
49     }
50   }
51   ...
52 }
```

Listing 8.1: Implementation of basic shutter control in JAVA.

```
1  class StopShutter extends
2        BaseShutter {
3
4    void stop() {
5      ((State)currentState).stop();
6    }
7
8    Moving newMoving() {
9      return new Moving();
10   }
11
12   Idle newIdle() {
13     return new Idle();
14   }
15
16   interface State extends
17             BaseShutter.State {
18     void stop();
19   }
20   class Idle extends BaseShutter.Idle
21             implements State {
22     void stop() {}
23   }
24
25   class Moving
26       extends BaseShutter.Moving
27       implements State {
28     void stop() {
29       /* stop the device */
30       transit(newIdle());
31     }
32     void setAperture(int ap) {
33       /* stop the device */
34       /* send aperture to device */
35     }
36   } ...
37 }
```

Listing 8.2: Implementation of stop behavior in JAVA.

As already shown in Section 2.3.1.1, the implementation of the pattern State in JAVA does not support type-safe extension with new events. There is no way to ensure that extensions of the *context class* are only used in combination with respective extensions of the *state class*.* The design of the extension presented in Listing 8.2 is not type-safe. For example, the type system does not prevent the programmer from assigning an instance of `BaseShutter.Moving` to `currentState` of an instance of `StopShutter`. As a result, a type cast is required at line 5 of Listing 8.2 to call the handler of the newly introduced event `stop`. This type cast would cause a runtime error if a wrong state object is set.

### 8.2.3 Implementation in CAESARJ

Chin and Millstein [CM08] show how to extend state machines with new events in a type-safe way using JAVA generics. A more concise and natural solution to these problems is enabled by virtual classes and propagating mixin composition as they are implemented in the language CAESARJ [AGMO06]. This section presents an implementation of the design pattern State using these language features and evaluate the resulting design.

The JAVA design of Section 8.2.2 implemented the different states of a class as its inner classes and used factory methods to make them extensible. In CAESARJ, state classes can be automatically made extensible by declaring them as virtual classes.

```
1  abstract cclass BaseDevListener {
2    abstract void apertureReached();
3  }
4
5  cclass BaseShutter extends
6         BaseDevListener {
7    State currentState = new Idle();
8
9    void setAperture(int ap) {
10     currState().setAperture(ap);
11   }
12
13   void apertureReached() {
14     currState().apertureReached();
15   }
16
17   void transit(State state) {
18     currentState = state;
19   }
20   abstract cclass State {
21     abstract void setAperture(int ap);
22     void apertureReached() {
23       transit(new Idle());
24     }
25   }
26
27   cclass Idle extends State {
28     void setAperture(int ap) {
29       /* send aperture to device */
30       transit(new Moving());
31     }
32   }
33
34   cclass Moving extends State {
35     void setAperture(int ap) {
36       throw new Exception();
37     }
38   } ...
39 }
```

Listing 8.3: Implementation of the basic shutter control in CAESARJ.

Listing 8.3 shows the implementation of the base shutter control in CAESARJ. It looks very similar to the implementation in JAVA. However, the keyword `cclass` declares the classes that have the special CAESARJ semantics†, i.e., the inner classes implementing

∗. We use here the terminology of [GHJV94]. The term *context classes* refers to the classes subject to state-dependent behavior, `BaseShutter` and its subclasses in our example. The term *state classes* refers to the classes implementing the various logical states, the classes implementing the interface `State` in our example.

†. The classes declared with the `class` keyword in CAESARJ preserve the standard JAVA semantics.

states and the state interface are virtual classes. The state interface is declared as an abstract virtual class.‡ This way, *default handlers* are provided to their events.

```
1  cclass StopShutter
2      extends BaseShutter {
3
4    void stop() {
5      currentState.stop();
6    }
7
8    abstract cclass State {
9      void stop() { }
10   }
11   cclass Moving {
12     void stop() {
13       /* stop the device */
14       transit(new Idle());
15     }
16
17     void setAperture(int ap) {
18       /* stop the device */
19       /* send aperture to device */
20     }
21   } ...
22 }
```

Listing 8.4: Implementation of stop behavior in CaesarJ.

```
1  abstract cclass StuckDevListener {
2    void gotStuck();
3  }
4
5  cclass StuckShutter extends
6         BaseShutter &&
7         StuckDevListener {
8
9    void gotStuck() {
10     currentState.gotStuck();
11   }
12   abstract cclass State {
13     void gotStuck() { }
14   }
15
16   cclass Moving {
17     void gotStuck() {
18       transit(new Stuck());
19     }
20   }
21
22   cclass Stuck extends State { }
23   ...
24 }
```

Listing 8.5: Implementation of stuck behavior in CaesarJ.

Since the interface and implementations of states are declared as virtual classes they can be refined in the subclasses of the context class. For example, Listing 8.4 shows the implementation of the extension with the stop behavior. The state interface is extended to include the `stop` event by refining the virtual abstract class `State`. The virtual class `Moving` is also refined in order to implement a handler for the new event and to override the inherited handling of `setAperture`. Implementation of the extension with the stuck behavior is analogous and shown in Listing 8.5.

As explained in Section 2.3, virtual classes implicitly inherit from their predecessor versions. The superclasses are implicitly inherited too, but rebound to their new implementations. Class instantiation expressions are also rebound to the refined versions of virtual classes, which makes explicit factory methods redundant. In `BaseShutter` the state classes are instantiated directly (see lines 15 and 24 of Listing 8.3), but instantiation of virtual classes is late-bound and determined by the dynamic type of the family object.§

‡. Actually, here is no special syntax for virtual interfaces in CaesarJ. However, this is not a limitation compared to Java because CaesarJ supports a form of multiple inheritance for classes.

§. In the instantiation expressions of Listing 8.3 the family object is determined implicitly, e.g. `new`

Virtual classes are also automatically rebound in type references, which enables a type-safe access to the methods introduced in extensions.

### 8.2.4 Comparison

The design of an extensible design pattern State with virtual classes and propagating mixin composition improves on the Java solution. The type system of CaesarJ enables type-safe extension of a state machine with handlers for new events. Moreover, type-safety is achieved without any sophisticated type declarations or any other code overhead. Since Java does not support multiple inheritance, the Java extensions of the base shutter control cannot be composed. For example, the stop behavior and the stuck behavior are independent extensions of the base shutter behavior. A shutter control that supports both stop commands and reports the stuck condition would need both of them.¶ Extensions of a state machine are composable in CaesarJ. The stop and the stuck extensions of the base shutter control can be composed using propagating mixin composition. This is done by simply declaring a class `CompleteShutter` which inherits from both `StuckShutter` and `StopShutter`:

```
cclass CompleteShutter extends StuckShutter & StopShutter { }
```

The amount of required glue code is also reduced. Virtual classes support late-bound instantiation directly making the factory methods of the Java solution redundant. Implicit inheritance of virtual classes and their superclasses avoids most of the explicit inheritance declarations required in Java; compare, e.g., the definitions of `StopShutter.Moving` in Listing 8.2 and in Listing 8.4. Finally, when introducing a new event with a default handler, only the classes implementing specific handling of the event have to be overridden. For example, Listing 8.4 and Listing 8.5 did not redefine the class `Idle`, because it was automatically updated to inherit from the refined version of `State`, which implements default handlers for new events.

## 8.3 The ECaesarJ Language

ECaesarJ [GNNM11] is a new language that extends CaesarJ with the model of implicit invocation developed in this dissertation. While we have shown the benefits of EJava by extending Java with such a model, the same benefits are transferred to ECaesarJ. The novelty of ECaesarJ, regarding EJava, is the support for virtual classes and propagating mixin composition. ECaesarJ inherits from EJava the integration of EBP, AOP and OOP. It completes such an integration by replacing the inter-type declarations of AspectJ by a more general mechanism: mixin composition. In addition, ECaesarJ includes special language support for state machines as presented in Section 8.3.4.

### 8.3.1 Principles

ECaesarJ eliminates the AO constructs of CaesarJ, pointcuts and pieces of advice, for the benefit of more regular and orthogonal class members: events and event handlers. ECaesarJ provides a unification of event handling and method execution as in EJava,

`Moving()` is equivalent to `BaseShutter.this.new Moving()`, where `BaseShutter.this` is Java syntax to refer to the enclosing object of an inner class.

¶. Because of the limitation of single inheritance, we could not provide either default implementations of states with default event handlers, which, for example, ignore the events.

so that methods are provided just as syntactic sugar. Thus, the only class members of ECAESARJ are fields, events and event handlers. The mixin-composition semantics of ECAESARJ is the semantics provided by CAESARJ. Indeed, the linearization algorithm of CAESARJ just organizes the hierarchy of classes and performs the necessary type rebinding. It does not depend on the kind of class members of the language. Since ECAESARJ is an extension of CAESARJ, many scenarios can be implemented as in CAESARJ. The benefit of ECAESARJ is its support for events, or in other words the combination of mixins and implicit invocation. This combination offers additional flexibility and expressiveness. Whereas mixins decouple the structure of complex applications, implicit invocation decouples their behavior. We have already observed this combination in Section 2.2. Structural aspects (inter-type declarations in ASPECTJ) are in general used to complement behavioral aspects (pointcut-advice model). ECAESARJ provides structural aspects as in CAESARJ and behavioral aspects as in EJAVA.

The remainder shows the combination of mixins and implicit invocation in action by implementing extensible state machines in ECAESARJ.

### 8.3.2 Pattern-based Implementation of State Machines

The CAESARJ implementations presented in Section 8.2.3 of the basic shutter control and its extensions are valid ECAESARJ implementations. The syntax used there is a subset of the ECAESARJ syntax. These examples use a method syntax, which in ECAESARJ, as in EJAVA, introduces imperative events and event handlers. The compiler is in charge of de-sugaring the code into the corresponding version using only events and event handlers.

```
1  cclass BaseShutter extends
2        StateMachine {
3
4    event setAperture(int ap);
5
6    event apertureReached() =
7          device.apertureReached();
8
9    void transit(State s1, State s2) {
10     undeploy(s1);
11     deploy(s2);
12   }
13   abstract cclass State {
14     abstract setAperture(int ap) =>;
15
16     apertureReached() => {
17       transit(this, new Idle());
18     }
19   }
20
21   cclass Idle extends State {
22     setAperture(int ap) => {
23       /* send aperture to device */
24       transit(this, new Moving());
25     }
26   }
27
28   cclass Moving extends State {
29     setAperture(int ap) => {
30       throw new Exception();
31     }
32   } ...
33 }
```

Listing 8.6: Implementation of base shutter control in ECAESARJ.

The unification of event handling and method execution provides additional programming alternatives. Listing 8.6 illustrates an alternative implementation of the base shutter using an implicit invocation scheme. In this implementation, the events `setAperture` are

explicitly defined with the `event` construct. The event `setAperture` is defined as imperative. The event `apertureReached` is defined as a declarative event representing an event `apertureReached` triggered by the physical device, avoiding in this way an implementation using listeners (the class `BaseDevListener` is not needed anymore). Since the events `setAperture` and `apertureReached` are in the scope of the state classes, these classes can handle them directly, avoiding the use of forwarding methods. As a whole, this implementation works by keeping a single instance of a state class deployed: the method `transit` undeploys the current state and deploys the new one. The reference to the current state is not needed anymore.

The solution of Listing 8.6 eliminates part of the glue code required by the previous solutions when implementing the shutter control. It eliminates the forwarding calls, eliminates the reference to the current state, and uses the notion of event to implement the interaction of the shutter control and the physical device. Let us now carry the example a little further with an implementation of *hierarchical state machines*.

### 8.3.3 Pattern-based Implementation of Hierarchical State Machines

Hierarchical state machines were first introduced in the Statecharts visual formalism [Har87] and later on included in the Unified Modeling Language (UML) notation with the name of UML statecharts. Hierarchical state machines are important because they provide a compact and structured way to express complex behavior. They define states which can be decomposed into further states, thus making it possible to reuse commonalities of multiple states by declaring them as substates of a state with an associated common behavior. They also enable decomposition of states in the extensions of a state machine.

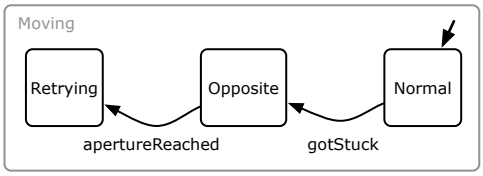


Figure 8.2 – Moving in a corrective stuck behavior.

Figure 8.2 shows an example of a hierarchical state. The figure extends the basic stuck behavior with corrective functionality trying to free the shutter by moving it into the opposite direction and then back. The state `Moving` is decomposed into further states to differentiate between moving as part of the normal device operation, represented by the substate `Normal`, and moving for the purpose of trying to free the shutter from the stuck condition, represented by the states `Opposite` and `Retrying`. Transitions toward the state `Moving` are translated into transitions toward the substate `Normal`, which has been marked as an initial substate using a small arrow. When a stuck event is received in the state `Normal`, the object will try to set a movement in the opposite direction in order to free the shutter by sending the physical device the corresponding aperture command. It will transit into the state `Opposite` and wait for an event `apertureReached` representing the success of the opposite movement. When this event is received, the shutter control will repeat the original aperture command transiting into the state `Retrying`. All the transitions defined for the state `Moving` are inherited by its substates, including the transitions of `Moving` defined in the basic stuck control. For example, when the shutter control is in the state

Normal, the reception of an event apertureReached will trigger a transition into the state Idle. Since gotStuck is not overridden for the states Opposite and Retrying, they will inherit the default handler of this event, handler that simply transits into the state Stuck. Hence, there will not be a repeated attempt to solve the problem. Something less obvious happens when in the state Normal an event gotStuck is received. In this case, the more specific transition (the one of the substate) overrides the transition defined in the super state, so that the shutter control transits into the state Opposite.

```
1  cclass UnstuckShutter
2         extends StuckShutter {
3
4    void transit(State state) {
5      super.transit(state.initial());
6    }
7
8    abstract cclass State {
9      State initial() { return this; }
10   }
11
12   cclass Moving {
13     State initial() {
14       return new Normal().initial();
15     }
16   }
17   cclass Normal extends Moving {
18     gotStuck() => {
19       /* send opposite aperture */
20       transit(new Opposite());
21     }
22   }
23
24   cclass Opposite extends Moving {
25     apertureReached() => {
26       /* send previous aperture */
27       transit(new Retrying());
28     }
29   }
30
31   cclass Retrying extends Moving { }
32   ...
33 }
```

Listing 8.7: Implementation of base behavior in CAESARJ with mixins.

Listing 8.7 shows the ECAESARJ implementation of the behavior of Figure 8.2. It introduces the family class UnstuckShutter as a subclass of the family class StuckShutter with Idle and Moving as its inner classes. The substates of the state Moving are implemented as subclasses of the class Moving, thus inheriting all the event handlers of Moving and overriding the needed ones (Figure 8.3 illustrates the relationships between these classes). For example, the class Normal overrides the handler gotStuck by sending an aperture to the physical device and transiting into Opposite. Finally, the method transit of the family class has been modified in order to implement the transitions between hierarchical states. When this method is called with a given state it asks the state for its initial substate so that it makes the machine transit into the corresponding substate. If the state does not have substates it responds with this.

The example of hierarchical state machines illustrates the inheritance of event handlers in ECAESARJ. The event handlers of the state classes have an identity given by the event they handle. This makes it possible to override them in subclasses. For example the event handler apertureReached of class Opposite overrides the event handler apertureReached of class Moving.

Plain state machines or hierarchical ones are widely used nowadays. This makes us consider as a good option to provide explicit support for them in mainstream languages. A big advantage is that dedicated language constructs can enforce a correct use of the state machines while this is harder to verify with pattern-based implementations. For illustration, in the Figure 8.1, there is a transition from Moving to itself when setAperture(int) is received. Now, consider the implementation of the stop shutter control shown in Listing 8.8.

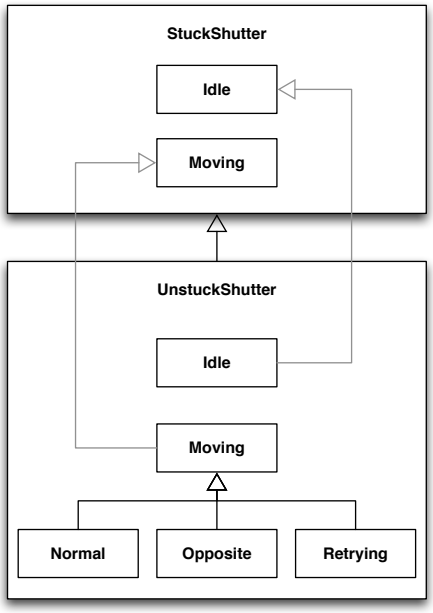


Figure 8.3 – Moving in a corrective stuck behavior.

```
cclass StopShutterAlt extends BaseShutter {
  cclass Moving {
    setAperture(int ap) => {
      stop();
      setAperture(ap);
    } ...
  } ...
}
```

Listing 8.8: Erroneous Implementation of Stop Behavior.

Following our previous considered semantics, this code is not correct because the instruction `stop()` introduces a transition into the state `Idle`. However, there is no way to statically detect this error. The next section introduces language support for stateful behavior in ECAESARJ.

### 8.3.4 Language Support for Stateful Behavior

A state machine can be seen as a set of atomic transitions of the form $(S_1, e \texttt{ => } a, S_2)$, which tells that in state $S_1$ an event $e$ causes the execution of the action $a$ and a transition of the machine into state $S_2$. The whole transition is atomic. This means that during the execution of the action $a$ the machine is in neither of the states, and the object cannot handle other events within the control flow of the action. Thus, an important semantics property of our state machines is that event handlers are considered to be non-reentrant.

The ECAESARJ state machines presented in the previous section provide a good structure on top of which to design dedicated language support for state machines ensuring a correct semantics. The language support is provided by using the keywords `smcclass`, `state` and `initial` introducing a state machine, a state of the state machine and an annotation of the initial state, respectively. An `smcclass` defines a context class and all the necessary infrastructure to implement hierarchical state machines. All the rules associated to standard classes are valid for an `smcclass`. As a standard class it can include any kind of members, can be instantiated and is subject to inheritance and mixin composition. Differently to a standard class, an `smcclass` includes `state` constructs. A `state` defines an inner state class. It also behaves as a standard class, however, it cannot be explicitly instantiated. Its instantiation is governed by the language. The event handlers inside a `state` are equipped with statement of the form `=>` *stateName*. These statements represent transitions into the given states.

The use of the new state-machine constructs impose additional semantic constraints, most of them verified at compile time, except reentrancy.

- All the events defined inside the `smcclass` are considered as the alphabet of the state machine.
- For the sake of determinism, every `state` has to provide a transition for each event in the alphabet.
- The execution of a state has to end with a transition.
- There is no reentrance.

Non-reentrance cannot be easily enforced statically. If at all feasible, it would at least require a complicated non-modular static analysis of the control flow of the program. Therefore, the language constructs impose dynamic checks that detect the reentrant cases and generate runtime errors.

The new state-machine constructs generate at compile-time an implementation analogous to the one seen in the previous section. The `smcclass` and `state` constructs are de-sugared in classes equipped with some additional infrastructure and the annotation `initial` is de-sugared in a method that sets the initial state.

Listing 8.9 shows the implementation of the base shutter control with the new constructs. When compared to Listing 8.3, most of the glue code required when implementing state machines in CAESARJ is avoided. Line 3 and 5 define the alphabet of the state machine. Every state has to provide handlers for these events. When an event is always handled in the same way, a class can declare default event handlers by defining branches valid for any state, denoted by the use of the anonymous state name `_`. For example, lines 9 defines

```
1  smcclass BaseShutter {
2
3    event setAperture(int ap);
4
5    event apertureReached() =
6          device.apertureReached();
7
8    state _ {
9      apertureReached() => Idle;
10   }
11   initial state Idle {
12     setAperture(int ap) => {
13       /* send aperture to device */
14       => Moving;
15     }
16   }
17
18   state Moving {
19     setAperture(int ap) => {
20       throw new Exception() => Moving;
21     }
22   } ...
23 }
```

Listing 8.9: Base shutter control with state machine constructs.

a default handler for the event `apertureReached`. If a state does not declare a specific handler for an event, the default handler is used. If a default handler is not available for an event and the class is not declared as abstract, then all the states of the class must provide handlers for the event. In this way it is statically ensured that all the events from the alphabet of a class are handled in all the states of the class. Line 11 defines `Idle` as the initial state. Line 14 shows how the transitions are defined by using the operator `=>`. The syntax used in the line 9 is a shortcut for an event handler whose body has just a transition.

```
1  smcclass StuckShutter
2          extends BaseShutter {
3    event gotStuck() =
4          device.gotStuck();
5
6    state _ {
7      gotStuck => _;
8    }
9    state Moving {
10     gotStuck() => Stuck;
11   }
12
13   state Stuck { }
14   ...
15 }
```

Listing 8.10: Stuck shutter control with state machine constructs.

Listing 8.10 shows the extension with stuck behavior. It illustrates how the `machine` and `state` constructs behave as standard classes and hence are subject to inheritance and mixin composition. Note how in a default handler, the anonymous state name can also be used to name a target state in order to indicate that the state should not change (line 7).

Listing 8.11 shows how hierarchical state machines are supported. Since the `state` construct acts as a standard class, substates are defined using the `extends` keyword.

### 8.3.5 Stateful Aspects

The support for events of the programming model of this dissertation combined with the stateful definition of a class makes it possible to implement stateful aspects (see Section 2.2.4.1). In particular, the dedicated support for stateful behavior of ECAESARJ makes this task easy.

As an example, Listing 8.12 shows a class `Monitor`, which each time that a user is added into the system explicitly instantiates an aspect `UserAsp` for the given user. The aspect

```
smcclass UnstuckShutter extends
         StuckShutter {

  initial state Normal extends
                Moving {
    gotStuck() => {
      /* send opposite aperture */
      => Opposite;
    }
  }
  state Opposite extends Moving {
    apertureReached() => {
      /* send previous aperture */
      => Retrying;
    }
  }

  state Retrying extends Moving {}
  ...
}
```

Listing 8.11: Unstuck shutter control with state machine constructs.

