POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione
DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA

Towards Software Components for Non Functional Aspects

Verso la realizzazione di componenti software per gli aspetti non funzionali di sistemi complessi

Tesi di dottorato di:
Mattia Monga

Relatore:
prof. Carlo Ghezzi
Tutore:
prof. Carlo Ghezzi
Coordinatore del programma di dottorato:
prof. Carlo Ghezzi

XIII ciclo

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci 32 I 20133 — Milano

POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione
DOTTORATO DI RICERCA IN INGEGNERIA INFORMATICA E AUTOMATICA

Towards Software Components for Non Functional Aspects

Ph.D. Dissertation of:
Mattia Monga

Advisor:
prof. Carlo Ghezzi
Tutor:
prof. Carlo Ghezzi
Supervisor of the Ph.D. Program:
prof. Carlo Ghezzi

XIII edition

A mio padre

Ringraziamenti

Due anni e mezzo; tanto è durato il dottorato e le persone che in questo periodo ho incontrato e che sento di dover ringraziare sono davvero tante. Lo faccio in italiano, in primo luogo perché nel mio inglese stentato i ringraziamenti fluirebbero naturali quanto un programma imperativo su di una *lisp-machine*, e poi perché questa è la lingua in cui per trent'anni ho parlato, scritto, letto, studiato, capito, imprecato: le ragioni dell'intelletto possono adattarsi anche all'"esperanto dei vincitori," ma per quelle del cuore mi serve l'unica lingua di cui controllo le sfumature.

Mark Twain ha detto una volta che gli unici cui spetti il diritto di usare il "noi" editoriale sono i monarchi o gli afflitti da problemi di vermi. Pur non rientrando, al momento, in nessuna delle due categorie, le persone che hanno in qualche maniera contribuito alla maturazione delle idee e alla realizzazione di questa tesi sono cosí tante che ho sentito davvero il dovere di sfidare l'ironia del sagace umorista. Alcune le voglio ricordare qui, ma molte altre ne dovrei ringraziare se la memoria e lo spazio me lo permettessero.

Inizio dai colleghi che si sono succeduti nell'ufficio 160 (in rigoroso ordine alfabetico): Alberto Coen, Alex Orso, Antonio Carzaniga, Daniela Gatti, Gianpaolo Cugola, Giovanni Denaro, Giovanni Vigna, Luciano Baresi, Matteo Pradella, Matteo Rossi, Matteo Valsasna, Ouejdane Mejri, Vincenzo Martena. A loro spetta il privilegio di essere citati per primi perché, pur costretti dalle circostanze a starmi vicini per parecchie ore al giorno, hanno saputo aiutarmi quando ne ho avuto bisogno, deridermi quando mi sono preso troppo sul serio, ignorarmi quando, piú spesso di quanto avrei voluto, ho concionato senza costrutto.

Corre l'obbligo di rinominare qui Gianpaolo Cugola, insieme a Gian Pietro Picco e Carlo Ghezzi, che tanta parte hanno avuto nella mia maturazione scientifica e professionale. Se questo lavoro ha qualche valore lo si deve ai loro incalzanti *"So what?"*, ai loro preziosi suggerimenti e alle loro pazienti e perspicaci revisioni.

Grazie a Mara. Ma ancora di piú grazie a Paola: forse un giorno riuscirò a spiegarle quanta importanza ha avuto in quest'avventura.

Spero di saperle essere altrettanto vicino nei momenti difficili del suo dottorato.

Riconoscenza infinita va naturalmente ai miei genitori, che oltre ad avermi trasmesso l'amore per il sapere, mi hanno accompagnato, incoraggiato e accudito in piú di vent'anni di studi. Non posso, inoltre, dimenticare mia zia Graziella, che ha sempre creduto in me molto di piú di quanto meritassi. E Andrea, amico di sempre, che negli ultimi tempi ho colpevolmente trascurato.

Permettetemi di concludere con un ringraziamento speciale alle pubbliche istituzioni italiane. Una di queste, l'ENEA, ha generosamente finanziato i miei studi di dottorato; un'altra, il Politecnico di Milano, mi ha ospitato fornendomi uno spazio dove stare, professori competenti con cui collaborare e innumerevoli occasioni per crescere professionalmente; le rimanenti hanno permesso che potessi godere di tutti i diritti di cui ogni uomo che nasce è titolare: mi piace pensare che il merito sia di tutti coloro che, credendo nella necessità del consorzio civile, hanno saputo sacrificarsi per il bene dell'Italia.

Sommario

Trattare i singoli problemi che la realizzazione di un sistema complesso pone, in maniera il piú possibile indipendente, è una tecnica fondamentale nel lavoro di ogni progettista. Ogni problema complesso viene decomposto in sotto-problemi piú semplici, le cui soluzioni vengono poi aggregate per formare la soluzione completa. In particolare nello sviluppo del software si cerca di incapsulare tali soluzioni in componenti sostanzialmente autonomi che possono essere sfruttati in occasioni diverse.

La tecnica del "divide et impera" è comune a tutti le fasi dello sviluppo. In particolare, risulta assere uno strumento insostituibile per il programmatore, il quale si aspetta di trovare nel linguaggio usato per esprimere le soluzioni strutture adatte a realizzare una effettiva separazione dei problemi. Di fatto qualsiasi linguaggio di alto livello fornisce appositi costrutti per isolare linguisticamente parti di programma: nei linguaggi tradizionali procedure e funzioni facilitano il ragionamento di tipo *top-down*, nei linguaggi orientati agli oggetti l'incapsulamento a livello di *classe* limita gli effetti del codice ad una parte ben definita di dati e l'ereditarietà permette l'evoluzione incrementale dei componenti grazie alla possibilità di aggiungere funzionalità o di ridefinirne le esistenti.

Anche la migliore suddivisione basata sulle funzionalità, però, è destinata a fallire per l'esistenza di problematiche che risultano essere *trasversali* all'intero sistema o ad alcune delle sue parti. Si pensi per esempio alla sincronizzazione di operazioni concorrenti, alla distribuzione dei componenti fra i nodi di una rete o all'implementazione della persistenza delle informazioni o della sicurezza. Questo tipo di problematiche non facilmente incapsulabili sono state chiamate *aspetti* del sistema e *programmazione orientata agli aspetti* è detto il recente filone di ricerca che cerca di identificare i costrutti linguistici appropriati per esprimerli in maniera separata. L'obiettivo è ovviamente di grande rilievo perché lo sviluppo di ogni sistema non banale coinvolge il lavoro di diverse persone, ciascuno con la propria visione degli obiettivi da raggiungere, e risulta fondamentale poter definire in maniera appropriata

le rispettive responsabilità.

L'idea fondamentale dei *linguaggi orientati agli aspetti* è quella di avere:

1. un'opportuna sintassi adatta a separare gli aspetti in unità con caratteristiche di modularità;

2. un modo di dichiarare quali sono i *punti di integrazione* fra il codice funzionale e gli aspetti.

Un apposito motore, chiamato *weaver*, provvederà poi (a tempo di compilazione o addirittura durante l'esecuzione) a "spalmare" gli aspetti sul codice funzionale per ottenere il sistema completo.

Il problema principale di tali tecniche sembra essere la rinuncia ai basilari principi dell'*information hiding*. Tale rinuncia rende difficile, se non impossibile, trattare gli aspetti come componenti veramente intercambiabili perché troppo legati alle peculiarità delle parti funzionali cui sono legate.

La programmazione orientata agli aspetti ha una particolare ricaduta sulle applicazioni distribuite. Infatti, la funzionalità dell'applicazione è percepita come sostanzialmente indipendente dall'effettiva distribuzione, dalla coesistenza di macchine differenti, dalla ridondanza dei componenti, dai processi concorrenti che ne realizzano il funzionamento, dalle politiche di sicurezza applicate, ecc. Sarebbe quindi auspicabile poter implementare l'applicazione come localizzata su di un'unica macchina e poi estenderla applicando gli aspetti voluti, guadagnando la possibilità di cambiarli liberamente.

Il contributo di questa tesi può essere cosí riassunto:

- sono stati analizzati diversi approcci alla programmazione orientata agli aspetti, mettendone in luce i problemi che ostacolano la possibilità di trattarli come veri componenti software;

- è stato proposto un nuovo linguaggio, chiamato Malaj, basato sulle seguenti idee:

  - gli aspetti che vale veramente la pena tentare di separare sono in numero limitato;
  - avendo in mente un aspetto ben preciso (per esempio la coordinazione) permette di concepire costrutti *ad hoc* che separano l'aspetto in maniera disciplinata e rispettosa dei principi dell'information hiding.

- sono stati definiti alcuni linguaggi specifici per la programmazione orientata agli aspetti nelle applicazioni distribuite:

  - la possibilità di coordinare funzionalità eseguite parallelamente;
  - la possibilità di rilocare sezioni di funzionalità in diversi spazi di indirizzamento;
  - la possibilità di dichiarare e gestire le situazioni eccezionali;
- è stato mostrato che l'approccio seguito permette di chiarire la relazione esistente fra aspetti e costrutti tipici dei linguaggi orientati agli oggetti come l'ereditarietà.

Contents

List of Figures

Listings

1 Introduction

Solution: The act of separating the parts of any body, or the condition of undergoing a separation of parts.
— Webster's Revised Unabridged Dictionary (1913)

Separation of concerns is a key engineering principle [18] applied in analysis, design, and implementation of systems. Designers want to think about one problem at a time and separation of concerns means decomposing a system into parts, each of which deals with, and encapsulates a particular area of interest, called a *concern*.

Software engineers learned how decomposition of a complex system into simpler sub-systems can make the problem tractable because the complete solution can be built out of sub-solutions, found relatively independently. In fact, most analysis and design notations and programming languages provide constructs for organising descriptions as hierarchical structures aggregating simpler modular units.

Moreover, the development of every non trivial system necessarily involves many people, and it is crucial to entrust individuals with precise responsibilities, according their skills, knowledge, and expertise. In fact, multiple perspectives of the system must exist to cope with different development phases, different issues, different stake-holders. These different *viewpoints* [20] need an explicit representation in order to make possible concurrently carrying out and integrating them in coherent analysis, design, implementation, testing and deployment artifacts.

Thus, smart techniques for expressing separate solutions to different concerns and composing them in the final system are needed at every stage of development. In particular, programming languages need proper idioms to help implementors in applying good "divide et impera" principles to their programs. Traditional programming languages have supported the partitioning of software in modular units of functionality. Such parts are then assembled to get the desired functionality of the whole system. The history of programming languages shows the evolution of linguistic constructs aimed at achieving isolation of a concern: in imperative and functional languages *procedures* and *functions* enable top-down reasoning. In object-oriented languages, *class encapsulation* limits the effects of change to localised portion of code and *inheritance* allows one to incrementally evolve a component by adding new features or redefining existing features.

However, even optimal functional decompositions omit to encapsulate some concerns because they *cross-cut* the entire system, or parts of it. As an example, suppose that a Java class is used to describe the pure functionality of certain objects. Additional separate issues may include the definition of:

- constraints on sequences of applicable operations (e.g., to get information from an object one must first apply a setup operation, and then one of a set of assignment operations);
- synchronisation operations to constrain concurrent access to the object (e.g., a consumer trying to read a datum from a queue must be suspended if the queue is empty);
- how objects are distributed on the nodes of a network, either statically or through dynamic migration;
- security or accounting policies (e.g., to get information from an object one must first ask some permission).

These kinds of separate issues that cannot be easily encapsulated in functional modules have been called *aspects* [37]. Such concerns are typically scattered over several units of encapsulation. Single modules become mix-ins of totally different concerns entangled in the same piece of code. *Aspect oriented programming* is the emerging research field born to identify appropriate linguistic means for isolating scattered concerns.

The basic idea is having an *aspect oriented language* providing:

1. some syntactic sugar to separate aspect code in novel modular units;
2. a way to declare *join points* that are the points in the functional code where aspect code will merge with it.

An engine, called *aspect weaver,* is responsible for mixing — either at compile time or at run time — functional and aspect code in order to produce a running coherent system.

The best known representative proposal in this area is AspectJ [70, 71] from Xerox Parc. The language provides an aspect oriented approach in which an **aspect** is a first class entity similar to a Java **class**. An **aspect** can have its own methods and attributes; furthermore it can *insinuate* any arbitrary piece of code in the Java **class**es composing the system. The join points between **aspect**s and **class**es are method signatures, that can be denoted also by using pattern matching, thanks to a regular expression syntax. Insinuated code has to be executed either at the very beginning or at the very end of the method acting as join point. However, it may freely manipulate **class** data to obtain the desired behaviour.

This is a very powerful mechanism that puts total control in programmers' hands, but it does not completely resolve the main issue of separation of concern. In fact, classes and aspects have to be designed together, because of the holes introduced in the information hiding boundaries.

