# ELEMENTS OF DESIGN OF AN OBJECT-ORIENTED FRAMEWORK PROTOTYPE FOR WAVELET-BASED IMAGE PROCESSING USING DESIGN PATTERNS

Yves Roy

A thesis
in
The Department
of
Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montréal, Québec, Canada

April 2004

© Yves Roy, 2004

National Library of Canada
Acquisitions and Bibliographic Services
395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale du Canada
Acquisisitons et services bibliographiques
395, rue Wellington
Ottawa ON K1A 0N4
Canada

*Your file Votre référence*
*ISBN: 0-612-91108-X*
*Our file Notre référence*
*ISBN: 0-612-91108-X*

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou aturement reproduits sans son autorisation.

---

In compliance with the Canadian Privacy Act some supporting forms may have been removed from this dissertation.

Conformément à la loi canadienne sur la protection de la vie privée, quelques formulaires secondaires ont été enlevés de ce manuscrit.

While these forms may be included in the document page count, their removal does not represent any loss of content from the dissertation.

Bien que ces formulaires aient inclus dans la pagination, il n'y aura aucun contenu manquant.

Canada

// Abstract

Elements of Design of an Object-Oriented Framework Prototype for Wavelet-Based Image Processing using Design Patterns

Yves Roy

Design patterns and frameworks are increasingly popular techniques for addressing key aspects of the design of complex software systems. Patterns support the reuse of design expertise by articulating the aspects of successful solutions to design problems in a particular context. Frameworks are concrete realizations of groups of patterns that enable code reuse and design reuse by capturing the common abstractions of an application domain while leaving control of application-specific structures and behaviors to application developers.

Application frameworks encapsulate expertise applicable to a wide range of programs and aim to provide a full range of functionality typically needed by an application thus encompassing a horizontal slice of functionality that can be applied across client domains. Domain frameworks encapsulate expertise in a particular domain, thus encompassing a vertical slice of functionality for a specific problem domain reducing the amount of work that needs to be done to implement domain-specific applications.

Wavelets and wavelet transform concepts originated from a synthesis of ideas developed during the last thirty years in engineering, physics, and pure mathematics. Wavelets have been very successful in many scientific and engineering fields and they have led to many successful applications in signal analysis and image processing.

In this thesis we are presenting design and implementation elements for the development of an object-oriented application and domain framework prototype for wavelet-based image processing applications using design patterns.

# Acknowledgments

I would like to thank my supervisor, Dr. Bui for his support during this long journey.

I would also like to thank Alessandro Pasetti, Geert Uytterhoeven, Doug Schmidt, and Micheal Stal for taking time to answer my numerous questions about design patterns and frameworks.

It should be stressed that my professional interest in design patterns and frameworks wouldn't have existed without the enthusiuasm, expertise and dedication of Mr. Achint Sandhu, my manager during my tenure at Nortel Networks. I will always remember the "bunker" time, the 80-hour weeks, and George Carlin wake-up breaks at 2:30 in the morning...

Many thanks to my friend, dragon boat and gym partner, Andrew Postans, for his support and late-night Linux crash recovery skills.

I am very grateful to my current employer, Dr. Julien Doyon, Scientific Director of RNQ (Réseau Neuroimagerie Québec) and REPRIC (Regroupement Provincial de Recherche en Imagerie Cérébrale) for his vision and understanding over the past few weeks.

I would like to sincerely thank Stan Swiercz for his help in numerous occasions during my undergraduate and graduate studies at Concordia.

Finally, my sincere thanks to my family and my wife's family, for their encouragements and to my love and counterpart, my wife Libei, for her unbelievable patience and caring support. To my little sunshines, Sophie, 27 months, and Olivier, 9 months who kept me as busy as ever, in the most wonderful kind of way.

# Contents

**List of Figures** xi

**List of Tables** xiv

**I Background and Domain Analysis: Object-Oriented Frameworks and Wavelets** 1

**1 Introduction** 2

1.1 Overview of Wavelet Research . . . . . 2

1.2 Overview of Object-Oriented Framework Research . . . . . 3

1.2.1 What is a framework? . . . . . 4

1.2.2 On Design Methodologies . . . . . 4

1.2.3 Documenting the framework development process . . . . . 5

1.2.4 On using design patterns . . . . . 6

1.3 What is Wave ImAgE? . . . . . 6

1.3.1 Transform-based image processing applications . . . . . 7

1.3.2 Goals of Wave ImAgE . . . . . 7

1.3.3 Functionality . . . . . 8

1.4 Design Problems . . . . . 9

1.5 Contribution of the Thesis . . . . . 10

1.6 Organization of the Thesis . . . . . . . . . . 11

1.7 Notation . . . . . . . . . . 13

**2 Object-Oriented Frameworks** **14**

2.1 Introduction . . . . . . . . . . 14

2.1.1 What are Design Patterns? . . . . . . . . . . 14

2.1.2 What are Frameworks? . . . . . . . . . . 15

2.2 Object-Oriented Frameworks . . . . . . . . . . 15

2.3 Framework of Frameworks . . . . . . . . . . 17

2.4 Comparing Frameworks with Other Reuse Techniques . . . . . . . . . . 18

2.4.1 Compound Design Patterns . . . . . . . . . . 24

2.4.2 Framework Construction Principles and Patterns . . . . . . . . . . 24

2.5 Framelet Overview . . . . . . . . . . 27

2.5.1 On the Iterative Nature of Framework Development . . . . . . . . . . 28

2.6 Overview of Wave ImAgE . . . . . . . . . . 29

2.6.1 Design Patterns in Wave ImAgE. . . . . . . . . . . 29

2.6.2 Frameworks in Wave ImAgE. . . . . . . . . . . 31

**3 Wavelets, Wavelet Transforms and Applications** **33**

3.1 Wavelets and Wavelet Transforms . . . . . . . . . . 33

3.1.1 What are Wavelets ? . . . . . . . . . . 33

3.1.2 Multiresolution Analysis (MRA) . . . . . . . . . . 34

3.1.3 The Wavelet Functions . . . . . . . . . . 35

3.1.4 The Discrete Wavelet Transform . . . . . . . . . . 35

3.1.5 An Example: The Haar Wavelet . . . . . . . . . . 35

3.1.6 Multiwavelets . . . . . . . . . . 36

3.1.7 Complex Symmetric Daubechies Wavelets . . . 40
3.1.8 Wavelet Transforms . . . 41
3.1.9 Preprocessing and Sampling . . . 41
3.2 Wavelet-Based Image Processing Applications . . . 44
3.2.1 Terminology of Digital Image Processing . . . 45
3.2.2 Wavelet-Based Processing . . . 45
3.2.3 Wavelet Denoising . . . 46
3.2.4 Wavelet-Based Image Compression . . . 52
3.2.5 3D Medical Image Fusion . . . 54
3.2.6 3-D Image Fusion . . . 54
3.3 Conclusion . . . 55

**II Frameworks Design** **56**

**4 Data Processing Framelet** **58**
4.1 Overview . . . 58
4.2 Context . . . 58
4.3 Problem . . . 59
4.4 Solutions . . . 60
4.4.1 The VISITOR Design Pattern . . . 61
4.4.2 Visitors in Frameworks . . . 62
4.4.3 The Staggered VISITOR . . . 63
4.4.4 The Acyclic VISITOR . . . 69
4.4.5 A STAGGERED VISITOR-Based Framework Candidate Solution . . . 73
4.4.6 An ACYCLIC VISITOR-Based Framework Candidate Solution . . . 75
4.5 Liabilities of a VISITOR-based Approach. . . . 76

4.6 Design Notes . . . 79
4.6.1 Generic Programming as an Alternative to Classic Polymorphism . . . 80
4.6.2 External Polymorphism and Processor Adapters . . . 80
4.6.3 Even More Adaptation using the Extension Object Pattern . . . 82
4.6.4 Mixing Paradigms: A Common Base Class for Different Parametrizations . . . 85
4.6.5 Genericity of Image Processing Algorithms . . . 93
4.6.6 Generic programming . . . 94
4.6.7 Classic polymorphism . . . 95

**5 Wavelet Transform Framework** **98**

5.1 Overview . . . 98
5.2 Context . . . 98
5.3 Problem . . . 99
5.4 Solution . . . 100
5.4.1 Wavelet Transform Steps . . . 100
5.4.2 Supporting Different Prefiltering Schemes . . . 108
5.5 Data Representation . . . 112
5.5.1 Base Data . . . 112
5.5.2 Data Subclass for Non-Transformed 1D Signals . . . 113
5.5.3 Data Subclass for Non-Transformed 2D Signals . . . 113
5.5.4 Templatized Data Subclasses . . . 114
5.5.5 Concrete Letters . . . 116
5.5.6 Iterators for Element Access . . . 116
5.5.7 Row and Colums Iterators for 2D Data . . . 117
5.5.8 Transformed Data . . . 117

5.5.9 Refining the Data Processing Framework . . . . . . . . . . . . . . . . . . . . 121

5.6 Design Notes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 132

5.6.1 From the Ground Up: Towards More Flexibility and Extensibility. . . . . . . 132

5.6.2 Standard and Non-Standard Wavelet Decomposition . . . . . . . . . . . . . 137

5.6.3 Basis Functions, Wavelets and Filters . . . . . . . . . . . . . . . . . . . . . 139

5.6.4 Wavelet Transform Process . . . . . . . . . . . . . . . . . . . . . . . . . . 139

5.6.5 Wavelet Transform Data Representations . . . . . . . . . . . . . . . . . . . 143

5.6.6 Finding an Interface for Wavelet Transforms . . . . . . . . . . . . . . . . . 145

**6 The Image Processing Experiment Configuration and Scheduling Framework 149**

6.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149

6.2 Context . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 149

6.3 Problem . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150

6.4 Solution . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 150

6.5 Configuration and Scheduling Sequence . . . . . . . . . . . . . . . . . . . . . 151

6.6 Resolved Example: From the Ground Up . . . . . . . . . . . . . . . . . . . . 153

6.6.1 Dynamically Loading Concrete Factories into Applications . . . . . . . . . . 155

6.6.2 The Component Configurator . . . . . . . . . . . . . . . . . . . . . . . . . 158

6.6.3 Experiment Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . . . 163

6.7 The Configuration Framework: Putting It All Together . . . . . . . . . . . . . 164

6.7.1 Generic Experiment and Application-Specific Experiment Lineup . . . . . . . 166

6.8 Framelet Constructs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 169

**7 Conclusion and Future R&D Focus Areas 171**

7.1 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 171

7.2 Future R&D Focus Areas . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172

7.2.1 Extensions . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172
7.2.2 Support for processors . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 172
7.2.3 Image Processing Functionality . . . . . . . . . . . . . . . . . . . . . . . . . . 172
7.2.4 Sub-Frameworks and Other Subsystems . . . . . . . . . . . . . . . . . . . . . 173

# List of Figures

1 Transform-Based Image Processing. . . . . 7

2 Manager-driven framework. . . . . 16

3 TEMPLATE METHOD design pattern [24]. . . . . 17

4 STRATEGY PATTERN in support of adaptation through composition (Separation principle). . . . . 17

5 Framework of frameworks architecture. . . . . 18

6 Application versus framework design activities. . . . . 19

7 Class library architecture . . . . . 20

8 Application framework architecture . . . . . 21

9 Unification principle . . . . . 25

10 Separation principle . . . . . 26

11 Abstract coupling. . . . . 26

12 Framelets and object subsets. . . . . 28

13 Wave ImAgE framework architecture. . . . . 29

14 The main patterns used in Wave ImAgE. . . . . 30

15 The Frameworks used in Wave ImAgE. . . . . 31

16 Multiwavelet analysis filter bank. . . . . 38

17 Prefilter and postfilter schematic illustration. . . . . 41

18 Single-level 2-D MWT using 1-D approximation preprocessing. . . . . 44

19 Single-level 2-D MWT using 2-D approximation preprocessing. . . . . . . . . . . . . 44

20 Transform-based image processing. . . . . . . . . . . . . . . . . . . . . . . . . . . 45

21 Filter bank implementation of a multilevel (three-stage) multi-band (two-band) analysis tree (descrete wavelet transform) of a 1D signal. . . . . . . . . . . . . . . . . . 46

22 Flowchart for a typical image denoising application. . . . . . . . . . . . . . . . . . . 49

23 Soft (a) and hard (b) thresholding. . . . . . . . . . . . . . . . . . . . . . . . . . . 50

24 Standard and non-standard decompositions. . . . . . . . . . . . . . . . . . . . . . . 51

25 Flowchart for a typical image compression application. . . . . . . . . . . . . . . . . 52

26 Flowchart for a typical image fusion application. . . . . . . . . . . . . . . . . . . . 54

27 VISITOR design pattern [24]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 61

28 VISITOR hierarchy in a framework context. . . . . . . . . . . . . . . . . . . . . . . 62

29 The STAGGERED VISITOR design pattern. . . . . . . . . . . . . . . . . . . . . . . . 64

30 STAGGERED VISITOR with Framework (pre-)defined Elements and Visitors . . . . . 65

31 The ACYCLIC VISITOR design pattern. . . . . . . . . . . . . . . . . . . . . . . . . . 69

32 STAGGERED VISITOR-based Data Processing Framework . . . . . . . . . . . . . . . 73

33 The Wave ImAgE ACYCLIC VISITOR-based Data Processor Framework. . . . . . . . 75

34 The PROXY design pattern. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

35 The STRATEGY design pattern. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 102

36 The TEMPLATE METHOD design pattern. . . . . . . . . . . . . . . . . . . . . . . . 103

37 The ITERATOR design pattern. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 103

38 Design of the wavelet transform variation-point. . . . . . . . . . . . . . . . . . . . . 105

39 DECORATOR design pattern [24]. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109

40 Data hierarchy in Wave ImAgE. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 113

41 2D Wavelet decomposition into subbands. . . . . . . . . . . . . . . . . . . . . . . . 118

42 Quadtree representation of a two-level non-standard wavelet decomposition. . . . . . 118

43 COMPOSITE design pattern [24]. . . . . 119

44 Overview of a STAGGERED VISITOR-based Wavelet Transform Framework. . . . . 121

45 Data class hierarchy with rich interfaces. . . . . 136

46 Shallow and wide transform class hierarchy. . . . . 136

47 Data class hierarchy. . . . . 137

48 Wavelet class hierarchy . . . . . 139

49 Wavelet transform recursive process yielding a recursive wavelet transform structure. 140

50 COMPOSITE design pattern [24]. . . . . 144

51 Wavelet transform data implemented using the COMPOSITE design pattern. . . . . 145

52 ABSTRACT FACTORY and PLUGGABLE FACTORY . . . . . 156

53 Class diagram for the COMPONENT CONFIGURATOR pattern. . . . . 159

54 Class diagram for the experiment scheduling subsystem. . . . . 163

55 Class diagram for the Image Processing Experiment Configuration and Scheduling Framework. . . . . 164

# List of Tables

1 Use case for a multiwavelet image denoising application . . . . . . . . . . . . . . . . . 47

2 Use case for a complex wavelet denoising application. . . . . . . . . . . . . . . . . . . . 48

# Part I

# Background and Domain Analysis: Object-Oriented Frameworks and Wavelets

# Chapter 1

# Introduction

## 1.1 Overview of Wavelet Research

Wavelets and wavelet transform concepts originated from a synthesis of ideas developed during the last twenty or thirty years in engineering, physics, and pure mathematics. Wavelets have been very successful in many scientific and engineering fields. As a consequence of these interdisciplinary origins, wavelets appeal to scientists and engineers of many different backgrounds. On the other hands, wavelets are a fairly simple mathematical tool with a great variety of possible applications.

The first wavelet transforms were *continuous wavelet transforms* (CWT). A signal is analysed by performing inner product with *basis functions*. Those basis functions are generated from an oscillating function with *finite support* called a mother *wavelet function*– wavelets are localized waves. The condition imposed on the wavelet function is that its integral be zero. By *translation* and *dilation* of the wavelet function, different basis functions are created. Translation and dilation allow for time and frequency customization of the window through which the analysis is done. This results in a customizable *time/frequency localization* of the analysis which proves advantageous when compared with the Fourier transform [32].

Then came the *discrete wavelet transform* (DWT) adapted for discrete signals. Translation is restricted to multiples of the sampling distances and dilation is dyadic. This leads to the important notion of *scale* (or resolution levels) and to a concept called *multiresolution analysis* [35].

The design of orthogonal wavelet functions with compact support [15] then leads to an efficient implementation of the wavelet transform called the *fast wavelet transform* (FWT), or filter-bank

algorithm, to use the language of subband coding and filter-bank theory [53].The wavelets made up from translation and dilation of a *single* scaling function and mother wavelet are called *single* wavelets.

A different kind of wavelet having important properties for image processing were also developed. These wavelets, called *multiwavelets* [54], exhibit simultaneously some properties that single wavelets can not. Multiwavelet basis uses translations and dilations of $L \geq 2$ scaling functions and $L$ mother wavelet functions. Because of those additional properties, multiwavelets can potentially give better results than single ones in image processing applications [54], but their implementation is also more intricate and complex. Since the multiwavelets coefficients are matrix coefficients, the transform requires vector input streams. Therefore, an additional processing step called *prefiltering* (or pre-processing) is necessary to provide that vector input from a scalar one.
In an effort to define a method to improve a given discrete wavelet transform, a scheme called the *lifting scheme* was developed [58, 59]. The lifting scheme became an efficient algorithm to compute a wavelet transform as a sequence of simple steps, providing a method to implement reversible integer wavelet transform ([57, 10, 16]).
Some more information on wavelets and wavelet transforms will be given in Chapter 3 of this thesis.

Wavelets have led to applications in signal analysis and image processing, image and video compression, mathematical physics, numerical analysis, communication, system theory, identification, geometric modeling, computer graphics, theoretical physics, approximation theory, stochastic processes, and many others. We refer to the litterature for more information [28, 38, 32, 62, 13, 15, 35]. More details on some wavelet-based processing applications will also be given in Chapter 4.