```
cclass Monitor {
  List aspects;

  event added(User user) =
        after(system.add(user));
  added(User u) => {
    aspects.add(new UserAsp(u));
    next;
  }
}

smcclass UserAsp {
  User user;
  DataBase db;

  event login() = user.login();
  event logout() = user.logout();
  event query(Query q) = db.query(q);

  state _ {
    login()  => { next; => In }
    logout() => { next; => Out }
  }
  initial state Out {
    query(Query q) => {
      next; => Out
    }
  }

  state In {
    query(Query q) => {
      log(q); next; => In
    }
  }
}
```

Listing 8.12: Stateful aspect in ECAESARJ.

is intended to log the queries of the user. It is implemented with the `smcclass` construct because its behavior is stateful. It defines an event `login` capturing the login of the user. A login makes the aspect transit into a state `In`. In such a state, the aspect is interested in two additional events. The event `logout` represents a logout of the user and the event `query` represents a query of the user in a database. On `logout` the aspect comes back into state `Out`. On `query`, the aspect logs the query. This aspect is an example of a stateful aspect because its observation depends on its internal state. The events `logout` and `query` are only taken into account when the user is logged in.

## 8.4 Related Work

**Patterns** Sane and Campbell [SC95] consider the incremental construction of state machines. Their premise is that it is not possible to simply model states as objects and derive new machines by inheritance because states cannot be inherited in isolation. This problem is solved by relying on family polymorphism.

Chin and Millstein [CM08] propose an extensible State design pattern, which uses JAVA generics to solve the problem of type-safe extension of a state machine with new events. While the proposed design solves the typing problem, it requires much more overhead than the pattern implementation with virtual classes, e.g. it requires two classes for each state machine - one class to support its extension and another for its instantiation. The pattern also does not address the problem of composition of extensions.

**Behavior Substitutability** There is a whole line of work (see for instance [BvdA01]) concerned with the meaning of inheritance, or more generally with substitutability with respect to behavior. For instance, linear substitutability requires that every trace in a superclass is a trace of its subclasses. This requires when extending a state machine to only add some behavior. As standard class inheritance does not provide any guarantee in this respect, we have chosen a very practical model whereby state machines can be extended with the same freedom as standard classes. In any case, the presence of explicit state machines may also help in case more control on the inheritance of behavior is required.

**Virtual Classes and propagating mixin composition** The concept of a virtual class stems from the programming language Beta [MMP89] and was further developed in gbeta [Ern99a], which introduced propagating mixin composition [Ern99b] and a type-safe family polymorphism [Ern01]. CAESARJ [AGMO06] integrates these concepts to JAVA, which has different method overriding semantics than Beta, and supports abstract methods and classes.

**Concurrent Object-Oriented Languages** Logical states are, at least implicitly, present in many concurrent object-oriented languages, where they implement *behavioral synchronization*, i.e. the need to delay the acceptance of a request until a service is available [BGL98]. Two proposals are explicitly related to our work. Both integrate explicit state machines, although in a different form, to promote a better separation of communication as an inter-object concern from the internal activities of the objects. These state machines implement behavioral synchronization.

PROCOL [vdBL89, vdBL91] is based on delegation. PROCOL objects interact through synchronous send and receive primitives. A *protocol* controls object access and completely

serialize interaction by specifying which are, depending on the logical state of the object, the message receipts that should be handled and the ones that should be delayed. Simple protocols are expressed using guarded regular expressions. These protocols can be composed using parallel composition. The definition of a primitive expression makes it possible to match message receipts against the type of the sender and also recognizes specific senders (for instance the creator of the object). So-called *constraints* can be used as a weak form of aspects, included unlike aspects within the object to which they apply, to specify additional code to be executed in case of specific message sends.

ACT++ is a hybrid of an actor language and C++ and combines the notion of behavior replacement typical of actor systems with inheritance (whereas actor systems usually rely on delegation) [KL89]. Instead of denoting an actor script, an ACT++ *behavior* denotes a set of methods that terminate by making the object obey to a new *behavior* using the standard `become` primitive of actor languages. Subclasses can be extended with new behaviors whereas previously defined behaviors can be redefined. These behaviors correspond quite forwardly to our logical states.

Our work generalizes these ideas, for the time being in a sequential setting, with the possibility of composing state machine extensions and generalized events, while resorting to a limited number of concepts at the implementation level.

**Hierarchical States** Harel's Statecharts [Har87] propose a graphical notation for hierarchical state machines, which are very expressive, but also very complex. Indeed, several authors such as [US94] have tried to give a proper semantics to Statecharts. Hierarchical states of ECAESARJ have taken many ideas from Statecharts, but they are simpler. One important difference in ECAESARJ is the overriding semantics for conflicting event handling defined in a substate and its enclosing state. This situation is considered as an ambiguous state machine in Statecharts. We have shown the utility of this feature. In addition, Statechart is a graphical notation, whereas we provide a programming language with extensible states.

## 8.5 Conclusion

This chapter has presented ECAESARJ, a language that combines advanced OO features such as propagating mixin composition with our model of implicit invocation. We have shown how the implementation of extensible state machines can be achieved in ECAESARJ without much glue code compared to a solution in JAVA or CAESARJ. Whereas virtual classes and mixin composition make the state machine extensible, events and event handlers reduce a big part of the glue code needed in a language such as JAVA. In addition, we have introduced dedicated language support for state machines, which is not so far from an ECAESARJ implementation using just classes, event and handlers. The main benefit of this language support is that it reduces a part of the remaining glue code and enforces a state machine semantics.

This chapter has shown how our model can be beneficial to the implementation of complex patterns. The next chapter confirms these benefits by showing how a smart home application can be implemented using ECAESARJ.

# Part III

# Validation and Conclusion

# Chapter 9

# Case Study

After having considered small illustrative examples in the previous chapters, this chapter validates our programming model and our prototypes on two case studies. The first case study is a minimal version of the drawing editor JHotDraw [Eri11]. The second one is an industrial-strength case study: the Smart Home case study [GNNM11]. Section 9.1 presents the principal elements of the implementation of the drawing editor in EJava. Section 9.2 illustrates the expressiveness of ECaesarJ when implementing the Smart Home case study. Finally, Section 9.3 concludes.

## 9.1 Mini JHotDraw

We have previously illustrated several features of our model by using examples inspired by a drawing editor. It is natural that our first case study consists of evaluating how EJava can actually facilitate the implementation of an application such as JHotDraw.

We have taken the current version of JHotDraw and we have eliminated part of its code in order to keep the core functionality. Thus, we can measure how much of this core functionality implements implicit invocation. We have commented some methods and their calls and eliminated some classes, being careful to maintain the original JHotDraw architecture. The result is a minimal version of JHotDraw that makes it possible to display text area figures, to select them with the mouse, to move them by dragging them into a different position or by using the keyboard, to resize them, to change text and the size of its font, and to delete the figures. As a whole the program consists of around 7K lines of code, distributed in around 70 classes.

A first evaluation studies how language support for events reduces the amount of lines of code of the application. A second evaluation studies how the crosscutting of event triggering can be reduced thanks to the unification of event handling and method execution. Finally, we have studied how some design intentions can be better expressed by using the EJava constructs.

### 9.1.1 Listeners versus Events

In JHotDraw, implicit invocation is programmed by using Java listeners. In our minimal JHotDraw application the methods declared by the interfaces `FigureListener`, `FigureSelectionListener` and `ToolListener` define the events of the application. We encountered events such as `areaInvalidated`, `figureChanged` and `toolStarted`, which are listed in the first column of Table 9.1. The last column of the table shows the classes that provide the necessary registration infrastructure. The last row of the table shows the amount of lines of code used by the declaration of listeners and the registration infrastructure. Table 9.2 lists the classes where the events are triggered and the classes where these events are handled. The table also shows the amount of lines of code. If we sum up the

| Event Name | Interface | Registration Infrastructure |
|---|---|---|
| areaInvalidated<br>attributeChanged<br>figureChanged | FigureListener | AbstractFigure |
| selectionChanged | FigureSelectionListener | DefaultDrawingView |
| toolStarted<br>toolDone<br>areaInvalidated<br>boundsInvalidated | ToolListener | AbstractTool |
| **Lines of code** | 19 | 121 |

Table 9.1 – Events of our minimal JHotDraw.

| Event Name | Triggering | Handling |
|---|---|---|
| areaInvalidated<br>figureChanged<br>attributeChanged | AbstractFigure<br>AbstractCompositeFigure<br>AbstractAttributtedFigure<br>AbstractAttributtedCompositeFigure | AbstractCompositeFigure<br>DefaultDrawingView<br>AbstractHandle<br>FloatingTextArea |
| selectionChanged | DefaultDrawingView | AbstractSelectedAction |
| toolStarted<br>toolDone<br>areaInvalidated<br>boundsInvalidated | AbstractTool<br>SelectionTool<br>DefaultDragTracker<br>TextAreaEditingTool<br>DefaultHandleTracker<br>DefaultSelectAreaTracker | DefaultDrawingEditor<br>SelectionTool |
| **Lines of code** | 39 | 120 |

Table 9.2 – Triggering and handling of the events of our minimal JHotDraw.

total amount of lines of code provided in both tables and we add the amount of lines of code needed to define subclasses of the listeners, to instantiate these subclasses, to call the registration infrastructure and to capture user input, we obtain around 500 lines of code. This number measures the amount of effort needed to connect the application functionality with the user input.

We have replaced the use of listeners by our events and event handlers. We have eliminated the 3 listeners, the registration code and some calls to the registration infrastructure. As a result, we eliminated around 300 lines of code, *i.e.* around 60% of the effort required when using listeners. We run the editor and the performance of the application was not visibly affected by the use of EJAVA.

This experience demonstrates the capacity of EJAVA to program traditional EB applications. The result would be analogous when using the imperative events of a language such as C#. The next section shows additional refactoring that highlights the benefits of declarative events and the unification of event handling and method execution.

### 9.1.2 Eliminating Crosscuting Concerns

The figure-change protocol of JHotDraw works in terms of the methods `willChange` and `changed` of the class `AbstractFigure` (see Listing 9.1). The method `willChange` of a figure is called each time the figure is about to be modified. It triggers the event `areaInvalidated` and invalidates the drawing area of the figure. The method `changed` is called each time the figure has been modified. It validates the new drawing area of the figure and triggers the

```
abstract class AbstractFigure implements Figure {
  ...
  public void willChange() {
    if (changingDepth == 0) {
      fireAreaInvalidated();
      invalidate();
    }
    changingDepth++;
  }

  public void changed() {
    if (changingDepth == 1) {
      validate();
      fireFigureChanged(getDrawingArea());
    }
    else if (changingDepth < 0)
      throw new InternalError();
    changingDepth--;
  }
}
```

Listing 9.1: Figure change protocol.

event `figureChanged`. These events are used by a drawing editor to clear an invalidated area and to repaint a validated one.

For example, the class `TextAreaEditingTool` has the following piece of code in its method `endEdit`:

```
textHolderFigure.willChange();
textHolderFigure.setText(newText);
textHolderFigure.changed();
```

This code calls `willChange` before changing the text of a text-holder figure and calls `changed` after the figure has been modified.

```
abstract class AbstractFigure implements Figure {
  ...
  event change() = transform(AffineTransform) ||
                   setBounds(Point2D.Double, Point2D.Double);

  void change() => {
    willChange();
    next;
    changed();
  }

  public void willChange() { ... }
  public void changed() { ... }
}
```

Listing 9.2: Figure change protocol.

The methods `willChange` and `changed` are called within 10 different methods in the

code of our minimal JHotDraw application, distributed among 8 different classes. It constitutes a crosscutting concern, also introducing tangling in the corresponding methods. In order to evaluate whether EJAVA can improve the situation, we have defined a new event `change` in the class `AbstractFigure` capturing the calls to the methods that modify a figure. At the same time we have implemented an event handler that calls the methods `willChange` and `changed` as illustrated in Listing 9.2. This refactoring replaced the 10 invocations of the methods `willChange` and `changed` by a single invocation in the handler `change` of `AbstractFigure`. We needed, however, the definition of the new event `change` in `AbstractFigure` and the refinement of this event in the classes `AbstractCompositeFigure` and `TextAreaFigure`. In terms of lines of code we just reduced around 8 lines of code, however we improved the modularity of the application as now only 3 classes are aware of the figure-change protocol and the definition is not tangled in any method.

### 9.1.3 Improving Expressiveness

A third study consisted of figuring out how to improve the design of JHotDraw by taking advantage of the expressiveness of EJAVA. Again we centered our attention on the figure-change protocol. If we look at Listing 9.1 the actual objective of the methods `willChange` and `change` is to invalidate and validate the drawing area of the figure. The notification of the events `areaInvalidated` and `figureChanged` are a result of these invalidation and validation. Indeed, by looking at the use of `figureChanged` in the application, we could perfectly rename it as `areaValidated`.

```
1 abstract class AbstractFigure implements Figure {
2   ...
3   event change() = transform(AffineTransform) ||
4                    setBounds(Point2D.Double, Point2D.Double);
5
6   change() when !isChanging => {
7     invalidate();
8     isChanging = true;
9     next;
10    isChanging = false;
11    validate();
12  }
13
14  event areaInvalidated(Figure source, Rectangle2D.Double area) =
15        after(invalidate() with source = this, area = getDrawingArea());
16
17  event figureChanged(Figure source, Rectangle2D.Double area) =
18        after(validate()) with source = this, area = getDrawingArea();
19
20 }
```

Listing 9.3: Figure change protocol.

Listing 9.3 shows the result of our third experience. We have eliminated the methods `willChange` and `changed` and placed some of their code in the event handler `change`. Specifically we call `invalidate` before continuing with the change (call to `next`) and `validate` afterwards. Then we have defined the events `areaInvalidate` and `figureChanged` in terms

of the calls to validate and invalidate in a declarative way. The variable `isChanging` prevents from nested change notifications in an equivalent way the variable `changingDepth` was used in Listing 9.1. As a result, we eliminated 12 lines of code, but more importantly we believe the intention of the programmer is clearer in Listing 9.3. We define what a change is, we express that when there is an enclosing change we invalidate the figure, we continue with the change and then we validate the changed figure. Finally we express that the area has been invalidated after the call to `invalidate` and that the figure has finally changed after the call to `validate`.

We have presented the first case study, which validates EJava. Now we continue with the second case study that validates the combination of our implicit invocation model and advanced OO features in ECaesarJ.

## 9.2 The Smart Home Case Study

This case study is part of a larger case study used in the project AMPLE [AMP10, GNNM11] in the context of building software product lines using model-driven engineering and aspect-oriented software development. The Smart Home case study is related to building automation. Modern buildings are equipped with a set of electrical sensors and actuators in order to allow for an intelligent sensing and controlling of building devices: temperature sensors and thermostats, electrically steered blinds and windows, fire and glass break sensors, etc. As there is the increasing demand to coordinate the various devices under control of an integrating IT platform, a Smart Home product emerges as software that networks these devices and enables the inhabitants to monitor and control their house from various user interfaces. A local network also allows the devices to coordinate their behavior in order to fulfill complex tasks without human intervention.

Houses are intrinsically different from each others, not only in terms of architecture, but also in terms of their available devices. Home-automation software needs to be adaptable to each possible setting. The Smart Home case study consisted of understanding how to build a software product line. It provides the necessary building blocks and mechanisms to automatically select and assemble suitable components and generate proper software products in terms of the set of features that better describes the requirements of a particular home.

Several techniques can be used in order to build a suitable smart home. First, modularization mechanisms such as family classes can help providing a good structural decomposition as they permit us to define the structure of objects in terms of features. Second, events can improve the decoupling of the different building blocks and provide good abstractions to describe automation behavior. Third, aspects can help modularizing crosscutting functionality. The case study is an example of large software that requires combining all these techniques. This section shows how ECaesarJ can be used to implement a smart home. This solution was initially described in [GNNM11].

Section 9.2.1 shows how the ECaesarJ virtual classes and propagating mixin are the starting point of our solution by providing good means to separate the structure of a smart home in terms of features. Then, Section 9.2.2 shows how, on top of this structural decomposition, the ECaesarJ event abstractions permit the implementation of the behavioral dimension of a smart home.

### 9.2.1 Structural Dimension

Every house is different. An important source of variability lies in the layout of a house. The number of floors, rooms, connecting doors and/or windows are typically unique for every house. On top of this (building) architectural setting, houses are equipped with connected devices. We have exploited the flexibility of virtual classes and propagating mixin composition in order to reflect in code the layout elements of a house and to express its variability in a modular way.

**Virtual-class decomposition.** The required functionality of a smart home can be separated in *features* [Pre97]. A feature may indicate for example a house with shutters, another may indicate a house with movement detectors and a certain behavior and another a house with several floors. They are combined to form the whole home functionality. This separation is called a *feature-oriented decomposition*.

```
cclass HouseStructure {
   abstract cclass Location { }

   abstract cclass CompositeLocation extends Location {
      abstract List<? extends Location> locations();
   }

   cclass Room extends Location { }

   cclass Floor extends CompositeLocation {
      List<Room> rooms;
      List<Room> rooms() { return rooms; }
      void addRoom(Room r) { rooms.add(r); }
      List<? extends Location> locations() { return rooms(); }
   }

   cclass House extends CompositeLocation {
      List<Floor> floors;
      List<Floor> floors() { return floors; }
      void addFloor(Floor r) { floors.add(r); }
      List<? extends Location> locations() { return floors(); }
   }

   House house = new House();
   House house() { return house; }
}
```

Listing 9.4: Implementation of a house structure as a family class.

We model a smart home feature as a family class and domain objects as its virtual classes. Since virtual classes can be refined in subclasses of the family class, a feature can extend the functionality of another feature by inheriting from it and selectively refining its classes. As an example, Listing 9.4 shows the implementation of the feature that defines the structure of the house. This feature is represented by the class `HouseStructure`, which contains virtual classes describing house locations.

The advantage of such class grouping is that it makes it possible to extend the classes

```
cclass HouseShutters extends HouseStructure {
   abstract cclass Location {
      abstract List<Shutter> shutters();
   }

   abstract cclass CompositeLocation {
      List<Shutter> shutters() {
         List<Shutter> shutters = new ArrayList<Shutter>();
         for (Location child : locations()) {
            shutters.addAll(child.shutters())
         }
         return shutters;
      }
   }

   cclass Room {
      List<Shutter> shutters;
      List<Shutter> shutters() { return shutters; }
   }
}
```

Listing 9.5: Extension of house structure with shutter devices.

with new features in a consistent way. For example, Listing 9.5 shows the extension of a house structure with shutter devices. This extension is defined in a new family class `HouseShutters`, declared as a subclass of `HouseStructure`. As a result, it inherits all virtual classes of its superclass, and can selectively extend them with new functionality. The class `Location` is extended with a declaration of an abstract method `shutters` providing access to the shutters of the location. The classes `CompositeLocation` and `Room` are extended to implement this method. In addition, the class `Room` is extended with a new field to store the list of shutter devices of its instances. As can be seen, the class `HouseShutters` defines only the very functionality related to organising the shutters in a house. All other definitions are implicitly inherited from `HouseStructure` and combined with the definitions of the extension. This means that `HouseShutters` inherits the definitions of `House` and `Floor` from its superclass, and the redefined classes `Location`, `CompositeLocation`, and `Room` implicitly inherit from the respective classes of `HouseStructure`.

**Particular Products.** Let us consider another extension of the house structure with light devices, presented in Fig. 9.6. This extension is analogous to the extension with the shutter devices of Fig. 9.5. The two extensions can be composed using propagating mixin composition by defining a class inheriting from them both as shown in Listing 9.7. The class `MyHouse` inherits all the members of both superclasses, and in particular their virtual classes. The virtual classes with equal names are automatically merged using the same mixin composition semantics. For example, the virtual class `MyHouse.House` inherits from the composed version of `CompositeLocation`, which implements operations for collecting all lights and all shutters in the house.

The class `MyHouse` represents a particular house product with two features: (1) a house with shutters and (2) a house with lights. More features can be added by composing the respective family classes. In the remainder we show how this structural implementation can

```
cclass HouseLights extends HouseStructure {
   abstract cclass Location {
      abstract List<Light> lights();
   }

   abstract cclass CompositeLocation {
      List<Light> lights() { ... }
   }

   cclass Room {
      List<Light> lights;
      List<Light> lights() { return lights; }
   }
}
```

Listing 9.6: Extension of house structure with light devices.

```
cclass MyHouse extends HouseShutters & HouseLights { }
```

Listing 9.7: Particular smarthome product.

be extended with behavior using the programming model introduced in this dissertation.

### 9.2.2 Behavioral Dimension

The structural decomposition described so far is the base framework on top of which we have defined the behavioral dimension of a smart home that we describe in this section. This behavior mainly corresponds to reactive functionality on the occurrence of several events, which are part of the daily life of the inhabitants of a home. For instance, a sunrise is an example of an event on which occurrence we would like shutters to be automatically opened in the morning.

This section shows how the different language abstractions and the integration of the different programming paradigms introduced by our programming model provide a simple and regular way to express the different behaviors of a smart home in a modular way respecting feature decomposition. The family classes that describe the layout of a house are extended here with events and handlers in order to implement the listed behaviors.

**Event Abstractions** Various events can be identified in a smart home application. How these events are defined corresponds to a variability point. Event definitions may depend on the kind of devices that are available in a particular house or on environmental conditions. For example, a sunrise event can be defined in terms of light intensity if the house is provided with a light sensor and it is summer, or as a timer signal otherwise. When defining a reactive behavior in a modular way we would like to abstract out from the manner events are defined. ECAESARJ permits this separation thanks to the fact that events can be defined as `abstract` permitting independence from their concrete definitions. This flexibility highlights the benefits of an integration of EBP and OOP allowing programmers to deal with events similarly to methods.

As an example, Listing 9.8 declares a *daytime detection* feature representing the ca-

```
abstract cclass IDaytime {
   abstract event sunrise();
   abstract event sunset();
   abstract boolean isDaytime();
}
```

Listing 9.8: Event Abstractions.

pacity (or feature) of a house to provide sunrise and sunset events. The `IDaytime` family class declares the `sunrise` event, emitted when it gets bright, and `sunset`, emitted when it gets dark. Additionally, the `isDaytime` predicate makes it possible to check whether it is daytime (in-between sunrise and sunset). The `sunrise` and `sunset` events are `abstract`, delaying definition to concrete subclasses. A smart home provided with this family class enables the implementation of behaviors on top of these events independently from the way the events are defined at different stages. A complete Smart Home product can be equipped with all the required behavior and just needs to be connected with the proper event sources depending on concrete situations.