This thesis argues that the problem of separating *every* concerns, still maintaining all the logical barriers between each of them, is not resolvable in its most general case and it proposes the less ambitious goal of trying to separate some *specific,* although important, predetermined concerns. In our vision, aspects should have their own separated production cycle. However, breaking encapsulation makes this very difficult, if not impossible. Thus our proposal gives up some flexibility and power in favour of understandability and ease of change.

We propose a new language, called Malaj, which has its roots in two basic assumptions:

1. only a (relatively small) set of possible aspects is worth separating from functional code;

2. having in mind a well defined aspect, it is possible to provide *ad hoc* constructs addressing this particular issue in a disciplined way.

Malaj constructs, thanks to their ad hoc nature, may have been carefully designed with "surgical" visibility on implementation secrets of functional modules, thus their relation with traditional object oriented code can be studied *a priori.* Furthermore, designing an aspect specific language may provide programmers a new tool embodying appropriate successful guidelines coming from the research work of experts of the field.

We focus on the particular domain of distributed systems. According to the definition given in [12], a distributed system is a collection of automata whose distribution is transparent to the users so that the system appears as one coherent machine. Users are not directly aware of the network, namely there are several machines, with different locations, storage replication, load balancing, concurrent processes, network failures, access protection, etc.. This makes it a perfect candidate for aspect oriented programming, because functionality is perceived as well separated issue, a goal to be reached notwithstanding any change in other concerns.

Thus, we concentrate on three aspects of distributed systems:

synchronisation: the ability of coordinating parallel threads of functionality;

network relocation: the ability of relocating the deployment status of piece of functionality;

exceptional behaviour: the ability of coping with exceptional states of the system.

The thesis is organised as follows: Chapter 2 presents the related problems of entangling and scattering of concerns and their relevance in modern software engineering; Chapter 3 details the approach followed by AspectJ and its drawbacks and pitfalls; Chapter 4 surveys other existing approaches; Chapter 5 describes Malaj, our aspect oriented language; Chapter 6 concludes the thesis by summing up the contribution of our work and envisioning improvements and further research.

2 Aspects of Complex Systems

*T*he astrologers call the evil influences
of the stars evil aspects.
— Francis Bacon

Separation of concerns is a very general and very powerful principle that applies to any large and complex human activity. It is especially used in software to express the ability to identify, describe, and handle important and critical facets of a software system separately.

Concerns are always related to a goal a stakeholder[1] wants to achieve with a software system or to anticipations or expectations he or she has on a system. A concern can be seen as a perspective that is taken by a stakeholder on a system. This is particularly true for systems which deploy a number of different technologies. For instance, building the software for controlling airplanes needs deep understanding of hardware and physical issues.

Well organized software systems are partitioned in modular units each addressing a well defined concern. Such parts are developed in relative isolation and then assembled to produce the whole system. A clean and explicit separation of concerns reduces the complexity of the description of the individual problems, thereby increasing the comprehensibility of the complete system.

Some concerns are only relevant in certain development stages, even though more often they are relevant during the complete software life cycle. Separation of concerns supports evolution and maintenance, facilitates reuse, it also enables consistency checking and correspondence between specification and implementation.

Even when the system is finished[2] having different models corresponding to different perspectives is a powerful tool useful for inferring

[1]With the term stakeholder we mean any person involved in the life cycle of the system: the entrepreneur who organizes the business venture and assumes the risk for it, the designer who sketches the system architecture, the programmer who writes the code, one of the marketing people who sell the product, the end-user who exploits the software in his own business, the customizer who adapts it to specific demands, etc.

[2]Of course, the notion of "finished system" is a controversial one: actually, complex

properties, deducting explanations of observed behaviors, or envisioning expected responses [54].

Separation and isolation are crucial if development of single parts has to be carried out concurrently to reduce time to market: people involved take different perspectives, exploit different development strategies, and have different responsibilities [20]. Only elements that are important for each stakeholder should be visible in her or his own viewpoint.

During implementation phases programming languages need to support programmers in isolating concerns and integrating them in coherent systems.

Traditionally, programming languages provide constructs to partition the software in modular units of functionality. Such parts are then assembled to get the desired functionality of the whole system. Traditional languages provide *procedures* and *functions*. Object-oriented languages break up programs in objects isolated by *class encapsulation*: this boundary limits the influence of pieces of code to localised regions; moreover, *inheritance* allows one to incrementally evolve components by adding new features or redefining existing features.

However, sometimes a concern is not easily factored out in a functional unit, because it *cross-cuts* the entire system, or parts of it. Synchronization, memory management, network distribution, load balancing, error checking, profiling, security are all *aspects* of computer problems that are unlikely to be separated in functional units. During last years hard research work was carried on concurrency, network distribution, security, etc. and skillful professionals in these areas are now emerging. Ideally, we would like to entrust such experts to implement the part of the system that impact on these aspects. However, it is typically not easy to give responsibility for these concerns to people different from implementors of other parts of the system, because the relevant code is often scattered across multiple components, tangled with other unrelated code.

Scattering is a problem because it hinders the possibility to reason about a concern in isolation, by temporarily ignoring what is currently irrelevent. For example, it may be difficult to predict deadlock or to detect deadlock when it occurs because it requires reasoning about two or more units at a time. Suppose (see Listing 2.1) we have a method

systems are never completely finished. They can be defective because they do not implement (or implement imperfectly) all the specifications given before building, but also because they neither cope with all requirements rose during the building process, nor with all needs that environmental changes impose. Here finished is to be taken as “deployed on users’ machines”

```
public void removeUseless(Folder file){
    synchronized (file){
        if (file.isUseless()){
            Cabinet directory = file.getCabinet();
            synchronized (directory){
                directory.remove(file);
            }
        }
    }
}
```
Listing 2.1: A method with synchronisation machinery

```
public void updateFolders(Cabinet dir){
    synchronized (dir){
        for (Folder f = dir.first(); f != null; f = dir.next(f)){
            synchronized (f){
                f.update();
            }
        }
    }
}
```
Listing 2.2: Another method with synchronisation machinery

`removeUseless()` in a database class. This method is called during the clean-up phase of the system. It receives a `Folder` object as parameter: this object represents some folder in the database system. The method controls the uselessness of the `Folder` by calling `isUseless()`. In order to act on the `Folder`, the method acquires the object lock of the `Folder`. If the `Folder` is really useless, the method simply remove the `Folder` from the `Cabinet`. The `Cabinet` can be found by a `getCabinet()` method, and the `Folder` can be deleted by invoking `remove()`. Just as with the `Folder` object, before it is possible act on the `Cabinet` object it is necessary to lock it. Now, let us suppose we have another method called `updateFolders()` (see Listing 2.2. This method receives a `Cabinet` object that represents a cabinet in the system. In order to act on this `Cabinet`, its object lock is needed. The act of updating the `Cabinet` is done by looping through all the `Folder`s in the `Cabinet` and calling the `update()` method. Again, the updating of `Folder`s needs the `Folder` lock.

This code can cause a deadlock. Suppose that a thread T_1 calls the method `updateFolders`(). It acquires the lock L_1 of the `Cabinet`. Now assume the `removeUseless`() method is called by a thread T_2. It locks the `Folder` with a lock L_2 and, after determining that it is indeed useless, it proceeds in locking the `Cabinet` with its lock L_1 in order to delete the `Folder`. At this point T_2 blocks and waits for the releasing of the `Cabinet` object lock.

But when the `Folder` on which `removeUseless`() is working is now accessed by `updateFolders`(), it tries to grab the object lock L_2. The deadlock situation arises because the `removeUseless`() method has the `Folder` lock L_2 and it is waiting for the `Cabinet` lock L_1 to be freed. In parallel the `updateFolders`() method holds the `Cabinet` lock L_1 and it is waiting for the `Folder` lock L_2 to be freed. This anomalous situation is hard to detect, but it is common to find code like this if it is written by people with no knowledge of each other work. A solution to the deadlock situation involves a major redesign of the system, with collaboration between the writer of `removeUseless`() and the one of `updateFolders`().

The problem dual to scattering is *code tangling:* modular units are often mixins of pieces of code that addresses different, unrelated concerns. For example, Listing 2.3 shows a synchronised `Stack` **class**. The core code is composed by statements that implement the stack functionality: a data structure in which elements can be accessed according the *last in, first out* policy. But the **class** contains also statements (indicated by `// synch`) that implement synchronisation among methods: no concurrent calls to `pop`() and `push`() are allowed, calls to `pop`() wait for elements in the stack, calls to `push`() wait for empty slots.

In the example functionality and synchronisation are tangled together, thus they have to be written at the same time. The same problem applies to the code in Listings 2.1 and 2.2. Moreover, because between the two there are no barriers of any kind, every change to one may have side effects to the other.

Thus, the question is: *"how non functional aspects, generally scattered and tangled within functional component can be managed as components as well?"*. Information hiding and encapsulation is the traditional way to introduce division of labor in software production, limiting the effects of change to localized portions of code, that become the bricks for building the whole system, but we need some new ideas to consider aspects real components, i.e., units of third-party composition [63].

```
package stack;

interface Constants{
    final int StackSize = 10;
}

public class Stack implements Constants{
    synchronized                              // synch
    public void push(Object o){
        while (top == StackSize−1) try{       // synch
            wait();                           // synch
        } catch (InterruptedException e){    // synch
            e.printStackTrace();              // synch
        }
        elements[++top] = o;
        if (top == 0) notifyAll();            // synch
    }
    synchronized                              // synch
    public Object pop(){
        Object ris;
        while (top == −1) try{                // synch
            wait();                           // synch
        } catch (InterruptedException e){    // synch
            e.printStackTrace();              // synch
        }
        ris = elements[top−−];
        if (top == StackSize−2) notifyAll(); // synch
        return ris;
    }

    private int top = −1;
    private Object[] elements = new Object[StackSize];
}
```

Listing 2.3: Scattering of the synchronisation concern

3 Aspect Oriented Programming

When the only hammer you have is C++, the whole world looks as a thumb.
— Keith Hodges

Software design processes and programming languages should mutually support each other. In particular, the necessity to maintain software imposes the two key principles of *factoring* and *locality* [22] in order to make programs readable and easily modifiable as much as possible. Thus, the language should allow programmers to factor concerns into one single unit and the effect of a language feature should be restricted to a well defined, "local", portion of the entire program.

Recently, some aspect oriented languages were proposed to make aspects clearly identifiable from functional code, which is written by using traditional constructs that are present in ordinary languages.

Aspect oriented languages (see Figure 3.1) provide support for writing encapsulated aspects thanks to constructs for:

1. syntactically isolating code for aspects;
2. identifying *join points* between aspect code and functional code;

Then, a *weaver* is responsible to mix — not necessarily at compile time — aspects and functional code in order to produce the running system.

Syntactic isolation establishes which entities can be manipulated by the weaver and scope rules among all program entities: how much does each concern know about each other? This is crucial to understand the coupling degree among the parts that compose the system.

The nature of the join points strongly affects the properties of the integration: its flexibility, the ability to understand the integrated system in terms of its components, reusability of components, and the nature and complexity of weaver and other supporting tools.

Integration, or weaving in the aspect oriented jargon, can be performed in different ways and at different time of developing. in fact, it can be done not just statically at compile, link, or even analysis time, but also dynamically at run time.

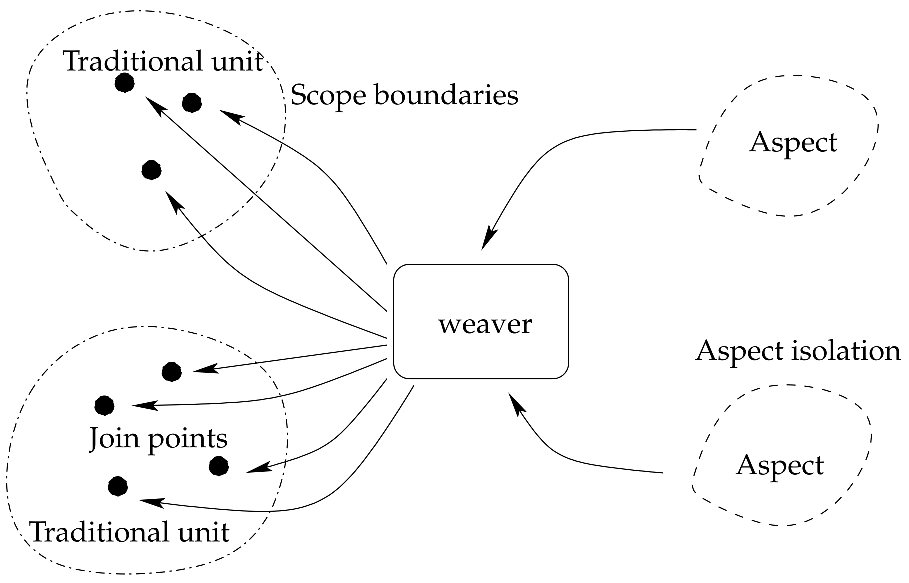

Figure 3.1: The basic idea of aspect-oriented programming

3.1 AspectJ

The best known system implementing an aspect-oriented approach is probably AspectJ [71, 70]. Designed and implemented at Xerox PARC, it is aimed at managing tangled concerns in Java programs.