## 1.2 Overview of Object-Oriented Framework Research

Crucial to the success of many software systems are *patterns* [24, 7, 51] and *frameworks* [42, 34] that document and reify the knowledge of how to develop these systems. Patterns support the reuse of *design expertise* by articulating the aspects of successful solutions to software design and implementation problems that arise in a particular context. In that sense, patterns, or *design patterns* as they are called, are *abstract* constructs. Frameworks on the other hands, are *concrete realizations* of groups of patterns that enable *code reuse* and *design reuse* by capturing the common abstractions of an application domain while leaving control of application-specific structures and behaviors to application developers.
During the past decade, a number of research and development efforts have focussed on documenting patterns and developing frameworks [24, 7, 51, 34, 42, 49, 50]. Object-oriented frameworks have emerged as a powerful technology for developing and reusing software systems possessing qualities like affordability, extensibility and flexibility. By leveraging domain knowledge and prior efforts of

experienced developers, frameworks embody common solutions to recurring software design challenges and application requirements that need not be created and validated over and over for each new application.

### 1.2.1 What is a framework?

A framework is a semi-finished system intended to be instantiated. It defines the architecture for a family of systems (the family of applications the framework supports) and provides the basic building blocks to create them. An object-oriented framework consists of abstract and concrete classes, and their collaborations. Instantiation of a framework consists of composing and subclassing the existing classes.
Frameworks will be covered in greater details in Chapter 2, and the instantiation process and reuse will be covered in Part III of the thesis.

The design of a large framework can be quite complex. The *framelet* concept was introduced as a way of simplifying the design process ([44, 19]). Framelets simplify the design process by applying the "divide and conquer" approach. At *design level*, a framework consists chiefly of a set of *variation points* (well identified parts of the framework that are kept flexible since they are most likely to be varying from application to application) and *design patterns* (design solutions of design problems in a certain context). Framelets partition this set into smaller subsets of related variation points and design patterns that are then handled, as far as the design is concerned and as far as possible, separately. The term "framelet" however is also used loosely to simply refer to a small framework and most of the time, at least in this thesis, framework and framelet are used interchangeably.
More details on framelets can be found in Chapter 2.

### 1.2.2 On Design Methodologies

Before undertaking the design of Wave ImAgE we have studied and reviewed a number of different framework development approaches and methodologies. Among them are *top-down* oriented approaches like *hot-spot driven* framework development [43], framework development *from domain knowledge* [2], where applications are build all at once following an exhaustive domain analysis phase, *domain engineering-based* approaches like SHERLOCK [56] and *bottom-up* oriented approaches like framework development by *systematic generalization* [47], where an initial application is gradually revisited (through generalization transformations) to accomodate a new application.
We also considered closer-to-implementation *recipe-based* approaches providing guidelines like Jones in [31], *hybrid* approaches like Butler and Xu in [9], and pattern language-based approach like Roberts and Johnson in [45]. Due to the lack of mature methodological tools targeted at framework development and to the fact that this thesis is *not* about design methodologies in themselves, we

decided to adopt a lightweight methodology borrowing from many of the approaches listed above to forge our own informal hybrid approach. We conducted an extensive litterature review of the field of wavelet-based image processing (see bibliography) to extract the key concepts and abstractions and to identify the commonalities accros a number of different applications. We made use of the concepts of hot-spots (or variation-points) and framework construction principles ([42] and [19]). We considered a single initial application and attempted generalizing it [47]. The generalization was guided by a knowledge of the multiple hot-spots identified during domain analysis ([2] and [43]). Use cases were useful in conducting commonality and variability analysis [56] for targetted applications.

### 1.2.3 Documenting the framework development process

Due to the abstract nature and complexity of the framework *design* process, it is of crucial importance to provide a good framework documentation in order to reduce the learning curve of framework re-users ([8], [21], [30]). By framework *re-users* we refer to two kinds of roles taken by people using the framework:

- *Application developers* who want to know how to *customize* the framework to produce specific applications.
- *Framework maintainers and developers* who are responsible for the maintenance and evolution of the framework.

In order to achieve our goal of assisting these users, this thesis provides the following documentation elements:

- A section of Chapter 2 provides an "at a glance" *overview of the framework* while chapters in Part II describing the design *process* and *products* go into more detail by presenting the framework as a collection of smaller frameworks. The *problem-domain* jargon is outlined in Chapter 3.
- *Design patterns* are used to document design solutions (by naming them whenever they are used and by using UML class diagrams to show the well-known structure of some solutions) to problems that are discussed in Part II.
- *Source code samples and commented C++ code fragments* are used throughout the text to illustrate the design and give a more concrete feel whenever we felt it was appropriate to do so.
- Each chapter in Part II really consists in a *design notebook* by providing *design notes* outlining some aspects of the design activity (*before* any candidate solution was in sight) tackling alternative or complementary elements of solution.

### 1.2.4 On using design patterns

The design of Wave ImAgE is full of design patterns. We are using design patterns for the following three reasons:

1. Recurring design problems are well served by *proven design solutions.*
2. Design patterns help *documenting* a software system.
3. Design patterns are *interesting* in themselves.

Facing a well-described design problem, we always look for ways to *encapsulate the concept that varies* in order to ensure that the parts that are most likely to change can be changed without incurring a redesign. And if there is a proven solution that addresses that problem, and if that solution is in the form of a design pattern, then we will reuse that solution by applying the design pattern to our problem.
Our use of design patterns is also motivated by a genuine interest in patterns and a desire to get familiar with them and truly *understand* them. Such a goal cannot be attained only by *reading about* design patterns. It is by putting them to the test by applying them in our own designs, in response to our own design problems, that we will come to have a good understanding of them, *i.e.* their strengths and liabilities.

## 1.3 What is Wave ImAgE?

Wave ImAgE stands for "*Wave*let-based *I*maging *A*daptive *E*nvironment", and it consists of a set of design elements for a prototypic object-oriented framework for wavelet-based image processing. The goal of the thesis is to illustrate the design process of this framework by providing design elements and some prototypic implementations for the framework. The word prototype is used to emphasize the fact that Wave ImAgE is a work in progress towards a proof of concept for a design. This means that some areas of the implemented framework only provide minimal interfaces that are sufficient to test the design and explore the viability of a proposed design solution.

As the name "*wavelet-based image processing*" implies, the framework is meant to facilitate the development of applications featuring:

1. the processing of input *images* (where "images" refers to 1D, 2D or 3D data signals),
2. the *transformation* of data into the *wavelet domain* via a process called wavelet transform.
3. the *processing* of the transformed data (also called *transform-based processing* or *transform-domain processing*, or in our case, *wavelet-based processing*).

This paradign is illustrated in Figure 1.

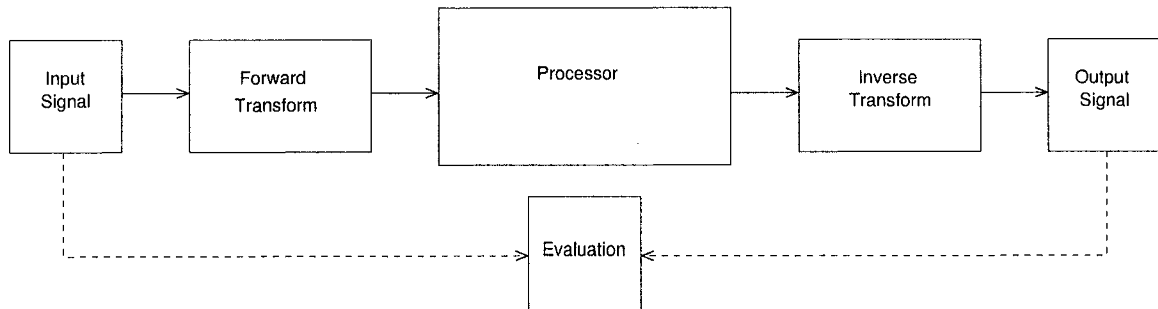



Figure 1: Transform-Based Image Processing.

### 1.3.1 Transform-based image processing applications

Examples of typical image processing applications are:

1. Real life image de-noising (white noise) using multiwavelet transform and univariate thresholding (using different prefiltering schemes needed by multiwavelets).
2. Satellite Aperture Radar (SAR) image de-noising (multiplicative noise) using complex-wavelet transform and elliptic thresholding.
3. 3D medical image fusion (using different fusion algorithms).
4. Face recognition using single wavelet approximation.
5. Image compression.
6. Image edge extraction and segmentation.
7. Geometry compression.

It is important to note that application-specific algorithms are *not* necessarily part of a framework *per se.* Their implementation is the responsibility of application developers adapting (or customizing) a framework and instantiating it to write applications. A framework however may or may not provide some application-specific components as default components.

### 1.3.2 Goals of Wave ImAgE

The main goal of Wave ImAgE is to make the development of wavelet-based image processing applications easier. Below is a list of some of the most important points that Wave ImAgE aims at providing for application developers:

- Wave ImAgE should capture the design decisions *common* to the application domain and the architecture of wavelet-based image processing applications. This amounts to providing the following:
  - Capture the *minimal* common logic of all applications which involves the processing of an image. The idea can be illustrated as follows:

```
Processor processor; // A processor of some sort...
Image image;         // Some input data...
image.apply(processor); // Let the processor process the data...
```

  - Capture the finer-grained *transform-based processing paradigm* (see Figure 1) consisting of the sequence data (acquisition), transformation of data, and transform-domain processing[1]
  - Support the *wavelet transform* flexibility and extensibility requirements: customization and extension of a wavelet transform algorithm should be made easy for an application developer. Similarly, the development and integration of *new* wavelet transform algorithms should be feasible and easy without redesign.
  - Provide a programming *environment* that allows application developers to *repeat* a set of experiments while modifying predefined parameters pertaining to each image-processing experiment (for example, configuration of a processor or change of input data).
  - Provide a programming environment that allows application developers to *reuse* image processing experiments as building blocks of other experiments in a modular way.
  - It should be easy and convenient for framework users to conduct comparative studies by performing a (preferably large) number of experiments by varying some parameters of a generic experiment.
  - Simplifying the development and testing of new wavelet-domain algorithms.
- Provide application developers with services external to wavelet-based image processing applications like logging facilities for example.

### 1.3.3 Functionality

As an integrated environment, the functionality of a mature Wave ImAgE framework [2] is grouped in the following categories:

[1] In this thesis, the focus is placed on the *forward* transform only. The inverse transform is just the same process "in reverse".

[2] We have conducted analysis and produced designs in all of these functional areas. However, due to space and time constraints, some areas are not discussed in this thesis and are relegated to future work (see the Conclusion of the thesis). Those areas ares: image loading/handling, the flexible design of the logging mechanism, and a viewing functionality.

- The *Image Handling Functionality* covering the acquisition (loading) of digital images, manipulation (changing format), and saving. *Not covered in this thesis.*
- The *Transform Functionality* covering the transformation of digital images from image-domain to transform-domain using wavelet transforms. It thus also covers the creation of wavelets filters and prefilters.
- The *Subband (transform-domain) Processing Functionality* covering the processing of transformed images according to various application-specific tasks such as coefficients and subbands access (for thresholding for example) and manipulation.
- The *Logging Functionality* covering logging of various messages (error, informational) for debugging/reporting purposes. A logger is implemented, but the flexible logging mechanism (framework) is not discussed in this thesis.
- The *Configuration of Experiment and Scheduling Functionality* covering the flexible configuration and orderly execution of a set of image processing experiments (or applications).

## 1.4 Design Problems

This section contains a brief overview of some of the main areas of Wave ImAgE's design where design problems occur:

1. *Data structure and representation.* The choice of internal representation for the data to be processed in Wave ImAgE affects many aspects of its design. From transforming and processing the data to displaying and logging, such activities will require accessing and traversing the representation. How does Wave ImAgE actually represents images and transforms (as a resulting *product* of a transform *process*) of different dimensionality and of different types and format?
2. *Transforming and processing.* The core of the framework consists of transforming and processing data for different purposes (i.e. to accomplish different tasks). What objects are responsible for performing different wavelet-based transforms and carrying out processing tasks on the transformed data? How do Wave ImAgE processors interact with the data's internal representation? How does the framework allows for the configuration of different transform and processing algorithms is a convenient way for the application developer, i.e. how can we minimize the number of classes an application developer has to write/modify in order to add new operations?
3. *Application development environment.* Since a large number of applications are likely to be designed using the framework, how can a fixed application structure ensure consistency among the different programs. Failing to provide and enforce a common programming model may

result in developers implementing their own unique versions of an execution model for each of the application they write. This leads to harder to understand, harder to maintain and harder to reuse code. How can such a programming model be enforced without restricting the flexibility of application specific classes?

Wavelet-based image processing applications are characterized by algorithms or processors *operating on* data (images, transforms). It should be possible, for example, for developers to reuse applications to *operate differently* on the *same data*, or *operate (the same way) on different data*. How can this be achieve in an orderly yet flexible fashion?

4. *Data acquisition and components configuration.* How can the framework assist in the configuration of the various components through some external configuration source? How can a configuration source be changed without impacting both the core of the framework and the application specific code? How can application-specific objects be created by the framework which has no knowledge of application-specific subclasses? How can application-specific objects be used (as such) by an application if they are provided by the framework? This is the issue of *type laundering* and the pull versus push model discussed in [63].

5. *Supporting multiple consistent logging mechanisms.* The framework and the applications need to keep trace of different steps of the processing as it is performed. How can multiple applications employ a straightforward and consistent message logging mechanism that can allow logging to various destinations?

## 1.5 Contribution of the Thesis

The main contributions of this thesis are described below:

1. As the title of the thesis indicates, we are offering *elements of design* for an *object-oriented framework prototype* using *design patterns.* Some areas are meant to be presented in an integrated manner, while some others are clearly meant to be presented in an *exploratory* manner. One of the main contribution of the thesis, and a rare quality, in our opinion, is to discuss a variety of design problems that are worth considering in themselves, and to show that the design process is often richer than what a crystallized finished solution can offer. However, since a framework is a concrete construct, we provide, even sometimes in prototypical form, working code samples.

2. A rare quality of this work is that it has "une teneur spéculative certaine". It might not always be evident, but the nature and inherent level of difficulty of the questioning undertaken in this thesis makes it worthwhile to read in its own right. For example, the sections where the problem of type versus composition is briefly discussed (design notes of Chapter 5) is of great interest to anybody seriously interested in software design. The decision to name design

patterns whenever the solution discussed can be considered an application of some well known patterns is deliberate. Very few people really take the time to study seriously design patterns. We did. And we feel that as a result, this thesis represents a good case study for whoever is interested in learning more about design pattern, see examples of them, and understand the context in which they may be used. The hard task of performing domain analysis for framework development is illustrated in great details throughout the thesis. The discussion on the multi-paradigm aspects of object-oriented software development in C++ (with regards to templates versus classic polymorphism) also raises some very interesting questions.

3. To our knowledge, and after a throughout review of the literature and internet search, we did not find any equivalent endeavour or existing systems aiming at accomplishing the same thing as what we are working towards in this thesis. Among the systems that we found that *might* be comparable in *intent*, or *design*, or *implementation*, are WAILI (Wavelets with Integer Lifting) [61], ImageLib (An Image Processing C++ Class Library) [29], Wavelet Image Compression Construction Toolkit [17], POOMA (Parallel Object-Oriented Methods and Applications) [41] or efforts like the generic image processing algorithms development conducted at the EPITA laboratory [18]. The domain analysis that we conducted prior to laying the ground work for the design, resulting in the identification of key domain abstractions, was time consuming but rewarding: to be able to step back and conceptualize at a higher level of abstraction the main elements of a domain is a good intellectual exercise. Thus, the domain analysis that was conducted before the framework design activity was undertaken, allowed us to gain some insights on the structure of some wavelet-based processing tasks, ranging from the wavelet transform itself, to the higher-level description of image processing applications.

4. We offer some interesting variations and combinations of the different VISITOR pattern implementations. This, in itself, is quite interesting, especially with the not so good reputation that the VISITOR pattern may have in a framework context.

5. We present some interesting design findings using generic programming and design patterns, taking as an example, a generic adapter based on the ADAPTER and EXTENSION OBJECT design patterns.

6. The different styles of different parts of the thesis (sometimes more conceptual and abstract, sometimes right down into the details of C++ code) makes it an interesting pedagogical reading. Also, our decision to include *design notes* sections, although these section may be harder to read because less structured and not yet synthetized is, in our opinion, a happy one. Mostly because, sometimes, the most interesting problem are to be found there.

## 1.6 Organization of the Thesis

The thesis is organized into three main parts:

**Part I**– Background Material and Domain Analysis

**Part II**– Framework Design and Development

Here is a detailed overview which describes the content of the thesis chapter by chapter:
**Part I**
Part I of the thesis is dedicated to this introduction, the *domain analysis* phase, and the framework development process. The chapters in Part I will thus be providing background information on the problem domain, *i.e.* wavelets, wavelet transforms and applications.
In **Chapter 2**, we provide an introduction to the concepts and methods of object-oriented frameworks.
In **Chapter 3**, we give a brief overview of the concepts of multi-resolution analysis, wavelets, wavelet transforms, together with a sketch of some applications. Among them are image denoising and image compression.
**Part II**
Part II focuses on the *design* phase of framework development. Part II is really the heart of the thesis and this is where the design activity is taking place. In Part II, we address the design problems mentioned above and many others, we present alternative solutions considered together with how we came to them. Some prototypic solutions are proposed and implemented, and numerous commented code samples are given.
Wave ImAgE is build as a collection of framelets, or small frameworks [3]. Due to the design-intensive nature of the framework development activity, a specific chapter will be dedicated for each framework. In each chapter, we will present not only the resulting design (the actual product of the design activity), but also in some cases, the design process itself. In doing so, our goal is two-fold:

1. Allow the reader to better appreciate and understand the design problems and trade-offs underlying the design decisions. Hence a better understanding of the design decisions and of the complexities inherent to the development of a framework.

2. Offer a documentation corpus that better represents the effort needed to design a framework. It might be argued that a design "speaks for itself", but we nevertheless believe that anyone interested in software design will be better served with a more exhaustive documentation that reflects the design *process* as well as the design *product*.

In **Chapters 4**, we present the design of the general *Data Processing Framework* which proposes a solution to the problem of setting up a framework that puts application-specific processor and data classes into motion without having any knowledge of those subclasses. It is also proposing solutions to the problem of supporting the development of different image-processing algorithms corresponding to different image-processing applications.

[3] According to W. Pree, who coined the term *framelet* in [44], the two concepts, framelet and framework, are not stricly equivalent, the former being more abstract (see [19]) and smaller (see [44]) than the later, but throughout the thesis we will use the terms interchangeably when referring to a small framework. More on that distinction in Chapter 2

In **Chapter 5**, we present the design of the *Wavelet Transform Framework* which provides a solution to the problem of performing a wavelet transform on data of various dimensionality while providing support for the flexibility requirements identified in Chapter 3.
In **Chapter 6**, we describe the design of the *Processor Configuration framework* which proposes a solution to the problem of providing a centralized configuration mechanism that allows for the dynamic configuration of Wave ImAgE components in a flexible manner as well as a solution to the problem of scheduling a series of image processing experiments for execution in an orderly fashion.
Finally, **Chapter 7** is our conclusion together with our suggestions for future research.