```
cclass DaytimeShutterAutomation
  extends HouseShutters & IDaytime & IShutterControl {
  cclass Room {
    sunrise() => {
      for (Shutter s : shutters())
        s.setAperture(100); // fully open
      next;
    }
    sunset()  => {
      for (Shutter s : shutters())
        s.setAperture(0);   // fully closed
      next;
    }
  }
}
abstract cclass IShutterControl {
  cclass Shutter {
    abstract void setAperture(int aperture);
  }
}
```

Listing 9.9: A feature that opens shutters at sunrise and closes them at sunset.

**Smart, Reactive Home Behavior** We have implemented the different reactive behaviors of a smart home on top of abstract events in a modular way. The standard EBP capabilities of ECAESARJ have been intensively used to implement reactive behaviors as event handlers, whereas other functions have been implemented as methods. As an example, the `DaytimeShutterAutomation` class of Listing 9.9 implements an automation behavior which opens and closes shutters on the occurrence of the `sunrise` and `sunset` events, respectively. This feature requires a house equipped with shutters and with `sunset` and `sunrise` events as illustrated by the dependence on the `HouseShutters` and `IDaytime` classes. In addition,

it requires the class `IShutterControl`, which imposes a specific behavioral interface to shutters with a command to set a specific aperture.

```
cclass SensorDaytime extends IDaytime requires ILightSensor {
  final int THRESHOLD = 80;
  protected boolean daytime;

  event sunrise() = lightSensor().intensityChanged()
    && if(!daytime && lightSensor().getIntensity() > THRESHOLD);

  event sunset() = lightSensor().intensityChanged()
    && if(daytime && lightSensor().getIntensity() < THRESHOLD);

  public boolean isDaytime() { return daytime; }

  sunrise() => { daytime = true; next; }
  sunset() => { daytime = false; next; }
}
```

Listing 9.10: Implementation of daytime events in terms of light intensity.

**Variability of Event Definition** When a house is not provided with a light sensor a second implementation can be used. This implementation imperatively define these events in terms of the hour of the day as shown in Listing 9.11.

```
cclass TimedDaytime extends IDaytime {
  public void run() {
    sunrise();
  }
}
```

Listing 9.11: Implementation of daytime events in terms of the hour of the day.

As previously said the complete functionality of a smart home can be expressed in terms of abstract events. In a second step, we need to provide proper event definitions in order to consolidate a final product. With ECAESARJ, events can be defined either by explicitly triggering them in code or by composing other existing events in a declarative way. The way events are defined is a variability point that depends on particular home settings. As an example, Listing 9.10 shows a possible implementation of the `sunrise` and `sunset` events. This implementation assumes that the `lightSensor` method provides a light sensor. Line 5 defines the `sunrise` event as a light intensity change taking place when it is still nighttime and exceeding a predefined threshold. Here, the event expression defining the `sunrise` event combines the definition of the `intensityChanged` event of the light sensor returned by the `lightSensor` method and a conditional expression testing the current state and light intensity. Line 8 defines the `sunset` event in an analogous way.

**Quantification** The quantification feature of ECAESARJ makes it possible to express in a compact way events that depend on several objects. For example, the presence of somebody in a location can be described in terms of events of motion sensors available at

```
cclass PresenceAtLocation {
  Location location;
  TimeService timeOut;
  event somebodyInLocation() = some(location.motionSensors()).motion();
  event somebodyOutOfLocation() = timeOut.time();
  ...
}
```

Listing 9.12: Presence in terms of activity in several sensors.

this location. A motion sensor provides an event `motion`. The constructor **some** is very useful in this case. As an example, in Listing 9.12 a presence is notified when some `motion` is detected.

**Method Calls as Events** Many events are made available for free thanks to the unification of method calls and events. Method calls as explicitly triggered events are observable by external entities, which can program reactive behavior on their occurrence. As a consequence the necessity of preplanning is reduced, simplifying in this way the implementation of the application and improving its extensibility.

```
cclass LightActivityDetection requires IHouseLights {
   event activityDetected() =
      some(house().lights()).turnOn() ||
      some(house().lights()).turnOff();
}
```

Listing 9.13: Light activity detection.

The implementation of an alarm system is an example of this. The alarm needs to be triggered when some activity is detected in the house in a period at which nobody should be at home. The alarm behavior is bound to an abstract event `activityDetected`, which indicates the situation when the alarm should be triggered. The event could be defined in several ways: in terms of the detection of a movement sensor, or the state of the device of the house. An additional source of events is provided by the activity of the Smart Home software. A call to a method `turnOn` on any device could be considered as an indicator that somebody is at home. Since a method call is an event in our model, the `turnOn` call can be used without additional effort to define the `activityDetected` event as shown in Listing 9.13.

We have show how our programming model, materialised in the ECAESARJ language, provides the necessary abstractions to build a Smart Home in a regular and simple way, all this thanks to a better integration of the techniques provided by the OOP, EBP and AOP paradigms.

## 9.3 Conclusion

Our first case study has shown that the amount of event-related code in a sizable application such as JHotDraw is not very big. In our minimal version we measured 500 lines of code in a total of 7K. These 500 lines of code are however fundamental as they

connect the user input with the application functionality. We saw that a big part of this code was just registration infrastructure and registration calls. We eliminated more than half of the number of lines of code and we improved their modularity. As a result, we got code that is easier to verify and maintain, with a direct mapping between the features at the design level and their implementation.

Our second case study was built from scratch. Whereas family classes made the decomposition of the application in features easy, our implicit invocation model motivated the use of events and facilitated the task. We showed how both technologies were used together.

The next chapter provides details on the implementation of our languages and some hints on their performance.

# Chapter 10

# Implementation

This chapter is dedicated to the implementation of the languages introduced in this dissertation. Sections 10.1 and 10.2 describe the main aspects of the implementation of EJava and ECaesarJ, respectively. Section 10.3 concludes with some final remarks.

## 10.1 Implementation of EJava

EJava is a language compiled into Java. The compilation process generates the infrastructure that implements our implicit-invocation model and the needed glue code. The code transformation/generation is performed by using the code-transformation tool Stratego XT [Vis01, Vis04]. The generated Java code is compiled by using a standard Java compiler. The remainder of this section describes the compilation process.

### 10.1.1 Principle

Most of the code of an EJava application consists of plain Java: definition of classes, definition of methods and invocations of these methods. As an important design decision, the plain Java code of an EJava program is not modified by the compilation process. It remains as plain Java code. This feature contributes to the efficiency of EJava but it also makes it possible to consider any Java program as an EJava program, in particular, Java libraries. All the generated EJava infrastructure depends on this design decision.

As explained in Chapter 5, plain Java code respects our implicit-invocation model. A method is conceptually seen as both an imperative event and an event handler, and a method call as the triggering of an imperative event. No additional compilation effort is needed if the method body is the only event handler registered with the imperative event: on the method call the method body is normally executed. However, the compilation of an EJava program must take into account the fact that an imperative event defined with method syntax can be used in the definition of other (declarative) events. Consequently, other event handlers can be (indirectly) registered with such an imperative event. Since we do not touch the plain Java code of an EJava program a mechanism is needed to intercept the relevant method calls and give the control to the EJava interpreter.

For the sake of simplicity, imperative events directly defined by using the `event` construct are translated into Java methods. The bodies of these methods are the bodies of the handlers directly registered with these imperative events. They are empty if there is no handler. This simplification allows all the imperative events of the program to be treated in the same way: by performing method-call interception.

The remainder describes the interpreter (Section 10.1.2) and the code-generation process (Section 10.1.3).

### 10.1.2 EJAVA Interpreter

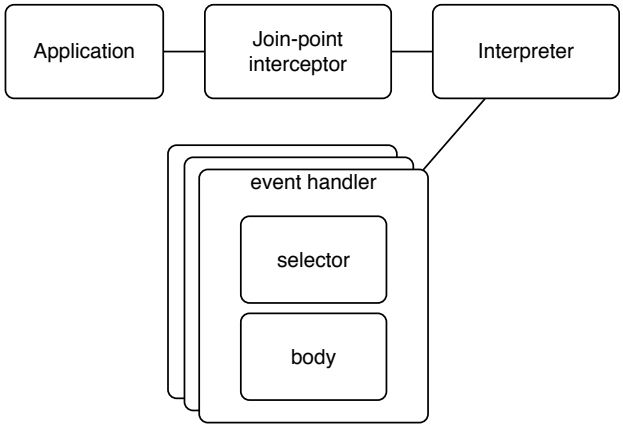


Figure 10.1 – Architecture of the EJAVA interpreter.

The EJAVA interpreter is a dedicated extension of the aspect-language interpreter CALI [AN09]. Figure 10.1 shows the architecture of the system. The three boxes on top represent the application and the interpreter, connected by a join-point interceptor. The box on the bottom represents a list of *deployed* event handlers. A selector of runtime events and a body is associated with each event handler. The join-point interceptor intercepts the relevant join-points from the application execution and passes the control to the interpreter. The latter executes a set of proper event handlers. This set contains the handlers whose selectors select the runtime event.

```
abstract aspect AbstractPlatform {
  abstract pointcut reifyBase();

  Object around(): reifyBase() {
    return Interpreter.eval(new RtEvent(thisJoinPoint));
  }
}
```

Listing 10.1: ASPECTJ aspect intercepting the relevant method calls in the application.

The join-point interceptor is implemented by the ASPECTJ aspect of Listing 10.1. The pointcut `reifyBase` indicates the list of join points to intercept in the application. It is defined as abstract so that the list of join points can be defined later on. The aspect defines a piece of advice that creates a runtime event, *i.e.* an instance of the class `RtEvent`. The `RtEvent` instance is created from the intercepted join point. The piece of advice evaluates this runtime event by calling the method `eval` of the class `Interpreter`, which implements the interpreter.

In the implementation of the interpreter we use the following classes:

- The class `EventSelector` defines an object that is used to select a runtime event. The events of the application are translated into instances of this class.
- The class `HandlerBody` defines an object with an executable method `run`. This method is executed in reaction to a runtime event given as a parameter. The bodies of the event handlers of the application are translated into instances of this class.

– The class `EventHandler` ties together an instance of `EventSelector` and an instance of `HandlerBody`. The event handlers of the application are translated into instances of this class.

The repository of event handlers consists of a list of `EventHandler` instances, representing all the event handlers deployed in the application. Each time a new object is deployed their event handlers are added to the repository. The method `eval` of the interpreter evaluates the selectors of all the deployed `EventHandler` instances and filters the ones whose selectors select the event.* It puts the resulting objects in a stack and executes the `HandlerBody` of the instance on the top of the stack. The class `Interpreter` provides a method `next`, which can be called from the running `HandlerBody` instance to nest the execution of the next handler on top of the stack. When `next` is called and there are no more handlers on the stack, a dedicated handler performs an ASPECTJ `proceed`.

### 10.1.3 Generated Infrastructure

```
class Connector {
  event out() = leftFigure().changed() || rightFigure().changed();

  out() => {
   next; update();
  }

  Figure leftFigure() { ...}
  Figure rightFigure() { ...}
}
```

Listing 10.2: Implementation of a connector between two figures.

The compilation of a class defining events and handlers consists of generating the corresponding instances of the previous classes and the needed additional infrastructure.

As described earlier, imperative events are translated into standard JAVA methods. If no handler has been bound to an imperative event, the resulting method has an empty body. Otherwise, the method body is the body of the corresponding event handler.

In order to describe the translation of declarative events and the translation of event handlers bound to declarative events, we will use as an example the translation of the EJAVA class `Connector` of Listing 10.2 into the JAVA class of Listing 10.3.

A declarative event *evt* is translated into a selector of runtime events. More concretely the event construct is translated into a field `event_`*evt* referencing an instance of `EventSelector` implemented in terms of the corresponding event expression. Lines 2-15 of Listing 10.3 show the translation of the event `out` of the class `Connector` into the field `event_out`.

An event handler *evt* is translated into a field `handler_`*evt* referencing an instance of `HandlerBody`. The method `run` of such an instance contains the handler body with the calls to `next` translated into a call to the method `next` of the interpreter. Lines 17-25 show the translation of the event handler `out` of the class `Connector` into the field `handler_out`.

Each class that defines at least one event handler is equipped with a method `deploy`. For each event handler *evt* defined in the class, this method deploys in the interpreter

∗. Section 10.1.4 describes some optimizations to reduce the amount of evaluations.

```
1  class Connector {
2    EventSelector event_out =
3      new EventSelector()
4      {
5        Object[] select(RtEvent evt) {
6          return getS1().select(evt) ||
7                 getS2().select(evt);
8        }
9        EventSelector getS1() {
10         Figure f = leftFigure();
11         return f != null?
12           f.evt_changed: FALSE;
13       }
14       EventSelector getS2() { ... }
15     };
16
17   HandlerBody handler_out =
18     new HandlerBody()
19     {
20       Object run(Object[] t) {
21         Interpreter.next();
22         update();
23         return null;
24       }
25     };
26
27   void deploy() {
28     Interpreter.deploy(
29       new EventHandler(
30         event_out, handler_out));
31   }
32 }
```

Listing 10.3: Translation of the EJAVA class of Listing 10.2 into JAVA.

an instance of `EventHandler`. The `EventHandler` instance ties together `event_`*evt* and `handler_`*evt*. Lines 27-31 show the method `deploy` added to the class `Connector`.

Finally, the compilation of an EJAVA program creates a concrete subclass of the aspect `AbstractPlatform` of Listing 10.1 defining the pointcut `reifyBase` with the list of method calls to intercept. This list is created by analyzing the application and obtaining the signature of the events used by the application.

### 10.1.4 Optimizations

EJAVA events are late-bound. The relationships between events are context-dependent. For example, the event `out` of the connector of Listing 10.2 is related to the events `changed` of the connected figures. The connected figures are returned by the methods `leftFigure` and `rightFigure`. The objects returned by these methods may depend on the context. This makes the event `out` of the connector potentially related to the event `change` of any figure in the application. Consequently, when any figure changes in the application the event selector of `out` needs to be evaluated to test whether it selects the corresponding runtime event. In the evaluation, the selector invokes the methods `leftFigure` and `rightFigure` to obtain the current figures (see line 10). Then it tests whether the runtime event corresponds to a change in one of these figures.

So far we have shown a very simplistic translation of an EJAVA class. The current implementation of EJAVA produces better code. For example, we avoid evaluating several times expressions such as the call to the method `leftFigure` of line 10 during the handling of the same runtime event. We also implement a more selective evaluation of event selectors. For example, a connector is only interested in the event `changed` of figures. The occurrence of an event called `changed` of another kind of object should not trigger the evaluation of the event selector of the connector.

Despite these optimizations, when there are many figures and many connectors the number of evaluations of event selectors can rapidly increase. A connector would observe the changes of several figures, whereas it is only interested in two particular figures. As a

```
class Connector {
  Figure leftFigure, rightFigure;

  event out() =
    leftFigure.changed() || rightFigure.changed();
  ...
}
```

Listing 10.4: Optimized event `out`.

solution to this problem, Lucas Satabin [GSM+11] implemented an optimization for the case an event is defined by accessing the events of the fields of a class (in a direct way). For example, this optimization is feasible for the event `out` of the class `Connector` of Listing 10.4, in which `leftFigure` and `rightFigure` are fields instead of methods. When the fields used in the definition of an event are immutable, the relationships between the defined event and the events of these fields are also immutable. Thus, it is possible to construct a graph expressing the relationships between the events of different objects. On the occurrence of an event in the graph, the occurrence can be propagated to the next nodes of the graph, avoiding the evaluation of the unconnected events. In the case the fields are mutable, a mechanism is needed to detect the modifications of the fields and update the graph.

We have borrowed these ideas so that EJAVA optimizes the case in which events are related through the fields of a class (in a direct way). Since in JAVA we cannot declare a field as immutable we have to deal with the case where a field is modified. This problem is solved with the introduction of a dedicated ASPECTJ aspect that intercepts the changes of relevant fields such as `leftFigure` and `rightFigure` and updates the corresponding graphs.

## 10.2 Implementation of ECAESARJ

The implementation of ECAESARJ consists of incorporating the previous implementation into CAESARJ. Since we generate methods and fields, the propagating mixin composition of CAESARJ can be applied to them without problems. The generated code is then compiled by using the CAESARJ compiler. Since the compilation process is the same as the one of EJAVA, the remainder of this section is just dedicated to the implementation of the language support for state machines in ECAESARJ.

State machines implemented with the `smcclass` construct are translated into plain ECAESARJ classes to be afterwards compiled with the ECAESARJ compiler. The translation generates an implementation similar to the one described in Section 8.3.2. In order to describe the translation let us consider the state machine `BasicShutter` of Listing 8.9. As a result of the compilation, `BasicShutter` is translated into the plain ECAESARJ class `BasicShutter` of Listing 10.5.

The class `StateMachine` of lines 1-23 defines the common infrastructure of state machines. It includes a reference to the current state of the state machine, a default constructor transiting into the initial state of the state machine, two classes `State` and `Internal`, and methods to transit between states. The initial state is defined with a method `initial`, left abstract. The class `State` represents a state. The class `Internal` represents an internal

```
1  cclass StateMachine {
2    State currentState;
3
4    StateMachine() {
5      transit(initial());
6    }
7
8    abstract State initial();
9
10   abstract cclass State {}
11   abstract cclass Internal
12            extends State {}
13
14   void transit(State aState) {
15     undeploy(currentState);
16     deploy(currentState = aState);
17   }
18
19   void transitInternal() {
20     undeploy(currentState);
21     currentState = new Internal();
22   }
23 }
24
25 cclass BaseShutter
26        extends StateMachine {
27
28   void setAperture(int ap) {
29     currentState.setAperture(ap);
30   }
31
32   event apertureReached() =
33         device.apertureReached();
34   cclass State {
35     abstract void setAperture(int ap);
36     apertureReached() => {
37       transit(new Idle());
38     }
39   }
40
41   State initial() {
42     return new Idle();
43   }
44
45   cclass Idle extends State {
46     void setAperture(int ap) {
47       transitInternal();
48       /* send aperture to device */
49       transit(new Moving());
50     }
51   }
52
53   cclass Moving extends State {
54     void setAperture(int ap) {
55       throw new Exception();
56     }
57   }
58
59   cclass Internal {
60     abstract void setAperture(int ap) {
61       throw new ReentranceException();
62     }
63   } ...
64 }
```

Listing 10.5: Translation of the base shutter control of Listing 8.9.

state between two transitions for reentrance purposes. The method `transit` undeploys the current state and deploys the new one. The method `transitInternal` makes the machine transit into the state `Internal` and will be called at the beginning of an event handler for reentrance purposes.

The translation of a `smcclass` construct goes through the following steps:

- The alphabet of the state machine is determined by looking at all the defined events. In the example, the alphabet consists of the events `setAperture` and `apertureReached`.
- All the imperative events are translated into forwarding methods. The reason is that in our implementation plain JAVA methods perform better than events. In the example, the event `setAperture` is translated into a method that forwards the call to the current state (lines 28-30).
- From the alphabet of the state machine and the default event handlers we create the class `State`. It includes the default event handlers and abstract members for the events of the alphabet lacking a default event handler. In the example, the class `State` of line 34 includes `setAperture` as abstract and the default handler of the event `apertureReached`.
- The `state` constructs are translated into classes extending `State`. The constructs `=>` within the event handlers of each state are translated into calls to the method `transit`.
- When the body of an event handler includes more instructions than just a transition we include a call to `transitInternal` at the beginning. This call makes the state machine transit into an internal state. The internal state, represented by the class `InternalState`, is formed of handlers for the imperative events of the alphabet of the state machine. These handlers throw a reentrance exception. Thus, inside an event handler of a state the state machine cannot handle other events. In the example, the event handler `setAperture` in the state `Idle` performs some instructions to send the aperture to the device. Before performing the instructions a call to `transitInternal` is introduced (line 47). As a result, while performing these instructions the state machine is in the internal state. The class `transitInternal` deals with `setAperture` by sending a reentrance exception.

As a result of the compilation of a state machine we get a standard ECAESARJ class that can be compiled by the ECAESARJ compiler.

## 10.3 Final Remarks

In our model an object is able to see any method call, even when the callee is another object. An efficient implementation of this model is not that easy to get as method-call observation grows with the number of objects of the application independently from the number of actually observed objects. Optimizations are important to get a scalable solution. Indeed, the standard OO metaphor can be seen as an optimized subset of our programming model such that an object only sees the calls to its own methods.

The implementation of our programming model has been optimized for dealing with methods. Methods are kept as plain JAVA code so that they are directly compiled by JAVA. In this way, the performance of a JAVA program is not affected by our model. Imperative events that have not been directly attached to event handlers are also translated to methods, with an empty body. As a result, every event triggering in the application corresponds to a method call and this method call is by default dealt with by JAVA. For

the case a method call is observed by other objects we have put in place a method-call interception mechanism using AspectJ.

# Chapter 11

# Conclusion

This chapter concludes this dissertation. Section 11.1 sums up the programming model introduced by this dissertation. Section 11.2 discusses the most important design decisions of this programming model. Section 11.3 highlights the main contributions of this work. Finally, Section 11.4 gives some guidelines on future work.

## 11.1 Summary

We have introduced a programming model, which, based on a general notion of event and event handler, unifies three programming paradigms: EBP, AOP and OOP. Classes in the model consist of fields, events and event handlers. Events are either imperative or declarative. An imperative event is explicitly triggered in the code of an application with a syntax similar to a method call. A declarative event is defined in terms of other events in a declarative way. Event handlers register with events such that they are executed when the respective events happen. Listing 4.17 illustrates what events and event handlers look like. The notion of declarative event contributes to the area of EBP as it makes it easy to program reactions to compositions of events. This is harder in most EB approaches, which only provide imperative events. The integration with OOP is enabled by the inclusion of methods in the model. A method is introduced as syntactic sugar for both an imperative event and an event handler registered with this event (see Listing 5.9). Thus, the model unifies the notion of method call of OOP with the notion of event triggering of EBP: the occurrence of an imperative event corresponds to a method call. In other words, when an object sends a message to another object, it explicitly triggers an imperative event. As a reaction to this imperative event, the method body is one of the event handlers to be executed. In this way, EBP and OOP can be provided in a regular way by using the same language constructs. The unification of event handling and method execution also enables AOP. A programmer working with methods *implicitly* trigger imperative events which can be observed by aspects, implemented as classes. These aspects implement their pieces of advice as event handlers reacting to such imperative events (see Listing 6.4). Pieces of advice can also react to declarative events, which combine events in a declarative expression similarly to the pointcuts of AO languages. Finally, the implementation of aspects as plain classes provides a good integration of AOP and OOP. First, aspects are explicitly instantiated like any class. Second, pointcuts, implemented as events, are full-fledged instance members. Third, aspects are subject to the standard inheritance rules of classes. The programming model has been implemented as an extension of Java called EJava, and as an extension of CaesarJ called ECaesarJ. The latter completes the integration by including support for structural aspects by using the notion of mixin composition and language support for state machines.

## 11.2 Discussion

This section clarifies some important points of our programming model. First, we clarify the respective role of explicit and implicit invocation resulting from the unification of event handling and method execution. Second, we make a parallel between the object-oriented nature of our events and the global nature of events in approaches based on event types. Finally, we discuss some technical issues such as expressiveness.

### 11.2.1 Explicit and Implicit Invocation

Whereas events are natural implementations of implicit invocation, methods are natural implementations of explicit invocation. Since our model unifies methods and events it is important to clarify how explicit and implicit invocation can both coexist in our model.