In AspectJ[1] it is possible to define a first-class entity called **aspect**. This construct is reminiscent of the Java **class**: it is a code unit with a name and its own data members and methods.

In addition, **aspect**s may **introduce** an attribute or a method in existing **class**es and **advise** that some code is are to be executed **before** or **after** the execution of an existing **class** method.

For example, the **aspect** in Listing 3.2 adds a method `main`() to the **class** in Listing 3.1 for testing purposes.

Join points are the calls to functional methods. It is possible to schedule the execution of arbitrary code when the thread of computation enters in the called method (with **before** clause), or when it returns normally from the method (**after** clause), when it returns with an exception[2] (**catch** clause), after any returning action (**finally** clause). Se-

[1]Following code and considerations refer to AspectJ v 3.0

[2]For the sake of precision: **catch** and **finally** clauses were introduced with AspectJ 0.4, together with some small syntactical changes; nevertheless they fit smoothly in the model of AspectJ 0.3, and for our considerations can be treated here. More "revolutionary" improvements are discussed separately in Section 3.3.

```
package stack;

interface Constants{
    final int StackSize = 10;
}

public class Stack implements Constants{
    public void push(Object o){
        elements[++top] = o;
    }
    public Object pop(){
        return elements[top−−];
    }

    private int top = −1;
    private Object[] elements = new Object[StackSize];
}
```

Listing 3.1: An example of stack implementation

```
aspect TestStack{

    introduce Stack {
        public static void main(String[] args){
                Stack s = new Stack();
                s.push(new Integer(5));
                s.push(new Integer(23));
                Integer i = (Integer)s.pop();
                i = (Integer)s.pop();
        }
    }
}
```

Listing 3.2: Implementation of a debugging routine for the stack in Listing 3.1)

```
Object wovenMethod(Object[] parameters){
    try{
        beforeClause_code(parameters);
        originalMethod_code(parameters);
        afterClause_code(parameters, thisResult);
    } catch (Exception e){
        catchClause_code(parameters);
    }
    finally{
        finallyClause_code(parameters);
    }
}
```

Listing 3.3: Equivalent code generated by the AspectJ weaver

mantically, the concrete method code generated by the weaver is equivalent to Listing 3.3.

This mechanism can be used for coding synchronisation around the `Stack` in Listing 3.1. Listing 3.4 shows how is possible to acquire a lock before performing a `push`() or a `pop`() on the `Stack` and to release it after operation. The implementation exploits services of a suitable class `Lock` showed in Listing 3.5.

The association between aspect instances and objects is one-to-one. However, this can be changed by using the keyword **static**, this way a single aspect instance is associated to all the objects of a class. Aspect code may refer to object instance by using the `thisObject` keyword and to aspect instance by using **thisAspect**. Further, it is possible to access the method where the junction between the aspect instance and the object is done by using the keyword **thisJoinPoint**.

Classes are unaware of aspects, i.e. it is not possible to name an **aspect** inside a **class**: this because aspects are conceptually *a posteriori* with respect to classes, they are augmentation of a given functionality. In the production cycle supported by AspectJ, aspects are downstream from the writing of classes, that they assume existing and fixed.

Actually, some variance degree is allowed by the join point mechanism. Join points are not requested to be specified with a unambiguous method signature. Instead, it is possible to specify a set of methods by using a special syntax. For example,

private ! **static** **void** $*$() & (`ScreenA | ScreenB`)

denotes the intersection of all private, non-static, void methods without parameters and all the methods of the classes `ScreenA` e `ScreenB`.

```
aspect SyncStack{
    advise void MyStack.push(Object o){
        static  before {
            class NotFull implements Condition{
                public boolean check(){
                    return top != StackSize−1;
                }
            }

            l.acquireLockIf(new NotFull());
        }

        static  after {
            l.releaseLock();
        }

    }

    advise Object MyStack.pop(){
        static  before{
            class NotEmpty implements Condition{
                public boolean check(){
                    return top != −1;
                }
            }

            l.acquireLockIf(new NotEmpty());
        }

        static  after{
            l.releaseLock();
        }
    }
}
```

Listing 3.4: Implementation of synchronisation aspect for the stack in Listing 3.1 (see also 3.5)

```
interface Condition{
    boolean check();
}

class Lock{
    public synchronized void acquireLockIf(Condition c){
        while (!c.check()
                || (locker != null
                    && locker != Thread.currentThread())){
            try{
                wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
        }
        countLock++;
        locker = Thread.currentThread();
    }

    public synchronized void releaseLock(){
        if ( locker == Thread.currentThread()){
            countLock−−;
            if (countLock == 0){
                locker = null;
                notifyAll();
            }
        }
    }

    private Thread locker = null;
    private int countLock = 0;
}
```

Listing 3.5: Support classes for synchronisation

The use of not and "wildcard" enables the possibility of denoting a method that was not foreseen when the **aspect** was written.

Moreover, the weaver is aware of polymorphism. This way, every **aspect** that **introduce**s or **advise**s some code in a **class** `C` keeps augmenting every **class** deriving from `C`.

3.2 Problems and Pitfalls

In principle, the various aspects should not interfere with functional code, they should not interfere with one another, and they should not interfere with the features used to define and evolve functionality, such as inheritance.

Instead, coding systems with AspectJ is error prone. If **aspect**s and **class**es are not carefully design *together,* it is easy that unwanted clashes occurs. For example:

1. *Possible clashes between functional code and aspects.* Usually such clashes result from the need of breaking encapsulation of functional units to implement a different aspect. In AspectJ **aspect** code may access the private attributes of a **class**. This can be useful in some situations but results in a potentially dangerous breaking of class encapsulation. Imagine a situation (see Listing 3.6)in which a **class** `Foo` has a private variable `i` that needs to be accessed by **aspect** `Bar`. Imagine also that subsequently class `Foo` is changed by changing type of variable `i` from **int** to **double**. This results in breaking the **aspect** code. In general, it is not possible to change the internals of a functional unit without changing other aspects.

2. *Possible clashes between different aspects.* Suppose (see Listing 3.7) that a class `Point` exists with two variables `x` and `y` and two methods, `setX`() and `setY`(). Suppose we have developed an **aspect** `TraceBefore` (see Listing 3.8) to trace the start of execution of methods of **class** `Point` and an **aspect** `TraceAfter` to trace the end of execution of the same methods. The two **aspect**s work perfectly when applied individually (for example, to trace the start of execution or to trace the end of it). Unfortunately, since they introduce the same method (i.e., method `print`()) with different definitions, they fail when applied together.

3. *Possible clashes between **aspect** code and specific language mechanisms.* One of the best known examples of problems that falls

```
class Foo{
    private int i = 1;
    public void method(){
        System.io.println(i);
    }
}

class DoubleFoo extends Foo{
    private double i = 1.0;
    public void method(){
        System.io.println(i);
    }
}

aspect Bar{
    advise void Foo.method(){
        static before {
            i++; // it does not work with a double!
        }
    }
}
```

Listing 3.6: Breaking encapsulation causes problem in program evolution

```
class Example1{
 public static  void main(String args[]){
         Point p=new Point();
          p.setX(1);
          p.setY(1);
 }
}

class Point{
 int x,y;
        public Point(){
  x=y=0;
 }
        public void setX(int x){
         this.x=x;
 }
        public void setY(int y){
         this.y=y;
 }
}
```

Listing 3.7: A *Point* class

```
aspect TraceBefore {
  introduce private void
    Point.print(String methodName) {
    System.out.println("Tracing method "
                       +methodName +" before");
    System.out.println("x="+x+" y="+y);
  }
  advise void Point.setX(int i),
         void Point.setY(int i) {
     static  before  {
       print(thisJoinPoint.methodName);
     }
  }
}
aspect TraceAfter {
  introduce private void Point.print(String methodName) {
    System.out.println("Tracing method "
                       +methodName +" after");
    System.out.println("x="+x+" y="+y);
  }
  advise void Point.setX(int i),
         void Point.setY(int i) {
     static  after  {
       print(thisJoinPoint.methodName);
     }
  }
}
```

Listing 3.8: Two *Trace* aspects that rise a clash

into this category is *inheritance anomaly* [47]. This term was first used in the area of concurrent object-oriented languages [73, 3, 55] to indicate the difficulty of inheriting the code used to implement the synchronisation constraints of an application written using one of such languages. In the area of aspect-oriented languages, the term can be used to indicate the difficulty of inheriting the aspect code in the presence of inheritance. As an example, consider **class** `Window` in Listing 3.9. Methods `show`() and `paint`() cannot be called before method `init`() is called. This behaviour is controlled by the **aspect** `WindowSync`. Now consider **class** `SpecialWindow` in Listing 3.11. It redefines method `show`() in such a way that it does not require a previous invocation of method `init`()[3]. In principle, it should be possible to "inherit" the `WindowSync` **aspect** just modifying the code associated to method `show`() (e.g., replacing it with the empty sequence). Unfortunately, this is not possible and it is necessary to rewrite entirely the aspect code (see **aspect** `SpecialWindowSync` in Listing 3.12).

Thus, AspectJ gives aspects full control on internal details of their associated functional classes. This results in violating the object-oriented principles of protection and encapsulation, increasing the chance that the different aspects could interfere with each other or with the functional code.

3.3 Recent Developments in AspectJ

AspectJ is subject to active research and is evolving at a fast pace. Today the current release (v. 0.7) has a number of new interesting features [58].

On one hand, the new version is more respectful of encapsulation principles:

- `aspect`s are real units of encapsulation that must obey the same access control rules as Java code when referring to members of other units (however, there is the possibility to declare an `aspect` as **privileged** bypassing all access control rules);
- `aspect`s may extend previously produced units, reusing their code, and advises can be overridden;

[3]Note that this way of sub-classing `Window` is consistent with the object-oriented type theory [44], which requires subclasses not to strengthen the precondition for redefined methods.

```
class Example2{
 public static  void main(String args[]){
  Window w=new Window();
   w.init();
   w.show;
 }
}

class Window{
 public void init(){
                // ...
        }
 // Requires initialization
        public void show(){
                // ...
        }
        // Requires initialization
        public void paint(){
                // ...
        }
}
```

Listing 3.9: An *Window* class

```
class Example2{
 public static  void main(String args[]){
  Window w=new Window();
   w.init();
   w.show;
 }
}

class Window{
 public void init(){
                 // ...
        }
 // Requires initialization
        public void show(){
                 // ...
        }
        // Requires initialization
        public void paint(){
                 // ...
        }
}
```

Listing 3.10: An aspect to control the sequence of invocation of different methods

```
class SpecialWindow extends Window {
  // This version of show does not
  // need any initialization
  public void show() {
     // ...
  }
}
```

Listing 3.11: An extension of the *Window* (see Listing 3.9) class

```
aspect WindowSync{
 introduce boolean Window.initDone=false;
 advise void Window.init(){
  static  after{
   initDone=true;
  }
 }
 advise void Window.paint(){
  static  before{
   if (!initDone)
    System.out.println("Error: init never called");
  }
 }
}
```

Listing 3.12: An example of inheritance anomaly

- declaration of join points is now separated (by using a **pointcut** construct) from the definition of the actions attached to them: this allows them to be reused and, from a conceptual viewpoint, gives them a first-class entity status;
- conflicts among different `aspects` are now disciplined by precedence rules: defaults can be changed by the programmer by using the **dominates** keyword.

On the other hand, the weaving mechanism is now greatly improved. Join points are no more restricted to be method calls and their granularity is greatly refined.

- In version 0.7 arbitrary code may be introduced when an attribute is referenced (keyword **gets**) or assigned to (keyword **sets**), and it is possible to distinguish among method call, method reception, and method execution. This distinction arises from the dispatch mechanism of Java. A *call* is every call of static or non-static method or constructor. In Java a non-static, non-private method can be executed in two ways, with difference in the dispatch algorithm.

  The normal one is by using a call in the form

  ```
  this.method("parameters");
  ```

  In this case the actual method to be executed is found at run-time, after the resolution of the object dynamically bounded to **this**.

The actual class of **this** (which may be a derived class from the one who defines the instruction showed above), and a method with signature `method(String)` is searched starting in that class and going up in the inheritance chain.

Another way for executing a method is by using a call in the form

```
super.method("parameters");
```

By so doing the compiler statically determines which `method(String)` to execute (the nearest one in the inheritance chain), and at run-time that method is simply executed without any lookup step. Therefore, code cannot be inserted between *reception* and *execution*. AspectJ makes it possible to distinguish these two cases: **before** or **after** join points **receptions** means before or after the moment in which calls to a method with a particular signature is run, but the particular implementation of that signature is not yet determined, while **before** or **after** join points **executions** means before or after the moment in which a particular implementation of method signature is actually executed.

- A new keyword **around** allows the introduction of code that is not executed neither before nor after a join point, but *instead of* it.