## 1.7 Notation

- The class diagrams in this thesis will follow the UML notation [5] and OMT-based notation ([46], [24]). Some elements of the UML-F notation, a UML profile for framework architecture [19], will be retained.
- Design patterns will be referred to using their name in small capital letters, as in STRATEGY, for the Strategy pattern.

# Chapter 2

# Object-Oriented Frameworks

## 2.1 Introduction

Patterns and frameworks are increasingly popular techniques for addressing key aspects of the design of complex software systems. While patterns support the reuse of design expertise by articulating the aspects of successful solutions to software design and implementation problems that arise in a particular context, frameworks are *concrete realizations* of groups of patterns that enable *code reuse* and *design reuse* by capturing the common abstractions of an application domain while leaving control of application-specific structure and behavior to application developers.

During the past decade, a number of research and development efforts have focussed on documenting patterns and developing frameworks [24, 7, 51, 34, 42, 49, 50]. Object-oriented frameworks have emerged as a powerful technology for developing and reusing software systems. By leveraging domain knowledge and prior efforts of experienced developers, frameworks embody common solutions to recurring software design challenges and application requirements.

### 2.1.1 What are Design Patterns?

Design patterns are abstract constructs providing design solutions to `recurring` design problems. Two widely accepted definitions of design patterns are given below:

- A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution, and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context [24]

- Descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context [24].

### 2.1.2 What are Frameworks?

Here are some common definitions of a framework:

- A semi-finished software (sub-) system intended to be *instantiated.* A framework defines the architecture for a family of (sub-) systems and provides the basic building blocks to create them. It also defines the parts of itself that must be adapted to achieve a specific functionality. In an object-oriented environment a framework consists of *abstract* and *concrete classes.* Instantiation of such a framework consists of composing and subclassing and subclassing the existing classes [51].
- A framework is a "semi-complete" application that programmers can customize to form complete applications by extending reusable components in the framework. In particular, frameworks help abstract the canonical flow of control of applications in a domain into product-line architectures and families of related components. At runtime, these components can collaborate to integrate customizable application-independent reusable code with customized application-defined code[48].
- A framework is a set of cooperating classes that makes up a reusable design for a specific class of software. A framework provides architectural guidance by partitioning the design into abstract classes and defining their responsibilities and collaborations. A developer customizes the framework to a particular application by subclassing and composing instances of framework classes.[24]

## 2.2 Object-Oriented Frameworks

There are many possible categorizations for frameworks. A common categorization uses the kind of domain that a framework encompass: *application* functions, *domain* functions, or *support* functions:
**Application frameworks.** Application frameworks encapsulate expertise applicable to a wide range of programs and aim to provide a full range of functionality typically needed by an application. Application frameworks encompass a horizontal slice of functionality that can be applied across client domains. Examples of application frameworks are MFC (Microsoft Foundation Classes) used to build applications for MS Windows and JFC (Java Foundation Classes).
**Domain frameworks.** Domain frameworks encapsulate expertise in a particular domain. These frameworks encompass a vertical slice of functionality for a specific problem domain and thus help reduce the amount of work that needs to be done to implement domain-specific applications. Examples of domain frameworks are control systems, banking or alarm systems.

**Support frameworks.** Support frameworks typically address specific system-level domains such as memory management or file systems. These frameworks are used by application developers to simplify program development. Support frameworks are typically used in conjunction with domain and/or application frameworks.
Wave ImAgE extends across the first two categories. Application framework because of the application (called experiment) scheduling and logging capabilities, and domain framework, since it encapsulates expertise in the wavelet-based image processing application *domain*.

A second possible categorization for frameworks is based on the *framework's high-level structures.* For example, a common kind of framework, called *manager-driven* frameworks, contain a single controlling function (in the manager) that triggers most of the framework actions. A call to the controlling function starts the framework which then creates the objects and calls the functions necessary to perform an application-specific task (See Figure 2).

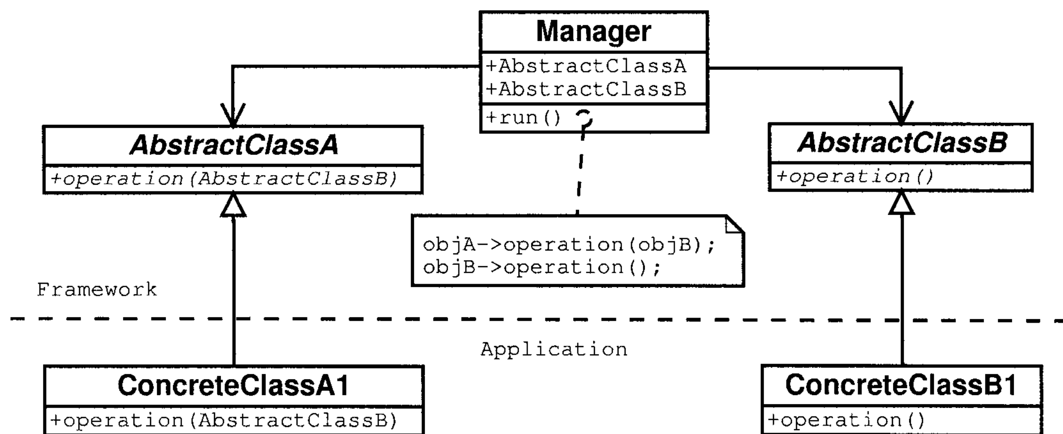



Figure 2: Manager-driven framework.

Another possible categorization for frameworks is based on *how clients use a framework.* When client usage is used as a criteria, we then talk about *architecture-driven* and *data-driven* frameworks [60] depending on whether clients customize the framework by *subclassing* (deriving new classes and overriding member functions) or whether customization is achieved through *composition.* Such frameworks are also called *whitebox* and *blackbox* frameworks respectively:

- **Whitebox frameworks**: In whitebox framework, customization (or adaptation, or extensibility) is achieve via object-oriented language features like inheritance and dynamic binding. Existing functionality can be adapted by inheriting from framework base classes and overriding pre-defined hook methods [19]. Design patterns such as TEMPLATE METHOD are typically

used for this kind of adaptation. See Figure 3.

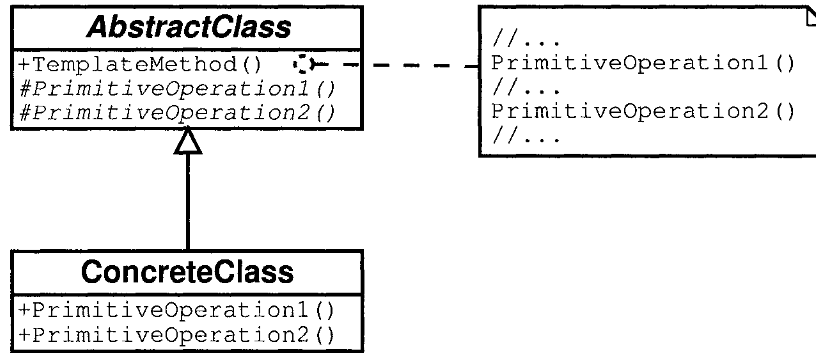



Figure 3: TEMPLATE METHOD design pattern [24].

- **Blackbox frameworks**: In blackbox framework, extensibility is achieve by defining interfaces that allow objects to be plugged in the framework through composition and delegation. Existing functionality can be extended by integrating into the framework new classes that comform to the existing interface. Design patterns such as STRATEGY, BRIDGE and DECORATOR are often used to support this kind of extensibility. See Figure 4.

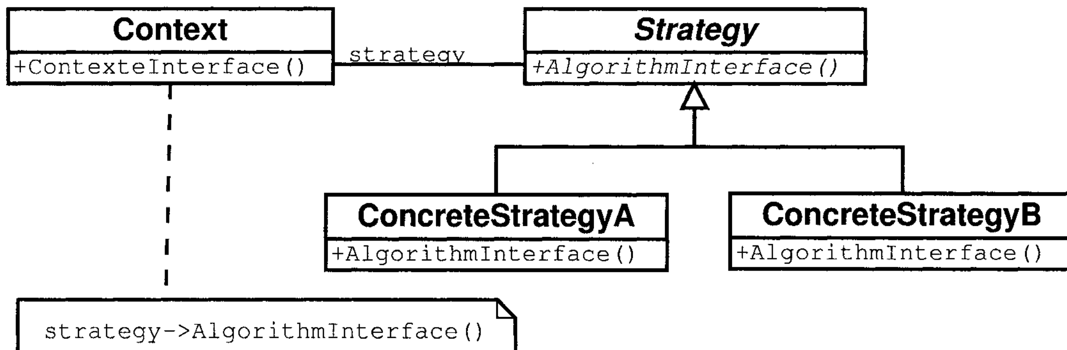



Figure 4: STRATEGY PATTERN in support of adaptation through composition (Separation principle).

## 2.3 Framework of Frameworks

One approach in building frameworks is to build a general, very flexible framework from which are derived additional smaller frameworks for narrower problem domains (see Figure 5). Wave ImAgE follows that approach, as it will be outlined in Part II of this thesis, when will be discussed the data processing, wavelet transform, and configuration frameworks.

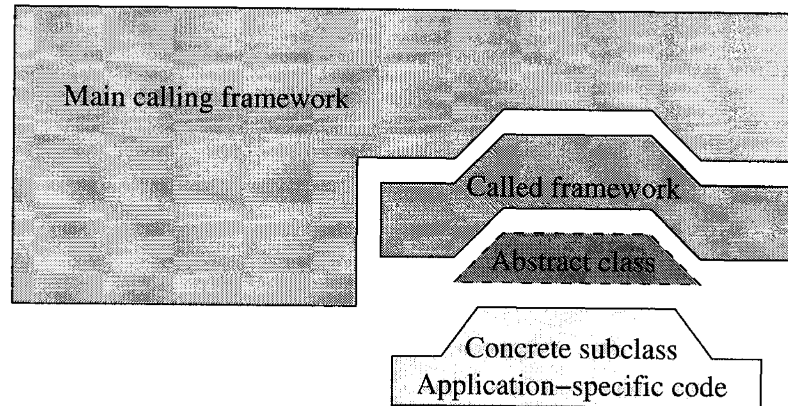



Figure 5: Framework of frameworks architecture.

## 2.4 Comparing Frameworks with Other Reuse Techniques

In this section, we compare frameworks with other widespread reuse techniques: patterns and class libraries. We start by comparing frameworks with the kind of reuse present in application programs ([50, 34]).

### Application Programs and Frameworks

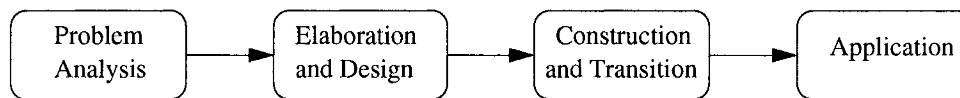
**Application programs**, as the name implies, are targeted towards a *single* application. Thus, *internal reuse* is a high priority. Internal reuse ensures that the application developer does not design and implement more than he has to: *code* is reused as much as possible within the application itself. A **framework** on the other end dictates the architecture for a *family* of applications. It defines the overall structure, its partitioning into classes and objects, their responsibilities, how they collaborate and the thread of control. The framework captures the design decisions that are common to all applications in the domain so that application developers can focus on the specifics of their applications. Thus, a framework emphasizes *design reuse* over code reuse.

These requirements placed on frameworks explain why frameworks are so hard to design: a framework designer gambles that a single architecture will work for all applications in a domain [24].

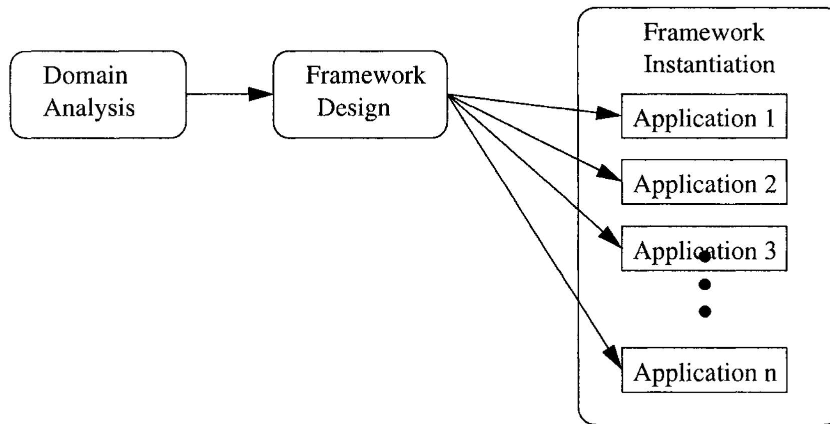
### Frameworks and Design Patterns

Design patterns and frameworks have similarities but they are different in three major ways [24]:

1. Design patterns are more *abstract* than frameworks. Frameworks are embodied in code, *i.e.*




Traditional Object–Oriented Design




Framework Development Process

Figure 6: Application versus framework design activities.

they are implementation artifacts written in a specific programming language and they can be reused directly. Design patterns can be studied, but only *examples*, or *instances* of design patterns can be embodied in code. Design patterns have to be re-implemented each time they are used.

2. Design patterns are smaller architectural elements than frameworks. A framework can contain many design patterns.

3. Design patterns are more general that frameworks. The same patterns can be found in many different frameworks.

Patterns are particularly useful in documenting recurring *micro*-architectures, which are abstractions of software components that experienced developers apply to resolve common design and implementation problems. Software abstractions documented as patterns *do not* yield reusable *code* directly. Frameworks on the other hand help developer to avoid costly reinvention of *software artifacts* by *implementing* common pattern languages and refactoring common implementation roles [50].
For example, consider the TEMPLATE METHOD design pattern [24].

The Intent section of this pattern says:

> Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

TEMPLATE METHOD lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

The Applicability section gives guidelines to help determine in which context the pattern should be used. For example, it says that the TEMPLATE METHOD pattern should be used

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.

The Structure section provides a class diagram showing the class structure of the pattern (see Figure 3). This is all pretty abstract, in the sense that there is no implementation that can be directly reused. The pattern can be applicable in many different contexts. It is up to a software designer to determine if the solution proposed by the design pattern is applicable in the context of his system. If its the case, then the designer has to implement a specific instance of the pattern.

## Frameworks and Class Libraries

Class libraries, the most common first-generation object-oriented reuse technique, support reuse-in-the-small do *not* capture the canonical flow of control, collaboration and variability among families of related software artifacts.

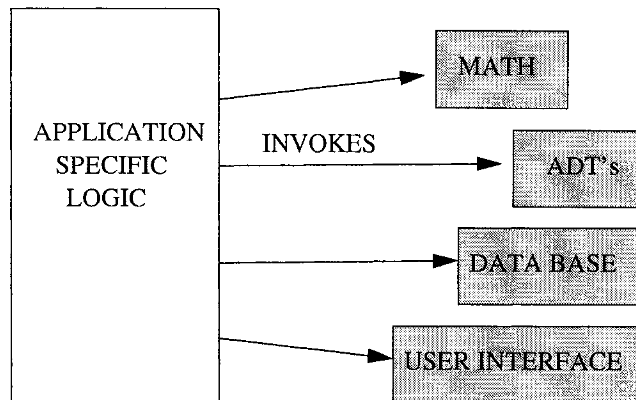



Figure 7: Class library architecture

Frameworks are a second-generation reuse technique that extends and complements class libraries in the following three ways:

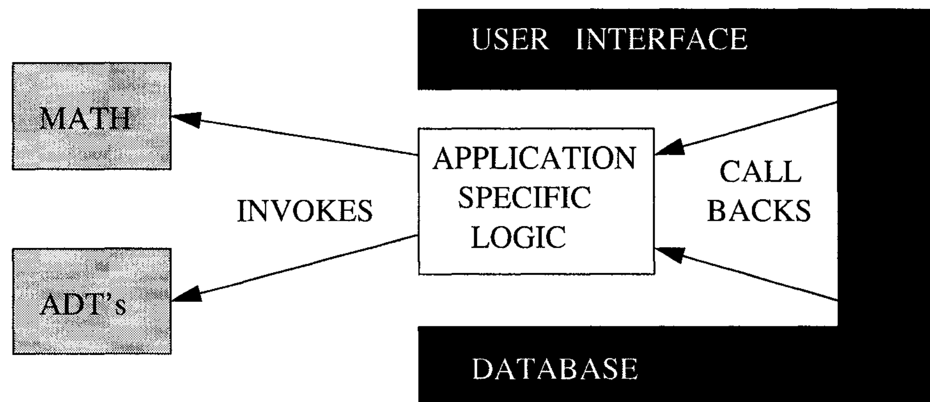



Figure 8: Application framework architecture

- **Frameworks are semi-complete applications that embody domain-specific structures and functionality.** Class libraries are often domain-independent and provide a relatively small granularity of reuse. For instance, the classes in the math library in Figure 7 are typically low level, relatively independent and general. Although they can be reused by many unrelated applications, the application developers are responsible for writing (and re-writing) the "glue code" that performs the biggest part of the application control flow.
  In contrast, classes in the framework collaborate to provide a reusable and customizable architecture for a family of related applications. As shown in Figure 8, the amount of application-specific code is greatly reduced since much of the domain-specific processing or logic common to the applications in the domain is factored into the generic components of the framework.