```
1 class SomeTool implements Tool {
2   Figure targetFigure;
3   ...
4   /* code that transforms the target figure */
5   targetFigure.changed();
6   ...
7 }
```

Listing 11.1: Event triggering or method call in EJAVA.

Consider Listing 11.1 implementing a JHotDraw tool in EJAVA. This tool transforms the figure `targetFigure` declared in line 2 and then executes the expression `targetFigure.changed()` in line 5. In our model, this expression means a triggering of the event `changed` of `targetFigure`. It notifies that the figure has changed after the transformation. The expression causes the implicit invocation of all the event handlers registered with the event `changed`.

When the class `Figure` declares the event `changed` using method syntax, *i.e.* when `changed` is a method, we can refer to the event triggering as a method call. In this case, the tool is aware of the existence of one of the event handlers registered with `changed`: the method body. The expression means the explicit invocation of the method body (as the tool is aware of it) and, at the same time, the implicit invocation of the other event handlers registered with the event (as the tool does not know them). This is the way both explicit and implicit invocation are combined in our model.

In other words, the invocation scheme depends on the protocol describing the relationship between objects. When there is a notion of required and provided interfaces, the invocation scheme is explicit, otherwise the scheme is implicit. This difference poses some challenges when combining aspects and components with explicit protocols as we explored in [NN07]. The model of this dissertation provides a seamless integration of both schemes.

### 11.2.2 Event Types versus OO Events

The interaction between objects is one of the most important constituents of an OO design. In models of explicit invocation, objects send messages to objects. In models of implicit invocation, such as the pattern Observer, objects register with objects. Similarly, in our model an object registers with the events of the object it explicitly knows. However,

in approaches based on event types object interaction is indirect. An object registers with event types, not with events of particular objects. Something similar happens in most AO approaches. In this section we would like to discuss about event types as there is a complete branch of EBP that is based on this notion [EGD01, RL08, SPAK10].

The idea of a design based on event types is to decouple destinations from sources: instances of an event type are provided by objects that destinations do not know. However, such a design is less object-oriented than the solution that uses the pattern Observer. Since an event type is in general provided by an application, it promotes the interaction between an object and the application instead of an interaction between objects. Thus, instances of event types have to provide the necessary information to be used by destinations, which have to apply specific filters on this information to select the relevant events associated to the relevant objects.

Events in AOP are compatible with event types. For example, an AspectJ aspect can monitor the calls to a method of a class. The aspect does not really know the objects that will provide such an event. They can be any instance of the class or a subclass. An event defined in this way is "application-level" or, in other words, static. Thus, AspectJ provides specific pointcut designators such as `this` to obtain the objects that are sources of the event. If an event of a specific object is expected then a specific filtering has to be done in the advice code to select such an event. The link between the event types and the events of AOP is made even more explicit in IIIA [SPAK10]. An event type in this approach can be implemented by a pointcut. Classes can be registered as advising an event type or as providing it.

Our programming model gives a bigger priority to an object-oriented design than to the decoupling of destinations from sources. In our model a destination registers with the events of the objects it is supposed to collaborate with, similarly to the pattern Observer. Thus, no additional filtering is necessary to select the relevant events. An object is only notified of the relevant events.

A way to turn an application-level event type into an object-level one is by restricting the scope of observation of an event type. If a destination object subscribes to a certain event type, scoping strategies [Tan08] can be used to restrict the space in which instances of such an event type are *collected*. This space can be set to the source objects that are relevant to the destination object. Thus, combinations of event types and proper scoping strategies can produce object-oriented interactions like the ones achieved with the pattern Observer.

The use of event types together with proper scoping strategies has the advantage that it *permits designers to address the tradeoff between specificity and reusability by deferring certain decisions to aspect deployment, leaving aspect definitions more reusable* [Tan08]. However, the definition of events as combinations of other events implies not only combining event types but also providing the proper combinations of the respective scoping strategies. The task is duplicated. The advantage of our model is that an event encapsulates an object-level event, so that it can be reused in a straightforward manner. We can still deal with specificity thanks to the late-bound nature of our events.

### 11.2.3 Technical Issues

Let us now discuss and justify some technical design decisions taken in our model. The discussion mainly concerns the design of event handling and the expressiveness of our model.

#### 11.2.3.1 Event Handlers

An event handler in our programming model is defined as bound to an event. This design decision contrasts with the flexible subscription and unsubscriptions of event handlers with particular events of traditional EB languages. In the pattern Observer, for instance, a subject provides methods to add and remove observers. In approaches such as C# the operators `+=` and `-=` are provided for the same purposes. The design proposed by this dissertation is less dynamic in this sense.

```
class A {
  B b;
  event evt2() = b.evt3();

  evt2() => { /* code of the handler */ }

  public static void main(String[] args) {
    A a = new A(); B b = new B();

    /* subscription */
    a.b = b;

    /* unsubscription */
    a.b = null;
  }
}
```

Listing 11.2: Emulating subscription and unsubscription of events.

Our design is natural when considering the definition of methods as event handlers (method bodies are statically defined). On the other hand, thanks to the late-bound nature of events, it is still possible to emulate the subscription and unsubscription of event handlers with events. A declarative event can be defined in terms of the events of a particular object and the value of the event depends on the value of the given object reference at runtime. If the object reference is null, then there is no observation of any event occurrence. By setting the object reference to null we can emulate unsubscription, whereas by assigning a new object we can emulate dynamic subscription. Listing 11.2 illustrates this idea. In addition, we provide the construct `undeploy` (see Section 4.4.5) that deactivates the observation of events of external objects.

#### 11.2.3.2 AspectJ Expressiveness

Quantification on the static structure of a program as provided by AspectJ is, in some cases, more expressive than quantification on a list of objects as achieved with our constructor `some`. This issue is a consequence of the OO nature of the events of our model: we observe events of particular objects. Let us describe this issue in more detail.

Consider as an example the AspectJ pointcut of Listing 11.3. It defines a pointcut `change` representing the change of a figure. The pointcut selects calls to either the method `moveBy` of the class `Figure`, the method `setBounds` of the class `Rectangle` or the method `setRadius` of the class `Circle`. The classes `Rectangle` and `Circle` are subclasses of `Figure` so that the pointcut designator `target` can be used to obtain the called object, as a `Figure` instance, no matter what method was called.

```
pointcut change(Figure figure):
        (  call(* Figure.moveBy(int, int))
        || call(* Rectangle.setBounds(double, double))
        || call(* Circle.setRadius(double))
        ) && target(figure);
```

Listing 11.3: ECAESARJ aspect.

```
Collection<Figure> figures;

event change(Figure figure):
         some(figure: figures).moveBy(int, int)
      || some(figure: getRectangles(figures)).setBounds(double,double)
      || some(figure: getCircles(figures)).setRadius(double);
```

Listing 11.4: EJAVA aspect.

Listing 11.4 shows the EJAVA version of the previous code snippet. We assume a collection `figures` containing `Figure` instances. By using the constructor `some`, we observe the events `moveBy`, `setBounds` and `setRadius` of the objects in the collection. However, the constructor `some` relies on the type of the elements of the collection in order to safely define the declarative event. Since the method `setBounds` is only defined in the class `Rectangle`, we need to filter the `Rectangle` instances from the collection of figures. We do that by calling a function `getRectangles`. We do something analogous for `Circle` instances. The problem is that filtering is far from being efficient.

```
Collection<Figure> figures;

event change(Figure figure):
         some(figure: figures).moveBy(int, int)
      || ((Rectangle) some(figure: figures)).setBounds(double,double);
      || ((Circle) some(figure: figures)).setRadius(double);
```

Listing 11.5: EJAVA aspect.

We consider as future work an extension of the constructor `some` improving on this issue. Listing 11.5 shows a possible extension in which the constructor `some` is equipped with a casting expression filtering the objects of the given type from the given collection. As provided by the language, the filtering could be done efficiently.

## 11.3 Contributions

The main contributions of the work presented in this dissertation are: (1) an EB programming model with declarative events, (2) a better integration of EBP, AOP and OOP through this model, (3) support for a region-in-time model of aspects by using a point-in-time model, (4) the concretization of the model as an actual programming language with two prototype implementations, (5) thanks to the prototypes, the validation of the programming model on the implementation of both a figure editor and a home-automation

application. We summarize these contributions in the remainder.

### 11.3.1 Declarative Events

As Chapter 4 described, when developing large applications, a programmer often needs to define events in terms of other events. In several EB languages, the composition of events is programmed by using imperative events. The composite events are triggered as imperative events within event handlers registered with the composed events. As a result, when several events are defined in this way, it is hard to understand the different relationships between the events of an application. This is source of complexity when developing such an application. Chapter 4 introduced an EB programming model that innovates with the notion of declarative events. A declarative event expresses in a declarative way the relationships between the events of an application. The different relationships between events are easier to understand. Imperative and declarative events can be made part of the interface of a system offering additional modularity.

### 11.3.2 Unification of Paradigms

A combination of several programming paradigms is often required in order to implement the requirements of a complex application and provide *good quality* software. Examples of paradigms that can often be found in applications are EBP, AOP and OOP. In most applications the different paradigms are implemented by using different language abstractions. These abstractions are provided as different language constructs. The proliferation of concepts and language constructs may result in complicated programs.

We have noted that some of these mechanisms are not that different from one another. This conflicts with the principles of programming-language design stated by MacLennan [Mac95]. To solve this issue we have introduced a programming model that integrates EBP, AOP and OOP. The model is an extension of the model of declarative events previously described. The cornerstone of this model is a unification of event handling and method execution. The contributions of the model are as follows:

- The model demonstrates that the same language constructs can be used to implement both explicit and implicit invocation. Thus, both EBP and OOP can be supported in a simple and orthogonal way by a programming model equipping classes with fields, events and event handlers.
- The model shows that the same mechanism of events and event handlers can be used to implement both EB and AO applications. Events and join points can be unified and a piece of advice can be implemented as an event handler. Thus, programmers with an understanding of EBP can get into AOP without additional efforts.
- The model shows the benefit of providing both events and event handlers as full-fledged instance members. It improves the integration between EBP and OOP and at the same time the integration of AOP and OOP. We have shown that aspects and classes can be provided in a common language construct: a (possibly aspectual) class. The same regular rules of inheritance, instantiation and treatment of class members can govern the semantics of classes and aspects. Thus, programmers with an understanding of OOP can more easily integrate in their programming practice the EBP and AOP paradigms.

### 11.3.3 Join-Point Model

The model presented in this dissertation makes it possible to implement aspects with a join-point model based on the join points in time introduced by Masuhara et al. [MEY06]. There has been discussions whether the region-in-time model could not be replaced by a simpler point-in-time model. We have actually seen that this is indeed possible, except that we have only implemented part of the join points that can be of interest. In Section 6.2.1 we have shown how our model support AspectJ-like aspects where the key is to consider pieces of advice and method bodies as event handlers.

### 11.3.4 Concrete Implementation and Case Studies

Two languages implement our programming model. EJava implements the essence of the model as an extension to Java. ECaesarJ combines our model and mixin composition. Both languages have been used to validate our model in the implementation of real applications. They improved the design of these applications and reduced glue code needed when using a mainstream language such as Java. These languages are freely available for further experimentation in this area.

### 11.3.5 Experimentation with Applications

We applied our programming model in the implementation of a known figure editor, JHotDraw [Eri11]. EJava allowed expressing the relationships between the events of figures, views and tools in a straightforward way. It reduced the number of lines of code of the application and increased its understandability. We also applied the model to the implementation of an industrial strength case study: a home-automation application [GNNM11]. In this case, we used ECaesarJ. Whereas family classes and virtual classes were used to decompose the structure of the application, events were used for representing context changes and event handlers implemented adaptive behavior. By implementing these application we validated the usability of our model.

## 11.4 Perspectives

### 11.4.1 General Improvements

**Formal treatment.** In order to prove the correctness of our model and other properties we have as a perspective a formalization of this model. The interest in a formal treatment was reflected in Chapter 4. There we introduced the model based on the formal language $\text{MiniMAO}_0$ [CL05].

**Richer join-point model.** The model presented in this dissertation makes it possible to implement aspects with a join-point model based on the join points in time introduced by Masuhara et al. [MEY06]. Only two of the four kinds of join points of the model of Masuhara are supported: call and reception. These join points make it possible to express the method-call join point of AspectJ. As future work we plan to extend the kinds of join points supported by our model. For example, we would like to include support for the execution and return join points of the model of Masuhara. This would enable the support for the execution join point of AspectJ. Since in our model the execution of a method (its method body) corresponds to the execution of an event handler, we should be able to

support execution join points by including a kind of event representing the execution of an event handler and its corresponding after event. This kind of event would make it possible to implement aspects interested in the execution of a particular event handler.

**Exception handling.** As described in Section 4.4.4, we deal with exceptions following the JAVA style of checked exceptions, which is not well-suited to EB applications. As a perspective we plan to investigate possible ways to deal with exceptions [PH07]. As a rough idea, it seems appropriate in this case to augment the expressiveness of our model to be able to define events representing exceptions. As a type of exception can be thrown from many places in code with many involved objects, it can be unsuitable to quantify on the list of all the objects that can throw a type of exception. In this case, it seems that a model of event types can be better suited.

**Improvement of the prototypes.** We have provided two implementations of the model, EJAVA and ECAESARJ. These prototypes still require improvements, especially in terms of performance. In addition, we still need to improve the semantic analysis of the EJAVA and ECAESARJ compilers and provide better tool support. Regarding the latter, an initial Eclipse plugin has been developed using Spoofax [KV10]. However, it still does not enable advanced features such as debugging.

**Concurrency.** Our work has focussed on a sequential setting. Studying the applicability of our model to a concurrent setting is seen as future work. The notion of event is often used to coordinate concurrent applications by using models taken from process algebras. Since our model is based on events the task should be facilitated in this sense.

**Exploring other design choices.** Now that we have a better understanding of the implications of our initial design decisions, we do not discard considering other design choices. In particular, we consider that the polymorphic events proposed in IIIA are especially interesting and it would be worth exploring the integration of event types with our OO events.

### 11.4.2 ESCALA

As a new/late spin-off of this work, a subset of our model has been implemented in the language ESCALA [GSM+11]. This language implements our model of declarative events on top of SCALA. There are some differences between this language and EJAVA both at the model and at the implementation level. First, whereas the basic operators to compose events, such as disjunction and conjunction, are similar in both languages, ESCALA provides operators that exploit the functional side of SCALA offering additional flexibility. Second, ESCALA does not integrate events and methods as in EJAVA. A method is a construct different from an event handler. Even if in ESCALA a method call can be seen as an event, method bodies are not considered as event handlers as in EJAVA. ESCALA does not make it possible to implement a design-side aspect that prevents the base program to perform a method call. In EJAVA this issue is just a matter of composition of event handlers. Thus, ESCALA is an EB language with some flavor of AOP rather than a language that integrates both EBP and AOP in a seamless way as EJAVA. Finally, the implementation of ESCALA, made by Lucas Satabin, is more efficient than the original implementation of EJAVA thanks to the use of a push-based notification process. The

implementation of EJava presented in this dissertation has borrowed the idea of using a push-based notification process and the treatment of references to events through mutable fields. Even though, we still differ in that our implementation uses AspectJ as a tool for code instrumentation. Regarding ECaesarJ, EScala does not support state machines and mixins.

Scala is a good language for pursuing this work thanks to its combination of functional and imperative programming and features such as type inference. EScala can be improved with the elements presented in this dissertation that have not been still considered, such as the unification of event handling and method execution and the support for state machines.

# Appendixes

# Appendix A

# Résumé

## A.1 Introduction

Une grande partie de la recherche sur l'ingénierie de logiciel s'intéresse à la qualité du logiciel. Un logiciel de bonne qualité garantit des propriétés comme la modularité, l'extensibilité, l'efficacité, la robustesse, la correction, entre autres.

La clé pour obtenir beaucoup de ces propriétés est une bonne séparation des préoccupations. La programmation par objets (PPO) a été proposée comme un paradigme structurant les programmes en termes d'objets, lesquels renferment des données et des comportements.

La PPO est le paradigme de développement de fait dans beaucoup de domaines. Pourtant, il présente certains manques pour le découplage de composants et pour la modularisation de préoccupations transverses. La programmation par événements (PPE) et la programmation par aspects (PPA) sont deux paradigmes qui complètent la PPO avec les mécanismes pour résoudre ces manques.

Nous supposons que le lecteur a des connaissances sur la PPO. Dans la suite, nous donnons des éléments sur la PPE et la PPA.

### A.1.1 La programmation par événements

Le flot d'un programme à base d'événements est déterminé en identifiant des événements, en déclenchant des notifications d'événements et en réagissant à ces notifications. Dans un programme à base d'événements les composants intégrés communiquent au moyen de ces notifications [FMG02]. Les composants intéressés par certains événements y souscrivent de sorte que lors de l'occurrence de ces événements, les notifications nécessaires leur soient automatiquement livrées.

Un événement représente un changement dans le contexte d'un calcul qui est reflété comme un changement dans l'état d'un tel calcul. Simplement, il est possible de considérer un événement comme un changement dans l'état d'un calcul. En conséquence, les événements sont inhérents à tout programme informatique. Toutefois, tous les programmes ne peuvent pas être considérés comme suivant les principes du paradigme de la PPE. La contribution de la PPE est de structurer la manière dont les événements spécifiques sont identifiés dans un programme et la façon dont les informations sur l'occurrence de ces événements sont transmises à des composants logiciels intéressés par ces événements. Une approche basée sur des événements est une approche qui découple les sources d'événements de leurs destinations en adoptant un schéma d'*invocation implicite*.

La figure A.1 montre les parties principales d'un système événementiel. Un *gestionnaire d'événement* est le code à exécuter lorsqu'un événement se produit. La *notification d'un événement* correspond à un processus dans lequel tous les gestionnaires d'événements attachés à un événement sont invoqués. Un *déclencheur d'événements* est une expression qui produit les notifications d'événements.

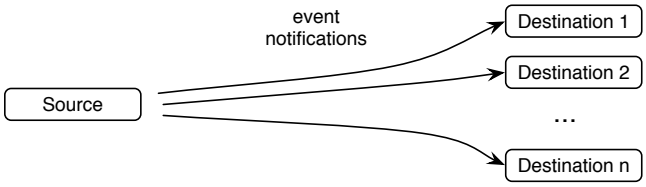


Figure A.1 – Conception à base d'événements.

Le support linguistique pour les événements correspond à l'inclusion de constructions spécialement dédiées à la PPE dans un langage de programmation. Ces constructions réduisent l'infrastructure nécessaire lors de l'utilisation des approches événementielles telles que le patron de conception *Observateur* et les JAVA *Listeners*. De façon générale, le support linguistique pour les événements est fourni sous la forme de deux constructions: un événement qui représente un changement d'état, qui dans certaines approches est juste une signature, et un gestionnaire d'événements, qui dans plusieurs approches est juste une méthode.

```
1 class Figure {
2  public event ChangeHandler changed;
3
4   void moveBy(int x, int y) {
5     /* change figure position */
6     if(changed != null)
7       changed(this);
8   } ...
9 }
```

Listing A.1: Un exemple d'événement en C#.

Le langage C# permet aux classes de définir explicitement des événements et de rester complètement ignorantes des observateurs potentiels de ces événements. Le langage fournit aux objets intéressés par ces événements des mécanismes pour effectuer les souscriptions correspondantes. Par exemple, le Listing A.1 montre une implémentation d'une classe `Figure` en C#. La classe `Figure` déclare explicitement un événement `changed` à la ligne 2, qui est déclenché comme un appel de méthode dans la méthode `moveBy` à la ligne 7. L'événement déclencheur produit une notification implicite de tous les objets intéressés par un tel événement, *i.e.* tous les objets qui ont déjà souscrit à l'événement.

En C# une fonction est attachée à un événement, en tant que gestionnaire de l'événement, en utilisant l'opérateur `+=`. De manière inverse, l'opérateur `-=` peut être utilisé pour détacher une fonction. Par exemple, le Listing A.2 attache la méthode `update` d'une instance de la classe `DrawingView` à l'événement `changed` d'une instance de la classe `Figure`. Notez que les fonctions sont, dans ce cas, des méthodes d'objets particuliers.

### A.1.2 Programmation par aspects

La PPA est le point de convergence d'un ensemble de technologies qui ont émergé pour répondre à l'incapacité des technologies des objets à atteindre une séparation des préoccupations à travers plus d'une dimension. À partir de la fin des années 90, la PPA a

```
1 class Test {
2   public static void Main() {
3     Figure figure = new Figure();
4     DrawingView view = new DrawingView();
5     figure.changed += new ChangeHandler(view.update); ...
6   }
7 }
8 class DrawingView {
9   public void update(Figure figure) {
10    repaint(figure.getArea());
11  } ...
12 }
```

Listing A.2: Registering to an event in C#.

été l'objet de nombreuses recherches et aujourd'hui elle couvre l'ensemble du processus de développement du logiciel.

**Modèle de points de jonction.** En général, les aspects sont invoqués sur des points bien définis dans l'exécution du programme appelés *points de jonction*. Les types possibles de points de jonction sont décrits dans un modèle de points de jonction. Les appels de méthode, les exécutions de méthode, les accès à des champs d'objet sont des exemples typiques de points de jonction.

**Modèle de points de coupure.** Les points de jonction qui sont pertinents pour un aspect sont sélectionnés à l'aide de prédicats appelés *points de coupure*. Dans la plupart des langages d'aspects, plus particulièrement en AspectJ, un point de coupure est une expression combinant un ensemble de prédicats dédiés qui raisonnent en termes du jeu complet des points de jonction possibles dans une application logicielle [BCRD08]. Dans AspectJ ces prédicats dédiés sont appelés *désignateurs de points de coupure*. Un point de coupure utilise ces prédicats pour sélectionner le sous-ensemble des points de jonction de l'application qui sont jugés pertinents pour l'aspect.