By exploiting these very powerful mechanisms, aspect programmers may insinuate arbitrary code between every instruction of functional code, potentially changing its semantic in a totally arbitrary way. The semantics of the resulting program is not easy to understand because of the subtleties of different constructs. This makes AspectJ a low level tool, for coding aspects in a stage subsequent to the one in which functional code was produced. The process supported by AspectJ is somewhat similar to the one sketched in Figure 3.2. Aspect production conceptually follows class production, and aspect-oriented programming becomes a new mechanism of *evolution* (indeed much more powerful than inheritance) of actual components (classes or sets of classes), of which some knowledge about implementation details is needed.

However, the ideal process is one that encourages *division of labour*, as showed in Figure 3.3. In this vision aspects becomes true components that can be developed separately and with some degree of independence: they can be later *integrated* by weaving them with functional code and even reused in other, similar, contexts. However, this approach cannot be achieved by using AspectJ.

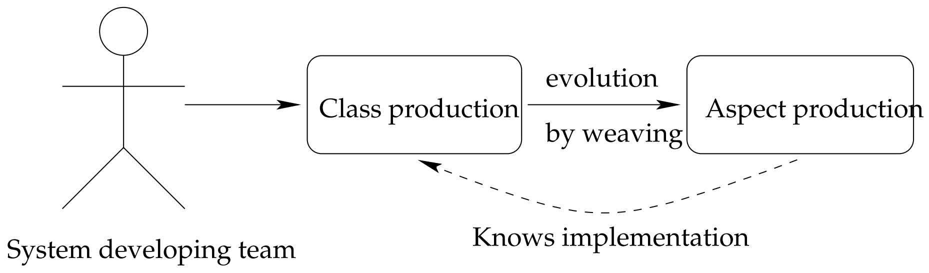

Figure 3.2: “Serial” process of software production supported by AspectJ

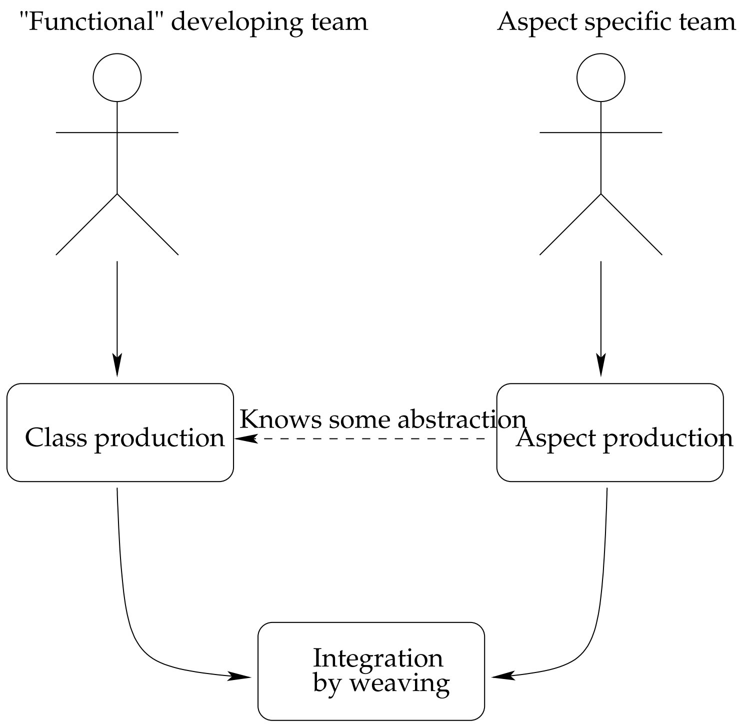

Figure 3.3: Ideal “parallel” process of software production

```
portal Library {
  Book find (String title){
    return:
      Book: {copy title, author, isbn;}
  }
}
```

Listing 3.13: A "portal" for a *Library* class

3.4 AspectJ Ancestors

The creators of AspectJ, in an early version of their system [4], provided two concern-specific languages: COOL, to control thread synchronisation; and RIDL, to program interactions among remote components. With these two aspect oriented languages it was possible to define *coordinators* and *portals*, which had total visibility of internal details of objects, but did not have the permission to change their state. With the constructor of **portal**, programmers could specify if data transfers across site boundaries are made possible by copying objects or transferring a reference to them: for example the **portal** of Listing 3.13 asserts that the `Book` object returned by the remote `find()` method of the **class** `Library` must contain a clone of the fields `title`, `author`, and `isbn` of the object computed by the functionality of the method.

With the constructor of **coordinator** programmers could specify self and mutual exclusion between class methods and pre/post-conditions on methods execution: for example, synchronisation of the `Stack` **class** can be obtained by using the **coordinator** showed in Listing 3.14. Currently, COOL has been transformed in a coordination library, whose features are woven into functional code by using the AspectJ engine. Again, **portal**s and **coordinator**s have total control on class secrets and there is no support to cope with inheritance or, in general, with evolution of **class**es.

This approach was abandoned by the AspectJ team because of its lack of generality. With COOL and RIDL it was possible to address only two specific concerns, while AspectJ is a tool of much lower level designed to cope with all conceivable cross-cutting aspects. We think that this ambitious goal represents also its weakness in achieving real separation of concern. AspectJ gives programmers a mechanism to manipulate units of Java code in almost arbitrary ways. It is difficult to encapsulate reusable manipulations because of their knowledge about

[4]For a detailed description see [45]

```
coordinator SyncStack : Stack{
    selfexclusive{pop, push};
    mutexclusive{pop, push};

    cond boolean full = false;
    cond boolean empty = true;

    push : requires !full;
    on_exit{
        if (top == StackSize−1) full = true;
        if (top == 0) empty = false;
    }

    pop : requires !empty;
    on_exit{
        if (top == −1) empty = false;
        if (top == StackSize−2) full = false;
    }
}
```

Listing 3.14: Coordination of the *Stack* class obtained with COOL

the details of functional code. In fact, in all the examples of AspectJ use appeared in literature (see, for instance, [32]), the reusable part of the code is encapsulated in ordinary classes. We suggest that anticipating the nature of the aspects we want to deal with can be the key to enable reuse, and in Chapter 5 we are going to propose a system supporting aspect-oriented programming in which we take a different approach: we define separate linguistic construct for *specific* aspect domains. This way we hope to reduce the clashes with traditional linguistic features and the need for breaking information hiding because the composition among known aspects and functionality can be carefully designed.

4 Other Approaches to Separation and Composition of Concerns

*I*t were not best that we should all think alike; it is difference of opinion that makes horse-races.
— Mark Twain

4.1 Subject-Oriented Programming and Hyper/J

Subject-Oriented Programming was proposed [57] as an extension of the object oriented paradigm to address the problem of handling different *subjective perspectives* on the objects to be modeled.

For example, the **class** representing `Employee`s for the administration people would include attributes such as `salary` or `socialSecurity`, whereas the human resource department would be interested in `skills` and `experience`.

Instead, one can have a **class** with all above attributes and "mark" each of them with the concern to which pertain (for example, the attribute skills could be marked as pertaining to the concern "human resource issues").

The need for different perspectives may emerge from different usage contexts, but also from different development viewpoints: one of the goals is the ability of adding unforeseen extensions to existing modules independently developed. A perspective is a system concern and *subjects* become the representation of the concern after filtering and composing modules. Furthermore, each subject constitutes a new dimension of the design process, each one should be, in principle, orthogonal to each other.

Hyper/J [64] is a meta-language to define multiple dimensions of concern within a software system written in Java. Hyper/J considers the system as a set of Java declarations (methods, attributes and classes) and provides:

1. a notation to map declaration units to arbitrary *concerns* and concerns to concern dimensions: i.e., each unit is assigned to one or more concern (see Figure 4.1);
2. a notation to describe various correspondences between declarations and definitions;
3. an engine to bind together declarations and definitions according to such correspondences.

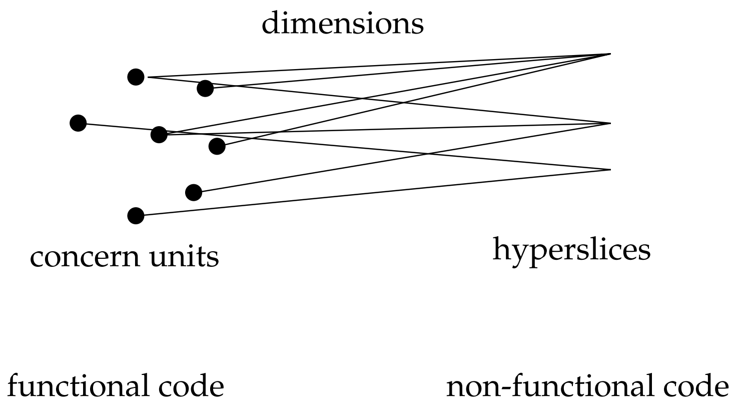

Figure 4.1: Hyper/J approach to separation of unencapsulable concerns.

Hyper/J considers the decomposition of the system in classes as a dimension of concern (the `ClassFile` dimension), not really different form others: its creators say that there is no tyranny of the dominant decomposition [64]. The set of all declaration units form a **hyperspace** from which the compiler can cut **hyperslices** containing all concerns pertaining to a dimension. **hyperslices** can be generated *declaratively complete* including in it all definitions needed for static checking. These slices are eventually integrated in executable **hypermodule**s by specifying which definitions must be linked to abstract declarations.

As an example, we can define a **hyperspace** composed by concern units of a **package** `stack` (see Listings 3.1, 4.1, and 4.2).

In this `hyperspace` each unit maps to exactly one concern, as specified in Listing 4.3: for instance, every `operation push` belongs to the solution addressing the dimension of concern `Features` and precisely the concern of implementing pure `Functionality`; instead, `operation stack.Stack.main` belongs to dimension of `Development` issues and address the problem of `Testing` the class. We can now build a `hypermodule HSyncStack` by merging

```
package stack;

public class SyncStack implements Constants {
    public void push(Object o){
        synchronized(lock){
            while (top == StackSize−1) try{
                lock.wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
            innerPush(o);
            if (top == 0) lock.notifyAll();
        }
    }

    public Object pop(){
        synchronized(lock){
            Object ris;
            while (top == −1) try{
                lock.wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
            ris = innerPop();
            if (top == StackSize−2) lock.notifyAll();
            return ris;
        }
    }

    private void innerPush(Object o) {}
    private Object innerPop() { return null; }

    private Object lock = new Object();

    private int top = −1;
    private Object[] elements = new Object[10];
}
```

Listing 4.1: A sample implementation for a synchronisation aspect for the *Stack* of Listing 3.1

```
hyperspace HStack
        composable class stack.∗;
        uncomposable class java.lang.∗, class java.io.∗;
```

Listing 4.2: An example hyper-space

```
package stack : Feature.Kernel // default
class stack.Stack : Feature.Kernel
class stack.SynchStack : Feature.Synch
operation stack.Stack.main : Development.Test
operation pop : Feature.Kernel
operation push : Feature.Kernel
field top : Feature.Kernel
field elements : Feature.Kernel
operation stack.SyncStack.innerPop : Feature.Synch
operation stack.SyncStack.InnerPush : Feature.Synch
field stack.SyncStack.popWait : Feature.Synch
field stack.SyncStack.pushWait : Feature.Synch
// class stack.Stack : ClassFile.Stack
// class stack.SyncStack : ClassFile.Stack
// interface stack.Constants : ClassFile.Constants
```

Listing 4.3: An example of concern mapping

the two (declaratively complete) `hyperslices` of `Features` and `Synchronisation`.

Moreover, we specify that during merge the code of unit `Features.-Synch.innerPush` has to be overridden by the code of `Features.-Functionality.push` and the code of unit `Features.Synch.inner-Pop` has to be overridden by the code of `Features.Functionality.-pop`. The produced module is a regular Java class that can be integrated in any application.

The power of composition mechanism is in the ability of specifying how the merging of units is actually performed: it can be a simple concatenation of their original definitions; it can be a reorder according to **before** and **after** relationships; instead, the last operation unit can override the preceding ones. Moreover, the value returned by a merged operation unit can be synthetized using an arbitrary aggregation of original definitions, by defining a **summary** unit (see Listing 4.5.

Despite (or because) of this powerful in expressiveness, this elegant approach presents some problems during the entire cycle of production

```
hypermodule HSyncStack
        hyperslices:
                Features.Functionality,
                Features.Synch
        relationships:
                overrideByName;

                equate operation
                        Features.Synch.innerPush,
                        Features.Functionality.push;

                equate operation
                        Features.Synch.innerPop,
                        Features.Functionality.pop;
end hypermodule
```

Listing 4.4: An example of hyper-module

```
// in the hypermodule specification
set summary function
        for action ExamplePkg.ExampleCls.check
                to ExamplePkg.ExampleCls.summarizeCheck;

// in a Java class

static void summarizeCheck(boolean[] returnResults){
        for(int i=0; i<returnResults.length; i++){
                if (! returnResults[i]){
                        return false;
                }
        }
        return true;
}
```

Listing 4.5: A sample aggregation of method results

of software:

- *Software creation.* Dimensions are partitions of the **hyperspace** composed by Java units: therefore, the declaration of these units has to precede the separation (mapping) of concerns. Thus, designers of the system receive no help from **hyperspace**s: they have to generate a bunch of **class**es to cope with all concerns. It is worth noting that it is not possible to assign the developing of a dimension to a separated team.
- *Software evolution.* Hyper/J can be very helpful to re-engineer an existing application, by identifying the different dimensions involved in the software. Evolution seems easier, but what about evolution of the evolved software? If designers do not want to go back to square one, they need a "hyper-Hyper/J" to evolve **hypermodule**s.
- *Software complexity.* The overall intricacy of the system does not diminish introducing different dimensions. There is no conceptual economy in the definition of dimensions of concern. Relations among software units become explicit, but their number do not decrease.