- **Frameworks are active and exhibit "inversion of control" at run-time** Class libraries are typically *passive* in the sense that they perform their processing by borrowing the thread of control defined in the applications that invoke the class library's methods. As a result, the control logic is not reused and the application developer must re-write the glue-code needed to drive the application.
  In contrast, frameworks are *active* in the sense that they direct the control flow of an application by calling back application code. The flow of control is thus inverted through a principle known as the *Hollywood Principle*, in other words, "Don't call us, we'll call you". A good illustration of this principle is the TEMPLATE METHOD pattern (Figure 3) which leads to such an inversion of control:

```
class FrameworkClass {
public:
    // Template method defining the
    // flow of control through a sequence of
```

```
    // calls to hook, primitive (abstract), and concrete methods.
    void TemplateMethod() {
        // Hook method
        Hook1();

        // Primitive (abstract) operation
        Primitive();

        // Concrete operation
        Concrete();

        // Hook method
        Hook2();
        // ...
     }
private:
    // Pure virtual, MUST be overidden by subclasses
    virtual void Primitive() = 0;
    virtual void Hook1() {
        // Default implementation (can be empty)
    }
    virtual void Hook2() {
        // Default implementation (can be empty)
    }
    void Concrete() {
       // Abstract class implementation.
    }
};
```

Framework customization is done here by subclassing a framework abstract class and overriding hook methods `Hook1()` and `Hook2()` (here, only `Hook1()` is overriden):

```
class ApplicationConcreteClass : public FrameworkAbstractClass {
public:
    void Hook1() {
        // Application-specific implementation
        // overiding default framework one.
    }
};
```

The application-specific subclass `ApplicationConcreteClass` *extends* (or *adapts*, or *customizes*) the parent class operation `TemplateMethod` by overriding the hook method `Hook1()`. Notice

also how the framework has control over how subclasses extend it: subclasses can only do so at well-defined locations, called *variation points* or *hot spots* implemented via hook methods. The framework is "put into motion" by a call to the framework controlling function, here `TemplateMethod`, which triggers the framework actions:

```
int main() {

    // Instantiation of application-specific subclass
    FrameworkAbstractClass *obj = new ApplicationConcreteClass;

    // A call to the framework class template method
    // will result in ApplicationConcreteClass's operations
    // being called (''Don't call us, we'll call you'')
    // in the sequence defined in the framework's
    // TemplateMethod (hence, controlling the flow).
    obj->TemplateMethod();
};
```

This is an example of a simple framework based on the TEMPLATE METHOD design pattern. We say that this framework *implements* the TEMPLATE METHOD pattern.
Subclasses can extend (or reimplement) the *variable* parts of an algorithm (which are the application-specific steps, whose implementation code goes into methods called "hooks") but they cannot alter the invariant part of the algorithm (which is the structure, or flow of control of the algorithm) captured in the template method. This is an example, in the small, of what is meant by the control or "architectural guidance" provided by the framework, which represents the commonality across applications.

- **Frameworks provide an integrated set of domain-specific structures and functionality.** Frameworks capture the commonalities and model the variabilities in application domains. By leveraging domain knowledge and prior efforts of experienced developers, frameworks embody common solutions to recurring software design challenges and application requirements. Thus, the design solutions and application requirements captured by the framework need not be recreated and revalidated everytime a new application is developed [48].

In practice, frameworks and class libraries are *complementary* technologies. For instance, frameworks can utilize class libraries (like the C++ Standard Template Library) internally to ease the development of the framework and application-specific code can use libraries to perform basic tasks, like string processing and numerical analysis, as shown in Figure 7.

To summarize, with class libraries:

- Classes are instantiated by clients

- Client calls the library functions
- There is no predefined flow of control
- There is no predefined interaction
- There is no default behavior

while with frameworks:

- Customization is provided by subclassing or through composition
- The framework calls client functions
- The framework takes care of the control flow of execution
- The framework defines object interaction
- The framework may provide default behavior

### 2.4.1 Compound Design Patterns

Compound design patterns are themselves patterns in that they name and document a recurring solution to a common problem. They are compound in nature in the sense that they express themselves in terms of other design patterns, even other compound patterns. The goal of compound patterns is to capture the synergy between patterns and make it explicit (See [65]).

### 2.4.2 Framework Construction Principles and Patterns

Wolfgang Pree (see [43, 19]) identified essential framework construction principles. These principles guide the development of frameworks extensible through the call-back style of programming. Adapting a framework can proceed according two styles of adaptation:

- *Overriding* methods of framework classes in subclasses (or implementation of interfaces).
- *Assembling* ready-made components (composition).

The construction principles that assist these two styles of adaptation are called:

- The *unification* construction principle (for whitebox frameworks).
- The *separation* construction principle (for blackbox frameworks).

#### 2.4.2.1 Template Methods and Hooks

*Hook methods* are placeholders that are invoked from within (often more complex) calling methods called *template methods*[1] The idea behind hook methods is that overriding them brings changes into the behavior of the template method from which they are called, and this, without having to alter the code of the class to which the template method belongs (not to mention the code of the template method itself). The classes containing the template method and the hook method are called respectively the *template class* and the *hook class.*

The construction principles identified by Pree are derived from the possible combinations of template and hooks with regards to the class they belong to.

#### 2.4.2.2 Unification Principle: Adaptation by Inheritance

The Unification principle requires overriding of the hook method in a subclass in order to change the behavior of the template method. Because the corresponding template and hook are in one class (they are united), the behavior of the template method can only be modified by defining a subclass. Runtime adaptations are not feasible.

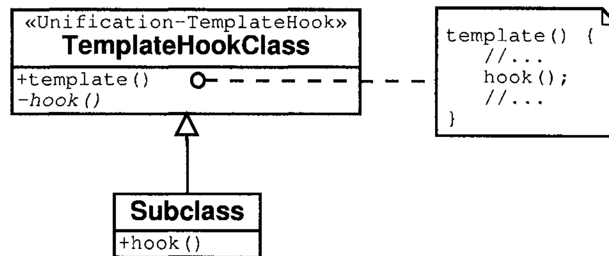



Figure 9: Unification principle

#### 2.4.2.3 Separation Principle: Adaptation through Composition

The Separation principle only requires an object instantiation and a redefinition of a reference that can be done at runtime. This is possible because the template and the hook methods are in different classes (they are separated).

[1] Not to be confused with the C++ template construct, which is a completely different thing, and with the TEMPLATE METHOD design pattern, which is a design pattern based on the *template/hook* methods concepts.

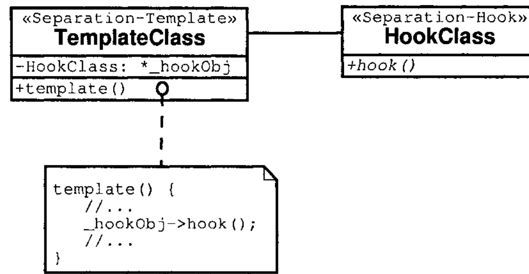



Figure 10: Separation principle

#### 2.4.2.4 Abstract Classes and Abstract Coupling

Abstract classes and abstract coupling are fundamental concepts to understand object-oriented frameworks. The purpose of a (pure) abstract class is to provide a signature so that other components in the software can be implemented based only on the signature of that class. For example, if we have an abstract class A providing a signature and that component B is implemented in terms of that signature, then we say that B is *abstractly coupled* with the abstract class A. If classes A and B are abstractly coupled classes in a framework capturing their interaction, then an adapted framework for which an implementation for the abstract class A is provided in a subclass A1 will work properly without change or recompilation because component B does not know of subclass A1.

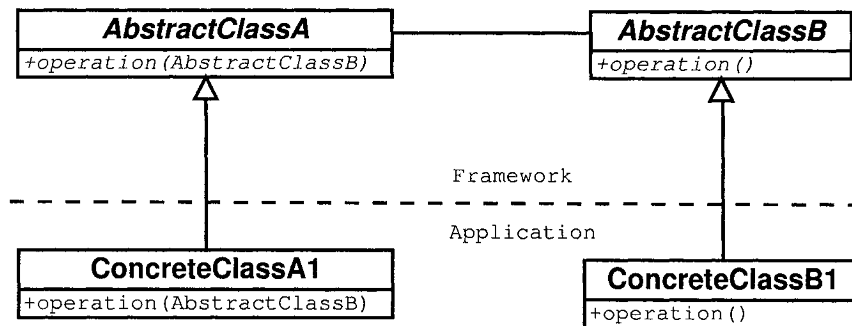



Figure 11: Abstract coupling.

## 2.5 Framelet Overview

The term *framelet* was introduced by Wolfgan Pree [44, 19] to designate non-overlapping groups of logically related design patterns and interfaces.
Some characteristics of framelets are:

- Framelets are small in size (typically less than 10 classes).
  As the name suggests, framelets are small frameworks. They usually consists in 2-3 design patterns and a few abstract interfaces.
- Framelets do not assume main control of an application.
  Since framelets are intended to be integrated with each other to form a framework, they must not have control of the application execution.
- Framelets have a clearly defined interface and are self-contained.
  They address a specific design problem and they can be developed and reused in isolation from other framelets.

#### 2.5.0.5 Framelets = Variation Points + Design Patterns

A useful distinction is made between *architecture* and *design*. The distinction between design and architecture corresponds to different levels of abstraction: concrete architecture versus abstract design. The difference between design and an architecture is the same as the difference between a design pattern and its instantiation. The pattern itself represents a pure design solution to a design problem while the instantiated design pattern into specific classes for a concrete architectural situation represents an *architectural* solution.

At the design level, frameworks consist mainly of a set of variation points and design patterns. *Framelets* partition this set into smaller subsets of related variation points and design patterns that are then handled independently. Framelets were introduced as a means to simplify the design process (since the design process of frameworks can be quite complex).
The design phase is chiefly concerned with incomplete abstract interfaces, independently handled design problems, and design patterns. During the transition to the architectural phase, the attention is shifted to concrete classes and objects.

The Wave ImAgE framework is being designed as a set of framelets, or small frameworks. The initial domain analysis is leading to the definition of a certain number of variation points which are being divided into a certain (smaller) number of framelets. Figure 12 illustrates the difference.

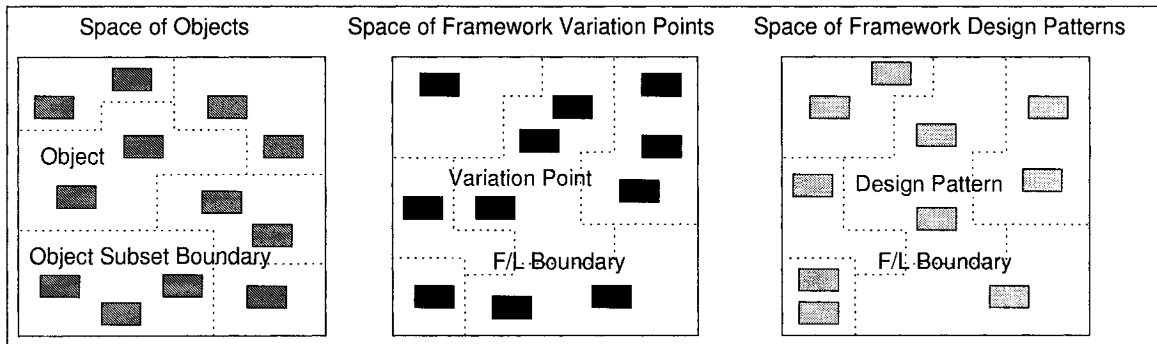



Figure 12: Framelets and object subsets.

### 2.5.1 On the Iterative Nature of Framework Development

Framework development is not an easy task. A framework is revisited often during its development. This constant evolution seems to be in contradiction with what a framework is supposed to achieve, *i.e.* to facilitate application development by providing a *stable*, core set of classes embedding design experience. Hopefully, it is not, as many authors recognize the iterative nature of framework development, even equating framework *development* with framework *evolution* [9]. Since what we are proposing are design elements of a prototype framework, Wave ImAgE will most likely go through many rounds of iteration before it can claim to have reached some sort of maturity.

## 2.6 Overview of Wave ImAgE

This section provides an overview of Wave ImAgE by listing and giving a brief description of the patterns and frameworks that are assembled into the main calling framework.

Figure 13 gives a global idea of the architecture of the framework. The wrapping box represents the core components of the framework which are calling back the application-specific components represented by the wrapped box.

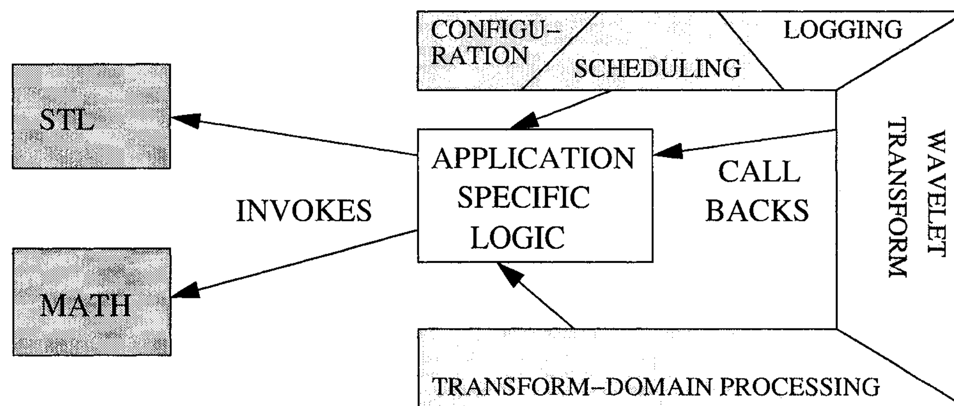



Figure 13: Wave ImAgE framework architecture.

### 2.6.1 Design Patterns in Wave ImAgE.

Some of the key patterns used in Wave ImAgE (see Figure 14) are described below:

- The COMPOSITE pattern [24] is used in the representation of wavelet transforms (as a recursive composition of subbands) and for the representation of composite processors.
- The DECORATOR pattern [24] is used to allow the dynamic extension of the wavelet transform and transform steps in terms of preprocessing and prefiltering steps.
- The VISITOR pattern [24] is used in the processing framework to ensure that processors and the data they are operating on within an application are brought together in a type-safe manner. Also, this pattern is used to allow the addition of new processors without requiring changing the classes in the data hierarchy on which they operate.
- The STRATEGY pattern [24] is used throughtly to configure wavelet transforms with transform steps for example.

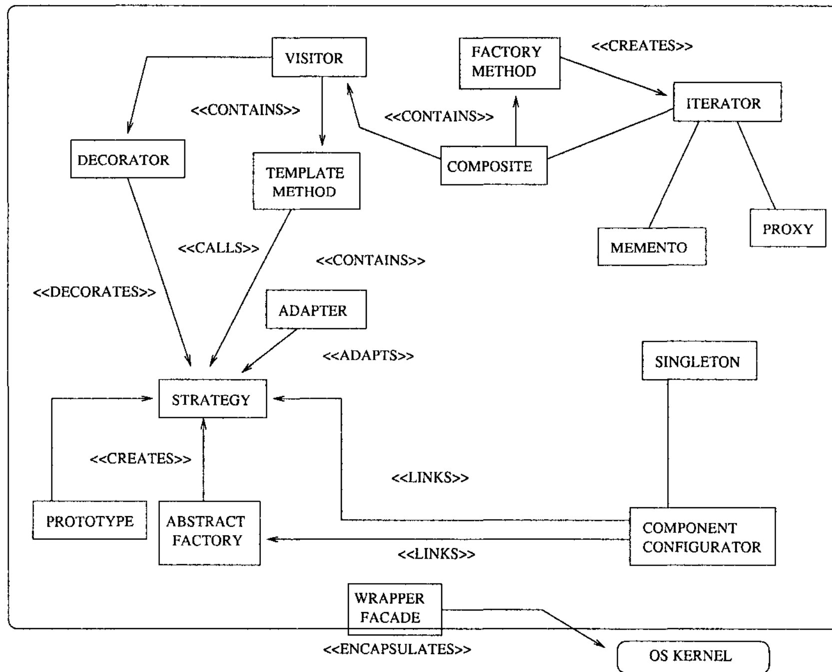



Figure 14: The main patterns used in Wave ImAgE.

- The SINGLETON pattern [24] is used to ensure that the configurator, the logger, the scheduler, the options have only one instance each so that they provide a global access point.
- The ITERATOR pattern [24] is used to encapsulate image and transform elements access and traversals as well as to iterate through image processing experiments lineups.
- The MEMENTO pattern is used to hold the state of a traversal in the wavelet transform descriptor.
- The ABSTRACT FACTORY pattern [24] is used to consolidate multiple strategies into semantically compatible processors and experiments configurations.
- The PLUGGABLE FACTORY pattern ([66] and [64]) is also used to consolidate multiple strategies into semantically compatible processors and experiments configurations. Pluggable factories are parametrized concrete factories.
- The PROTOTYPE pattern [24] is used to implement the PLUGGABLE FACTORY pattern.
- The FACTORY METHOD pattern [24] is use to create strategies (processors) and iterators.
- The TEMPLATE METHOD pattern [24] is used to capture the structure of some image processing algorithms (like the wavelet transform).

- The ADAPTER pattern [24] is used to allow objects that have different interfaces to work together. Wave ImAgE uses this pattern's generic form (using templates as an example of external polymorphism) as a helper for application developers to write and reuse derived Processors' implementation in experiment-specific concrete strategies.
- The PROXY pattern [24] is used in the ITERATOR pattern to ensure automatic destruction of iterators.
- The ENVELOP-LETTER idiom is used to separate a data abstraction from its implementation.
- A variant of the COMPONENT CONFIGURATOR pattern [51] which implements the MANAGER pattern is used to centralize configuration of image processing experiments with processor strategies.
- The WRAPPER FACADE [51] pattern is used to encapsulate low level functionality at the OS level.

### 2.6.2 Frameworks in Wave ImAgE.

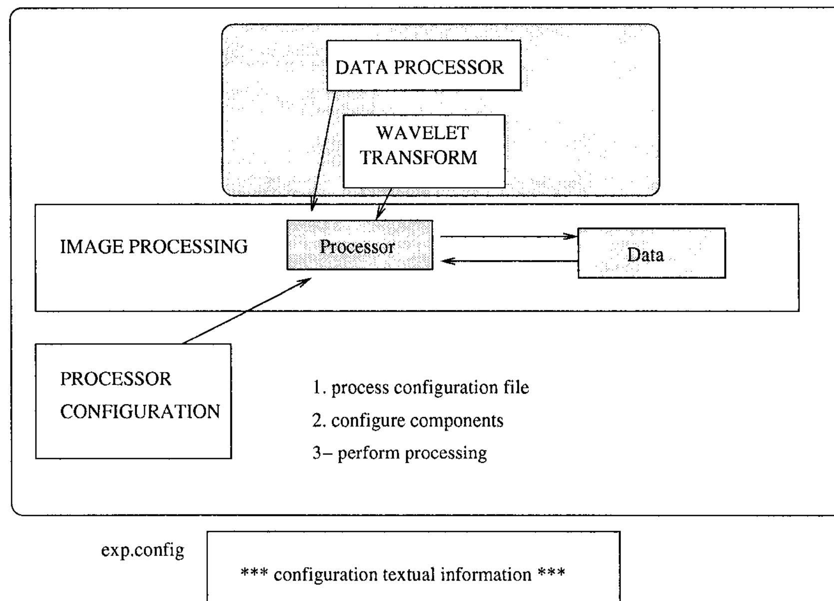



Figure 15: The Frameworks used in Wave ImAgE.