```
pointcut change(Figure figure):
         execution(* Figure.moveBy(..,..)) && this(figure)
```

Listing A.3: Point de coupure AspectJ.

Le Listing A.3 montre un point de coupure écrit en AspectJ. Il sélectionne les exécutions de la méthode `moveBy` des instances de `Figure`. Le désignateur de points de coupure `execution` signifie l'exécution d'une méthode et est utilisé conjointement avec un motif qui identifie la méthode. Des constructions similaires sont fournies par le langage AspectJ pour sélectionner les appels de méthodes et les accès aux champs, entre autres. En outre, une point de coupure peut obtenir des informations contextuelles sur le point de jonction à l'aide de désignateurs de points de coupure particuliers. Par exemple, le désignateur de points de coupure `this` est utilisé pour obtenir l'objet contextuel impliqué dans l'exécution de la méthode (celui qui est obtenu lors de l'utilisation de `this` dans le corps de la méthode).

La plupart des désignateurs de points de coupure d'ASPECTJ peuvent être liés à des endroits précis dans le code d'une application. Il est possible de dire que les points de coupure ASPECTJ sont statiques, car ils ne peuvent pas se référer à un contexte dynamique: variables ou méthodes de la structure dans laquelle ils sont définis. Même si ASPECTJ comprend un désignateur de points de coupure `if`, ce qui permet de tester des conditions, ce désignateur ne peut utiliser que des variables statiques, des méthodes statiques ou des variables déclarées dans le point de coupure.

**Greffon.** La fonctionnalité d'un aspect est mis en œuvre dans des morceaux de code appelés *greffons*. En général, un greffon lie un point de coupure et une action, le corps du greffon, exécutée lorsque le point de coupure sélectionne un point de jonction. Les greffons permettent également de définir l'emplacement de l'action: avant, après ou au lieu de l'exécution du point de jonction.

```
void around(Figure figure) : change(figure) {
  if(!figure.isReadOnly())
    proceed(figure);
}
```

Listing A.4: Greffon ASPECTJ `around`.

En particulier, les greffons étiquetés comme `around` sont exécutés à la place du point de jonction. Le mot-clé `proceed` offre ensuite la possibilité d'exécuter le point de jonction original à l'intérieur du corps du greffon. À titre d'exemple, le Listing A.4 montre un greffon, écrit en ASPECTJ, qui lie le point de coupure du Listing A.3 avec le code qu'exécute le point de jonction initial, par l'utilisation de `proceed`, uniquement lorsque la figure n'est pas en mode lecture seule.

**Définition des aspects et leur instanciation.** Les points de coupure et les greffons sont mis ensemble dans un module appelé *aspect*. Un aspect encapsule l'implémentation d'une préoccupation du logiciel.

La plupart des approches de programmation par aspects s'appuient sur une technique de décomposition asymétrique: les aspects sont définis comme un type particulier de modules qui sépare les préoccupations transversales. Les aspects sont équipés de points de coupures et de greffons et peuvent partager des éléments du langage de base dans lequel ils ont été embarqués tels que méthodes et champs. Dans les langages tels que ASPECTJ et ASPECTWERKZ, par exemple, les aspects sont définis de façon similaire aux classes standard, mais ils suivent des stratégies d'instanciation particulières. Contrairement aux classes, les aspects ne peuvent pas être explicitement instanciés. Un aspect est implicitement instancié par le langage au moment du chargement, par défaut comme un singleton. ASPECTJ fournit également une instanciation implicite par objet pour des aspects définis en utilisant `perthis(`*pointcut*`)` ou `pertarget(`*pointcut*`)`. Ces stratégies instancient un aspect par objet spécifié. Une stratégie `perthis(`*pointcut*`)` exclut les point de jonction qui ne sont pas sélectionnés par le point de coupure *pointcut* et associe à chaque nouvel *objet courant* (objet au sein duquel s'exécute le point de jonction) une nouvelle instance d'aspect. La stratégie `pertarget` fonctionne de la même manière en considérant l'objet *cible* du point de jonction (l'objet appelé dans le cas d'un appel de méthode) au lieu de l'objet courant.

```
1 aspect Aspect {
2   DrawingView view; List relevantFigures;
3
4   pointcut change(Figure figure):
5            execution(* Figure.moveBy(..)) && this(figure)
6
7   after(Figure figure) : change(figure) {
8     if(relevantFigures.contains(figure))
9       view.repaint(figure.getArea());
10  }
11 }
```

Listing A.5: Aspect AspectJ.

À titre d'exemple, le Listing A.5 définit un aspect AspectJ qui, étant donné une vue et une liste de figures pertinentes, met à jour la vue dès que l'une des figures change.

Les aspects AspectJ peuvent inclure des membres de classe standard, qui sont accessibles dans le code des greffons, *i.e.* le code d'un greffon est spécifique à l'instance. Par exemple, la ligne 8 du Listing A.5 utilise la variable d'instance `relevantFigures` pour tester si la figure liée par le point de coupure est une figure pertinente. Ce n'est cependant pas le cas pour les points de coupure, qui sont statiques. Le point de coupure de la ligne 4 observe l'exécution de toutes les figures de l'éditeur, pas seulement celles qui sont pertinentes. Même si AspectJ permet d'utiliser un désignateur de points de coupure `if` pour tester l'état du calcul lors de la sélection d'un point de jonction, il n'est pas possible de faire directement référence à des méthodes ou variables d'instance de l'aspect.

## A.2 Problématique

La conception des langages de programmation décrite par MacLennan [Mac95] met en avant des principes comme l'orthogonalité, la simplicité et la régularité. Le principe d'orthogonalité dicte que les fonctions indépendantes d'une langage doivent être contrôlées par des mécanismes indépendants. Selon le principe de simplicité un langage doit être aussi simple que possible. Il devrait y avoir un nombre minimum de concepts avec des règles simples pour leur combinaison. Le principe de régularité correspond à une utilisation cohérente et systématique des règles qui régissent la syntaxe et la sémantique de ces concepts.

La coexistence des paradigmes de programmation exige que les langages fournissent des mécanismes appropriés à chacun de ces paradigmes. C'est potentiellement une source de complexité en contradiction avec les principes ci-dessus. La faible intégration des différents paradigmes de programmation dans une application augmente sa complexité. Par exemple, l'utilisation intensive de *listeners* lors de la programmation des fonctionnalités événementielles dans langages comme Java augmente considérablement la taille des programmes et complique leur compréhension. La situation s'aggrave lorsque ils sont utilisés en combinaison avec les aspects qui nécessitent un ensemble complet de nouveaux concepts.

Cette situation appelle à une meilleure intégration des paradigmes tels que la PPE, la PPA et la PPO, afin de fournir des langages plus simples et plus réguliers, incluant tous les concepts nécessaires. Cela correspond à la problématique générale de cette thèse.

### A.2.1 La PPE et la PPO

La manière classique de programmer des applications événementielles avec les langages à objets traditionnels tels que JAVA consiste à utiliser des *callbacks* ou des *listeners*. Les gestionnaires d'événements sont encapsulés en tant qu'objets conformes à une interface connue, qui est invoquée lors de l'occurrence d'un événement. Certains langages à objets tels que C# ont inclut un support linguistique spécial pour les événements afin de réduire l'infrastructure nécessaire lors de l'utilisation de *callbacks*. Les événements sont définis en tant que membres des objets. Fait intéressant, les événements sont aussi conçus comme des membres d'instance dans les langages à objets tels que *e* [IEE08] et SystemVerilog [IEE09], focalisés sur la vérification de circuits. Cependant, aucun des langages mentionnés ci-dessus n'intègre véritablement les événements au sein de la PPO.

### A.2.2 La PPE et la PPA

La PPE et la PPA améliorent la séparation des préoccupations en réduisant le couplage entre les composants logiciels. Les deux paradigmes utilisent le même mécanisme: *l'invocation implicite*. Dans les systèmes événementiels, un gestionnaire d'événement programmé dans un composant est implicitement exécuté lors du déclenchement d'un événement dans un autre composant sans garder de dépendance explicite entre les deux composants. De même, dans les systèmes aspectuels, le comportement d'un aspect est implicitement invoqué dans l'implementation d'autres composants. Les développeurs de ces autres composants peuvent être largement ignorants des préoccupations transverses. En dépit du fait que la PPE et la PPA partagent le même mécanisme, il n'existe pas de langages offrant une intégration transparente de la gestion des événements et de l'action des greffons. PTOLEMY [RL08] a pris des mesures intéressantes dans cette direction, mais sa prise en compte des aspects reste très limitée.

### A.2.3 La PPA et la PPO

La PPA fournit une composition *asymétrique* des préoccupations, en distinguant les classes, lesquelles mettent en œuvre des préoccupations de base, et les aspects, lesquels mettent en œuvre des préoccupations transverses. Dans son incarnation la plus réussie, ASPECTJ, les aspects ressemblent beaucoup aux classes, mais contiennent deux types de membres nouveaux: points de coupure et greffons. Bien qu'un aspect ressemble beaucoup à une classe, les règles régissant sa sémantique statique et dynamique sont différentes. En particulier, il peut être étendu en tant que classe, mais de manière limitée et il ne peut pas être explicitement instancié. Cela ajoute beaucoup de complexité, en contradiction avec les principes de simplicité, de régularité et d'orthogonalité de MacLennan [Mac95]. D'un côté, il y a un certain nombre de nouvelles constructions. D'autre part, les principes de base pour travailler avec ces constructions (instanciation, héritage) ne sont pas réguliers. Les choix de conception correspondants peuvent en partie s'expliquer par la volonté de mieux saisir l'intention du programmeur et d'éviter les problèmes rencontrés lors de l'utilisation de la réflexion et des protocoles à méta-objets [KdRB91], une approche plus globale et intégrée mais difficile à maîtriser. Ils peuvent aussi être en partie expliqués par des considérations de performance.

Pour résoudre la problématique de cette thèse nous commençons avec la présentation de notre modèle de programmation, lequel est ensuite utilisé pour mettre en œuvre l'intégration des paradigmes. Pour illustrer notre modèle nous utilisons la syntaxe du langage

EJava qui est un des deux langages qui concrétisent notre modèle.

## A.3 Un modèle d'événements déclaratifs et polymorphes

### A.3.1 Motivation

Des constructions spécifiques aux événements ont été introduites dans les langages à objets traditionnels tels que C# afin de faciliter la mise en œuvre des applications réactives. Cependant, ces constructions ne fournissent pas les moyens appropriés pour définir des événements comme compositions d'autres événements. Les grandes applications ont généralement besoin d'événements composites. Dans les applications en couches, par exemple, les couches supérieures peuvent définir des événements en termes d'événements prévus dans les couches inférieures. Il est naturel pour les programmeurs de penser à des événements composites sous forme de combinaisons logiques d'événements plus primitifs. Malheureusement, dans les langages comme C# il y a un fossé entre la manière dont les événements composites sont écrits et l'intention finale des programmeurs. Les événements composites sont exprimés à l'aide d'événements impératifs et ne peuvent pas être définis déclarativement.

Cette thèse introduit un modèle de programmation avec événements déclaratifs. Le modèle est utilisé ensuite pour intégrer les différents paradigmes de programmation introduits précédemment.

### A.3.2 Événements et gestionnaires propriétés des objets

Nous proposons un modèle objet simple avec méthodes et champs et avec deux nouveaux membres de classe: événements et gestionnaires d'événements. Un événement peut être déclaré comme abstrait ou défini comme étant impératif ou déclaratif. Un événement abstrait ajourne sa définition aux sous-classes. Un événement impératif est utilisé pour déclencher explicitement l'occurrence d'un événement semblable aux événements de C#. Un événement déclaratif combine d'autres événements comme l'autorise le langage *e* [IEE08]. Un événement déclaratif est un événement composite (à moins qu'il n'associe tout simplement un nouveau nom à un événement primitif), tandis qu'un événement impératif est un événement primitif.

Les événements sont des propriétés des objets. L'accès externe à un événement doit être qualifié par une expression retournant l'objet propriétaire de l'événement. Cela signifie que les événements sont à liaison tardive puisque leur valeur concrète dépend de l'évaluation de l'expression correspondante à l'exécution. L'accès interne à un événement est qualifié avec `this`. Les événements sont accédés lors de la définition des événements déclaratifs et lors du déclenchement des événements impératifs. Dans notre modèle tout objet peut déclencher un événement impératif d'un autre objet.

Un gestionnaire d'événement est défini en indiquant la signature d'un événement (c'est-à-dire son nom et sa liste de paramètres formels) et un corps. Le gestionnaire d'événement est défini comme étant lié à l'événement correspondant, de sorte qu'il est exécuté lorsque l'événement survient. Afin de fournir des gestionnaires d'événements avec une identité, un gestionnaire d'événement ne peut qu'être lié à un événement défini dans la même classe ou dans une superclasse. Ainsi, l'identité d'un gestionnaire d'événement correspond à la signature de l'événement correspondant. En conséquence, une classe peut avoir des paires

(événement, gestionnaire) partageant la même signature. Il n'y a aucune ambiguïté puisque les événements et les gestionnaires d'événements sont utilisés dans des contextes différents.

Les gestionnaires d'événements sont également des propriétés des objets. Cependant, un gestionnaire d'événement est uniquement accessible de manière explicite dans la hiérarchie de classe à des fins d'héritage. Le fait que les événements et les gestionnaires sont des propriétés des objets implique également qu'ils ont accès au contexte de leur objet propriétaire, tels que des références à des champs ou des méthodes. En outre, les événements et les gestionnaires d'événements peuvent être hérités, comme les champs et les méthodes.

### A.3.3 Événements impératifs et occurrences d'événements

Comme il a été dit précédemment, notre modèle permet le déclenchement externe: tout objet peut déclencher un événement impératif d'un autre objet. Des occurrences d'événements impératifs sont déclenchés pour signaler des changements dans l'état d'une application. Nous utilisons le terme *occurrence de l'événement* pour l'utilisation d'un événement à l'exécution, alors que nous utilisons le terme *événement* pour la construction syntaxique correspondante. Quand nous disons que l'événement impératif est déclenché, nous voulons dire qu'une occurrence de l'événement impératif est déclenchée. Un événement impératif est déclenché lors de l'exécution en fournissant à chaque paramètre déclaré une valeur. L'exécution d'un programme produit une séquence d'occurrences d'événements indiquant plusieurs changements d'état.

```
1 abstract class Figure {
2   abstract event changed(Figure source);
3 }
4 class AbstractFigure extends Figure {
5   event changed(Figure source);
6
7   void moveBy(int x, int y) {
8     /* implementation*/
9     changed(this);
10  }
11 }
```

Listing A.6: Définition d'un événement impératif `changed` et son déclenchement.

Les événements abstraits et les événements impératifs se composent d'un nom et de zéro, un ou plusieurs paramètres formels. À titre d'exemple, regardons le Listing A.6 écrit en EJAVA, l'un des langages qui concrétisent notre modèle. Un événement `changed` est déclaré abstrait dans une classe `Figure` à la ligne 2. En effet, dans notre modèle les événements abstraits ont du sens car les sous-classes peuvent prévoir soit une définition déclarative soit une définition impérative de l'événement. La classe `AbstractFigure` définit l'événement comme impératif à la ligne 5 et l'utilise pour déclencher une occurrence à l'intérieur de la méthode `moveBy` à la ligne 9. L'objet `this` est fourni comme paramètre de l'événement.

### A.3.4 Événements déclaratifs

Un événement déclaratif est un sélecteur d'occurrences d'événements, une fonction qui sélectionne les occurrences d'événements qui informent du changement d'état représenté

par l'événement. Un événement impératif est également un sélecteur d'occurrences d'événements, mais il choisit ses propres occurrences. Les occurrences d'un événement déclaratif sont les occurrences que l'événement déclaratif sélectionne. On dit qu'un événement se produit lorsque le sélecteur a sélectionné une occurrence de l'événement.

Les événements déclaratifs se composent d'un nom, de zéro, un ou plusieurs paramètres formels et d'une expression d'événement. Cette expression peut être un accès à un événement défini ailleurs, une disjonction de deux expressions d'événements, ou une conjonction de deux expressions d'événements, entre autres. Une expression d'événement est également un sélecteur d'occurrences d'événements, de sorte qu'un événement déclaratif sélectionne les occurrences que son expression d'événement sélectionne. Les différentes expressions d'événements sont décrites comme suit:

**Accès à un événement.** Un accès à un événement permet d'accéder à un événement d'un objet. C'est une expression d'événement primitive qui sélectionne les occurrences que l'événement accédé sélectionne.

```
1 class CompositeFigure extends Figure {
2   Figure fig1, fig2;
3
4   event changed(Figure source) =
5       fig1.changed(source) || fig2.changed(source);
6 }
```

Listing A.7: Un événement défini en termes de deux autres événements.

**Disjonction.** La disjonction de deux expressions d'événement sélectionne les occurrences sélectionnées par l'une ou l'autre des expressions. À titre d'exemple, le Listing A.7 montre comment l'événement `changed` d'une figure composite est défini comme une disjonction des événements `changed` de ses enfants. Ainsi, chaque fois que l'événement `changed` d'un enfant se produit, l'événement `changed` de la figure composite se produit également.

```
1 class DrawingView {
2   Figure drawing;
3   Boolean isVisible;
4
5   event invalidated(Figure changedFig) =
6       drawing.changed(changedFig) when isVisible;
7
8   invalidated(Figure changedFig) {
9     /* implementation */
10    next;
11  }
12 }
```

Listing A.8: Un événement défini en termes d'un autre événement et raffiné avec une condition.

**Raffinement.** Une expression d'événement peut représenter le raffinement d'une autre expression d'événement avec une condition. Elle sélectionne les occurrences choisies par l'expression d'événement donnée qui ont lieu lorsque que la condition est vraie. À titre d'exemple, le Listing A.8 montre comment l'événement `invalidated` d'une vue est défini à la ligne 5 comme un raffinement de l'événement `changed` d'un dessin avec la condition `isVisible`. Ainsi, chaque fois que l'événement `changed` d'un dessin se produit, l'événement `invalidated` se produit également lorsque la vue est visible.

**Paramètres.** Comme il a été vu précédemment, la signature d'un événement peut déclarer des paramètres. Un événement impératif est déclenché lors de l'exécution en fournissant à chaque paramètre déclaré une valeur. Ces valeurs peuvent être saisies et liées à des paramètres d'événements déclaratifs, concrètement dans un accès d'événement, et se propager à d'autres événements déclaratifs. Un gestionnaire d'événement attaché à un événement peut utiliser les paramètres liés pour exécuter ses actions. Par exemple, la ligne 9 du Listing A.6 déclenche l'événement `changed` d'une figure en fournissant l'objet `this` comme paramètre. Cet objet est capturé et lié au paramètre `source` lors de l'accès aux événements des enfants de la figure composite dans la définition de l'événement déclaratif `changed` de la ligne 5 du Listing A.7. La propagation du paramètre continue jusqu'à atteindre le gestionnaire d'événement `invalidated` du Listing A.8, qui utilise le paramètre lié pour repeindre la surface de la figure correspondante.

Un événement lie les valeurs de ses paramètres quand il choisit une occurrence. Les paramètres peuvent jouer un rôle dans le processus de sélection, car ils peuvent être utilisés dans la condition d'un raffinement. Les paramètres d'un événement déclaratif sont implicitement liés lors de l'accès à d'autres événements. En outre, le mot-clé `with` permet de lier explicitement un paramètre avec une valeur personnalisée, par exemple le résultat d'un appel de fonction.

En général, les conditions d'une expression d'événement peuvent utiliser les paramètres liés par des événements déclaratifs et tout champ ou méthode de l'objet courant. De plus, notre modèle permet de déclarer des variables d'événements locales. Ces variables sont locales à une expression d'événement et peuvent être liées et utilisées dans des conditions à l'intérieur de l'expression d'événement. Elles ne sont pas visibles à l'extérieur.

```
1 event newRedCars(List redCars) =
2   (List cars)
3     newCars(cars)
4     with redCars = filterRedCars(cars)
5     when redCars.isNotEmpty())
```

Listing A.9: Exemple de liaison explicite et variable d'événement locale.

Le Listing A.9 montre l'exemple d'une variable d'événement locale explicitement liée. Il définit un événement déclaratif `newRedCars` en termes d'un événement impératif `newCars`. L'événement `newCars` représente la disponibilité d'une liste de voitures neuves et l'événement `newRedCars` la disponibilité d'une liste de voitures rouges neuves. L'événement `newRedCars` est défini par l'accès à `newCars` à la ligne 3. La liste des voitures fournies par `newCars` est liée à une variable locale `cars` définie à la ligne 2. Une liaison explicite est utilisée à la ligne 4 pour lier la variable `redCars` avec le résultat du filtrage des voitures rouges depuis la liste `cars`. L'événement `newRedCars` se produit lorsque l'événement

`newCars` survient et la liste des voitures rouges n'est pas vide, ce qui est testé à la ligne 5.

```
1 event newHotCars(List redCars, List cheapCars) =
2   newRedCars(redCars) && newCheapCars(cheapCars);
3
4 event newHotCars(List cars) =
5     (List redCars, List cheapCars)
6         newHotCars(redCars, cheapCars)
7         with cars = intersect(redCars, cheapCars);
```

Listing A.10: Exemple de conjonction d'événements.

**Conjonction.** Dans notre modèle plusieurs événements peuvent se produire simultanément, car ils peuvent sélectionner les mêmes occurrences. Ainsi, il est logique d'offrir, en dehors de la disjonction, la conjonction d'événements. La conjonction de deux expressions d'événements sélectionne les occurrences que les deux expressions sélectionnent. À titre d'exemple, le Listing A.10 définit un événement déclaratif `newHotCars`, à la ligne 1, comme la conjonction des événements `newRedCars` et `newCheapCars`. L'événement `newRedCars` est celui présenté plus tôt, soit la disponibilité d'une liste de voitures rouges neuves. L'événement `newCheapCars` représente la disponibilité d'une liste de voitures neuves bon marché et il est défini en fonction d'un événement impératif `newCars` d'une façon analogue à `newRedCars`. Lorsque l'événement `newCars` se produit, tant `newRedCars` que `newCheapCars` peuvent arriver simultanément, chacun filtrant la liste correcte de voitures. L'événement `newHotCars`, défini comme une conjonction de ces événements, représente la disponibilité de voitures neuves dont certaines sont rouges et les autres sont bon marché. L'événement définit deux paramètres qui capturent chaque type de voitures. L'événement `newHotCars` de la ligne 4 utilise l'événement `newHotCars` de la ligne 1 pour fournir une liste unique de voitures, qui sont à la fois rouges et bon marché. Dans l'exemple précédent, l'événement `newHotCars` aurait pu être défini directement en fonction de l'événement `newCars` sans avoir besoin d'une conjonction. Cependant, nous illustrons ici la réutilisation d'événements existants.