4.2 Adaptive Programming

Another interesting approach is *adaptive programming* [41]. Lieberherr et. al. envision a complete development process called Demeter in which an application is built by adding "enhancements" to a simple core program. Thus, if P is the initial program, B_i are behavioural enhancements, S is a synchronisation policy, and D is a distribution policy, the development of the application α_1 can be viewed as:

$$\alpha_1 = P + B_1 + B_2 + \cdots + B_n + S + D \tag{4.1}$$

and a similar application α_2 as:

$$\alpha_2 = P + B_1 + B_2 + \cdots + B_n + S_1 + D_1 \tag{4.2}$$

The problem is that the composition appearing in (4.2) may be completely different from the one in (4.1), thereby adaptions have to be applied to enhancements in order to fit them in the composition.

$$\alpha_1 = P + A_1(B_1) + A_2(B_2) + \cdots + A_n(B_n) + A_{n+1}(S) + A_{n+2}(D) \tag{4.3}$$

Functions A_j are called *adapters.*

The basic idea is to break up each enhancement in two parts: a general, reusable, one, and a specific adapter to the current context. As showed in Figure 4.2 the adapter is an intermediate layer responsible for doing the "dirty" work of manipulating the secrets of the functional code. Instead, the aspect code sees just a suitable abstraction of adapter, and this is the key of its reusability.

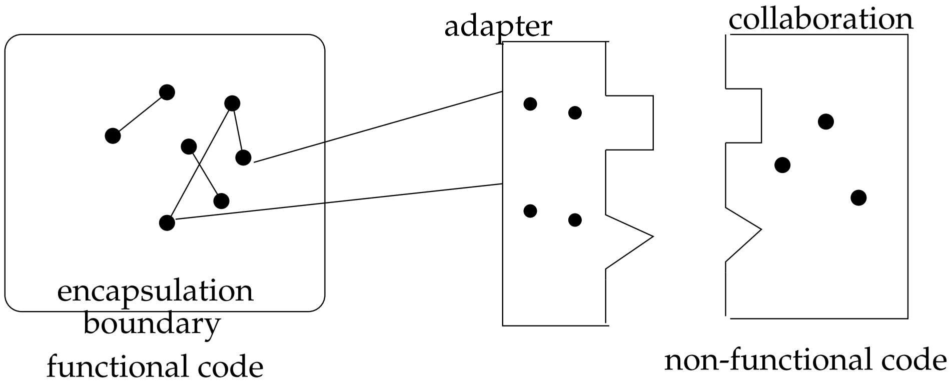

Figure 4.2: Adaptive approach to separation of concerns

The general part is called **collaboration** (see Listing 4.6).

Each **collaboration** has **participant**s with an **expect**ed interface. The code of the **collaboration** specifies how to use **participant**s' code to achieve its purposes: typically it **replace**s some **expect**ed methods with its own code. The **expect**ed interface is what the collaboration perceives as an useful abstraction of functional code for its own goals.

The **adapter** (see Listing 4.7) makes real **class**es suitable to the **collaboration expect**ed interface. It can freely make use of all features of classes (also the private ones) and special constructs exist to predicate on an entire graph of classes (traversal strategies[1]) to address the cross-cutting problem.

What to put in the adapter or in the collaboration is a crucial design decision. As AspectJ demonstrates, adapters can be empty, but this limits the potential reusability of collaborations. Viceversa, small collaborations are harmless, but rarely useful.

Inheritance can be used among **adapter**s, with special attention to traversal strategies. The main problem here is that **collaboration**s are

[1]Traversal strategies are graphs defined on a graph of classes G as any connected subgraph of the transitive closure of G. The transitive closure of $G = (V, E)$ is the graph $G^* = (V, E^*)$ where $E^* = \{(v, w) : \exists$ a path from vertex v to vertex w in $G\}$

```
collaboration BoundedDataStructure {
  participant S {
    expect void put(Object o);
    expect Object take();
    expect int used();
    expect int length();
    protected boolean full=false;
    protected boolean empty=true;
    replace synchronized void put(Object o){
      while (full) {
        try{
          wait();
        } catch (InterruptedException e) {};
      expected(o);
      if (used() == 1) {
        notifyAll();
      }
      empty = false;
      if (used()==length()) {
        full=true;
      }
    }
    replace synchronized Object take(){
      while (empty) {
        try{
          wait();
        } catch (InterruptedException e) {};
      Object r = expected();
      if (used() == length() − 1) {
        notifyAll();
      }
      full = false;
      if (used() == 0) {
        empty = true;
      }
      return r;
    }
```

Listing 4.6: Collaboration for synchronising a bounded data structure

```
adapter BoundedDataStructureToStack {
  Stack is BoundedDataStructure.S
    with {
      void put(Object o) {
        push(o);
      }
      Object take() {
        return pop();
      }
      int used() {
        return top+1;
      }
      int length() {
        return Constants.StackSize;
      }
    }
}
```

Listing 4.7: Adaption of collaboration of Listing 4.6 to the *Stack* class

"patterns" that one must use *as are.* No evolution is permitted, thus programmers are forced to keep them extremely general. This is intentional: Lieberherr team claims that this allows the exploitation of what is called the *Inventor Paradox* [42]: sometimes it is more easy specializing (adapting) a general solution to a problem than finding a specialized solution. However, the hard part is finding the suitable generalization. It is also worth noting that adaptive programming techniques are inherently static: composition of enhancements must be performed before ordinary compilation.

4.3 Design Patterns

Design patterns are good solutions to recurring problems of object-oriented design. Some smart solutions to some specific problems of scattering and tangling were published in the literature. For example, a well know pattern for separating synchronisation code from the functionality by using inheritance is described in [40] and its exploitation in the `Stack` example is showed in Listing 4.8: note that the solution showed is based on the assumption that the attribute `top` of `Stack` was declared **protected**, foreseeing its use in sub-classes.

```
package stack;

public class SyncStack extends Stack{
    public void push(Object o){
        synchronized(lock){
            while (top == StackSize−1) try{
                lock.wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
            super.push(o);
            if (top == 0) lock.notifyAll();
        }
    }

    public Object pop(){
        synchronized(lock){
            Object ris;
            while (top == −1) try{
                lock.wait();
            } catch (InterruptedException e){
                e.printStackTrace();
            }
            ris = super.pop();
            if (top == StackSize−2) lock.notifyAll();
            return ris;
        }
    }

    private Object lock = new Object();
}
```

Listing 4.8: A solution to the “Stack” synchronisation problem based on a design pattern by [40]

Solutions based on design patterns — while elegant, because they use only idioms based on inheritance and object composition to decouple and encapsulate the aspects of complex systems — present a number of problems [1] that limit their effectiveness.

The first problem is known as *object schizophrenia.* This problem arise when an object that conceptually is a single entity is split in one or more objects for implementation reasons. For example, if an algorithm is part of a given class and it needs to access some of the methods of the class, it can simply refer to **this**. But if the algorithm was factored out in an other class (as common in solutions based on the "strategy" pattern) some added complexity is needed to provide the algorithm with data and methods it needs from the original class.

Another problem is *traceability.* Given a concrete implementation of a solution, it is usually unclear which design patterns were actually exploited. The code implementing a pattern is itself intertwined with other aspects and scattered over a number of components. This indirect representation hinders maintenance, reuse and evolution. Moreover, design patterns often increase the fragmentation and complexity of the design by introducing extra methods and classes. That is, design patterns sometimes can worst the illness they are supposed to cure.

Furthermore, there is the so called *preplanning problem*. Design patterns can improve adaptability of software only if adaptations were anticipated during the design phase. One of the goals of aspect oriented programming is unanticipated evolution: for instance, we would like to add the distribution aspect to an application conceived for a single machine environment.

5 Malaj: a Multi Aspects LAnguage for Java

To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one's subject matter in isolation.
— Edsger Dijkstra

In Chapter 3 we analysed AspectJ, probably the best-known, general purpose aspect-oriented language. We argued that its "general-purpose" approach provides the maximum expressive power at the expense of violating the object-oriented principles of protection and encapsulation, thus increasing the chance that the different aspects could interfere with each other or with the functional code. All the problems discussed in Section 3.2 show that aspect-oriented programming is still in its infancy. However, in our opinion, many problems might result more from the linguistic choices made in developing AspectJ, rather than from intrinsic limitations of the aspect-oriented approach. In AspectJ (see Figure 5.1), aspects have full control on internal details of their associated classes. This results in violating the object-oriented principles of protection and encapsulation, thus increasing the chance that the different aspects could interfere with each other or with the functional code.

These theoretical considerations are supported by empirical studies on programmers working with AspectJ [69]: these studies demonstrate that programmers may be better able to understand an aspect-oriented program when the effect of the aspect code has a *well-defined scope.* In another paper, Kendall describes the problems that they had in understanding the relation among traditional constructs and aspect-oriented ones [31].

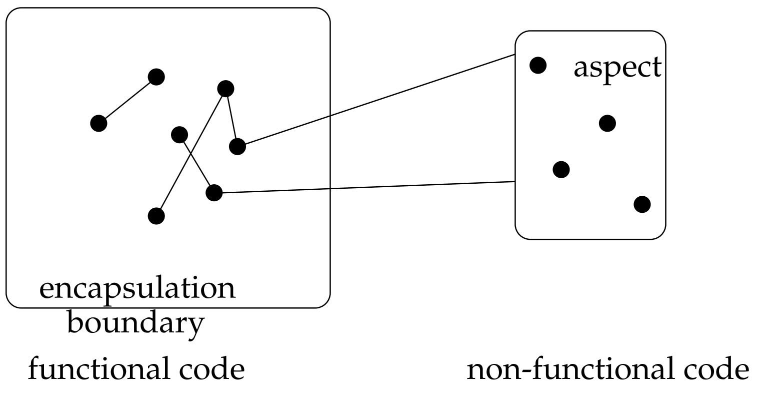

Figure 5.1: AspectJ approach to separation of unencapsulable concerns.

Our proposal, called Malaj[1], as opposed to other approaches mentioned in the previous chapters is not a general purpose aspect oriented language. It focuses on a well-defined set of aspects and provides a different linguistic construct for each aspect. Figure 5.2 illustrates the philosophy adopted: we aim at identifying some *concern-specific* relationships, emphasising the need for restricted visibility and for clear rules of composition with traditional constructs.

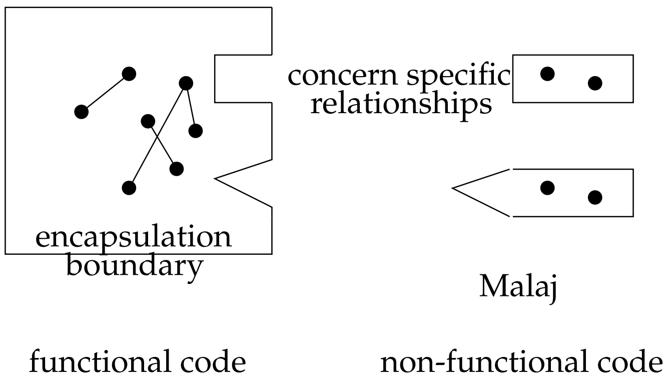

Figure 5.2: Malaj approach to non-functional aspect separation

Predefining the set of possible aspects an aspect-oriented language should deal with, and then providing ad-hoc constructs to implement these aspects, makes it feasible to provide limited visibility of the fea-

[1]the name Malaj was chosen after the Malay archipelago that comprises the Java island, suggesting that functional code (Java) addresses just one concern of building a system (though often the most important one), but other concerns need to be taken in account in other islands of the mind.

tures of the functional module to which the different aspects apply. As the next sections will show, this approach offers a good compromise between flexibility and power, on the one side, and understandability and ease of change on the other. It does not allow programmers to use the aspect-oriented language to code any conceivable aspect, but it limits the problems encountered with a general purpose approach.