Wave ImAgE contains three frameworks. Below we describe each framework, outline which pattern(s) it implements, and discuss how it enhances reuse.

- The **Data Processing** framework prototypes (see Chapter 4) implements two implementation variants of the VISITOR pattern, the ACYCLIC VISITOR and the STAGGERED VISITOR patterns

to allow for the double-dispatching of processors implementation. It enhance reuse by allowing easy addition of new image processors without modification of the stable data hierarchy and facilitate reuse by grouping related operations together.

- The **Wavelet Transform** framework (see Chapter 5) implements the TEMPLATE METHOD, STRATEGY, DECORATOR, COMPOSITE, ITERATOR, and ADAPTER patterns to allow for the adaptation and extension of image processing algorithms.

- The **Image Processing Experiment Configuration and Scheduling** framework (see Chapter 6) implements a variant of the the COMPONENT CONFIGURATOR and the ABSTRACT FACTORY pattern to facilitate dynamic configuration of image processing experiments with processors.

# Chapter 3

# Wavelets, Wavelet Transforms and Applications

## 3.1 Wavelets and Wavelet Transforms

This section provides background information about the concepts and theory of wavelets and wavelet transforms.

### 3.1.1 What are Wavelets ?

A *wave* is defined as an oscillating function of time or space, such as a sinusoid in Fourier analysis. A *wavelet* is a "small wave", *i.e.* a wave which has its energy concentrated in time (non zero only on a finite interval). This "local" aspect of the wavelet makes it a nice tool for the analysis of transient, time-varying signals.
It is known that the wavelet transform is well suited for localized frequency analysis because the basis functions have short time resolution for high frequency (which corresponds to a narrow window able to detect high-frequency burst) and long time resolution for low frequency (which corresponds to a long window to analyse slowly varying low-frequency components). The main reason behind the adoption of the wavelet representation, the most powerful aspect, is that a function is decomposed into a *multiresolution* hierarchy. The mathematics and practical interpretations of wavelets are best served using the concept of *resolution*, referring to the effects of changing scale. I will briefly introduce that representation in this section, and then move to wavelet functions and transforms..

### 3.1.2 Multiresolution Analysis (MRA)

A set of *scaling functions* is defined in terms of integer translates of the basic scaling function by

$$\varphi(t) = \varphi(t-k) \qquad k \in \mathbb{Z} \qquad \varphi \in \mathbb{L}^{\nvDash} \tag{1}$$

The function space $L^2(\mathbb{R})$ is the space of all square integrable functions $f(t)$. The subspace of $L^2(\mathbb{R})$ *spanned* by these functions is defined as

$$\mathcal{V}_{\prime} = \overline{\underset{\|}{\text{Span}}\{\varphi_{\|}(\sqcup)\}} \tag{2}$$

for all integers $k$, and where the over-bar denotes closure. *This means that* functions $f(t)$ can be expressed as a linear combination of the $\varphi_k(t)$'s

$$f(t) = \sum_k a_k \varphi_k(t) \qquad \text{for any} \qquad f(t) \in \mathcal{V}_{\prime} \tag{3}$$

And here comes the notion of a scale. The size of the spanned subspace can be increased by changing the time scale of the scaling functions such that

$$\varphi_{j,k}(t) = 2^{j/2}\varphi(2^j t - k) \tag{4}$$

whose span is

$$\mathcal{V}_{|} = \overline{\underset{\|}{\text{Span}}\{\varphi_{\|}(\in^{|}\sqcup)\}} \tag{5}$$

This means that if $f(t) \in \mathcal{V}_{|}$, then

$$f(t) = \sum_k a_k \varphi(2^j t + k) \tag{6}$$

The basic requirement for a multiresolution analysis is a nesting of subspaces as

$$\ldots \subset \mathcal{V}_{-\in} \subset \mathcal{V}_{-\infty} \subset \mathcal{V}_{\prime} \subset \mathcal{V}_{\infty} \subset \mathcal{V}_{\in} \subset \ldots \subset \mathcal{L}^{\in} \tag{7}$$

or

$$\mathcal{V}_{|} \subset \mathcal{V}_{|+\infty} \tag{8}$$

with

$$\mathcal{V}_{-\infty} = \prime, \qquad \mathcal{V}_{\infty} = \mathcal{L}^{\in} \tag{9}$$

And because of the nested subspaces

$$f(t) \in \mathcal{V}_{|} \Leftrightarrow \{(\in\sqcup) \in \mathcal{V}_{|+\infty} \tag{10}$$

*Elements in a space are scaled versions of the elements in the next space.* Because of the nested structure of the subspaces, $\varphi(t)$ can be written as a weighted sum of scaled and shifted $\varphi(t)$ such that

$$\boxed{\varphi(t) = \sum_n h(n)\sqrt{2}\varphi(2t-n), \qquad n \in \mathbb{Z}} \tag{11}$$

where the $h(n)$'s are the *scaling function coefficients.*

### 3.1.3 The Wavelet Functions

The orthogonal complement of $\mathcal{V}_{|}$ in $\mathcal{V}_{|+\infty}$ is defined as $\mathcal{W}_{|}$. We define the *wavelet spanned subspace* $\mathcal{W}_{\prime}$ by

$$\mathcal{V}_{\infty} = \mathcal{V}_{\prime} \oplus \mathcal{W}_{\prime} \tag{12}$$

which yields

$$L^2 = \mathcal{V}_{\prime} \oplus \mathcal{W}_{\prime} \oplus \mathcal{W}_{\infty} \oplus \dots \tag{13}$$

where $\mathcal{V}_{\prime}$ is the initial space spanned by the scaling function $\varphi(t-k)$. Since the *wavelets* as they are defined are in the space spanned by the next narrower scaling function ($\mathcal{W}_{\prime} \subset \mathcal{V}_{\infty}$), we can represent them as

$$\boxed{\psi(t) = \sum_n h_1 \sqrt{2}\varphi(2t-n), \qquad n \in \mathbb{Z}} \tag{14}$$

for some set of coefficients $h_1(n)$, the *wavelet coefficients*, which are related to the scaling coefficients, for orthogonality, in the case of a finite (even) length-N signal by $h_1(n) = (-1)^n h(N-1-n)$. Similarly as in (4) for the scaling function, we have a family of functions generated by (14), the *mother wavelet* of the form

$$\psi_{j,k}(t) = 2^{j/2}\psi(2^j t - k) \tag{15}$$

### 3.1.4 The Discrete Wavelet Transform

Since we have $L^2$ expressed as in (13), using (4) and (15) we have the *discrete wavelet transform* (DWT)

$$f(t) = \sum_k c_{j_0}(k)\varphi_{j_0,k}(t) + \sum_k \sum_{j=j_0}^{\infty} d_j(k)\psi_{j,k}(t) \tag{16}$$

where the coefficients themselves (the $c_{j_0}(k)$ and $d_j(k)$) in the above expansion are called the *discrete wavelet transform*. They are calculated by inner products (convolution products):

$$c_j(k) = \langle f(t), \varphi_{j,k}(t) \rangle \tag{17}$$

and

$$d_j(k) = \langle f(t), \psi_{j,k}(t) \rangle \tag{18}$$

### 3.1.5 An Example: The Haar Wavelet

As mentioned above, the basic constructions of wavelets using scaling functions require first a *scaling function* $\varphi$ which is essentially a function $\varphi(t)$ that can be written as a linear combination of $\varphi(2t-k)$,

a 1/2 scaled and $k/2$ translated version of $\varphi(t)$:

$$\varphi(t) = \sum_{k=-\infty}^{\infty} h_k \varphi(2t-k). \tag{19}$$

This is the *two-scale relation for the scaling functions* and the sequence $\{h_k\}$ is the *two-scale sequence* of $\varphi$:

$$\varphi(t) = \varphi(2t) + \varphi(2t-1) \tag{20}$$

which means that (11) is satisfied for the coefficients $h(0) = 1/\sqrt{2}$ and $h(1) = 1/\sqrt{2}$, which correspond to the set of scaled and translated "box" functions *i.e.*

$$\varphi(t) = \begin{cases} 1 & \text{for } 0 \leq t < 1, \\ 0 & \text{elsewhere} \end{cases} \tag{21}$$

Associated with the scaling function $\varphi$ is the wavelet function $\psi$ defined by

$$\psi(t) = \varphi(t) - \varphi(2t-1) \tag{22}$$

and thus (14) is satisfied for $h_1(0) = 1/\sqrt{2}$ and $h_1(1) = -1/\sqrt{2}$. The wavelets corresponding to the box basis are known as the *Haar wavelets* and are given by

$$\psi(t) = \begin{cases} 1 & \text{for } 0 \leq t < 1/2, \\ -1 & \text{for } 1/2 \leq t < 1, \\ 0 & \text{elsewhere} \end{cases} \tag{23}$$

### 3.1.6 Multiwavelets

Multiwavelets have recently been developed by using translates and dilates of more than one mother wavelet functions [54]. They are known to have several advantages over scalar wavelets such as short support, orthogonality, symmetry, and higher order of vanishing moments which make them better suited to image processing than single wavelets.

Multiwavelet basis uses translations and dilations of $L \geq 2$ scaling functions $\{\varphi_k(x)\}_{1 \leq k \leq L}$ and $L$ mother wavelet functions $\{\psi_k(x)\}_{1 \leq k \leq L}$. If we write $\Phi(x) = (\varphi_1(x), \varphi_2(x), \ldots, \varphi_L(x))^T$ and $\Psi(x) = (\psi_1(x), \psi_2(x), \ldots, \psi_L(x))^T$, then we have

$$\Phi(x) = 2 \sum_{k=0}^{2N-1} H_k \Phi(2x-k), \tag{24}$$

and

$$\Psi(x) = 2 \sum_{k=0}^{2N-1} G_k \Phi(2x-k). \tag{25}$$

where $\{H_k\}_{0\leq k\leq 2N-1}$ and $\{G_k\}_{0\leq k\leq 2N-1}$ are $L\times L$ filter matrices.

The two functions $\varphi_1(x)$ and $\varphi_2(x)$ can be generated via (24). Similarly, the two mother wavelet functions $\psi_1(x)$ and $\psi_2(x)$ can be constructed by (25).

Let $V_J$ be the closure of the linear span of $2^{J/2}\varphi_l(2^J t-k), l=1,2; k\in Z$. With the above constructions, it has been proved that $\varphi_l(t-k), l=1,2; k\in Z$ form an orthonormal basis for $V_0$, and moreover the dilations and translations $2^{j/2}\psi_l(2^j-k), l=1,2; j,k\in Z$ form an orthonormal basis for $L^2(R)$. In other words, the spaces $V_j, j\in Z$, form an orthogonal multiresolution analysis of $L^2(R)$. Let

$$H(w) = \sum_{k=0}^{3} H_k e^{iwk}, \tag{26}$$

$$G(w) = \sum_{k=0}^{3} G_k e^{iwk}. \tag{27}$$

Let $f\in V_0$, then

$$\begin{aligned} f(t) &= \sum_{k\in Z}(c_{1,0,k}\varphi_1(t-k)+c_{2,0,k}\varphi_2(t-k)) \\ &= \sum_{k\in Z}(c_{1,J_0,k}2^{J_0/2}\varphi_1(2^{J_0}t-k) \\ &\quad + c_{2,J_0,k}2^{J_0/2}\varphi_2(2^{J_0}t-k)) \\ &\qquad + \sum_{J_0\leq j<0}\sum_{k\in Z}(d_{1,j,k}2^{j/2}\psi_1(2^j t-k) \\ &\qquad + d_{2,j,k}2^{j/2}\psi_2(2^j t-k)) \end{aligned} \tag{28}$$

where

$$c_{i,j,k} = \int f(t)2^{j/2}\varphi_i(2^j t-k)dt \tag{29}$$

$$d_{i,j,k} = \int f(t)2^{j/2}\psi_i(2^j t-k)dt \tag{30}$$

for $i=1,2; j,k\in Z$ and $J_0<0$. By the dilation equations (24) and (25), we have the following recursive relationship between the coefficients $(c_{1,j,k}, c_{2,j,k})^T$ and $(d_{1,j,k}, d_{2,j,k})^T$:

$$\begin{pmatrix} c_{1,j-1,k} \\ c_{2,j-1,k} \end{pmatrix} = \sqrt{2}\sum_{l=0}^{3} H_l \begin{pmatrix} c_{1,j,2k+l} \\ c_{2,j,2k+l} \end{pmatrix}, \quad j,k\in Z \tag{31}$$

and

$$\begin{pmatrix} d_{1,j-1,k} \\ d_{2,j-1,k} \end{pmatrix} = \sqrt{2}\sum_{l=0}^{3} G_l \begin{pmatrix} d_{1,j,2k+l} \\ d_{2,j,2k+l} \end{pmatrix}, \quad j,k\in Z \tag{32}$$

Moreover,

$$\begin{pmatrix} c_{1,j,l} \\ c_{2,j,l} \end{pmatrix} = \sqrt{2}\sum_{k=0}^{3}\left(H_k^T \begin{pmatrix} c_{1,j-1,2k+l} \\ c_{2,j-1,2k+l} \end{pmatrix} + G_k^T \begin{pmatrix} d_{1,j-1,2k+l} \\ d_{2,j-1,2k+l} \end{pmatrix}\right). \tag{33}$$

Multiwavelets have some advantages in comparison to single wavelets like Haar and Daubechies. For example, short support, orthogonality, symmetry, and higher order of vanishing moments, are important properties for image processing. A single wavelet cannot exibit all these properties at the same time. Therefore, multiwavelets can potentially give better results than single ones in image compression [54] and denoising [6], but their implementation is also more intricate and complex. Since the multiwavelets coefficients are matrix coefficients, the transform requires vector input streams (2 in the case of GHM multiwavelet, see next section), and then prefiltering becomes an important issue (discussed in the next sections).

#### 3.1.6.1 An Example of Multiwavelets: Geronimo-Hardin-Massopust

An important multiwavelet system was constructed by J. Geronimo, D. Hardin and P. Massopust (GHM) [26]. Their system uses translations and dilations of $L = 2$ scaling functions $\{\varphi_k(x)\}_{k=1,2.}$ and $L$ mother wavelet functions $\{\psi_k(x)\}_{k=1,2.}$. If we write $\Phi(x) = (\varphi_1(x), \varphi_2(x))^T$ and $\Psi(x) = (\psi_1(x), \psi_2(x))^T$, then we have

$$\Phi(x) = 2 \sum_{k=0}^{2N-1} H_k \Phi(2x - k), \tag{34}$$

and

$$\Psi(x) = 2 \sum_{k=0}^{2N-1} G_k \Phi(2x - k). \tag{35}$$

where $\{H_k\}_{0 \leq k \leq 2N-1}$ and $\{G_k\}_{0 \leq k \leq 2N-1}$ are $2 \times 2$ filter matrices.(see Fig. (16)).

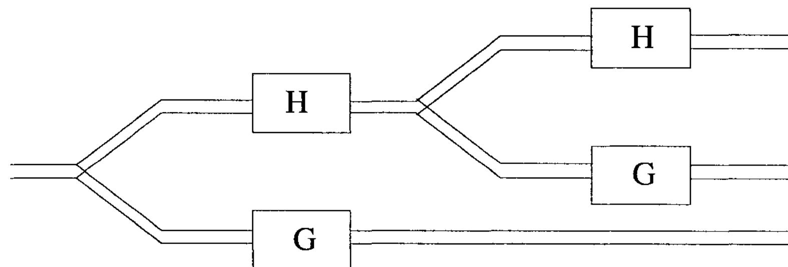



Figure 16: Multiwavelet analysis filter bank.

The GHM scaling and wavelet equations have four coefficients

$$\Phi(x) = \begin{bmatrix} \varphi_1(x) \\ \varphi_2(x) \end{bmatrix} = C[0]\Phi(2x) + C[1]\Phi(2x-1) + C[2]\Phi(2x-2) + C[3]\Phi(2x-3). \tag{36}$$

where

$$H_0 = \begin{bmatrix} 3/10 & 2\sqrt{2}/5 \\ -\sqrt{2}/40 & -3/20 \end{bmatrix} \qquad H_1 = \begin{bmatrix} 3/10 & 0 \\ 9\sqrt{2}/40 & 1/2 \end{bmatrix},$$

$$H_2 = \begin{bmatrix} 0 & 0 \\ 9\sqrt{2}/40 & -3/20 \end{bmatrix} \qquad H_3 = \begin{bmatrix} 0 & 0 \\ -\sqrt{2}/40 & 0 \end{bmatrix}.$$

and

$$\Psi(x) = \begin{bmatrix} \psi_1(x) \\ \psi_2(x) \end{bmatrix} = D[0]\Phi(2x) + D[1]\Phi(2x-1) + D[2]\Phi(2x-2) + D[3]\Phi(2x-3). \qquad (37)$$

where

$$G_0 = \begin{bmatrix} -\sqrt{2}/40 & -3/20 \\ -1/20 & -3\sqrt{2}/20 \end{bmatrix} \qquad G_1 = \begin{bmatrix} 9\sqrt{2}/40 & -1/2 \\ 9/20 & 0 \end{bmatrix},$$

$$G_2 = \begin{bmatrix} 9\sqrt{2}/40 & -3/20 \\ -9/20 & 3\sqrt{2}/20 \end{bmatrix} \qquad G_3 = \begin{bmatrix} -\sqrt{2}/40 & 0 \\ 1/20 & 0 \end{bmatrix}.$$

then the two functions $\varphi_1(x)$ and $\varphi_2(x)$ can be generated via (34). Similarly, the two mother wavelet functions $\psi_1(x)$ and $\psi_2(x)$ can be constructed by (35).

There are four remarkable properties of the GHM scaling functions:

- They have **short support** (on [0,1] and [0,2]).
- **Symmetry**: The scaling functions are symmetric and the wavelets form a symmetric/antisymmetric pair.
- **Orthogonality**: Integer translates of the scaling functions are orthogonal.
- **Approximation**: The system has a second order of approximation.[1]

A scalar wavelet cannot possess all these properties at the same time. Therefore, multiwavelets are expected to give better results than single ones in image compression and de-noising [54].