```
event changed(Figure source) = some(figures).changed(source);
```

Listing A.11: Exemple de quantification sur les événements d'une liste d'objets.

**Quantification.** Jusqu'ici nous avons considéré des objets liés par des relations un-à-un, mais un objet peut également être lié à une collection d'autres objets et avoir besoin d'observer globalement les occurrences d'un événement spécifique défini par l'ensemble de ces objets. Notre modèle permet une forme de quantification existentielle sur une liste d'objets. Un événement peut être défini comme une disjonction d'expressions d'événements toutes liées au même nom d'événement en utilisant une expression de quantification. Par exemple, le Listing A.11 montre comment un changement d'une figure, qui est une composition d'une liste de figures (variables `figures`), peut être décrit en termes des événements observés sur ses enfants. Il définit que l'événement `changed` de la figure composite se produit lorsque l'événement `changed` d'une figure composée survient.

### A.3.5 Gestionnaires d'événements

Un gestionnaire d'événement est défini en indiquant la signature d'un événement (c'est-à-dire son nom et sa liste de paramètres formels) et de zéro, une ou plusieurs expressions implémentant le corps du gestionnaire d'événement. Ces expressions sont soit des expressions standard du corps d'une méthode, soit les expressions *Trigger* ou *Next*, utilisées respectivement pour déclencher un événement et pour composer des gestionnaires d'événements.

## A.4 Intégration des paradigmes

### A.4.1 Unification des événements impératifs et des appels de méthode

```
class Callee extends Superclass {
  T m(){ methodBody }
}
```

Listing A.12: Une méthode `m` définie dans une classe `Callee`.

Dans la métaphore de la PPO les objets communiquent en interchangeant des messages. Un objet envoie un message à un autre, et ce dernier réagit en recherchant une méthode et en exécutant le corps de la méthode. Par exemple, la méthode `m` de la classe `Callee` du Listing A.12 définit un morceau de code *methodBody* à exécuter en réponse à un appel à `m`.

Les gestionnaires d'événements et les méthodes jouent un rôle similaire. Alors qu'un gestionnaire est exécuté au déclenchement d'un événement, une méthode (le corps) est exécutée en réponse à un appel de méthode. Qu'en est-il d'offrir un traitement uniforme des deux ?

#### A.4.1.1 Principe

```
class Callee extends Superclass {
  event T m();
  T m() { methodBody }
}
```

Listing A.13: La méthode `m` de la class `Callee` du Listing A.12 désucrée.

Notre modèle fournit un traitement uniforme de la gestion d'événements et de l'exécution de méthodes par une modélisation des appels de méthodes comme le déclenchement d'événements impératifs. La syntaxe standard des méthodes est fournie comme un raccourci pour définir à la fois un événement impératif et un gestionnaire d'événement pour cet événement. Le gestionnaire d'événement contient le corps de la méthode. Concrètement, la définition de classe du Listing A.12 est du sucre syntaxique pour le Listing A.13. Ainsi, l'appel d'une méthode d'un objet est équivalent au déclenchement de l'événement impératif correspondant. Le gestionnaire contenant le corps de la méthode est exécuté dans le processus de gestion de l'événement déclenché. En d'autres termes, le corps du gestionnaire d'événement gère l'appel de la méthode correspondante.

```
abstract class AbstractCompositeFigure
       extends CompositeFigure
{
  void draw(Graphics2D g) {
    /* method body */
  }
}
```

```
abstract class AbstractCompositeFigure
      extends CompositeFigure
{
  event draw(Graphics2D g);
  draw(Graphics2D g) => {
    /* method body */
  }
}
```

Listing A.14: Syntaxe standard de méthode à gauche et syntaxe désucrée à droite.

Le Listing A.14 montre une autre exemple. Le côté gauche du Listing A.14 montre la classe `AbstractCompositeFigure` définissant une méthode `draw`. Cette définition de la classe est équivalente à celle sur le côté droit. Cette dernière définit un événement impératif `draw` et un gestionnaire d'événement `draw` contenant le corps de la méthode. Quand l'événement `draw` se produit le corps de la méthode est exécuté avec le code de tout le reste des gestionnaires de l'événement. Comme il a déjà été expliqué, un appel de méthode est équivalente au déclenchement de l'événement impératif correspondant et l'exécution de la méthode associée est équivalente à la gestion de l'événement.

#### A.4.1.2 Événements avec type de retour

Un gestionnaire d'événement est défini avec un type de retour. Il retourne la valeur qui résulte de l'évaluation de la dernière expression de son corps. Un événement est également défini avec un type de retour, ce qui indique que le déclenchement d'un événement impératif (ou un appel de méthode) renvoie la valeur retournée par le premier gestionnaire exécuté. Un gestionnaire d'événement peut composer sa valeur de retour avec la valeur retournée par le reste des gestionnaires d'événements. Cette valeur peut être obtenue en utilisant l'expression `next`. L'exécution de `next` retourne la valeur retournée par le gestionnaire suivant dans la liste des gestionnaires à exécuter.

```
1  class Agenda {
2    event List meeting(Date date);
3    List meeting(Date date) {
4      return new ArrayList();
5    }
6  }
7
8  class Employee extends AbstractEmployee {
9    Agenda agenda;
10
11   List meeting(Date date) = agenda.meeting(date);
12   List meeting(Date date) { return next.add(this); }
13 }
```

Listing A.15: Utilité des événements avec une valeur de retour.

À titre d'exemple, le Listing A.15 met en œuvre un agenda avec un événement `meeting` qui représente la planification d'une réunion à une certaine date. Plusieurs employés peuvent observer cet événement et confirmer leur présence. L'événement `meeting` renvoie la liste

des employés qui peuvent participer à la réunion. Le gestionnaire d'un employé est censé indiquer si l'employé peut participer à la réunion ou pas. L'alternative a été omise pour simplifier. À la ligne 12, le gestionnaire utilise l'expression `next` pour obtenir la liste renvoyée par les autres gestionnaires et pour ajouter son objet propriétaire dans la liste. La méthode `add` d'une liste retourne la liste résultante.

Un gestionnaire d'événement directement associé à un événement impératif est toujours le dernier gestionnaire à être exécuté lorsque l'événement est déclenché. En particulier, c'est le cas des gestionnaires d'événements implicitement définis en utilisant la syntaxe des méthodes. À la ligne 3 du Listing A.15, la classe `Agenda` définit un gestionnaire d'événement qui retourne une liste préliminaire vide. Le fait que ce gestionnaire est le dernier gestionnaire à être exécuté fait en sorte que les gestionnaires précédents obtiennent une liste initiale. Le gestionnaire d'événement de la ligne 3 est un gestionnaire par défaut. Notez que l'événement `meeting` et le gestionnaire d'événement `meeting` auraient pu être définis en utilisant une méthode.

Le lecteur peut se demander ce qui se passe s'il n'y a pas de gestionnaire associé à un événement impératif défini avec un type de retour, en particulier, ce qui arrive si une valeur est attendue lors du déclenchement de l'événement. Dans un langage comme JAVA cela pourrait être un problème, par exemple, lorsque l'événement est défini comme retournant un type primitif comme `int`. Il est nécessaire d'imposer qu'une classe définissant un événement impératif avec un type de retour (distinct de `void`) fournisse un gestionnaire d'événement par défaut comme à la ligne 3 du Listing A.15. En d'autres termes, dans ce cas nous devrions être toujours en mesure d'utiliser une syntaxe de méthode.

#### A.4.1.3 Les événements et les gestionnaires comme membres des objets

Les trois principes suivants, inclus dans notre modèle, assurent un traitement régulier des événements et des gestionnaires d'événements par rapport aux autres membres des objets.

**Ajouts d'événements et de leur gestionnaires comme des propriétés polymorphes des objets.** Tout d'abord, l'accès à un événement doit être qualifié avec une expression retournant l'objet propriétaire de l'événement. Deuxièmement, il est possible de faire référence à un événement sans avoir une connaissance de sa valeur concrète, qui dépend de l'objet retourné à l'exécution par l'expression de qualification de l'événement. Troisièmement, la définition d'un événement peut faire référence aux membres de son instance propriétaire et a donc accès au contexte complet de l'exécution. Les mêmes remarques s'appliquent aux gestionnaires d'événements.

**Support pour l'héritage des événements et des gestionnaires d'événements.** Les événements et les gestionnaires d'événements d'une classe sont hérités par les sous-classes. De même que pour les méthodes, les sous-classes peuvent remplacer les événements et les gestionnaires d'événements définis dans la superclasse. Lors de la redéfinition d'un événement ou gestionnaire d'événement, une expression `super` est disponible pour faire référence à la définition de l'événement et du gestionnaire d'événement fournie dans la superclasse.

Dans le contexte d'un événement, un appel à `super` est considéré comme une expression d'événement qui rend possible l'accès à la définition d'un événement de la superclasse. Sa sémantique est similaire à la sémantique d'un accès à un événement standard. Un

```
class Employee2 extends Employee {
  Boolean isAvailable;,

  event List meeting(Date date) = super.meeting(date) when isAvailable;
}
```

Listing A.16: Définition d'un événement déclaratif utilisant un appel à `super`.

appel à `super` peut être composé avec d'autres expressions d'événements d'une manière standard. Par exemple, le Listing A.16 montre une sous-classe de la classe `Employee` du Listing A.15 qui redéfinit l'événement `meeting` en attachant une condition à l'événement de la superclasse.

```
1 class LabeledConnectionFigure extends ConnectionFigure {
2   transform(AffineTransform tx) => {
3     super.transform(tx)
4     /* rest of the implementation */
5   }
6   ...
7 }
```

Listing A.17: Héritage des gestionnaires d'événements.

L'héritage des gestionnaires d'événements est équivalent à l'héritage des méthodes. Les gestionnaires d'événements ont une identité donnée par le nom de l'événement qu'ils manipulent. Cela permet de les remplacer dans les sous-classes comme dans le Listing A.17. La classe `LabeledLineConnectionFigure` remplace le gestionnaire `transform` défini dans sa superclasse. Le gestionnaire `transform` est en effet un gestionnaire lié à l'événement impératif `transform` défini dans la hiérarchie de la figure. Cette conception fait que la classe a deux membres de même nom. Aucune ambiguïté n'est produite parce que ces membres sont accessibles dans des contextes différents. Par exemple, la déclaration `super.transform(tx)` à la ligne 3 fait référence au gestionnaire `transform` défini dans la superclasse et elle ne correspond pas à un déclenchement d'événement. La même expression sans le mot clé `super` serait interprétée comme un déclenchement de l'événement `transform`.

**Événements et gestionnaires d'événements abstraits.** Les événements et les gestionnaires d'événements peuvent être déclarés abstraits avec le mot-clé `abstract`, et ainsi différer leur définition aux sous-classes.

L'importance des événements abstraits est que le comportement réactif d'une application peut être entièrement programmé sans aucune connaissance sur la manière dont les événements vont être concrètement définis. À un stade ultérieur, les différentes versions d'un logiciel peuvent être facilement instanciées en fournissant différentes définitions d'événements adaptées à chaque situation, sans modifier le comportement réactif déjà programmé. À titre d'exemple, la classe `Figure` du Listing A.6 définit un événement abstrait `changed` représentant un changement dans une figure, comme un mouvement, un changement dans sa taille, etc. L'événement `changed`, comme les méthodes abstraites de la classe, n'a pas de définition. Les définitions des membres abstraits sont reportées à une étape distincte dans laquelle des paramètres spécifiques peuvent fournir les détails nécessaires

pour appliquer ces définitions.

```
1 class AbstractEmployee {
2   Agenda agenda;
3
4   event List meeting(Date date) = agenda.meeting(date);
5   abstract List meeting(Date date);
6 }
```

Listing A.18: Gestionnaire d'événement abstrait.

Un gestionnaire d'événement abstrait oblige les sous-classes concrètes à implementer l'événement associé. Cela facilite la compréhension des relations entre les objets de l'application. À titre d'exemple, le Listing A.18 montre une classe abstraite représentant des salariés. La classe définit un gestionnaire abstrait à la ligne 5 pour l'événement `meeting` de la ligne 4. Puisque cet événement fait référence à l'événement `meeting` d'un agenda, le gestionnaire d'événement indique d'une manière appropriée qu'un employé doit gérer l'événement de l'agenda.

### A.4.2 Unification de la gestion des événements et de l'action des greffons

Dans notre modèle, une méthode est donc du sucre syntaxique pour la définition d'un événement impératif et d'un gestionnaire d'événement associé. Ce gestionnaire contient le corps de la méthode. Par conséquent, les greffons peuvent être simplement modélisés comme des gestionnaires d'événements gérant les occurrences du même événement impératif. La sémantique du modèle assure que le gestionnaire d'événement mis en place par une méthode est le dernier gestionnaire d'événement à être exécuté lorsque l'événement est déclenché. Ainsi, le contrôle sera toujours donné aux greffons avant qu'il ne soit donné aux corps de la méthode. Les greffons peuvent contrôler, de cette manière, l'exécution du corps de la méthode de la même manière qu'un greffon standard contrôle son point de jonction. Nous allons décrire ce point plus en détail.

```
aspect Aspect {
  pointcut m() = call(void Callee.m());

  void around(): m() {
    /* before code */ proceed(); /* after code */
  }
}
```

Listing A.19: Un aspect ASPECTJ appliqué à l'appel de la méthode `m` d'une classe `Callee`.

#### A.4.2.1 Principe

**Les aspects en ASPECTJ.** Étant donné une classe `Callee` avec une méthode `m`, l'aspect ASPECTJ du Listing A.19 définit un greffon `around` dont l'exécution remplace l'appel à la méthode `m` d'une telle classe. L'instruction `proceed` permet d'exécuter le point de jonction au sein du greffon, si nécessaire. En d'autres termes, un greffon `around` prend le contrôle

de l'exécution du point de jonction, ce qui empêche le programme de base de l'exécuter lorsque `proceed` n'est pas invoquée.

Si plusieurs greffons `around` ont été définis pour un appel à `m`, ils sont composés en utilisant `proceed`. Lorsque le premier greffon est exécuté, un appel à `proceed` exécute le prochain greffon en son sein. Le greffon suivant fait la même chose, et ainsi de suite. Lorsque le dernier greffon est exécuté, un appel à `proceed` exécute le point de jonction. Si un greffon n'appelle pas `proceed`, alors le reste des greffons n'est pas exécuté, le point de jonction non plus.

```
class Aspect {
  Callee callee;

  event void m() = callee.m();

  void m() => {
    /*beforeCode*/
    next;
    /*afterCode*/
])
```

Listing A.20: Un aspect avec un événement et un gestionnaire.

**Les aspects en EJAVA** Du fait de l'équivalence entre définition de méthode et définition simultanée d'un événement et de son gestionnaire, une classe peut jouer le rôle d'un aspect en définissant un gestionnaire pour l'appel de méthode à observer. Ce gestionnaire se comporte comme un greffon. L'aspect ASPECTJ du Listing A.19 peut être exprimé comme dans le Listing A.20. L'aspect est implementé comme une classe avec un champ `callee`. Le point de coupure est implementé comme un événement déclaratif `m` accédant l'événement `m` de `callee` (l'appel de méthode). Le greffon correspondant est programmé comme un gestionnaire de l'événement `m`. En conséquence, deux gestionnaires d'événements sont liés à un même événement: celui du Listing A.13, contenant le corps de la méthode, et celui du Listing A.20, contenant le corps du greffon. L'effet de l'aspect ASPECTJ du Listing A.19 est obtenu parce que le gestionnaire programmé dans l'aspect est exécuté en premier et donc contrôle l'exécution du corps de la méthode. Ceci est dû au fait qu'un gestionnaire d'événement associé directement à un événement impératif est toujours le dernier gestionnaire à être exécuté lorsque l'événement survient. L'expression `next` dans le gestionnaire d'événement de l'aspect exécute le gestionnaire d'événement suivant, à savoir le corps de la méthode, se comportant comme `proceed` dans l'exemple ASPECTJ.

#### A.4.2.2 Événements *after* et modèle de points de jonction de Masuhara

Notre modèle est compatible avec le modèle de points de jonction de Masuhara et al. [MEY06], c'est-à-dire que les points de jonction (événements) sont des points d'exécution atomiques. Le modèle de Masuhara est plus fin que le modèle de points de jonction d'ASPECTJ, qui considère un point de jonction en tant que région. Par exemple, cette région comprend l'exécution du corps de la méthode dans le cas d'un point de jonction `call` (voir la figure A.2).

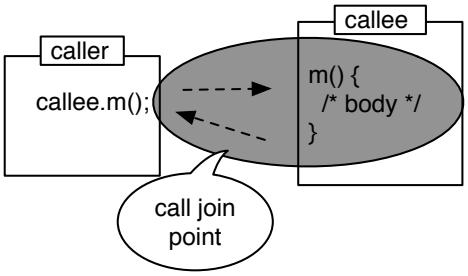


Figure A.2 – Modèle de point de jonction d'ASPECTJ.

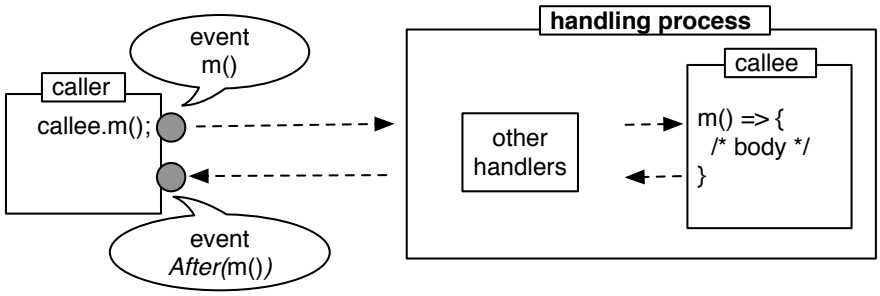


Figure A.3 – Modèle de point de jonction d'EJAVA.

Dans notre modèle, un appel de méthode correspond a une opération atomique: le déclenchement de l'événement impératif correspondant. La figure A.3 illustre cette opération par le petit cercle gris du haut. Cette opération atomique est similaire au point de jonction `call` du modèle de Masuhara et al. [MEY06]. En réaction à cette opération, un processus de traitement est démarré. Tous les gestionnaires sont exécutés, le dernier correspondant au corps de la méthode.

Notez dans la figure A.3, un deuxième cercle gris sous le précédent. Ce cercle représente un événement *after* `m` qui se déclenche une fois que l'événement `m` a été géré. Cet événement est appelé un événement *after* et est équivalent au point de jonction *reception* du modèle de Masuhara et al. [MEY06]. Notre modèle introduit des événements *after* implicitement pour être en mesure de notifier qu'un événement a été géré. Pour chaque définition de l'événement, l'événement *after* associé est mis à disposition. Cet événement peut être utilisé dans la définition d'autres événements en utilisant le mot-clé `after`. Les événements *after* sont utiles lorsque le traitement d'un événement introduit des changements et que réagir à ces changements est nécessaire. Par exemple, un événement *after* peut être utilisé pour définir l'événement `changed` d'une figure, déclenché après la gestion d'événements tels que `moveBy`.

### A.4.3 Aspects asymétriques flexibles

L'intégration de la PPA et la PPO en EJAVA permet la mise en œuvre d'aspects asymétriques qui sont plus flexibles que les aspects asymétrique de langages telles qu'ASPECTJ. Par flexibilité nous faisons référence à la possibilité d'instancier librement les aspects (en fait, les classes faisant fonction d'aspect) et à l'intégration des fonctionnalités de la PPA discutées précédemment.

Voyons comment EJAVA permet de mettre en œuvre le patron de conception *Média-*

*teur* [GHJV94] d'une manière vraiment modulaire. Nous illustrons cela en utilisant un exemple tiré de la présentation de Classpects [RS09]. Nous avons choisi cet exemple parce qu'il a déjà servi à illustrer les avantages de l'unification des classes et des aspects. Nous illustrons l'expressivité d'EJava pour programmer un tel exemple.

L'exemple se compose d'un système intégrant deux types de composants: capteurs de mouvement et de température et appareils photo. Un appareil photo peut être relié à plusieurs capteurs et un capteur à plusieurs appareils photo. Quand un certain capteur détecte un signal chaque appareil photo lié prend une photo. Une fois la photos prise, chaque appareil réinitialise le capteur afin qu'il puisse à nouveau capter un mouvement ou un changement de température.

```
class Sensor {
  void movementDetect() { ... }
  void temperatureDetect() { ... }
  void reset() { ... }
}
```

```
class Camera {
  void click() { ... }
}
```

Listing A.21: Classes implémentant des capteurs et des appareils photo.

L'implémentation de ce système en Classpects, présentée dans [RS09], illustre la possibilité d'utiliser le patron de conception *Médiateur* [GHJV94]. Une telle solution considère les classes `Sensor` et `Caméra` du Listing A.21, mettant respectivement en œuvre le code du capteur et de l'appareil photo.