As its name suggests, Malaj is an aspect-oriented extension to Java. Malaj is born to provide support to the aspects that arise when implementing applications whose functionality is spread on a set of machines, communicating via a network. Such distributed systems often appear to end-users as one coherent system. Users are not directly aware of the network, namely there are several machines, with different locations, storage replication, load balancing, concurrent processes, network failures, access protection, etc.. Thus, seems a sensible choice to have separated aspects for addressing these issues, and plug them in the core applications according needs, instead of tangling these concerns in functional code. We have focussed on three important aspects of distributed system, namely synchronisation, network relocation, and exceptional behavior. In distributed system synchronisation is need because threads of control are executed concurrently. Network relocation is the ability to change the network node where an object resides, in order to exchange data, code or balance the system load. Exceptional behavior management is indeed not peculiar to distributed systems, nevertheless is important when you have components produced or deployed by different parties.

To describe the Malaj constructs for aspect programming, the following sections refer to a common example: a very simplified electronic commerce system. The system (see Listing 5.1) is composed of two main classes: `Shop` and `Customer` (see Listing 5.3 and Listing 5.2.

The `Shop` class provides methods to add and remove items from the list of available articles. It also exports two methods to ask for the price of a specific article and to deliver the article to a specific address after it has been bought. Class `Customer` models the behaviour of an e-commerce agent: it goes through a list of shops in search of a given article looking for the best price. At the end of the process, it buys the article. Finally, class `ECom` creates two customers and starts them.

5.1 Synchronisation

Synchronisation between the different units that compose an application is a central aspect for any, non-trivial, software. According to

```
public class ECom {
    private Shop[] shops;
    private Customer paul, john;

    public ECom(){
        Shop[] shops = new Shop[5];
        for(int i=0; i<5; i++) {
            shops[i] = new Shop();
            // add new articles to shops[i]
            shops[i].addArticle("book", 40−i);
            shops[i].addArticle("CD", 10+i);
        }
        paul = new Customer(shops, "paul home","book", 60);
        john = new Customer(shops, "john home","CD", 30);
    }

    public void startShopping(){
        paul.start();
        john.start();
    }

  public static  void main(String[] args){
      ECom e = new ECom;
      e.startShopping();
  }
}
```

Listing 5.1: An example of a simplistic e-commerce application

```
public class Customer extends Thread {
  private Shop bestShop;
  private int bestPrice;
  private Shop[] shops;
  private String myAddress;
  private String desire;
  private int wallet;

  public Customer(Shop[] shops, String address,
                 String article, int wallet) {
    this.shops = shops;
    myAddress = address;
    desire = article;
    this.wallet=wallet;
    bestShop = null;
    bestPrice = wallet;
  }

  public void shopping(Shop s) {
    int price = s.query(desire);
    if(price <= bestPrice) {
      bestShop = s;
      bestPrice = price;
    }
  }
  public void buy() {
    if(bestShop != null) {
      bestShop.deliver(desire, myAddress);
      System.out.println("I bought a "
                        + desire+" at "+ bestShop);
    }
  }
  public void run() {
    for(int i=0; i<shops.length; i++) {
      shopping(shops[i]);
    }
    buy();
  }
}
```

Listing 5.2: A *Customer* class

```
public class Shop {
    private Hashtable goodies;
    public Shop() {
        goodies=new Hashtable();
    }
    public void addArticle(String article, int price) {
        goodies.put(article, new Integer(price));
    }
    public void removeArticle(String article) {
        goodies.remove(article);
    }
    public int query(String article) {
        return ((Integer)goodies.get(article)).intValue();
    }
    public void deliver(String article, String address) {
        // deliver the article to the address specified
    }
    public boolean isEmpty() {
        return goodies.isEmpty();
    }
    public boolean hasArticle(String article) {
        return goodies.containsKey(article);
    }
}
```

Listing 5.3: A *Shop* class

the WordNet Dictionary [5] synchronisation is “the relation that exists when things occur at the same time”.

To express this aspect it is necessary to clearly state what happens when a functional unit is invoked in a context where other functional units can be invoked concurrently as well. When a service is requested three cases may arise:

1. the call violates the synchronisation policy and an exception is returned to the caller;

2. the call is suspended until some condition becomes true according to the synchronisation policy;

3. the call may proceed because no synchronisation rules are violated.

To support this aspect, Malaj provides the **guardian** construct. Each **guardian** is a distinct source unit with its own name, possibly coded in a different source file. Each **guardian** is associated with a particular **class** (i.e., it **guards** that class) and expresses the synchronisation constraints of a set of related methods of that class. Each **class** has at most one **guardian**. Listing 5.4 and Listing 5.5 show respectively the **guardian**s for classes `Shop` and `Customer` mentioned above.

The Malaj core language is a reduced version of Java, which does not include the features that are provided through the separately specified aspects. More specifically, the Java keyword **synchronized** cannot be used in Malaj, and the same is true for the methods `wait`(), `notify`(), and `notifyall`() of the standard Java class `Object`.

For each **class** `C`, the **guardian** `G` of `C` basically represents the set of **synchronized** methods of `C`. As an example of the **synchronized** statement see the **guardian** `ShopGuardian` in Listing 5.4. `G` expresses also the conditions that, if not satisfied, result in an exception when `m` is called (i.e., the **deny** guards), and the conditions that, if not satisfied, result in suspending the caller of `m` (i.e., the **suspend** guards).

As an example of the **deny** and **suspend** statements, see methods `addArticle` and `deliver` of guardian `ShopGuardian` in Listing 5.4, respectively. Observe that **deny** guards are always considered before **suspend** guards and they are considered in the order in which they appear in the code. This means that if different **deny** and **suspend** guards are **true**, only **deny** guards are considered and among them the first is taken and the exception it defines is returned to the caller.

A **guardian** may include also a set of local attributes and method definitions to code guards that depend on state conditions (e.g., attribute `elements` of guardian `ShopGuardian` in Listing 5.4). Finally, for

```
guardian ShopGuardian guards Shop {
  HashSet elements=new HashSet();
  synchronized {
    addArticle, removeArticle,
    query, deliver
  }
  void addArticle(String article, int price):
    deny A: (price<0) with new PriceTooLow();
    before {
        elements.add(article);
    }
  void removeArticle(String article):
    deny A: (!elements.contains(article))
            with new ArticleNotFound();
    before {elements.remove(article);}
  int query(String article):
    deny A: (!elements.contains(article))
            with new ArticleNotFound();
  void deliver(String article, String address):
    suspend A: (!elements.contains(article));
}
```

Listing 5.4: A “guardian” for the *Shop* (see listings 5.3) class

```
guardian CustomerGuardian guards Customer{
  void Customer(Shop[] shops, String address,
              String article, int wallet):
    deny A: (shops==null || shops.length<1)
            with  new NotEnoughShops();
         B: (wallet<1) with new NotEnoughMoney();
}
```

Listing 5.5: A "guardian" for the *Customer* (see listings 5.2) class

each method `m` of the guarded class, the **guardian** may introduce a fragment of code to be executed before or after `m` (e.g., method `addArticle` of **guardian** `ShopGuardian` in Listing 5.4).

An important point is that, in order to avoid breaking object encapsulation and to increase separation between the functional and synchronisation aspects, **guardian** code — i.e., **deny** and **suspend** guards, and **before** and **after** clauses — cannot access private elements of the guarded class and has read-only access to the **public**, **protected**, and package attributes of the guarded **class**.

The ultimate goal of Malaj to enable a separate production cycle for aspects, which means that **class**es may evolve independently from respective **guardian**s. A precise relationship between the synchronisation aspect and inheritance is needed, as stated by the following rules:

1. The **guardian** of a **class** `C` is inherited by all the extensions of `C` that do not have a different **guardian**.

2. A **guardian** `G1` always extends a parent **guardian** `G`. If not explicitly mentioned, the parent **guardian** of `G1` is the **guardian** `malaj.Guardian`, which is part of the Malaj library. `G1` inherits all the synchronisation constraints specified by `G` and it may add new guards, redefine existing ones, or remove them. To distinguish between added and redefined guards, each guard of a given method `m` has its own label (see Listing 5.4). A guard in `G1` that has the same label of a guard in `G` redefines it, otherwise it is considered as a new guard.

3. The **guardian** of a **class** `C` must extend the **guardian** of the parent **class** of `C`.

4. The guards redefined in `G1` cannot be stricter than the original ones. In fact, as the next point explains, a sub-guardian `G1` guards a **class** that extends the **class** guarded by the parent **guardian** of

G1 and, as observed by [44], the precondition of a sub-class cannot be stronger than the precondition of the parent class to preserve correctness of polymorphic calls.

5. To reduce the impact of inheritance anomaly[2], the **before** and **after** clauses of a **guardian** G may refer to the corresponding clauses of the parent **guardian** through the statement **super**(). Similarly, in redefining a guard it is possible to refer to the original guard through the construct **super**().

5.2 Relocation

Distributed systems have to be aware of networks and code implementing network awareness is typically dispersed among functional units, thus representing a good candidate to be *aspectified*. In particular, programmers should be able to move objects among sites, in order to exchange data and code.

We identify two relationships to be maintained as objects move:

Ownership: if an object A owns an object B, then A is the only object entitled to move B. By default, B follows A in its movements.

Interest: if an object A is interested in B, A has to be always able to reach B, but A and B move completely independently.

If an object A does not own B and is not interested in it, it simply does not care of B's location, and even of its existence. Evidently, ownership implies interest. Each object may have at most one owner.

These relationships are inherently dynamic: they are subject to change during program execution, as objects change their interest in other objects according to the programmers' needs.

Malaj provides the **relocator** construct. Each **relocator** is a distinct source unit with its own name, possibly coded in a different source file. A relocator is associated with a particular **class** (i.e., it **relocates** the objects of that **class**). Relocation actions can be executed before or after the execution of any method. To specify this, the **relocator** provides **before** and **after** clauses that allow programmers to introduce the pieces of code that will be executed before or after the execution of the method (see example in Listing 5.6).

[2]Inheritance anomaly [47] problem appears when a new definition in a sub-class forces to redefine all super-class operations to cope with new synchronisation constraints

In order to preserve class integrity, within **before** and **after** clauses one is not allowed to change attributes (i.e., the internal state of an object can be changed only by using the methods it provides). However, it is possible to:

- take or release the ownership of an object, by using the methods:

  ```
  takeOwnership(Object owned)
      throws ObjectOwnedException

  releaseOwnership(Object owned)
      throws NotOwnerException
  ```

  Only the owner is allowed to release ownership and only objects that have no owner can be arguments of `takeOwnership`(). By default, each newly created object is owned by the object that has created it.

- express or retract the interest in an object, by using the methods:

  ```
  expressInterest(Object o)

  retractInterest(Object o)
  ```

- fix the location of an owned object, by using the methods:

  ```
  pin(Site s, Object owned)
      throws NotOwnerException

  unpin(Object owned)
      throws NotOwnerException
  ```

  Unpinned objects reside in the same site of their owner. Pinning an object in a site different from the one in which it currently resides means moving it on a new location, together with all owned objects that were not previously pinned.

- refer to variable and method definitions that are local to the **relocator**.

Listing 5.6 and Listing 5.5 show respectively the **relocator**s for **class**es `ECom` and `Customer` introduced above. After creation of an `ECom` instance, some `shops` are distributed in a worldwide `market`.

`Customer`s `pin`() their `wallet` in a secure site (`secureHome`) and query around for best prices.

```
relocator EComRelocator relocates ECom{
  Site[] market;
  EComRelocator(){
    market = new Site[shops.length()];
    for(int i=0; i<shops.length(); i++){
      // just an example...
      Site[i] = new Site("host[i]");
  }
  after ECom(){
    for(int i=0; i<shops.length(); i++){
      try{
        this.takeOwnership(shops[i]);
        this.pin(market[i], shops[i]);
      }
      catch(Exception e){
        e.printStackTrace();
    }
  }
}
```

Listing 5.6: A "relocator" for the *ECom* application (see Listing 5.1)

```
relocator CustomerRelocator relocates Customer{
  Site secureHome = new Site("home");
    after Customer(){
      try{
        // Customer creates wallet
        // next instruction is redundant
        this.takeOwnership(wallet);
        // pinned wallet doesn't follow me
        this.pin(secureHome, wallet);
      }
      catch(Exception e){
        e.printStackTrace();
      }
   }
   before void shopping(Shop s){
     this.expressInterest(s);
   }
}
```

Listing 5.7: A "relocator" for the *Customer* class (see Listing 5.2)

The limited semantic of the relocation aspect, enables the definition of simple rules for clarifying the relationship between this aspect and inheritance, the following rules exist:

1. The **relocator** of a **class** `C` is inherited by all the extensions of `C` that do not have a different **relocator**.
2. The **relocator** `L1` of a **class** with a relocated ancestor (by **relocator** `L`) may add **before** and **after** clauses for methods not considered in `L` and may redefine `L` clauses.
3. To reduce the impact of inheritance anomaly, the **before** and **after** clauses of a **relocator** `L1` may refer to the corresponding clauses of the parent **relocator** `L` through the statement **super**().