[1]The higher the *approximation order* of a wavelet base is, the better we are able to approximate smooth functions using a small number of terms. However, the higher the approximation order, the longer the *support*, and this is the price to pay for high approximation order since longer support means larger coefficients generated by each singularities. Thus, to obtain a compact representation of a function with singularities, we need to find a *good balance* between the approximation order needed to *efficiently approximate smooth portion* of the function, and the length of the support that *sufficiently localizes singularities*. For image processing applications, a good balance is attained with two to four order of approximation.

### 3.1.7 Complex Symmetric Daubechies Wavelets

Symmetric Daubechies (SD) wavelets and their underlying scaling functions are obtained from a complex MRA.

$$\varphi(x) = 2\sum_k a_k \varphi(2x-k) \tag{38}$$

$$\psi(x) = 2\sum_k b_k \varphi(2x-k) \tag{39}$$

where $\varphi(x)$ and $\psi(x)$ are the complex scaling and wavelet functions, with the complex-valued coefficients $a_k = a_{2J+1-k}$ and $b_k = (-1)^k a^*_{2J+1-k}$, $J = 0, 2, 4, \ldots$, $k = 0, \ldots, 2J+1$ where $*$ stands for complex conjugate. SD wavelets are known to share the same properties as the real Daubechies wavelets (*i.e.* compact support, orthogonality and vanishing moments) but also features such as symmetric wavelets and scaling functions and better interpolation capabilities (for a detailed presentation we refer to [22, 23, 33]).

From equations (38) and (39) it is clear that there is a link between scalar complex wavelets and the multiwavelet framework as noted in [22] and [33]. Rewriting the dilation equation and equation (39) in matrix notation, it is obvious that SD wavelets can have a multiwavelet interpretation,

$$\varphi(x) = \begin{bmatrix} \varphi^{\Re}(x) \\ \varphi^{\Im}(x) \end{bmatrix} = 2\sum_k A[k] \begin{bmatrix} \varphi^{\Re}(2x-k) \\ \varphi^{\Im}(2x-k) \end{bmatrix} \tag{40}$$

$$\psi(x) = \begin{bmatrix} \psi^{\Re}(x) \\ \psi^{\Im}(x) \end{bmatrix} = 2\sum_k B[k] \begin{bmatrix} \psi^{\Re}(2x-k) \\ \psi^{\Im}(2x-k) \end{bmatrix} \tag{41}$$

where $A[k]$ and $B[k]$ are the $2 \times 2$-matrix scaling and wavelet function coefficients.

#### 3.1.7.1 Complex Transform as a special case of Multiwavelet Transform

It is possible to *implement* the complex wavelet transform as a special case of multiwavelet (for $r = 2$). In order to do that, we need to translate the operations on complex-valued coefficients in terms of operations on vectors and matrices.

For example, the product of two complex numbers $a$ and $b$ defined as

$$a * b = (a_1 + ja_2) * (b_1 + jb_2) = (a_1b_1 - a_2b_2) + j(a_1b_2 + a_2b_1) \tag{42}$$

can be written in matrix notation as the product

$$\begin{bmatrix} a_1 & -a_2 \\ a_2 & a_1 \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = \begin{bmatrix} (a_1b_1 - a_2b_2) \\ (a_1b_2 + a_2b_1) \end{bmatrix}. \tag{43}$$

So, having a set of complex filter coefficients $c_i^{\mathbb{C}} = c_i^{\Re} + jc_i^{\Im}$ and an input sequence of complexe numbers $x^{\mathbb{C}} = [x_0^{\mathbb{C}}, x_1^{\mathbb{C}}, \ldots, x_{n-1}^{\mathbb{C}}]$, the complex convolution product $y = c * x$ can be written as

$$\mathbf{Y[i]} = \begin{bmatrix} c_i^{\Re} & -c_i^{\Im} \\ c_i^{\Im} & c_i^{\Re} \end{bmatrix} \begin{bmatrix} x_{i-k}^{\Re} \\ x_{i-k}^{\Im} \end{bmatrix} = \begin{bmatrix} (c_i^{\Re} x_{i-k}^{\Re} - c_i^{\Im} x_{i-k}^{\Im}) \\ (c_i^{\Re} x_{i-k}^{\Im} + c_{i-k}^{\Im} x_{i-k}^{\Re}) \end{bmatrix}. \tag{44}$$

Thus, the complex-values filter coefficients can simply be written in matrix notation and the resulting implementation of the complex low pass and high pass filters follow the one of the multiwavelet transform.

### 3.1.8 Wavelet Transforms

In this section, we introduce the concept of wavelet transform. We will briefly look at the different components of a wavelet transform.

### 3.1.9 Preprocessing and Sampling

In applications like image denoising, we are given a set of *function samples* to transform rather than a set of scaling function coefficients as it is assumed by the transform. Thus, the working assumption in the single wavelet case, is that the scaling function coefficients are a good approximation to the function samples.

In the multiwavelet case however, when the multiscaling functions differ significantly, this assumption is not useful anymore since a map from image samples to scaling function coefficients introduces high-frequency artefacts. Thus a method to approximate multiscaling function coefficients has to be defined, *i.e.* a process for obtaining the coefficients "at scale 0", i.e. $C_k^{(0)}$. Such a method is called *preprocessing* or *prefiltering* and has been the object of several recent papers (see for example [54, 55, 68, 27]). Also, realizable as matrix-valued filter banks leading to wavelets bases, multiwavelets differ from scalar (single) wavelets in requiring two or more input streams to the multiwavelets filter bank. Methods for obtaining such a vector input stream from a one-dimensional signal are called preprocessing schemes and are implemented through prefilters . The aim of preprocessing is to associate to a given scalar input signal of length $N$ a sequence of length-2 vectors for input to the analysis algorithm.

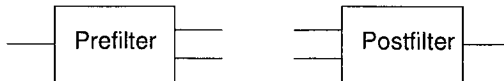



Figure 17: Prefilter and postfilter schematic illustration.

We will consider here two kinds of preprocessing schemes:

- **Oversampled scheme:** if it produces $N$ length-2 vectors from the length-$N$ input signal.
- **Critically sampled scheme:** if it produces $N/2$ length-2 vectors from the length-$N$ input signal.

We briefly introduce the two general schemes.

#### 3.1.9.1 Oversampling

As mentionned above, when the preprocessing produces $N$ length-2 input vector it is said to be an oversampling scheme. A well known such scheme called "*Repeated-row preprocessing*" is presented here.

**Repeated-row preprocessing** An input length-2 vectors $\{\vec{u}_{0,k}\}$ is formed from an original one-dimensional sequence $X$ of length $N$ via

$$\vec{u}_{0,k} = \begin{bmatrix} u^0_{0,k} \\ u^1_{0,k} \end{bmatrix} = \begin{bmatrix} X_k \\ \alpha X_k \end{bmatrix}, k = 0, \ldots, N-1.$$

where $\alpha$ is a constant[2]. Representing the preprocessing by a matrix multiplication we have

$$\mathcal{P}\mathcal{X} = \mathcal{V}_0,$$

where $\mathcal{P}$ is a $2N \times N$ prefilter operator, $\mathcal{X}$ is a $N \times 1$ input sequence, and $\mathcal{V}_0$ the $2N \times 1$ prefiltered sequence. We have

$$\mathcal{P} = \begin{bmatrix} 1 & 0 & 0 & \ldots \\ \alpha & 0 & 0 & \ldots \\ 0 & 1 & 0 & \ldots \\ 0 & \alpha & 0 & \ldots \end{bmatrix} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \vdots \end{bmatrix}_{N\times 1} = \begin{bmatrix} X_0 \\ \alpha X_0 \\ X_1 \\ \alpha X_1 \\ \vdots \end{bmatrix}_{2N\times 1} = \begin{bmatrix} \begin{bmatrix} u^0_{0,0} \\ u^1_{0,0} \end{bmatrix} \\ \begin{bmatrix} u^0_{0,1} \\ u^1_{0,1} \end{bmatrix} \\ \vdots \end{bmatrix}_{2N\times 1} \tag{45}$$

#### 3.1.9.2 Critical Sampling

Critical sampling produces $N$ length-2 vectors from a input signal of length $N$ thus preserving the sampling rate of the input signal.

Some implementations of the multiwavelet transform create a 2-input signals by simply using 2 adjacent rows (and columns in the vertical direction) as input. Better results could be obtained by

[2]The choice of $\alpha$ is not discussed here, neither the properties that a "good" prefiltering scheme should possess. These matters are fully discussed in [54],[55] and [68]. However, a conveniently implemented scheme for 2D GHM repeated row preprocessing uses $\alpha = 1/\sqrt{2}$.

using of the two-dimensionality of the matrix filter coefficients [54] through a better understanding of the approximation property (and intricacies) of multiwavelets. Strela and Walden in [55] however claimed to have obtained good results using repeated-row preprocessing. We note that no true 2-D preprocessing scheme exists now and that the preprocessing scheme conveniently used for the GHM multifilter bank, called "approximation preprocessing" ([54, 55, 68]) is in fact a 1-D scheme for 2-D image signals and that further investigation towards such a 2-D scheme constitute and interesting avenue for research.

#### 3.1.9.3 Matrix Preprocessing

Matrix preprocessing can be represented by:

$$\mathbf{v_{0,k}} = \sum_{\mathbf{m=0}}^{\mathbf{M}} \mathbf{Q_m} \begin{bmatrix} X_{2(m+k)} \\ X_{2(m+k)+1} \end{bmatrix}, \qquad \mathbf{k = 0, \ldots, N/2 - 1}$$

where $X$ is an input sequence, $\mathbf{Q_0}, \mathbf{Q_0}, \ldots, \mathbf{Q_0}$ are $2 \times 2$ matrices, and $\mathbf{v}$ the resulting prefiltered vector signal.

If we represent the matrix preprocessing by a matrix multiplication,

$$\mathcal{Q}\mathbf{X} = \mathbf{V_0},$$

where $\mathcal{Q}$ is $N \times N$, $\mathbf{X}$ is $N \times 1$, and $\mathbf{V_0}$ is $N \times 1$, then we have

$$\mathcal{Q} = \begin{bmatrix} \mathbf{Q_0} & \mathbf{Q_1} & \ldots & \mathbf{Q_M} & \mathbf{0_2} & \ldots & \ldots & \ldots \\ \mathbf{0_2} & \mathbf{Q_0} & \mathbf{Q_1} & \ldots & \mathbf{Q_M} & \mathbf{0_2} & \ldots & \ldots \\ & & & & & & & \ddots \end{bmatrix}_{N \times N} \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ X_3 \\ \vdots \end{bmatrix}_{N \times 1} = \begin{bmatrix} \begin{bmatrix} u^0_{0,0} \\ u^1_{0,0} \end{bmatrix} \\ \begin{bmatrix} u^0_{0,1} \\ u^1_{0,1} \end{bmatrix} \\ \vdots \end{bmatrix}_{N \times 1} \qquad (46)$$

where the $\mathbf{Q_m}$ matrices are $2 \times 2$ and $\mathbf{0_2}$ is an all-zeros $2 \times 2$ matrix.

#### 3.1.9.4 Approximation Preprocessing

Approximation preprocessing is a special case of matrix prefiltering for which $M = 1$.

For example, for the GHM multifilter bank we have the interpolation prefilter:

$$\mathbf{Q_0} = \begin{bmatrix} 3/8\sqrt{6} & 5/4\sqrt{6} \\ 0 & 0 \end{bmatrix} \qquad \text{and } \mathbf{Q_1} = \begin{bmatrix} 3/8\sqrt{6} & 0 \\ 1/\sqrt{3} & 0 \end{bmatrix}$$

which can be applied to a 1D signal to prepare it for a transform using GHM multiwavelets [55] (see Figure 18).

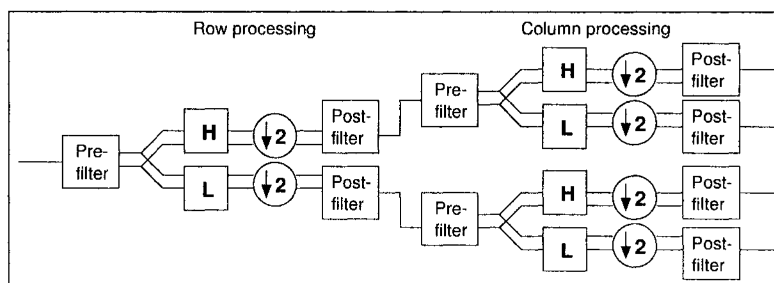



Figure 18: Single-level 2-D MWT using 1-D approximation preprocessing.

#### 3.1.9.5 Two-dimensional proprocessing scheme

The prefiltering schemes discussed so far are called 1D schemes, since they are applied to 1D signals, *i.e.* even in the case of image processing, the scalar-vector conversion is performed on rows or columns (1D) before the matrix filter/subsampling unit. However, there exist 2D prefilters that are applied before the transform process to map the 2D signal to coefficient vectors suitable for direct filtering by the matrix filters (see Figure 19) [12].

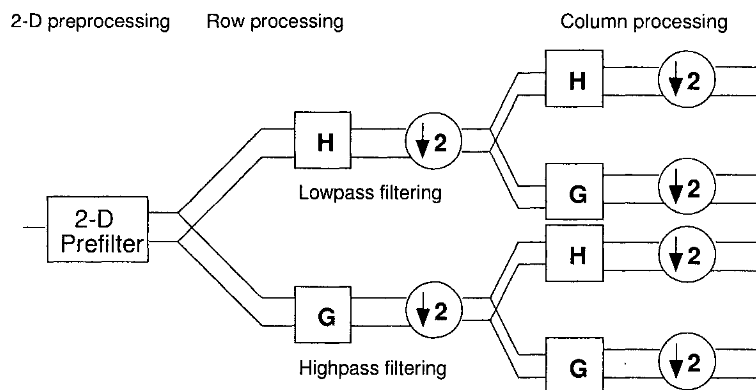



Figure 19: Single-level 2-D MWT using 2-D approximation preprocessing.

## 3.2 Wavelet-Based Image Processing Applications

In this section, we present briefly some typical wavelet-based image processing applications. We will identify the main processing units at work in the applications, at a high level of abstraction (*i.e.* at a semantic level), together with a more detailed overview of some representative algorithms. The main goal in our current context is simply to get a feel for the kind of algorithmic organization at work in wavelet-based image processing applications.

### 3.2.1 Terminology of Digital Image Processing

We will restrict the general definition of an image to the following narrower definition: a *digital image* is a sampled, quantized function of two (or three) dimensions, sampled in an equally spaced grid pattern, and quantized in equal intervals of amplitude. Thus, a digital image is a two or three dimensional array of quantized sampled values.
*Processing* is the act of subjecting something to a process. A *process* is a series of operations leading to a desired result (see [11]).

*Digital image processing* can be defined as subjecting a numerical representation of an object to a series of operations in order to obtain a desired result.

*Digital image processing* starts with one (or many) image(s) and produces a modified version of that image (those images).

### 3.2.2 Wavelet-Based Processing

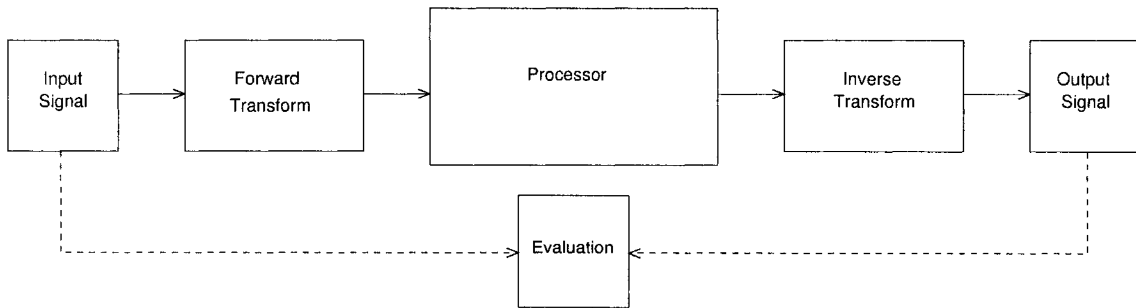



Figure 20: Transform-based image processing.

In order to describe wavelet-based (wavelet transform-based) image processing applications (see Figure 20) it is important to agree on the language used to describe such applications.
Wavelets and wavelet transforms can be presented in at least two languages: mathematics (wavelets and scaling functions) and signal processing (digital filters, filter banks and signals), reflecting the multi-disciplinary origin of wavelet theory and wavelet transforms. We adopt the point of view and language of filters since in many applications, one never has to deal directly with the scaling functions or wavelets and only the filter coefficients are considered. [53]. Let us define some important terms:

- ***Transform***. The *transform* of a signal is a *new representation* of that signal. For an input $x[n]$ the output of a transform is a set of coefficients $b_{jk}$, which express the input in the wavelet basis. The $b_{jk}$ can be found *recursively*. The coefficients $b_{jk}$ in this wavelet expansion are called the *discrete wavelet transform* (DWT) of the signal $f(t)$. The central idea in this recursion is *multiresolution*.

- ***Filter.*** A *filter* is a time-invariant operator. It acts on input vectors $x$. The output vector $y$ is the *convolution* of $x$ with a fixed vector $h$. The vector $h$ contains the filter coefficients $h(0), h(1), h(2), \ldots$.

- ***Filter bank.*** A *filter bank* is a set of filters. The *analysis bank* has often two filters, lowpass and highpass. These filters separate the input signal into frequency bands. When processing is applied to those subbands, we are talking about subband coding or processing in the transform-domain.

- ***Wavelet.*** *Wavelets* are basis functions in continuous time. A basis is a set of linearly independent functions that can be used to produce all admisissible functions $f(t)$. Corresponding to the lowpass filter there is a continuous time scaling function $\varphi(t)$. Corresponding to the highpass filter, there is a wavelet $\omega(t)$.