```
class Mediator {
  Sensor s; Camera c;

  Mediator(Sensor s, Camera c) {
    this.s = s; this.c = c;
    addObject(s); addObject(c);
  }

  after(): execution(void Sensor.movementDetect()) ||
           execution(void Sensor.temperatureDetect()): onDetected();
  after(): execution(void Camera.click()): onClicked();

  onDetected() { c.click(); }
  onClicked()  { s.reset(); }
}
```

Listing A.22: Implémentation d'un médiateur entre un capteur et un appareil photo en Classpects.

La classe `Mediator` du Listing A.22 modularise la relation entre un appareil photo et un capteur en Classpects. La classe est instanciée pour un capteur et un appareil photo. Elle capte l'exécution des méthodes `movementDetect` ou `temperatureDetect` du capteur et déclenche l'appareil photo. De façon analogue, elle capte les exécutions de la méthode `click` sur l'appareil et réinitialise le capteur. La conception à l'aide d'un médiateur est un exemple d'un style de programmation à la AspectJ, maintenant l'ignorance du code de base vis-à-vis des aspects et l'asymétrie des aspects. L'appareil photo et le capteur peuvent être considérés comme faisant partie du programme de base, tandis que le médiateur peut

être considéré comme l'aspect. Les avantages d'une unification des classes et des aspects sont évidentes puisque ces médiateurs peuvent être instanciés d'une manière flexible pour représenter chaque relation individuelle entre plusieurs capteurs et appareils. En ASPECTJ, le mécanisme d'instanciation doit être simulé par le programmeur au sein d'un aspect singleton.

```
class Mediator {
  Sensor s; Camera c;

  Mediator(Sensor s, Camera c) {
    this.s = s; this.c = c;
  }

  event onDetected() = after(s.movementDetect() || s.temperatureDetect());
  event onClicked()  = after(c.click());

  onDetect() => { c.click(); next; }
  onClick()  => { s.reset(); next; }
}
```

Listing A.23: Implémentation d'un médiateur entre un capteur et un appareil photo en EJAVA.

EJAVA partage avec CLASSPECTS l'unification des classes et des aspects qui permet de programmer des aspects asymétriques flexibles tels que des médiateurs. Le Listing A.23 montre l'implémentation de l'exemple précédent en EJAVA. Comme le médiateur de CLASSPECTS, le médiateur d'EJAVA est instancié pour un capteur et un appareil photo. Il observe les événements du capteur afin de déclencher l'appareil photo, en même temps il observe les événements de l'appareil photo afin de réinitialiser le capteur. Toutefois, EJAVA est plus souple que CLASSPECTS. D'abord, EJAVA permet de définir des événements plus simplement avec une portée plus précise. Deuxièmement, il permet une conception véritablement modulaire en définissant des événements du côté de la source. Ces caractéristiques sont définies ci-dessous.

**Simplicité et portée des définitions d'événements.** Un événement est défini en EJAVA en utilisant une seule construction. En CLASSPECTS, la définition d'un événement se fait en deux étapes. Tout d'abord, un point de coupure à la ASPECTJ décrit les points de jonction à observer dans un programme de base. Deuxièmement, la construction `addObject` permet de restreindre l'observation à un ensemble donné d'objets. Par exemple, une instance de la classe `Mediator` du Listing A.22 observe l'exécution des méthodes telles que `movementDetect` et `click` restreinte à un capteur particulier et un appareil photo. La différence dans la façon dont les événements sont définis en EJAVA et en CLASSPECTS a un impact sur la capacité de ces langages à discriminer entre les événements de différents objets.

En CLASSPECTS il n'est pas facile de distinguer entre les événements de deux objets du même type. Prenons comme exemple le code EJAVA du Listing A.24. Il implémente une classe qui calcule le temps mis par une particule pour aller d'un point à un autre, en supposant qu'à chaque point il y a un capteur. La classe est instanciée avec deux capteurs, un pour chaque point. Lorsque le premier capteur détecte la particule, le code

```
class ParticleTracker {
  Sensor s1, s2; Long time;

  event onDetect1() = after(s1.movementDetect());
  event onDetect2() = after(s2.movementDetect());

  onDetect1() => {
    time = currentTime();
    next;
  }

  onDetect2() => {
    report(currentTime() - time);
    s1.reset();
    s2.reset();
    next;
  }
}
```

Listing A.24: Détection double en EJava.

de la classe calcule un temps initial. Lorsque le second capteur détecte la particule, le code de la classe calcule le temps final. La classe associe un événement et son gestionnaire à chaque capteur. Cet exemple ne peut pas être mis en œuvre en Classpects d'une manière directe. La raison est que Classpects ne peut différencier entre événements qu'en termes des structures statiques des programmes qui génèrent le point de jonction. Puisque dans cet exemple le point de jonction est généré par la même méthode `movementDetected`, Classpects ne peut pas définir deux événements différents.

**Une conception à la Open Modules.** Dans l'exemple des capteurs et des appareils photo, une caractéristique importante de la conception présentée est l'absence d'information au sein du code des appareils photo et des capteurs sur les hypothèses nécessaires à leur intégration. Le code d'intégration, qui est une préoccupation transverse, a été déplacé hors du code des appareils photo et des capteurs. Toutefois, la solution présentée est *fragile* car elle ne reflète pas correctement les relations entre le programme de base et le médiateur. Un programmeur modifiant indépendamment le programme de base n'est pas conscient d'éventuelles violations des hypothèses nécessaires au bon fonctionnement du médiateur. Par exemple, la classe `Mediator` du Listing A.23 suppose que la détection du capteur est mise en œuvre par les méthodes `movementDetect` ou `temperatureDetect`. Il y a de gros risques qu'une évolution indépendante du code de détection viole cette hypothèse.

L'absence complète de référence aux aspects dans le programme de base (en anglais, on parle d'*obliviousness*) n'est pas compatible avec une véritable modularité. Aldrich [Ald05] a proposé l'idée d'Open Modules comme une conception réellement modulaire en sacrifiant l'absence de référence aux aspects. L'idée est que le programme de base fournit les points de coupure que les aspects vont utiliser. Ainsi, la sémantique de ces points de coupure est maintenue par le programme de base le long de son évolution. Ces points de coupure appartiennent maintenant à l'interface du programme de base, permettant aux aspects et au programme de base d'évoluer indépendamment, tout en préservant cette interface. Le Listing A.25 montre qu'EJava est suffisamment expressif pour fournir une solution à la

```
class Sensor {  ...
  /* provided event */
  event onDetected() = after(movementDetect() || temperatureDetect());
}

class Camera {  ...
  /* provided event */
  event onClicked() = after(click());
}

class Mediator {
  Sensor s; Camera c;

  event onDetected() = s.onDetected();/* required event */
  event onClicked()  = c.onClick();/* required event */

  onDetected() => { c.click(); next; }
  onClicked()  => { s.reset(); next; }
}
```

Listing A.25: Solution asymétrique à la *Open Modules*.

*Open Modules*. Notez comment les événements `onDetected` et `onClicked` sont maintenant fournis respectivement par les classes `Sensor` et `Camera`. Un objet `Mediator` peut observer ces événements en leur associant un gestionnaire.

La définition des points de coupure côté programme de base est déjà possible statiquement en AspectJ. L'avantage d'EJava est le fait que les événements sont membres d'instance à part entière, ce qui contraste avec le caractère statique des points de coupure d'AspectJ. Notez comment l'objet `Mediator` observe l'événement `onDetected` associé à la bonne instance de `Sensor`, définie dans le cadre de son interface. Ce n'est pas possible en AspectJ, qui nécessite de quantifier toutes les instances possibles de `Sensor` et d'inclure une déclaration conditionnelle additionnelle dans le greffon pour choisir la bonne instance.

## A.5 Conclusion

### A.5.1 Discussion

Clarifions maintenant certains points importants de notre modèle. D'abord, nous précisons les rôles respectifs de l'invocation explicite et de l'invocation implicite dans notre modèle. Deuxièmement, nous faisons un parallèle entre la nature objet de nos événements et la nature globale des événements dans les approches basées sur des *types* d'événements.

#### A.5.1.1 Invocation explicite et invocation implicite

Les événements sont des implémentations naturelles de l'invocation implicite. Les méthodes sont des implémentations naturelles de l'invocation explicite. Puisque notre modèle unifie les méthodes et les événements, il est important de préciser comment l'invocation explicite et l'invocation implicite peuvent coexister dans notre modèle.

Le Listing A.26 implémente une fonction d'un éditeur de figures en EJava. Cette fonction transforme la figure `targetFigure` déclarée à la ligne 2, puis elle exécute l'expression