The model of distribution depicted by **relocator**s, while simple, is powerful enough to implement most frequently used relocation policies. For example, in their system FarGo, Holder et al. use five basic types of reference [30] between objects[3] possibly located on different sites:

1. $Link(\alpha, \beta)$, meaning that α and β may or may not be located in the same site and that relocation of α does not affect the location of β and vice-versa. This is the same that neither α is interested (thus it does not own) in β, nor β is interested in α.
2. $Pull(\alpha, \beta)$, meaning that α and β are located in the same site and a relocation of α moves β. This can be obtained by stating that α owns β and β is not pinned.
3. $Duplicate(\alpha, \beta)$, meaning that α and a copy of β are located in the same site and a relocation of α moves the copy of β. With $\gamma = \beta.clone()$ this is equivalent to a $Pull(\alpha, \gamma)$
4. $Stamp(\alpha, \beta)$, meaning that α and an instance of β's type are located in the same site and a relocation of α moves the instance of β's type. With $\gamma = new\beta.getClass()$ this is equivalent to a $Pull(\alpha, \gamma)$
5. $BiDirectionalPull(\alpha, \beta)$, meaning that α and β are located in the same site and either a relocation of α moves β or a relocation of β moves α. This can be obtained by stating that α owns β and β owns α; after pinning α or β in a site they have to be unpinned immediately.

[3]Actually, they speak about *complets:* a group of objects that is their unit of migration.

5.3 Exceptional Behaviour

Design by contract [48, 49] is a programming style popularised by Bertrand Meyer of programming based on the idea that the relationship between a class and its clients may be seen as a formal agreement, expressing each party's rights and obligations. Such a precise definition of every module's claims and responsibilities can be used also to decide how to deal with run-time errors, i.e. with contract violations, defining an exceptional behaviour alternative to the normal one (that is executed when contracts are satisfied).

The basic idea of design by contract is that a specification of conditions that are guaranteed to be valid before and after the execution of a service should be made explicit. This can be done by using the so called *Hoare triples* [29] specifying a service S as $\{P\}S\{Q\}$, meaning that any execution of S, starting in a state where P holds, if the program terminates, leads to state where Q holds. The *precondition* P expresses the constraints under which a service will performed properly; The *postcondition* Q expresses properties of the state resulting from the service execution.

Preconditions and postconditions describe the properties of individual services. There is also a need for expressing global properties which must be preserved by a family of services. Such properties are called *class invariant*, capturing the deeper semantic properties and integrity constraints characterizing a class. Invariants apply not only to the methods actually present in a class, but also to any ones that might be added later, thus serving as control over its future evolution. This will be reflected in the inheritance rules. In the contract metaphor invariants are regulations that apply to all contracts within a certain category.

To express the contractual aspect of a class, Malaj provides the **notary** construct. Each **notary** is a distinct source unit with its own name, possibly coded in a different source file. Within a **notary** it is possible to assert properties that some part of the class should satisfy at certain stages of execution of its methods [2]. A **notary** may **witnesses** that:

1. a precondition is **require**d to hold **before** a method is executed;
2. a postcondition is **ensure**d to hold **after** a method execution;
3. a condition is **always** preserved by calling any method of the class. This is merely a shortcut for a condition that otherwise appeared in all preconditions and all postconditions.

For each precondition it is possible to indicate an `Exception` to be thrown if the condition does not hold when the method is called. For each postcondition it is possible to indicate an `Error` to raise if the condition is not true after the execution of the method (see Listing 5.8). Using different exceptions or errors allows programmers to distinguish the condition that does not hold. Preconditions and postconditions are **boolean** expressions written by using only public features (according to Meyer's Availability rule [49]) of the class and they must not use neither methods with side effects nor assignments. It is indeed possible, and usually desirable, to define helpers to express complex conditions (for example, the method `wellFormed`() in Listing 5.8). A postcondition can refer to the return value of a method by using the keyword **result**.

Preconditions bind the client of a method: they define the conditions under which a call is legitimate. It is an obligation for the client and a benefit for the supplier. The postconditions bind the class: it defines the conditions that must be ensured by the execution of the method. It is a benefit for the client and an obligation for the supplier. The benefits are, for the client, the guarantee that certain properties will hold after the call; for the supplier, the guarantee that certain assumptions will be satisfied whenever the routine is called. The obligations are, for the client, to satisfy the requirements as stated by the precondition; for the supplier, to do the job as stated by the postcondition.

To cope with contractual exceptions, notaries apply the principle *"You broke it, you buy it"*. When a precondition fails to hold the responsible for that is the client who called the method: the **notary** throws an `Exception` that the client should manage and the service is not provided. When a postcondition fails to hold the responsible is the supplier and an `Error` is raised. However, it is possible to specify a **rescue** clause in which some code tries to remedy to the situation restoring invariants or restoring preconditions to enable a **retry** (see Listing 5.9). The resulting thread of control is equivalent to the one showed in Listing 5.10.

A precise relationship between this aspect and inheritance may be derived from Meyer's Assertion Redeclaration rule [48]: a routine redeclaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger. Pragmatically:

1. the logical conjunction of **require** clauses of a **notary** `N1` that **witnesses** a **class** `D` derived from a **class** `B` witnessed by a **notary** `N` is in logical *or* (`||`) with the logical conjunction of **require** clauses of `N`;

```
notary ShopNotary witnesses Shop {
  void addArticle(String article, int price):
    before {
        require (article != null) with new NullArticle();
        require (!article.equal(""))
            with new EmptyArticle();
    }
    after {
        ensure (!isEmpty()) with new EmptyShopError();
        ensure (!hasArticle(article)) with new ShopError();
    }
  void removeArticle(String art):
    before {
        require (art != null) with new NullArticle();
        require (hasArticle(art)) with new EmptyArticle();
    }
    after {
        ensure (!hasArticle(article)) with new ShopError();
    }
  int query(String art):
    before {
        require (article != null) with new NullArticle();
        require (hasArticle(art)) with new EmptyArticle();
    }
    after {
        ensure (result >= 0) with new PriceError();
    }
  public static boolean wellFormed(String address){
        // true if address is a true address
  }
  void deliver(String article, String address):
    before {
        require (article != null) with new NullArticle();
        require (wellFormed(address)) with new Address();
    }
    after {
        ensure (getCustomer(address).hasArticle(article))
        with new ShippingError();
    }
  always { true // default }
}
```

Listing 5.8: A "notary" for the *Shop* (see listings 5.3) class

```
notary ShopNotary witnesses Shop {
  public static  boolean wellFormed(String address){
        // true if address is a true address
  }
  void deliver(String article, String address):
    before {
        require (article != null) with new NullArticle();
        require (wellFormed(address)) with new Address();
    }
    after {
        ensure (getCustomer(address).hasArticle(article))
        with new ShippingError();
    }
    int retries = 0;
    rescue (ShippingError e){
 if (retries == 0){
    String.upcase(address); // sometimes it works
    retry;
 } // else fail
    }
}
```

Listing 5.9: A rescue clause

```
void thread(){
    boolean retry = false;
    do{
        try{
            method();
            if (! postcondition()){
                throw new Exception();
            }
        }catch (Exception e){
            System.out.println(stato);
            // set retry according rescuer
        }
    } while(retry);
}
```

Listing 5.10: Retry semantics

2. the logical conjunction of **ensure** clauses of a **notary** `N1` that **witnesses** a **class** `D` derived from a **class** `B` witnessed by a **notary** `N` is in logical *and* (&&) with the logical conjunction of **ensure** clauses of `N`;

3. the **always** clause of a **notary** `N1` that **witnesses** a **class** `D` derived from a **class** `B` witnessed by a **notary** `N` is in logical *and* (&&) with the **always** clause of `N`;

5.4 Considerations

The three aspect languages described above share a common approach: we gave up with the intent of separating *every* conceivable concerns, and we are trying to separate some *specific,* although important, predetermined concerns, still maintaining all the logical barriers among each of them, and the functional code. The ultimate goal is being able of identifying some *concern-specific* relationships.

Aside this, the three languages are quite different, because the aspects they address are very different. Synchronisation is a "low-level" issue: in fact, Java itself provides suitable (and sufficient) for obtaining thread synchronisation. The problem is that by using simple Java statements synchronisation and coordination become scattered over many functional units: aspect-oriented programming aims at eliminating exactly this. In a sense, synchronisation is a *pure* aspect, that has to be applied to *pure* functional code (i.e., Java code, cleaned from **synchronized** statements). Relocation, instead, is performed by pieces of code that could be dispersed all around functional code: the aspect language helps in grouping these pieces, but the structure of the relationship between functional and aspect code is not really specific. The most different one is the aspect language addressing exceptional behavior. In real life contracts are signed *before* services are served. Thus, conceptually, this aspect precedes functional code and it serves as control over the current use or future evolution (thanks to inheritance rules) of the class. In fact, many existing proposals exist for adding "design by contract" in a preprocessing step on Java code (see for example [2]).

Notwithstanding their differences, the three aspects can be implemented by exploiting a common architecture. An architecture view of the Malaj system is sketched in Figure 5.3. For each aspect A_i there is a corresponding weaver W_i. The weaver takes in input an aspect A, the class C to which A refers, and any other class needed for ordinary Java compilation. The aspect A is transformed in a suitable Java class α, from which aspect objects will be instatiated.

The weaving phase produces also a class K equivalent to C (i.e., it has the same interface of C), but instrumented to be able to use associated aspect objects. The instrumentation of class K is independent from the kind of aspect: the methods of K are the same methods of C with an added part responsible of managing a list of aspects. For example, if C has a method $m_c()$, the class K has a method $m_k()$ acting roughly as follow:

1. it examines the list of associated aspects by using reflection and it decides which actions are to be taken to fulfill aspects' **before** directives;

2. it performs housekeeping actions on aspect list;

3. it execute the code of $m_c()$

4. it examines the list of associated aspects by using refelection and it decides which action are to be taken to fulfill aspects' **after** directives;

5. it performs other housekeeping actions on aspect list;

Finally, classes K_i and α_j, that are ordinary Java classes, can be integrated in the final program X.

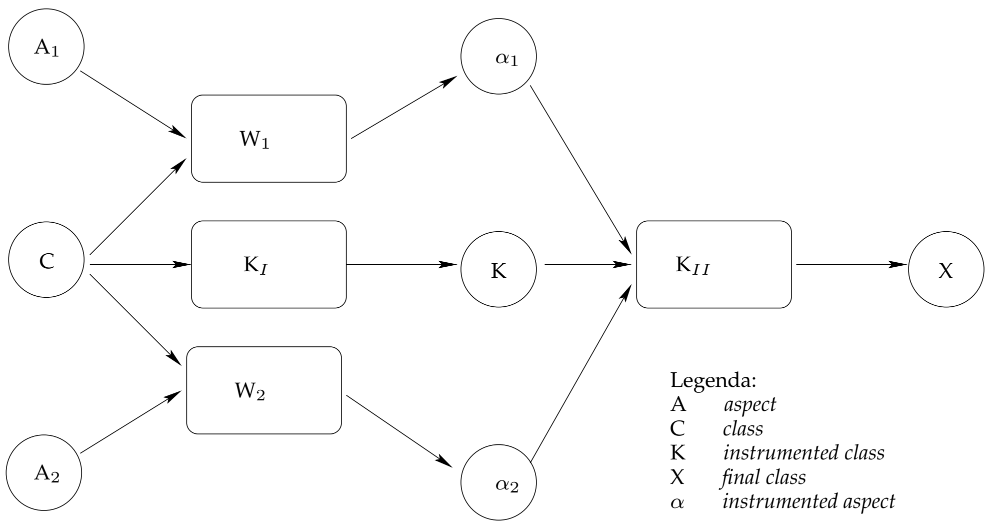

Figure 5.3: Malaj architecture

This strategy is flexible enough for managing any kind of aspect, provided that its effects can be reduced to "before" and "after" actions. This is the case of the three aspects actually present in Malaj and seems a reasonable assumption also for the future plans of extensions. Moreover, the list of aspects exists for the entire execution of the program, thus aspects could be changed at run-time. The choice of having a different weaver for each ad-hoc aspect languages is driven by the goal of building Malaj as a collection of languages easily extendible: adding a new language means just implementing a new weaver.

6 Conclusions

Will your long-winded speeches never end? What ails you that you keep on arguing?
— Job 16,3

Separation of concern is the main issue of all engineering works: in fact, the human mind seems well suited to cope with just one problem at a time. In particular, software engineers are accustomed to partition software systems in modular units. Such parts are developed in relative isolation and then assembled to produce the whole system. Separation is useful and effective when issues are well localized and their handling is explicit.

However, often some concerns are difficult to isolate, because they spread among several units. Thus, several design decisions are not addressed in a specific module, but result as an implicit property of the overall structure.

Cross-cutting concerns are termed *aspects* and aspect oriented programming tries to identify proper linguistic mechanisms to factor out different aspects of a program, which can be defined, understood, and evolved separately. It pushes the idea of separation of concerns one step forward with respect to existing programming language constructs, which simply provide ways to encapsulate a single functionality in a unit. Aspect oriented techniques, however, are still in their infancy.