- ***Tree structured filter bank.*** tree-structured filter bank implements the discrete wavelet transform. This is known as *iterating the filter bank.*

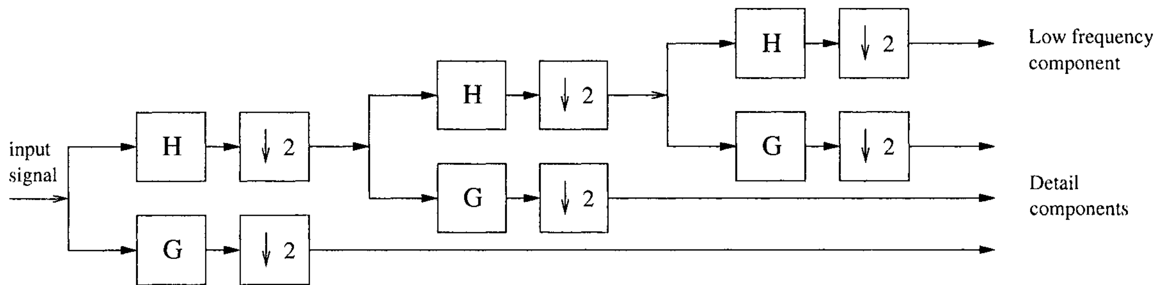



Figure 21: Filter bank implementation of a multilevel (three-stage) multi-band (two-band) analysis tree (descrete wavelet transform) of a 1D signal.

In order to *adapt* or *instantiate* a framework to create an application (activity referred to as a framework *instantiation*), an application developer must have a set of requirements for the target application. *Use cases* are a common technique of capturing, representing and discussing application requirements. Tables 1 and 2 sketch examples of use cases for 2 wavelet-based image processing applications: a multiwavelets image denoising, a complex wavelet image denoising. Other typical applications also briefly presented are: non-linear wavelet image compression, and a 3D medical image fusion.

### 3.2.3 Wavelet Denoising

Typical transform-based denoising consists in performing the three following steps:

- Transform the noisy data into the transform domain (orthogonal or non-orthogonal).

- Apply thresholding to the resulting coefficients.

*Use case: Multiwavelet Image Denoising*

*Actors*

Image, desired multiwavelet, transform, prefiltering scheme, thresholding scheme, performance evaluation routine.

*Pre-conditions*

1. Image available.
2. Wavelet available.

*Post-conditions (objectives)*

1. Image is denoised and statistics are available.
2. Intermediate and final results (sub-images, metrics) are logged to a chosen destination.

*Basic flow*

1. Acquire a noisy (white noise) image.
2. Configure transform.
   - Create filter, a specific multiwavelet: GHM.
   - Set transform type to non-standard (rectangular).
   - Set the depth of the transform.
3. Determine the thresholding scheme to be use: *univariate thresholding.*
4. Perform a wavelet transform on the noisy image.
5. Threshold the transform image.
6. Evaluate performance.
7. Log metrics and save resulting image to desired destination/format.

Table 1: Use case for a multiwavelet image denoising application

*Use case: Complex wavelet denoising*

*Actors*

Image, complex wavelet, transform, thresholding scheme, performance evaluation routine.

*Pre-conditions*

1. Image available.
2. Wavelet available.

*Post-conditions (objectives)*

1. Image is denoised and statistics are available.
2. Intermediate and final results (sub-images, metrics) are logged to a chosen destination.

*Basic flow*

1. Acquire a noisy image (with multiplicative noise).
2. Configure transform.
   - Create filter, a specific *complex wavelet.*
   - Set transform type to *standard* (square).
   - Set the depth of the transform.
   - Select translation invariance algorithm: *average over diagonal shifts.*
3. Determine the thresholding scheme to be use: *elliptic thresholding.*
4. Perform transform on noisy image.
5. Threshold the transform image with selected thresholding scheme.
6. Evaluate performance.
7. Log metrics and save resulting images to desired destination/format.

Table 2: Use case for a complex wavelet denoising application.

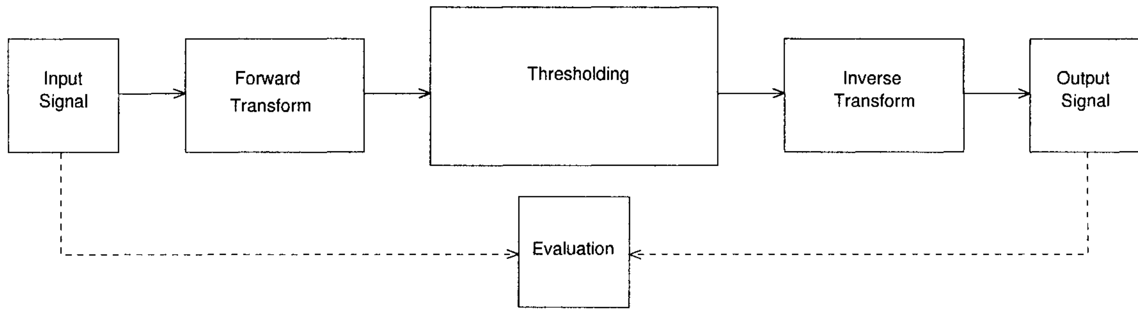



Figure 22: Flowchart for a typical image denoising application.

- Transform back into original domain.

#### 3.2.3.1 Denoising by Thresholding

Once an image has been transformed, some kind of operations are performed on the coefficients in the transformed domain. In the case of image compression as well as in the case of image denoising, this operation consists in shrinking the coefficients. For image compression the goal is to reduce the number of non-zero coefficients to represent the image more compactly. For image denoising, based on the assumption that the noise predominantly lies in the wavelet coefficients at higher resolution level, the goal is to reduce those coefficients to remove the amount of noise.

We briefly present here two types of thresholding (or wavelet coefficient shrinkage): univariate and multivariate. But first, we illustrate two types of "shrinking" that is performed on individual coefficients: hard and soft thresholding.

#### 3.2.3.2 Hard and Soft Thresholding

The most straightforward approach to denoising uses the absolute value of the wavelet coefficient as a regularity measure: a coefficient close to zero contains little information and is *relatively* strongly influenced by noise. When all wavelet coefficients having an absolute value below a certain threshold $\delta$(and hence classified as "noisy"), they are replaced by zero. The coefficients with absolute value above the threshold are kept intact. This is *hard thresholding.* (see Fig. (23)(b)) The coefficients above the same threshold $\delta$ are shrunk by the value of the threshold. This is called *soft thresholding.*(see Fig. (23)(a))

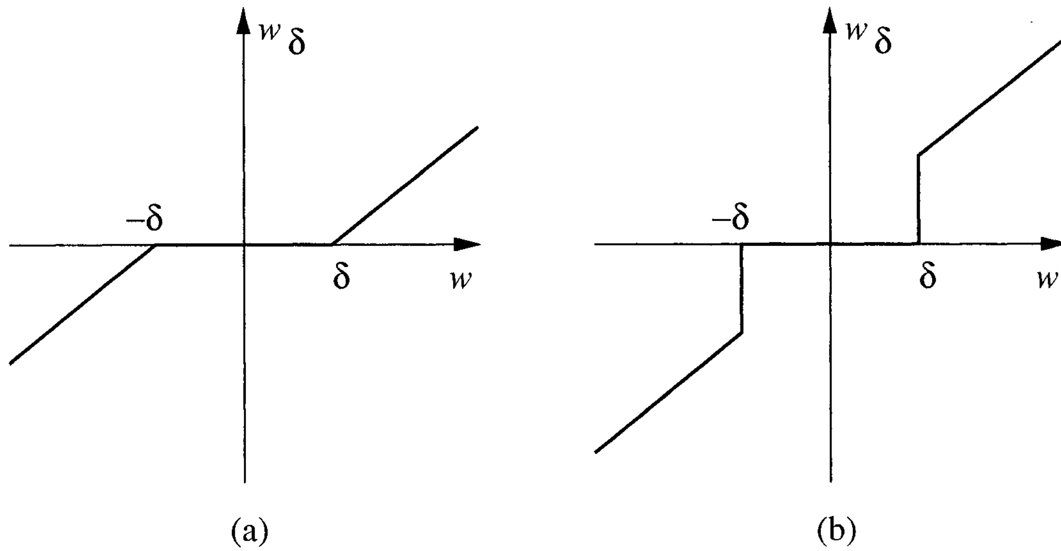



Figure 23: Soft (a) and hard (b) thresholding.

#### 3.2.3.3 Univariate Thresholding

The simplest wavelet coefficient method (Visu-Shrink) as it applies to N data points (1-D) signal consist of 4 steps:

1. Perform a N-level forward transform on the signal
2. Estimate the noise variance by calculating the standard variation $\sigma$ of the wavelet coefficients at highest resolution level.
3. Set a threshold level $T = \sqrt{2log(N)}\sigma$ and apply a soft thresholding ($w_k^j \rightarrow sgn(w_k^j)(|w_k^j|-T)_+$) or hard thresholding ($w_k^j \rightarrow 0 \;\; \text{for} \;\; (|w_k^j| <= T)$) on all wavelet coefficients uniformly (universal threshold).
4. Perform an inverse wavelet transform.

#### 3.2.3.4 Multivariate (or Vector) Thresholding: Elliptic Thresholding

The essential difference between univariate thresholding and multivariate (or vector) thresholding is that the latter considers (multi or complex) wavelets coefficients vectors as a whole rather that thresholding individual elements.

The algorithm presented above can also be applied to noisy images encoded with a 2-D wavelet transform. The difference here is that the coefficients are collected in three different spectral bands (subbands) corresponding to the vertical, horizontal and diagonal directions (VV, WV and WW blocks).

Since the noise characteristics can differ in each of these blocks, they can be processed separately. For instance, the noise characteristics in the WV subband can be used for thresholding all the lower resolution WV blocks (and similarly for WW and VW), given that the transform is a non-standard (square) transform (see Figure 24).

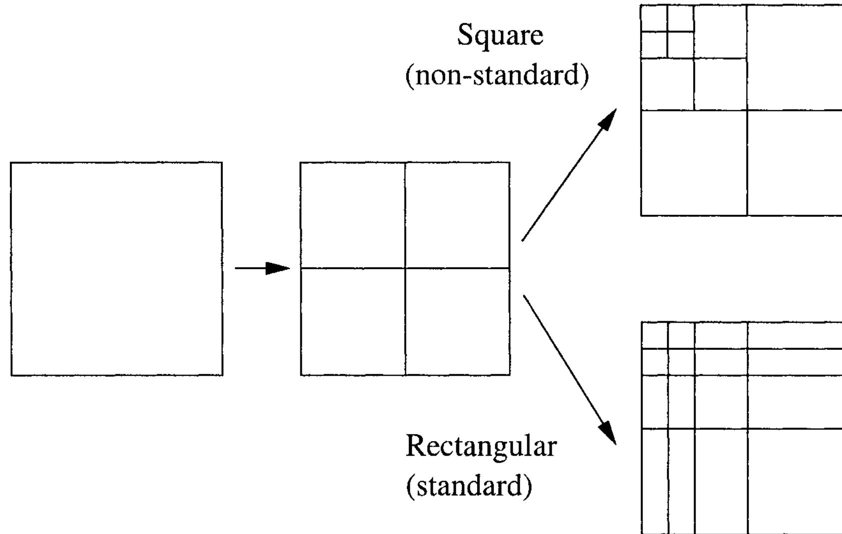



Figure 24: Standard and non-standard decompositions.

In the case of complex wavelet coefficients, it has been observed that the coefficients follow a bi-normal distribution of mean $(\mu_x, \mu_y)$ with covariance matrix $R$ which is not diagonal (meaning that the real and imaginary parts are correlated. The term "elliptic" thresholding comes from the fact that thresholding of the complex coefficients should be performed with respect to the principal axes of the 2-D distribution and angle-dependent (within an area that extends in proportion to the eccentricity of the centered dispersion ellipse (thus preserving the elliptic shape of the distribution). The algorithm proceeds as follows [22, 23]:

- Perform a N-level wavelet transform.
- For each subband type (VW, WV, WW) and level $j$ ($j = 1, 2, \ldots, N$), perform the following:
- Calculate the mean $(\mu_x, \mu_y)$ and the independent element $r_{11}$, $r_{12}$ and $r_{21}$ of the covariance matrix $R$.
- Calculate the orientation angle $\alpha$ of the main axis $\xi$ using

$$\alpha = 1/2 \arctan \frac{2r_{12}}{r_{11} - r_{22}}$$

.

- Evaluate the standard deviations $\sigma_\xi$ and $\sigma_\eta$ using

$$\sigma_\xi = \sqrt{r_{11} \cos^2 \alpha + r_{12} \sin 2\alpha + r_{22} \sin^2 \alpha}$$

and

$$\sigma_\eta = \sqrt{r_{11}\sin^2\alpha + r_{12}\sin 2\alpha + r_{22}\cos^2\alpha}$$

- Apply the elliptical thresholding rule $|w_k| \to$ if $\xi^2/t_\xi^2 + \eta^2/t_\eta^2 <= 1.0$ and $|w_k| \to |w_k| - T(\Theta)$ if $\xi^2/t_\xi^2 + \eta^2/t_\eta^2 > 1.0$ where $T(\Theta) = \frac{t_\xi t_\eta}{\sqrt{(t_\xi \sin\Theta)^2 + (t_\eta \cos\Theta)^2}}$ and $t_\eta = \delta\sigma_\xi^1, t_\eta = t_\xi \sigma_\eta/\sigma_\xi$
- Perform an inverse wavelet transform.

### 3.2.4 Wavelet-Based Image Compression

Wavelet coders are all derived from the transform coder paradigm. According to that paradigm, there are three basic *components* that underly wavelet coders:

- A decorrelating function,
- A quantization procedure,
- An entropy coding procedure

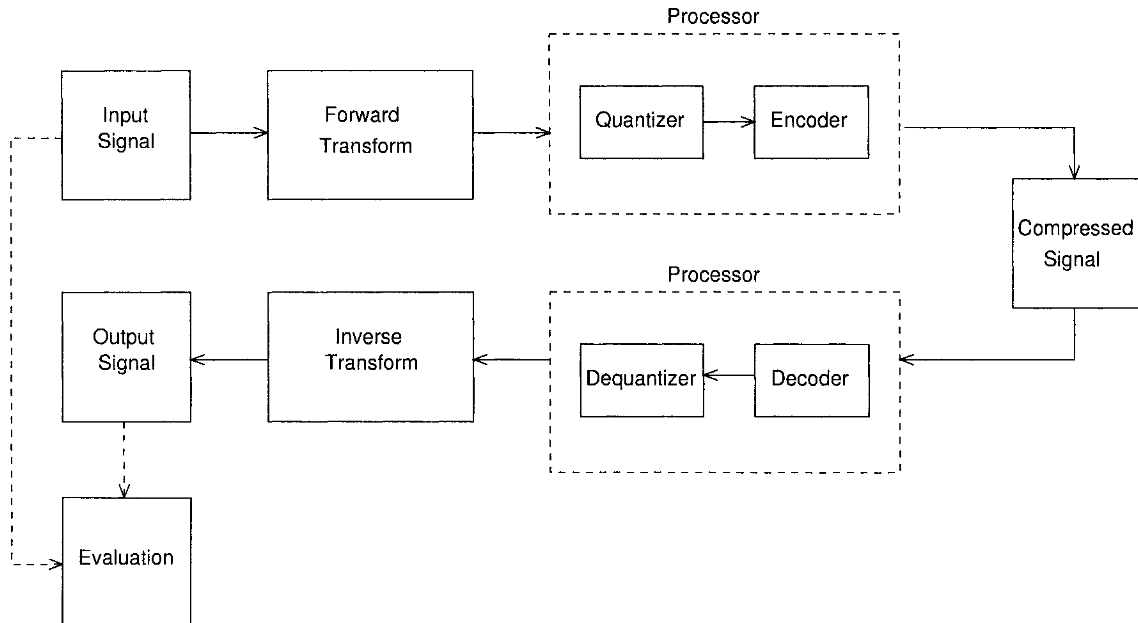



Figure 25: Flowchart for a typical image compression application.

We apply thresholding only to the wavelet coefficients (*i.e.* to the coefficients representing the details in the image), since wavelets form what is called an unconditional basis, and the wavelet expansion of a signal have coefficients that drop off very rapidly. Thus, the signal can be represented efficiently with only a small number of them.

#### 3.2.4.1 Wavelet Basis

The success of transform coding depends on how well the basis functions of the transform represent the features of the signal. A property of wavelets that makes them useful for image compression is energy compaction *i.e.* the ability the wavelet transform has to represent an image (viewed as a matrix of coefficient) with an acceptable level of accuracy using only a fraction of the original data. Wavelets have better energy compaction than the Fourier transform which makes them suited for the compression of signals with discontinuities.

#### 3.2.4.2 Quantization

A useful view of quantizers is to see them as concatenation of two mappings. An *encoder*, which takes partitions of the $x-$axis to the set of integers $-2, -1, 0, 1, 2$, and a *decoder* which takes integers to a set of values $\hat{x}_k$.

### 3.2.5 3D Medical Image Fusion

Image fusion can be defined as the process by which several images, or some of their features, are combined together to form a single image [40]. In many applications, such as medical imaging, there is a growing need for 3-D image fusion algorithms capable of combining multimodality or multisource images. An example from the 3-D medical imaging domain is the fusion of Magnetic Resonance (MR) and Computed Tomography (CT) images or fusion of MR and ultrasound (US) images.

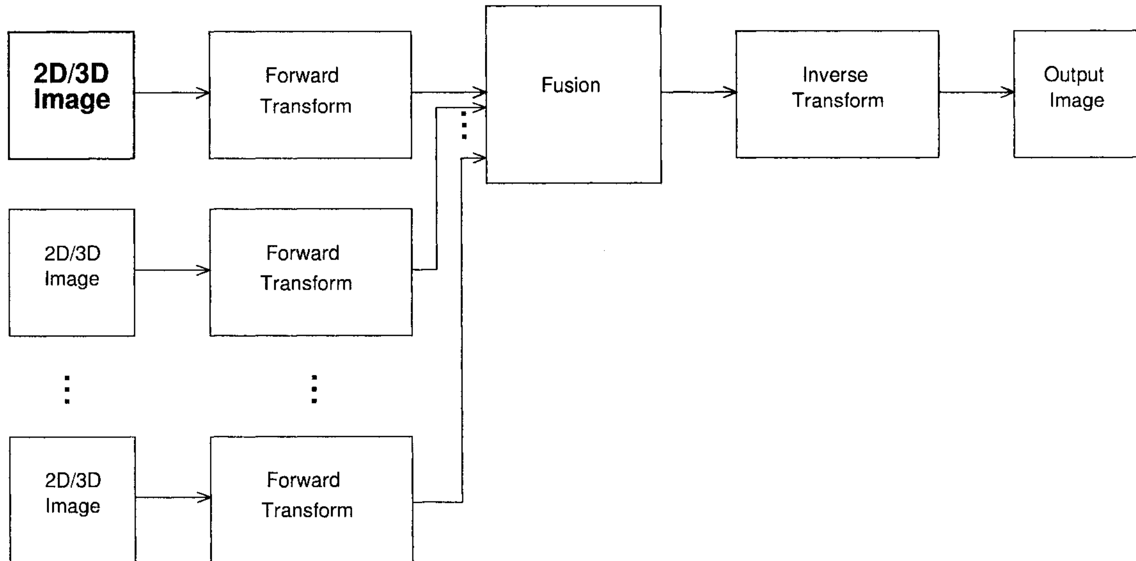



Figure 26: Flowchart for a typical image fusion application.