```
1 class SomeTool implements Tool {
2   Figure targetFigure;
3   ...
4   /* code that transforms the target figure */
5   targetFigure.changed();
6   ...
7 }
```

Listing A.26: Déclenchement d'événements et appel de méthode en EJava.

`targetFigure.changed()` à la ligne 5. Dans notre modèle, cette expression signifie un déclenchement de l'événement `changed` de `targetFigure`. Elle avertit que la figure a changé après la transformation. L'expression provoque l'invocation implicite de tous les gestionnaires d'événements associés à l'événement `changed`.

Lorsque la classe `Figure` déclare l'événement `changed` en utilisant une syntaxe de méthode, c'est-à-dire lorsque `changed` est une méthode, nous pouvons considérer le déclenchement de l'événement comme un appel de méthode. Dans ce cas, le programmeur de la fonction est conscient de l'existence de l'un des gestionnaires d'événements liés à `changed`: le corps de la méthode. L'expression signifie l'invocation explicite du corps de la méthode (connu du programmeur) et, en même temps, l'invocation implicite d'autres gestionnaires d'événements associés à l'événement (inconnus du programmeur). C'est de cette façon que l'invocation explicite et l'invocation implicite sont combinées dans notre modèle.

En d'autres termes, le régime d'invocation dépend du protocole décrivant la relation entre les objets. Quand il y a une notion d'interfaces requises et fournies, le régime d'invocation est explicite, sinon, il est implicite. Cette différence pose certains défis lorsqu'on combine les aspects et les composants avec des protocoles explicites, combinaison que nous avons explorée en [NN07]. Cette thèse propose une intégration transparente de ces deux régimes.

#### A.5.1.2 Les types d'événements versus les événements des objets

Les interactions entre les objets est l'un des constituants les plus importants d'une conception par objets. Dans les modèles d'invocation explicite, les objets envoient des messages aux objets. Dans les modèles d'invocation implicite, tels que le patron de conception Observateur, les objets indiquent leur intérêt pour certains événements. De même, dans notre modèle un objet peut indiquer son intérêt pour des événements issus d'objets qu'il connaît explicitement. Cependant, dans les approches basées sur des types d'événements, l'interaction entre les objets est indirecte. Un objet indique son intérêt pour des types d'événements, pas pour des événements issus d'objets particuliers. Quelque chose de semblable se passe dans la plupart des approches de programmation par aspects. Il y a une branche complète de la PPE qui est basée sur cette notion [EGD01, RL08, SPAK10]. Il faut donc clarifier comment notre modèle se situe face cette branche.

L'idée de base d'une conception par types d'événements est le découplage des destinations vis-à-vis des sources: les instances d'un type d'événement sont fournies par des objets que les destinations ne connaissent pas. Cette conception s'éloigne de la conception par objets utilisant le patron Observateur. Il y a bien interaction entre objets, mais les types d'événement, de portée globale, viennent s'intercaler entre les sources et les destinations, qui reçoivent tous les événements d'un type donné, sans distinguer leur source. Si une

telle distinction est nécessaire, de l'information supplémentaire, portée par les instances de types d'événement, doit être passée de la sources aux destinations, qui appliquent des filtres spécifiques sur cette information pour sélectionner les événements pertinents liés aux objets pertinents.

Les événements (points de coupure) dans la PPA sont compatibles avec les types d'événements. Par exemple, un aspect ASPECTJ peut surveiller les appels à une méthode d'une classe. L'aspect ne sait pas quels objets déclencheront l'événement. Ce peut être n'importe quelle instance de la classe ou une sous-classe. Un événement défini de cette façon a l'application pour portée. En d'autres termes, il est statique. Aussi, ASPECTJ fournit des désignateurs de point de jonction spécifiques telles que `this` pour obtenir l'objet qui est la source de l'événement. Si un événement d'un objet spécifique est attendu un filtrage spécifique doit être en suite fait dans le code des greffons pour sélectionner la bonne occurrence. Le lien entre les types d'événements et les événements de la PPA est rendu encore plus explicite en IIIA [SPAK10]. Un type d'événement dans cette approche peut être mis en œuvre par une point de coupure. Les classes peuvent enregistrer des greffons pour des types d'événements ou fournir des instances de ces types.

Notre modèle de programmation donne une plus grande priorité à la conception par objets qu'au découplage des destinations vis-à-vis des sources. Dans notre modèle, une destination s'intéresse aux (ou est notifiée des) événements issus des sources avec lesquelles elle collabore, à l'instar du patron Observateur. Ainsi, aucun filtrage supplémentaire n'est nécessaire pour sélectionner les événements pertinents. Un objet n'est notifié que des événements pertinents.

### A.5.2 Contributions

Les principales contributions de cette thèse sont: (1) un modèle de programmation événementielle avec des événements déclaratifs, (2) une meilleure intégration de la PPE, de la PPA et de la PPO à travers ce modèle, (3) la concrétisation du modèle sous la forme de deux langages de programmation réels avec leur implémentation, et grâce à ces prototypes, la validation du modèle de programmation sur la mise en œuvre d'un éditeur de figures et d'une ligne de produits domotiques. Nous résumons ces contributions ci-dessous.

**Événements déclaratifs** Lors du développement de grandes applications, une programmeur a souvent besoin de définir des événements en termes d'autres événements. Dans plusieurs langages incluant des événements, la composition des événements est programmée en utilisant des événements impératifs. Les événements composites sont déclenchés sous la forme d'événements impératifs au sein des gestionnaires d'événements associés aux événements composés. En conséquence, quand plusieurs événements sont définis de cette façon, il est difficile de comprendre les différentes relations entre les événements d'une application. Ceci est source de complexité. Pour pallier à ce problème, notre modèle de programmation événementiel introduit la notion d'*événement déclaratif*. Un événement déclaratif exprime de manière déclarative les relations entre les événements d'une application. Les différentes relations entre les événements sont plus faciles à comprendre. Des événements impératifs et déclaratifs peuvent faire partie de l'interface d'un système améliorant sa modularité.

**Unification des paradigmes** La combinaison de plusieurs paradigmes de programmation est souvent nécessaire pour mettre en œuvre les exigences d'une application complexe

et de fournir un logiciel de *bonne qualité*. La PPE, la PPA et la PPO s'avèrent particulièrement utiles. Ces différents paradigmes sont implémentés en utilisant des abstractions linguistiques différentes. Ces abstractions sont aussi fournies comme des constructions linguistiques différents. La prolifération des concepts et des constructions linguistiques donne lieu à des programmes complexes.

Nous avons noté que certains de ces mécanismes ne sont pas si différents les uns des autres. C'est en contradiction avec les principes de la conception des langages de programmation mis en avant par MacLennan [Mac95]. Pour résoudre ce problème, nous avons introduit un modèle de programmation qui intègre la PPE, la PPA et la PPO. Le modèle est une extension du modèle d'événements déclaratifs décrit précédemment. La pierre angulaire de ce modèle est une unification des méthodes et des événements. Les contributions de ce modèle sont les suivantes:

- Les mêmes constructions linguistiques peuvent être utilisées pour mettre en œuvre les deux formes d'invocation, explicite et implicite. Ainsi, la PPE et la PPO peuvent être réalisées de façon simple et orthogonale par un modèle de programmation qui équipe les classes avec des champs, des événements et des gestionnaires d'événements.
- Le même mécanisme d'événements et de gestionnaires d'événements peut être utilisé pour mettre en œuvre des applications événementielles et des aspects. Les événements et les points de jonction peuvent être unifiés et un greffon peut être mis en œuvre comme un gestionnaire d'événement. Ainsi, les programmeurs ayant une bonne compréhension de la PPE peuvent entrer dans la PPA sans efforts supplémentaires.
- Le modèle montre l'avantage de fournir des événements et des gestionnaires d'événements en tant que membres d'instance à part entière. Il améliore l'intégration entre la PPE et PPO et, en même temps, l'intégration de la PPA et de la PPO. Nous avons montré que les aspects et les classes peuvent être fournies sous la forme d'une structure linguistique unique: une classe (éventuellement aspectuelle). Les mêmes règles régulières de l'héritage, de l'instanciation et du traitement des membres de la classe peuvent gouverner la sémantique des classes et des aspects. Ainsi, les programmeurs ayant une bonne compréhension de la PPO peuvent intégrer plus facilement dans leur pratique de programmation la PPE et la PPA.

**Implémentation et cas d'étude** Deux langages concrétisent notre modèle de programmation. EJava implémente l'essence du modèle comme une extension de Java. ECaesarJ combine notre modèle et de la composition de mixins. Les deux langages ont été utilisés pour valider notre modèle dans la mise en œuvre d'applications réelles. Ils ont amélioré la implémentation de ces applications. Ces langages sont librement disponibles pour l'expérimentation dans ce domaine.

### A.5.3 Perspectives

**Traitement formel.** Afin de notamment prouver la correction de notre modèle, il est nécessaire de le formaliser. Dans un premier temps, nous avons utilisé le langage formel MiniMAO$_0$ [CL05] pour en décrire la syntaxe abstraite.

**Un modèle de points de jonction plus riche.** Le modèle présenté dans cette thèse permet de mettre en œuvre des aspects avec un modèle de point de jonction basé sur les points de jonction de Masuhara et al. [MEY06]. Seulement deux des quatre types de points de jonction du modèle de Masuhara sont supportés: *call* et *reception*. Dans le futur, nous

envisageons d'étendre les types de points de jonction pris en charge par notre modèle. Par exemple, nous aimerions inclure les points de jonction *execution* et *return* du modèle de Masuhara. Cela permettrait la prise en compte des points de jonction *execution* d'ASPECTJ. Puisque dans notre modèle l'exécution d'une méthode (corps de la méthode) correspond à l'exécution d'un gestionnaire d'événement, nous devrions être en mesure d'introduire les points de jonction *execution* en incluant un type d'événement représentant l'exécution d'un gestionnaire d'événement. Ce type d'événements permettra de mettre en œuvre des aspects intéressés par l'exécution d'un gestionnaire d'événement particulier.

**Gestion des exceptions.** Nous traitons les exceptions en adoptant le style des exceptions vérifiées de JAVA, ce qui n'est pas bien adapté aux applications événementielles. Dans le future nous prévoyons d'étudier d'autres moyens possibles pour faire face aux exceptions [PH07]. Intuitivement, il semble approprié d'augmenter l'expressivité de notre modèle pour être en mesure de définir les événements qui représentent des exceptions. Comme un type d'exception peut être lancé à partir de nombreux endroits dans le code avec de nombreux objets en jeu, il peut être impropre de quantifier sur la liste de tous les objets qui peuvent lancer un type d'exception. Dans ce cas, il semble qu'utiliser des types d'événement soit mieux adapté.

**Amélioration des prototypes.** Nous avons fourni deux implémentations du modèle, EJAVA et ECAESARJ. Ces prototypes peuvent être améliorés, notamment en termes de performances. L'analyse sémantique des compilateurs d'EJAVA et ECAESARJ et l'outillage des langages peuvent aussi être améliorés. En ce qui concerne l'outillage, un plugin Eclipse initial a été développé en utilisant Spoofax [KV10]. Cependant, il ne permet toujours pas des fonctionnalités avancées telles que le débogage.

**Concurrence.** Notre travail a porté sur un cadre séquentiel. Étudier l'applicabilité de notre modèle à un cadre concurrent reste à mener. La notion d'événement est souvent utilisé pour coordonner les applications concurrentes en utilisant des modèles tirés des algèbres de processus. Comme notre modèle est basé sur cette notion le passage à un cadre concurrent devrait être facilité.

**Exploration d'autres choix de conception.** Maintenant que nous avons une meilleure compréhension des implications de nos décisions de conception initiales, nous n'excluons pas de considérer d'autres choix de conception. En particulier, nous considérons que les événements polymorphes proposés dans le cadre d'IIIA [SPAK10] sont particulièrement intéressants et qu'il serait utile d'explorer l'intégration entre les types d'événements et nos événements liés aux objets.

Une retombée de notre travail a été l'implémentation d'une variante de notre modèle sous la forme d'une extension de SCALA, ESCALA [GSM+11]. Ce langage implémente notre modèle d'événements déclaratifs sur le langage SCALA. Il y a des différences entre ce langage et EJAVA, au niveau du modèle et de l'implémentation. Premièrement, alors que les opérateurs de composition de base, tels que la disjonction et la conjonction, sont similaires dans les deux langages, ESCALA offre des opérateurs qui exploitent le côté fonctionnel de SCALA en offrant une flexibilité supplémentaire. Deuxièmement, ESCALA n'intègre pas les événements et les méthodes comme en EJAVA. Les méthodes et les gestionnaires d'événement sont des notions distinctes. Même si en ESCALA un appel de méthode peut être

considéré comme un événement, les corps de méthode ne sont pas considérés comme des gestionnaires d'événements comme en EJava. EScala ne permet pas de mettre en œuvre un aspect qui empêche le programme de base d'effectuer un appel de méthode. Dans EJava cette question est juste une question de composition des gestionnaires d'événements. Ainsi, EScala est un langage événementiel avec une saveur de PPA plutôt que d'un langage qui intègre à la fois la PPE et la PPA d'une manière transparente comme en EJava. Enfin, la mise en œuvre d'EScala, faite par Lucas Satabin, est plus efficace que la mise en œuvre originale de EJava grâce à l'optimisation du processus de notification. La mise en œuvre d'EJava a emprunté l'optimisation du processus de notification et le traitement des références à des événements à travers des champs mutables. Elle reste toutefois très différente, par exemple, en ce qui concerne l'instrumentation du code, puisque nous utilisons AspectJ plutôt qu'un mécanisme spécifique. En ce qui concerne ECaesarJ, EScala ne supporte pas les machines à états et les mixins.

Scala est un bon langage pour la poursuite des travaux développés dans cette thèse, grâce à sa combinaison de programmation fonctionnelle et impérative et de fonctionnalités telles que l'inférence de types. Par exemple, EScala peut être amélioré avec les éléments présentés dans cette thèse qui n'ont pas encore été considérés comme l'unification des événements et des méthodes et le support des machines à états.

# Bibliography


[ADGI08] Jagrati Agrawal, Yanlei Diao, Daniel Gyllstrom, and Neil Immerman. Efficient pattern matching over event streams. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 147–160, New York, NY, USA, 2008. ACM Press.

[AGMO06] Ivica Aracic, Vaidas Gasiūnas, Mira Mezini, and Klaus Ostermann. An overview of CaesarJ. In *Transactions on Aspect-Oriented Software Development I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer-Verlag, February 2006.

[Ald05] Jonathan Aldrich. Open modules: Modular reasoning about advice. In Black [Bla05], pages 144–168.

[AM97] Mehmet Aksit and Satoshi Matsuoka, editors. *ECOOP '97 - Object-Oriented Programming - 11th European Conference*, volume 1241 of *Lecture Notes in Computer Science*. Springer-Verlag, June 1997.

[AMP10] AMPLE. Ample project. http://www.ample-project.net, 2007-2010.

[AN09] Ali Assaf and Jacques Noyé. Flexible pointcut implementation: An interpreted approach. In Bernard Carré, editor, *Langages et Modèles à Objets (LMO 2009)*, pages 45–60, Nancy France, March 2009. Olivier Zendra, Cépaduès-Editions.

[aos08] *Proceedings of the 7th International Conference on Aspect-Oriented Software Development, AOSD '08*, Brussels, Belgium, March 2008. ACM Press.

[BC11] Paulo Borba and Shigeru Chiba, editors. *Proceedings of the 10th International Conference on Aspect-Oriented Software Development, AOSD '11, Porto de Galinhas, Brazil, March 21-25, 2011*. ACM Press, 2011.

[BCRD08] Johan Brichau, Ruzanna Chitchyan, Awais Rashid, and Theo D'Hondt. Aspect-oriented software development: an introduction. In Wah [Wah08].

[BDG+07] Lars Brenna, Alan Demers, Johannes Gehrke, Mingsheng Hong, Joel Ossher, Biswanath Panda, Mirek Riedewald, Mohit Thatte, and Walker White. Cayuga: a high-performance event processing engine. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 1100–1102, New York, NY, USA, 2007. ACM Press.

[BEG08] Lodewijk Bergmans, Erik Ernst, and Kris Gybels, editors. *6th Workshop on Software-engineering Properties of Languages and Aspect Technologies (SPLAT 2008)*, Brussels, Belgium, March 2008.

[BGL98] J.-P. Briot, R. Guerraoui, and K.-P. Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3), September 1998.

[BJ87] K. Birman and T. Joseph. Exploiting virtual synchrony in distributed systems. *SIGOPS Oper. Syst. Rev.*, 21(5):123–138, 1987.

[Bla05] Andrew P. Black, editor. *ECOOP '05 - Object-Oriented Programming, 19th European Conference*, volume 3586 of *Lecture Notes in Computer Science*, Glasgow, UK, July 2005. Springer-Verlag.

[Bon04] Jonas Bonér. AspectWerkz - dynamic AOP for Java. In Lieberherr [Lie04], pages 51–62.

[BPS99] Viviana Bono, Amit Patel, and Vitaly Shmatikov. A core calculus of classes and mixins. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, ECOOP '99, pages 43–66, London, UK, 1999. Springer-Verlag.

[Bru07] Bruce Eckel. Does Java need checked exceptions? http://www.mindview.net/Etc/Discussions/CheckedExceptions, 2007.

[BV04] Jonas Bonér and Alexandre Vasseur. AspectWerkz. http://aspectwerkz.codehaus.org/index.html, February 2004.

[BvdA01] Twan Basten and Wil M. P. van der Aalst. Inheritance of behavior. *Journal of Logic and Algebraic Programming*, 47(2):47–145, 2001.

[CL05] Curtis Clifton and Gary T. Leavens. Minimao: Investigating the semantics of proceed. In Curtis Clifton, Ralf Lämmel, and Gary T. Leavens, editors, *FOAL 2005 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD '05, Chicago, IL*, pages 57–67, Ames, IA, 50011, March 2005. TR 05-05, Dept. of Computer Science, Iowa State University.

[CM08] Brian Chin and Todd Millstein. An extensible state machine pattern for interactive applications. In Vitek [Vit08].

[Coo89] S. Cook, editor. *Proceedings of the Third European Conference on Object-Oriented Programming, 1989 (ECOOP '89)*, Kaiserslautern, Germany, 1989. Cambridge University Press.

[DFS04] Rémi Douence, Pascal Fradet, and Mario Südholt. Composition, reuse and interaction analysis of stateful aspects. In Lieberherr [Lie04].

[DHW94] Umeshwar Dayal, Eric N. Hanson, and Jennifer Widom. Active database systems. In *Modern Database Systems*, pages 434–456. ACM Press, 1994.

[Dij79] E. Dijkstra. *Programming considered as a human activity*, pages 1–9. Yourdon Press, Upper Saddle River, NJ, USA, 1979.

[DLBNS06] Rémi Douence, Didier Le Botlan, Jacques Noyé, and Mario Südholt. Concurrent aspects. In *Proceedings of the 4th International Conference on Generative Programming and Component Engineering (GPCE '06)*, pages 79–88. ACM Press, October 2006.

[Dro09] Sophia Drossopoulou, editor. *ECOOP '09 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, volume 5653 of *Lecture Notes in Computer Science*. Springer, 2009.

[EFGK03] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[EGD01] Patrick Th. Eugster, Rachid Guerraoui, and Christian Heide Damm. On objects and events. In *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '01, pages 254–269, New York, NY, USA, 2001. ACM Press.

[EJ09] Patrick Th. Eugster and K. R. Jayaram. EventJava: An extension of Java for event correlation. In Drossopoulou [Dro09], pages 570–594.

[Eri11] Erich Gamma and Jochen Quante and Wolfram Kaiser. JHotDraw as open-source project. http://sourceforge.net/projects/jhotdraw/, 2011.

[Ern99a] Erik Ernst. *gbeta - a language with virtual attributes, Block Structure, and Propagating, Dynamic Inheritance*. PhD thesis, Department of Computer Science, University of Aarhus, Denmark, 1999.

[Ern99b] Erik Ernst. Propagating class and method combination. In Guerraoui [Gue99], pages 67–91.

[Ern01] Erik Ernst. Family polymorphism. In Knudsen [Knu01], pages 303–326.

[FECA05] Robert E. Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Akşit, editors. *Aspect-Oriented Software Development*. Addison-Wesley, 2005.

[FF05] Robert E. Filman and Daniel P. Friedman. Aspect-oriented programming is quantification and obliviousness. In Filman et al. [FECA05], pages 21–35.

[FH02] Robert E. Filman and Klaus Havelund. Realising aspects by transforming for events. In Kris De Volder, Kim Mens, Tom Mens, and Roel Wuyts, editors, *Proc. Workshop on Declarative Meta Programming to Support Software Development*, September 2002.

[FMG02] Ludger Fiege, Gero Mühl, and Felix C. Gärtner. Modular event-based systems. *Knowl. Eng. Rev.*, 17(4):359–388, 2002.

[GB03] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69, New York, NY, USA, 2003. ACM Press.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GN91] David Garlan and David Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development (VDM '91)* [vdm91], pages 31–44.

[GNNM11] Vaidas Gasiūnas, Angel Núñez, Jacques Noyé, and Mira Mezini. Product line implementation with ECaesarJ. In Awais Rashid, Jean-Claude Royer, and Andreas Rummler, editors, *Aspect-Oriented, Model-Driven Software Product Lines - The AMPLE Way*. Cambridge University Press, October 2011.

[Gre75] Irene Greif. Semantics of communicating parallel processes. Technical Report 154, Massachusetts Institute of Technology, Project MAC, September 1975.

[GS94] David Garlan and Mary Shaw. An introduction to software architecture. Technical Report CMU-CS-94-166, Carnegie Mellon University, January 1994.

[GSM+11] Vaidas Gasiūnas, Lucas Satabin, Mira Mezini, Angel Núñez, and Jacques Noyé. EScala: modular event-driven object interactions in Scala. In Borba and Chiba [BC11], pages 227–240.

[Gue99] R. Guerraoui, editor. *Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP '99)*, volume 1648 of *Lecture Notes in Computer Science*, Lisbon, Portugal, June 1999. Springer-Verlag.

[Har87] David Harel. Statecharts: A visual formulation for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[Hew10] Carl Hewitt. Actor model for discretionary, adaptive concurrency. *CoRR*, abs/1008.1459, 2010.

[Hir02] Robert Hirschfeld. AspectS - aspect-oriented programming with Squeak. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a NetworkedWorld: International Conference NetObjectDays, NODe 2002*, volume 2591 of *Lecture Notes in Computer Science*, pages 216–232. Springer-Verlag, 2002.

[HO09] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2-3):202–220, February 2009.

[Hoa78] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[ics05] *Proceedings of the 27th International Conference on Software Engineering*, St. Louis, MO, USA, May 2005. ACM Press.

[IEE08] IEEE Standard for the Functional Verification Language *e*. IEEE Computer Society, August 2008. IEEE Std 1647$^{TM}$-2008.

[IEE09] IEEE Standard for SystemVerilog - Unified Hardware Design, Specification, and Verification Language. IEEE Computer Society, December 2009. IEEE Std 1800$^{TM}$-2009.

[JBo10] JBoss AOP. JBoss AOP reference manual. version 2.0. http://docs.jboss.org/jbossaop/docs/index.html, 2010.

[JF88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, June/July 1988.

[JHD+10] Rod Johnson, Juergen Hoeller, Keith Donald, Colin Sampaleanu, Rob Harrop, Alef Arendsen, Thomas Risberg, Darren Davison, Dmitriy Kopylenko, Mark Pollack, Thierry Templier, Erwin Vervaet, Portia Tung, Ben Hale, Adrian Colyer, John Lewis, Costin Leau, Mark Fisher, Sam Brannen, Ramnivas Laddad, Arjen Poutsma, Chris Beams, Tareq Abedrabbo, and Andy Clement. *The Spring Framework - Reference Documentation 3.0.RC3*. SpringSource, 2010. http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/.

[JP94] Bengt Jonsson and Joachim Parrow, editors. *Proceedings of the 5th International Conference on Concurrency Theory (CONCUR '94)*, volume 836 of *Lecture Notes in Computer Science*, Uppsala, Sweden, 1994. Springer-Verlag.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Meta-Object Protocol*. MIT Press, 1991.

[Kee89] Sonya E. Keene. *Object-Oriented Programming in Common Lisp*. Addison Wesley, Reading, Mass., 1989.

[KHH+01] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. An overview of AspectJ. In Knudsen [Knu01], pages 327–353.

[Kic02] Gregor Kiczales, editor. *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD '02)*, Enschede, The Netherlands, April 2002. ACM Press.

[KL89] D. Kafura and K.H. Lee. Inheritance in actor based concurrent object-oriented languages. In Cook [Coo89].

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In Aksit and Matsuoka [AM97], pages 220–242.

[Knu01] Jørgen Lindskov Knudsen, editor. *ECOOP '01 - Object-Oriented Programming, 15th European Conference*, number 2072 in Lecture Notes in Computer Science, Budapest, Hungary, June 2001. Springer-Verlag.

[KP88] Glenn E. Krasner and Stephen T. Pope. A cookbook for using the model-view controller user interface paradigm in smalltalk-80. *J. Object Oriented Program.*, 1(3):26–49, 1988.

[KS04] Christian Koppen and Maximilian Störzer. PCDiff: Attacking the fragile pointcut problem. In Kris Gybels, Stefan Hanenberg, Stephan Herrmann, and Jan Wloka, editors, *European Interactive Workshop on Aspects in Software (EIWAS)*, September 2004.

[KV10] Lennart C. L. Kats and Eelco Visser. The Spoofax language workbench. Rules for declarative specification of languages and IDEs. In Martin Rinard, editor, *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA '10, October 17-21, 2010, Reno, NV, USA*, pages 444–463, 2010.

[Lie04] Karl Lieberherr, editor. *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, Lancaster, UK, March 2004. ACM Press.

[Luc97] David C. Luckham. Rapide: a language and toolset for simulation of distributed systems by partial orderings of events. In *Proceedings of the DIMACS workshop on Partial order methods in verification*, pages 329–357, New York, NY, USA, 1997. AMS Press, Inc.

[Luc01] 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.

[Mac95] Bruce J. MacLennan. *Principles of Programming Languages: Design, Evaluation, and Implementation*. Oxford University Press, 2nd edition, 1995.

[Mey89] N. Meyrowitz, editor. *OOPSLA '89, Conference Proceedings*, New Orleans, Louisiana, USA, October 1989. 24(10).

[MEY06] Hidehiko Masuhara, Yusuke Endoh, and Akinori Yonezawa. A fine-grained join point model for more reusable aspects. In *Proceedings of 4th Asian Symposium on Programming Languages and Systems (APLAS '06)*, volume 4279 of *Lecture Notes in Computer Science*, pages 131–147, Sydney, Australia, November 2006. Springer-Verlag.

[Mil82] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1982.

[MK06] J. Magee and J. Kramer. *Concurrency: State Models and Java*. Wiley, 2nd edition, 2006.

[MMP89] O. L. Madsen and B. Møller-Pedersen. Virtual classes: A powerful mechanism in object-oriented programming. In Meyrowitz [Mey89], pages 397–406. 24(10).

[Moo86b] David A. Moon. Object-oriented programming with flavors. In *Conference proceedings on Object-oriented programming systems, languages and applications*, OOPLSA '86, pages 1–8, New York, NY, USA, 1986. ACM.

[MW08] Antoine Marot and Roel Wuyts. Composability of aspects. In Bergmans et al. [BEG08].

[ND66] K. Nygaard and O.-J. Dahl. Simula - an ALGOL-based simulation language. *Communications of the ACM*, 9(9), September 1966.

[ND78] Kristen Nygaard and Ole-Johan Dahl. The development of the Simula languages. 13(8):245–272, 1978.

[NN07] Angel Núñez and Jacques Noyé. A seamless extension of components with aspects using protocols. In Reussner et al. [RSW07].

[NN08] Angel Núñez and Jacques Noyé. An event-based coordination model for context-aware applications. In Doug Lea and Gianluigi Zavattaro, editors, *Coordination Models and Languages (COORDINATION '08)*, volume 5052 of *Lecture Notes in Computer Science*, pages 232–248. Springer Berlin / Heidelberg, 2008.

[NnNG09] 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 Press.

[OMB05] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Black [Bla05].

[Par69] David L. Parnas. On simulating networks of parallel processes in which simultaneous events may occur. *Communications of the ACM*, 12(9):519–531, 1969.

[PH07] Jan Ploski and Wilhelm Hasselbring. Exception handling in an event-driven system. In *Proceedings of the The Second International Conference on Availability, Reliability and Security*, pages 1085–1092, Washington, DC, USA, 2007. IEEE Computer Society.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Aksit and Matsuoka [AM97], pages 419–443.

[RL08] Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In Vitek [Vit08].

[RS05] Hridesh Rajan and Kevin J. Sullivan. Classpects: Unifying aspect- and object-oriented language design. In *Proceedings of the 27th International Conference on Software Engineering* [ics05], pages 15–21.

[RS09] Hridesh Rajan and Kevin J. Sullivan. Unifying aspect- and object-oriented design. *ACM Transactions on Software Engineering and Methodology*, 19(1), August 2009.

[RSW07] Ralf Reussner, Clemens Szyperski, and Wolfgang Weck, editors. *WCOP 2007 - Components beyond Reuse - 12th International ECOOP Workshop on Component-Oriented Programming*, July 2007.

[RW97] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC '97/FSE-5, pages 344–360, New York, NY, USA, 1997. Springer-Verlag New York, Inc.

[Sam08] Miro Samek. *Practical UML Statecharts in C/C++, Second Edition: Event-Driven Programming for Embedded Systems*. Newnes, Newton, MA, USA, 2008.

[SC95] Aamod Sane and Roy H. Campbell. Object-oriented state machines: Subclassing, composition, delegation and genericity. In *OOPSLA '95, Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 17–32, Austin, TX, USA, October 1995. ACM Press.

[Sch78] J. William Schmidt. Introduction to simulation modeling. In *WSC '78: Proceedings of the 10th conference on Winter simulation*, pages 3–12, Piscataway, NJ, USA, 1978. IEEE Computer Society.

[SGC02] Kevin Sullivan, Lin Gu, and Yuanfang Cai. Non-modularity in aspect-oriented languages: integration as a crosscutting concern for AspectJ. In Kiczales [Kic02], pages 19–26.

[SMU+04] Kouhei Sakurai, Hidehiko Masuhara, Naoyasu Ubayashi, Saeko Matsuura, and Seiichi Komiya. Association aspects. In Lieberherr [Lie04], pages 16–25.

[SPAK10] Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Transactions on Software Engineering and Methodology*, 20(1), July 2010.

[Sun97] Sun Microsystems. Javabeans(tm) specification. version 1.01. http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html, 1997.

[Tan08] Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *Proceedings of the 7th International Conference on Aspect-Oriented Software Development, AOSD '08* [aos08], pages 168–179.

[TOHS99] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton, Jr. N degrees of separation: multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, pages 107–119, Los Angeles, CA, USA, May 1999. ACM Press.

[US94] Andrew C. Uselton and Scott A. Smolka. A compositional semantics for statecharts using labeled transition systems. In Jonsson and Parrow [JP94], pages 2–17.

[vdBCC05] K.G. van den Berg, J.M. Conejero, and R. Chitchyan. AOSD ontology 1.0 - public ontology of aspect-orientation, May 2005. AOSD-Europe-UT-01.

[VdBdJKO00] M. G. T. Van den Brand, H. A. de Jong, P. Klint, and P. A. Olivier. Efficient annotated terms. *Softw. Pract. Exper.*, 30:259–291, March 2000.

[vdBL89] Jan van den Bos and Chris Laffra. PROCOL: a parallel object language with protocols. In Meyrowitz [Mey89], pages 95–102. 24(10).

[vdBL91] Jan van den Bos and Chris Laffra. PROCOL: A concurrent object-oriented language with protocols, delegation and constraints. *Acta Informatica*, 28(6):511–538, 1991.

[vdBSVV02] Mark G. J. van den Brand, Jeroen Scheerder, Jurgen J. Vinju, and Eelco Visser. Disambiguation Filters for Scannerless Generalized LR Parsers. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 143–158. Springer-Verlag, April 2002.

[vdm91] *Proceedings of the 4th International Symposium of VDM Europe on Formal Software Development (VDM '91)*, Springer-Verlag, 1991.

[Vis97a] Eelco Visser. Scannerless Generalized-LR Parsing. Technical Report P9707, Programming Research Group, University of Amsterdam, July 1997.

[Vis97b] Eelco Visser. *Syntax Definition for Language Prototyping.* PhD thesis, University of Amsterdam, September 1997.

[Vis01] Eelco Visser. Stratego: A Language for Program Transformation Based on Rewriting Strategies. In *RTA*, pages 357–362, 2001.

[Vis04] Eelco Visser. Program Transformation with Stratego/XT. Rules, Strategies, Tools, and Systems in Stratego/XT 0.9 . Technical Report UU-CS-2004-011, Department of Information and Computing Sciences, Utrecht University, 2004.

[Vit08] Jan Vitek, editor. *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP '08)*, volume 5142 of *Lecture Notes in Computer Science*, Paphos, Cyprus, July 2008. Springer-Verlag.

[Wah08] Benjamin W. Wah, editor. *Wiley Encyclopedia of Computer Science and Engineering.* John Wiley & Sons, Inc., 2008.

[WC96] Jennifer Widom and Stefano Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing.* Morgan Kaufmann, 1996.

[WDR06] Eugene Wu, Yanlei Diao, and Shariq Rizvi. High-performance complex event processing over streams. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 407–418, New York, NY, USA, 2006. ACM.

[WM01] Robert J. Walker and Gail C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In *Workshop on Advanced Separation of Concerns in Software Engineering. In Proceedings of ICSE '01*, 2001.

# Un modèle de programmation intégrant classes, événements et aspects

Angel Rodrigo Núñez López

**Résumé**

Le paradigme de la programmation par objets (PPO) est devenu le paradigme de programmation le plus utilisé. La programmation événementielle (PE) et la programmation par aspects (PPA) complètent la PPO en comblant certaines de ses lacunes lors de la construction de logiciels complexes. Les applications actuelles combinent ainsi les trois paradigmes. Toutefois, la POO, la PE et la POA ne sont pas encore bien intégrées. Leurs concepts sous-jacents sont en général fournis sous la forme de constructions syntaxiques spécifiques malgré leurs points communs. Ce manque d'intégration et d'orthogonalité complique les logiciels car il réduit leur compréhensibilité et leur composabilité, et augmente le code d'infrastructure.

Cette thèse propose une intégration de la PPO, de la PE et de la PPA conduisant à un modèle de programmation simple et régulier. Ce modèle intègre les notions de classe et d'aspect, les notions d'événement et de point de jonction, et les notions d'action, de méthode et de gestionnaire d'événements. Il réduit le nombre de constructions tout en gardant l'expressivité initiale et en offrant même des options de programmation supplémentaires.

Nous avons conçu et mis en œuvre deux langages de programmation basés sur ce modèle : EJava et ECaesarJ. EJava est une extension de Java implémentant le modèle. Nous avons validé l'expressivité de ce langage par la mise en œuvre d'un éditeur graphique bien connu, JHotDraw, en réduisant le code d'infrastructure nécessaire et en améliorant sa conception. ECaesarJ est une extension de CaesarJ qui combine notre modèle avec de la composition de *mixins* et un support linguistique des machines à états. Cette combinaison a grandement facilité la mise en œuvre d'une application de maison intelligente, une étude de cas d'origine industrielle dans le domaine de la domotique.

**Mots-clés :** programmation par objets, programmation événementielle, programmation par aspects, langages de programmation

# A Programming Model Integrating Classes, Events and Aspects

**Abstract**

Object-Oriented Programming (OOP) has become the de facto programming paradigm. Event-Based Programming (EBP) and Aspect-Oriented Programming (AOP) complement OOP, covering some of its deficiencies when building complex software. Today's applications combine the three paradigms. However, OOP, EBP and AOP have not yet been properly integrated. Their underlying concepts are in general provided as distinct language constructs, whereas they are not completely orthogonal. This lack of integration and orthogonality complicates the development of software as it reduces its understandability, its composability and increases the required glue code.

This thesis proposes an integration of OOP, EBP and AOP leading to a simple and regular programming model. This model integrates the notions of class and aspect, the notions of event and join point, and the notions of piece of advice, method and event handler. It reduces the number of language constructs while keeping expressiveness and offering additional programming options.

We have designed and implemented two programming languages based on this model: EJava and ECaesarJ. EJava is an extension of Java implementing the model. We have validated the expressiveness of this language by implementing a well-known graphical editor, JHotDraw, reducing its glue code and improving its design. ECaesarJ is an extension of CaesarJ that combines our model with mixins and language support for state machines. This combination was shown to greatly facilitate the implementation of a smart home application, an industrial-strength case study that aims to coordinate different devices in a house and automatize their behaviors.

**Keywords:** object-oriented programming, event-based programming, aspect-oriented programming, programming languages