The main contribution of this thesis is in understanding implications of adopting these new techniques: we showed that some of the existing proposals have problems that hinder software evolution. Problems descend from the fact that these techniques exploit some form of elusion of good information hiding principles. This is needed because aspects are, by definition, intertwined with the functional code and full generality requires that aspect composition should be possible at every level of granularity.

Thus, no clean interface can be interposed among functionality and aspects: if you have to mix fluids you cannot assemble them as Lego bricklets. However, we claim that even if the problem has no solution in general, it can be solved in particular cases. We focussed on a particular class of systems, namely distributed applications. In this domain there are critical aspects, as concurrency, network relocation, exception handling, that software engineers are willing to think separately, nevertheless current state of the art forces to code them together with the other features of the system.

We have designed three ad hoc aspect oriented languages for these aspects, and we have found that predefining the set of possible aspects to deal with, enables the definition of constructs with limited visibility of specific features of functional modules to which the different aspects apply. This is key to understand the properties of the composition between aspects and functionality and the relation with traditional composition mechanisms as inheritance.

We think our approach offers a good compromise between flexibility and power, on the one side, and understandability and ease of change on the other. It does not allow programmers to code any possible concern, but it enables the comprehension of concern specific relations with functional code. Future work is needed to cover a spectrum of concerns far beyond these three, and to complement the programming support with a formal model that can be used to reason about aspects interaction. Research work at the programming language level should go hand-in-hand with experimental work, which should try to assess the usefulness and usability of the languages.

We are convinced that the isolation of aspects by using ad hoc languages may have an impact also on testing activities. Verification of software system is often performed by applying the following strategy: (1) a model of application is derived from artifacts produce during implementation (formal or semi-formal specifications, source code, documentation); (2) the model is exploited to infer interesting properties of the system; (3) the model may be also used to generate a number of test cases sufficient for having significant statistical evidence on the correctness of the program.

Thereby, the derivation of the model (step (1)) is the critical phase for achieving a successful verification. The ontology of the model is dependent from the properties one wants to prove. For example, a Petri net model is a suitable choice if interested in properties regarding concurrency. Having different languages for different aspects should allow one to derive the proper model more easily and naturally. Moreover, in some cases, the integration test can be reduced to a unit test on the aspect.

The ultimate goal is finding a proper methodology for dealing with aspect oriented units as true components: they should be deployed independently, implementors should be able to substitute them in applications according needs, and their composition should preserve the safety of the system.

Bibliography

[1] Subject-oriented programming and design patterns. `http://www.research.ibm.com/sopcpats.htm`. IBM Thomas J. Watson Research Center, Yorktown Heights, New York.

[2] Jass. `http://semantik.informatik.uni-oldenburg.de/~jass`, 1999. Jass is copyrighted by the Semantics Group, theoretical informatics, department of informatics at Carl von Ossietzky University of Oldenburg, Germany.

[3] G. Agha. Concurrent object-oriented programming. *Communications of the ACM*, 33(9):125–141, September 1990.

[4] Architecture Project Management, Cambridge, UK. *The Advanced Network Systems Architecture (ANSA)*, 1989.

[5] Cognitive Science Laboratory at Princeton University. Wordnet (r) 1.6. `http://www.cogsci.princeton.edu/~wn/`, 1997. Available via DICT protocol at `http://www.dict.org`.

[6] S. Baker. *CORBA Distributed Objects Using Orbix*. Addison-Wesley, Reading, MA, USA, 1997.

[7] L. M. J. Bergmans and M. Aksit. Composing software from multiple concerns: A model and composition anomalies. In *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, page 16, Limerick, Ireland, June 2000.

[8] T. Bolognesi. Toward constraint-object-oriented development. *IEEE Transactions on Software Engineering*, 26(7):594–616, jul 2000.

[9] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley publishing company, USA, anniversary edition, 1995.

[10] L. Cardelli and A. D. Gordon. Mobile ambients. In Maurice Nivat, editor, *Proceedings of the First International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '98), Held as*

Part of the Joint European Conferences on Theory and Practice of Software (ETAPS'98), (Lisbon, Portugal, March/April 1998), volume 1378 of *lncs*, pages 140–155. sv, 1998.

[11] M. C. Chu-Carrol and S. Sprenkle. Software configuration management as a mechanism for multidimesional separation of concerns. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 28–32, Limerick, Ireland, June 2000.

[12] G. Colouris, J. Dollimore, and T. Kindberg. *Distributed Systems: Concepts and Design*. Addison Wesley, 2nd edition, 1994.

[13] G. Cugola, E. Di Nitto, and A. Fuggetta. Exploiting an event-based infrastructure to develop complex distributed systems. In *Proceedings of the 20th International Conference on Software Engineering (ICSE98)*, Kyoto (Japan), April 1998.

[14] G. Cugola, E. Di Nitto, and A. Fuggetta. The jedi event-based infrastructure and its application to the development of the opss wfms. *IEEE Transactions on Software Engineering (TSE)*, "To appear".

[15] G. Cugola, C. Ghezzi, and M. Monga. Coding different design paradigms for distributed applications with aspect-oriented programming. In *WSDAAL (Workshop su Sistemi Distribuiti: Algoritmi, Architetture e Linguaggi)*, L'Aquila, Italy, sep 1999.

[16] G. Cugola, C. Ghezzi, and M. Monga. Language support for evolvable software: An initial assessment of aspect-oriented programming. In *Proceedings of International Workshop on the Principles of Software Evolution*, Fukuoka, Japan, jul 1999.

[17] G. Cugola, C. Ghezzi, M. Monga, and G. P. Picco. Malaj: A proposal to eliminate clashes between aspect-oriented and object-oriented programming. In *Proceedings of International Conference on Software: Theory and Practice*, Bejing, China, August 2000.

[18] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[19] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, Ltd., Baffins Lane, Chichester
West Sussex PO19 1UD, England, 2000.

[20] A. Finkelstein, J. Kramer, B. Nuseibeh, L. Finkelstein, and M. Goedicke. Viewpoints: A framework for integrating multiple perspectives in systems development. *International Journal of Software Engineering and Knowledge Engineering*, 1(2):31–58, 1992.

[21] A. Fuggetta, G. P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[22] C. Ghezzi and M. Jazayeri. *Programming language concepts*. John Wiley & Sons, third edition, 1997.

[23] M. Grand. *Patterns in Java*, volume I. John Wiley & Sons, Inc., first edition, 1998.

[24] M. Grand. *Patterns in Java*, volume II. John Wiley & Sons, Inc., first edition, 1998.

[25] W. G. Griswold. Coping with software change using information transparency. Technical Report CS98-585, Department of Computer Science and Engineering, University of California, San Diego, USA, 1998.

[26] P. Gruenbacher, A. Egyed, and N. Medvidovic. Dimensions of concerns in requirements negotiation and architecture modeling. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 50–54, Limerick, Ireland, June 2000.

[27] R. Guerraoui. Strategic directions in object-oriented programming. *ACM Computing Surveys*, 28(4):691–700, dec 1996.

[28] W. Ho, F. Pennaneac'h, J. Jézéquel, and N. Plouzeau. Aspect-oriented design with uml. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 60–64, Limerick, Ireland, June 2000.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, oct 1969.

[30] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in fargo. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[31] E. A. Kendall. Reengineering for separation of concerns. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 65–68, Limerick, Ireland, June 2000.

[32] M. A. Kersten and G. C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. Technical Report TR-99-04, Department of Computer Science – University of British Columbia, 201-2366 Main Mall – Vancouver BC Canada V6T 1Z4, apr 1999.

[33] G. Kiczales, J. de Rivières, and D. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, 1991.

[34] G. Kiczales, J. Hugunin, M. Kersten, J. Lamping, C. V. Lopes, and W. G. Griswold. Semantic-based crosscutting in aspectj^{TM}. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 69–74, Limerick, Ireland, June 2000.

[35] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. *ACM SIGPLAN Notices*, pages 435–451, 1992. OOPSLA'92.

[36] G. Kiczales, J. Lamping, C. V. Lopes, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, May 1997. IEEE.

[37] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, Finland, June 1997. Springer-Verlag.

[38] A. Lai, G. C. Murphy, and R. J. Walker. Separating concerns with hyper/j^{TM}: An experience report. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 79–91, Limerick, Ireland, June 2000.

[39] J. Lamping. Typing the specialization interface. *ACM SIGPLAN Notices*, pages 201–214, 1993. OOPSLA'93.

[40] D. Lea. *Concurrent Programming in Java*. Addison-Wesley Longman, 1997.

[41] K. Lieberherr, D. Lorenz, and M. Mezini. Building modular object-oriented systems with reusable collaborations. `http://www.ccs.neu.edu/research/demeter`, 2000.

[42] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, 1996.

[43] K. J. Lieberherr, I. Holland, and A. J. Riel. Object-oriented programming: an objective sense of style. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'88)*, volume 23 of *ACM SIGPLAN Notices*, pages 323–334, 1988.

[44] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transaction on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[45] C. V. Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, nov 1997.

[46] C. V. Lopes and G. Kiczales. Recent developments in AspectJ^{TM}. In Serge Demeyer and Jan Bosch, editors, *Object-Oriented Technology: ECOOP'98 Workshop Reader*, volume 1543 of *Lecture Notes in Computer Science*, pages 398–401. Springer, 1998.

[47] S. Matsuoka and A. Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, Cambridge, MA, 1993.

[48] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[49] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, New York, NY, second edition, 1997.

[50] R. Miller and A. Tripathi. Issues with exception handling in object-oriented systems. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, volume 1241 of *Lecture Notes in Computer Science*, pages 85–103. Springer-Verlag, New York, NY, June 1997.

[51] R. Milner. The polyadic π-calculus: A tutorial. In M. Broy, editor, *Logic and Algebra of Specification*. Springer-Verlag, 1992.

[52] M. Monga. Ad-hoc constructs for non functional aspects. In *WS-DAAL (Workshop su Sistemi Distribuiti: Algoritmi, Architetture e Linguaggi)*, Ischia, Italy, sep 2000.

[53] M. Monga. Concern specific aspect-oriented programming with malaj. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, page 101, Limerick, Ireland, June 2000.

[54] M. Monga and S. Novelli. Omissys: Managing temporal information in multi-model diagnosis. Master's thesis, Politecnico di Milano, June 1996. In Italian.

[55] O. Nierstrasz. Composing active objects. In P. Wegner G. Agha and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 151–171. MIT Press, 1993.

[56] S. Oaks and H. Wong. *Java Threads*. O'Reilly & Associates, first edition, 1997.

[57] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3):179–202, 1996.

[58] Xerox PARC. Aspectj design notes. `http://aspectj.org/documentation/designNotes/changes.html`, 2000.

[59] G. P. Picco. μCode: A Lightweight and Flexible Mobile Code Toolkit. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *Lecture Notes in Computer Science*, pages 160–171, Stuttgart, Germany, 1998. Springer-Verlag, Berlin.

[60] K. Rustan, M. Leino, and R. Stata. Checking object invariants. Technical Report 1997-007, Digital Systems Research Center, jan 1997.

[61] M. Skipper. A model of composition oriented programming. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 120–125, Limerick, Ireland, June 2000.

[62] R. Stata and J. V. Guttag. Modular reasoning in the presence of subclassing. *ACM SIGPLAN Notices*, 30(10):200–214, October 1995. *Proceedings of OOPSLA '95 Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*.

[63] C. Szyperski. *Component Software — Beyond Object-Oriented Programming*. Addison Wesley Longman Limited, 1998.

[64] Peri Tarr and Harold Ossher. *Hyper/J^{TM} User and Installation Manual*. IBM Research, 2000. Also available at `http://www.research.ibm.com/hyperspace`.

[65] J. Viega and D. Evans. Separation of concerns for security. In Peri Tarr, William Harrison, Harold Ossher, Anthony Finkelstein, Bashar Nuseibeh, and Dewayne Perry, editors, *ICSE 2000 Workshop on Multi-Dimensional Separation of Concerns in Software Engineering*, pages 126–129, Limerick, Ireland, June 2000.

[66] G. Vigna. *Mobile Code Technologies, Paradigms, and Applications*. PhD thesis, Politecnico di Milano, 1997.

[67] P. Wadler. A taste of linear logic. In *Mathematical Foundations of Computing Science*, Gdansk, Poland, August 1993. Springer Verlag LNCS 711.

[68] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. In *Mobile Object Systems*, volume 1222 of *Lecture Notes in Computer Science*, pages 49–64. Springer-Verlag, Berlin, 1997.

[69] R. J. Walker, E. L. Baniassad, and G. C. Murphy. An initial assessment of aspect-oriented programming. In *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.

[70] XEROX Palo Alto Research Center. *AspectJ: User's Guide and Primer*, 1998.

[71] XEROX Palo Alto Research Center. *AspectJ: User's Guide and Primer*, 1999.

[72] Z. Yang and K. Duddy. CORBA: A platform for distributed object computing. *OSR*, 30(2):4–31, April 1996.

[73] A. Yonezawa and M. Tokoro, editors. *Concurrent Object-Oriented Programming*. The MIT Press, Cambridge, Mass., 1987.