According to [1], image fusion can be performed at different levels of the information representation: signal, pixel, feature and symbolic levels. Here we are interested in pixel level fusion.

Pixel fusion algorithms vary from very simple to very complex. Simple schemes include image averaging for example. More complex schemes include principal component analysis and wavelet transform fusion.

### 3.2.6 3-D Image Fusion

The general idea (the high-level algorithm) behind wavelet-based image fusion is the following:

- The wavelet transform $\omega$ of $N$ input images $I_1(x, y, z), I_2(x, y, z), \ldots, I_N(x, y, z)$ (usually $N = 2$) are computed.

- These transforms are combined using some kind of *fusion rule* $\varphi$.
- The inverse wavelet transform $\omega^{-1}$ is computed and the fused image is reconstructed.

The process can be expressed as follows:

$$
\begin{aligned}
I(x,y,z) &= \omega^{-1}(\varphi(\omega(I_1(x,y,z))), \omega(I_2(x,y,z)), \ldots, \omega(I_N(x,y,z))) && (47) \\
&= \omega^{-1}(\varphi(W_1(x,y,z))), W_2(x,y,z)), \ldots, W_N(x,y,z))) && (48)
\end{aligned}
$$

## 3.3 Conclusion

This concludes our brief overview of wavelets, wavelet transforms and applications. This chapter was meant to reflect a fraction of the domain analysis activity that was performed prior to the development of the framework. We are now ready to embark on the design of the framework outlined in part II of the thesis.

# Part II

# Frameworks Design

# Wave ImAgE Framelets

The framelet construct was intoduced in Part I. The initial domain analysis led to the identification of a number of variation points. These variation points were then partitioned into three main framelets, each containing a subset of the variation points.

The adaptability and flexibility requirements intrinsic to the variation points represent design problems for which candidate design solutions were developed in each framelet.

Since, at design level, the framework is composed of design patterns and variation points, each framelet is composed of a subset of the framework variation points and related design patterns. The following chapters (one per framelet) contain descriptions of the design patterns and variation-points for each framelet.
The Wave ImAgE framework is made up of the following framelets:

- *The Data Processing Framework:*
  This framelet proposes a design solution to the problem of developing general data processing applications.
  This framelet is covered in Chapter 4.
- *The Wavelet Transform Framework:*
  This framelet proposes a design solution to the problem of performing wavelet transforms on data signals.
  This framelet is covered in Chapter 5.
- *The Configuration and Experiment Scheduling Framework:*
  This framelet proposes a design solution to the problem of flexibly configuring image processing experiments with image processors and to schedule the execution of these experiments.
  This framelet is covered in Chapter 6.

# Chapter 4

# Data Processing Framelet

## 4.1 Overview

This chapter describes the data processing framelet for the Wave ImAgE framework. The framework is described at both the framelet design level and at the framelet architectural level.

This framework (or the proposed frameworks) proposes candidate design solutions to the problem of developing data processing applications, which are applications performing processing steps on data signals.
In this chapter we provide outlines of candidate solutions for that category of applications featuring *processors*, as building blocks or units of processing functionality, *operating on* data.

The idea is to offer a way to build complex image processing tasks (wavelet transforms and wavelet-domain image processing algorithms) that can be performed in a sequential way and configured by the user at various levels of abstraction (from the details of fine grained low-level functionality to high level tasks).

## 4.2 Context

The context for the design of the framework is described in Chapter 3, Wavelets, Wavelet Transforms and Image Processing Applications. This chapter assumes that the reader is familiar with Design

Patterns [24] and more specifically, with the VISITOR family of design patterns[1].

## 4.3 Problem

The design solution proposed should be influenced by the following *forces*:

- The number and type of *operations* applied on the data will most likely proliferate as more applications are developed. The framework *should* allow for the development of an open-ended number of processing functionality without having to modify the data. Basically, a primary challenge when designing the Data interface is to come up with a minimal set of operations that lets clients build open-ended functionality. Performing "surgery" on Data and its subclasses for *each* new capability is both invasive and error-prone. The risk is that the Data interface evolving as a hodgepodge of operations eventually obscuring the essential properties of Data objects. When that happens, the classes just get harder to understand, to extend and to use.
- Exchange of processing units *should* easily be done at run-time.
- The framework *should* allow for the occasional addition of new kinds of data to be treated as they should without modification to the framework.
- The framework *should* allow for the composition of complex processing tasks from smaller processing units (building blocks).
- The framework *should* be able to capture the relationships and collaborations between some abstract processors, in order to alleviate the responsibilities of application developers.
- The data to be processed by different applications could be of different dimensionality (1D, 2D and 3D) and represent different types of objects (images, geometries, etc.). The framework must instantiate classes and define the collaborations between those classes but it only knows about abstract classes (the applications know about their application-specific classes). The data *should* be represented in the framework in its most generic form.
- As the domain analysis conducted in earlier chapters revealed, the wavelet-based image processing domain can be modeled at a high level of abstraction using two kinds of basic abstract objects. Objects that that are *processed*, or *operated on*, and *processing* objects (called processors), operating on the first kind of objects. From a user's perspective, these more passive objects are signals, images, transforms (as transform-domain representations of data objects that were transformed) and geometries. We want the framework to be able to treat those objects uniformly (*i.e.* without concern for their application-specific features), *i.e.* as objects

[1]The patterns in this family are: conventional VISITOR, ACYCLIC VISITOR, STAGGERED VISITOR, DECORATOR, and EXTENSION OBJECT [37]

that are *processable* and *processed.* At that level at least, it could mean that those objects should have a common interface. It could also mean that the classes representing those objects be derived from a common (abstract) base class (or using some other means to obtain genericity, as it is discussed later in this chapter). How can such a base class look like, and what should its interface contain? The difficulty of this question stems from the *diversity* of the objects belonging to that class. This diversity can be expressed along three main *axes of variability*:

1. *data type* for the elements contained within the data objects (which are some sort of *containers* holding data elements: integers, real, complex numbers, matrices, etc...),
2. *dimensionality* of the data (1D, 2D or 3D),
3. *data structure* represented (signal or geometry).

These issues will be discussed in the next section below.

## 4.4 Solutions

The candidate solutions proposed in this chapter are all based on the VISITOR family of design patterns.
The patterns in this family are:

- VISITOR
- ACYCLIC VISITOR
- STAGGERED VISITOR
- DECORATOR

What those patterns have in common is that they allow new operations to be added to existing hierarchies without modifying the hierarchies.

In this section we will thus review the VISITOR pattern (and the kinds of problems it's good at solving), motivate our use of this family of patterns, and finally present candidate solutions, which are based on different implementations of the VISITOR pattern.
Our intent is not to present a definitive unique solution, but rather to articulate and appreciate the trade-offs of the various solutions considered.

### 4.4.1 The Visitor Design Pattern

The intent section of the VISITOR design pattern says:
VISITOR

> Represent an operation to be performed on the elements of an object structure. VISITOR lets you define a new operation without changing the classes of the elements on which it operates [24].

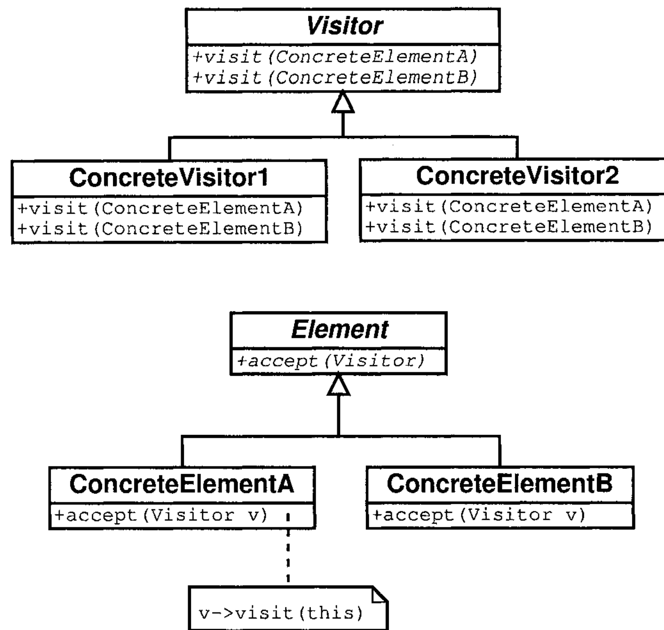



Figure 27: VISITOR design pattern [24].

In light of the *forces* identified in the previous section, let us look at what VISITOR has to offer.

1. It contains two class hierarchies. One for passive (visited) elements operated on, and one for visitor classes containing the implementations for these operations to perform.

2. It allows adding new operations to a hierarchy of class without modifying a class hierarchy.

3. It provides a way to reduce the number of classes to implement $m \times n$ behaviors (this is called the $m \times n$ problem for $m$ element classes and $n$ types of behavior on those elements). For that problem, double-dispatch probably offers the best solution. And VISITOR offers one way to implement double-dispatch.

4. It provides a way to recover lost type information from an instance without resorting to dynamic cast. This relies on *double-dispatch* to execute the correct application-specific implementation.

The third point is not really relevant in our case since in the context of this framework, we are not really interested in reducing the number of classes containing an implementation: these are provided by the applications.

This approach has the following advantages:

- *Adding a new processing algorithm becomes easy.* To add a new operation does not require opening the Data hierarchy interface to perform surgery on it but it simply requires adding a derivative in the processor hierachy.
- *Related operations are gathered together while unrelated ones are separated.* The set of operations implementing the same kind of processing *could be* grouped together in the same class since each algorithm (visitor) has its own class. Also, algorithm-specific data structures can be hidden in the visitor [24].
- *The class defining the data are much simpler and cleaner.* The data classes only have to define an `accept(Visitor&)` method (or `apply(Processor&)` in our case), no matter how many different transform algorithms there are.

### 4.4.2 Visitors in Frameworks

In the context of framework development (see Figure 28), it would be desirable if the framework-provided abstract classes `Visitor1` and `Visitor2` were base classes for two different kinds of algorithms (1 and 2) that applications would subclass. However, for those base classes to be more useful to application developers, there should be either some sort of collaboration between `Visitor1` and `Visitor2` (*i.e.* an abstract coupling capturing the collaboration between different algorithms) and/or the addtion of template methods capturing common behaviors across applications. A prob-

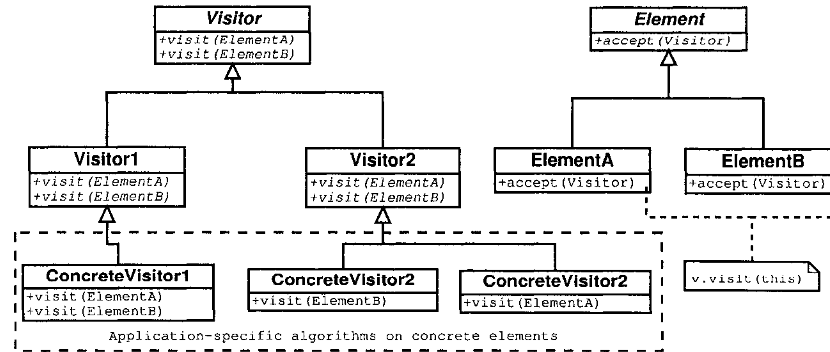



Figure 28: VISITOR hierarchy in a framework context.

lem with such a VISITOR-based approach in the context of framework development is that the visitor

base class contains references to application-specific subclasses. In order to get rid of this problem, we will look at two alternative implementations of the VISITOR pattern:

- The STAGGERED VISITOR (see [67] and [63])
- The ACYCLIC VISITOR (see [36], [37] and [3])

We will introduce those implementations in the next sections.

### 4.4.3 The Staggered Visitor

The problem with the VISITOR pattern as we know it is that the Visitor base class which is a framework class, contains references to *concrete (i.e. application-specific) subclasses*:

```
class Visitor {
   //...
   virtual void visit(ConcreteElementA *);
   virtual void visit(ConcreteElementB *);
   // etc. for all concrete element classes
};
```

A framework interface should not refer to concrete subclasses. Since application-specific subclasses will be written long after the framework classes interfaces are defined, a framework class depending on subclasses will need to be changed to parallel the application developments. This is just not possible for a framework. What is needed is a *place* to put new Element subclasses as they arise, without changing the Visitor interface [63]. The idea is to commit to an application-specific Visitor interface *as deeply as possible* in the Visitor hierarchy. An implementation called the STAGGERED VISITOR was proposed to achieve that ([63], see Figure 29).

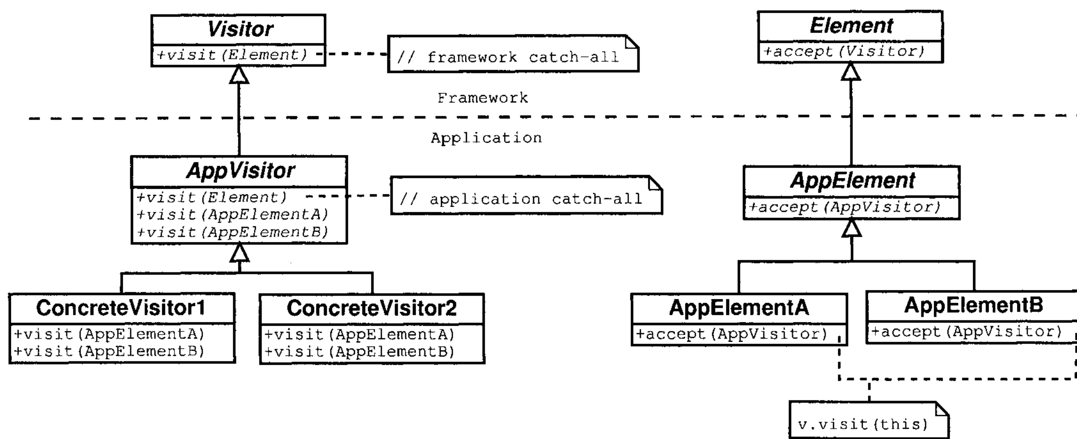



Figure 29: The STAGGERED VISITOR design pattern.

The two crucial things that are done to the conventional VISITOR implementation are:

1. Move the application-specific visitor operations out of the base class.

2. New AppVisitor and AppElement abstract classes bridge the gap between application-specific VISITOR classes and the framework's side of the pattern.

This is achieve through a clever use of catch-all methods as we will see below (see also Figure 30).

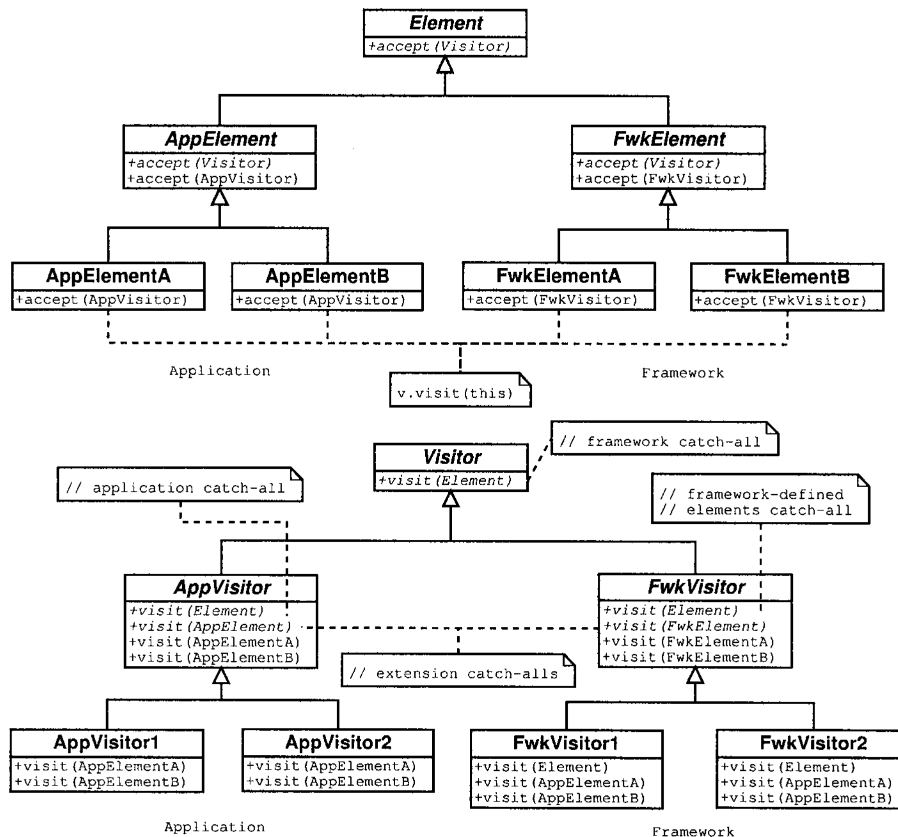



Figure 30: STAGGERED VISITOR with Framework (pre-)defined Elements and Visitors

#### 4.4.3.1 Staggered Visitor Commented Code Sample

We now present a simple commented implementation of the STAGGERED VISITOR. First we have the framework core Visitor abstraction. Notice that no mention of application-specific subclasses is made. The framework only knows about abstract Element, as it should:

```
class Visitor {
public:
   virtual void visit(Element *e) { // Framework catch-all
      // Empty implementation
   }
};
```

Then the framework core Element abstraction defining the `accept()` method:

```
class Element {
public:
   virtual void accept(Visitor &v) {
      v.visit(this);
   }
};
```

A base class for application-specific visitors on application-specific elements is defined.

```
class AppVisitor : public Visitor {
public:
   virtual void visit(Element *e);       // Application catch-all
   virtual void visit(AppElement *appe); // Extension catch-all
   // visit for new application-specific elements
   virtual void visit(AppElementA *e);
   virtual void visit(AppElementB *e);
protected:
   void defaultVisit(AppElement *appe);
};
```

where the application catch-all relies on a dynamic cast to "catch" the application-defined elements:

```
void
AppVisitor::visit(Element *e) {
   AppElement *ae;
   if (ae = dynamic_cast<AppElement *>(e)) {
      ae->accept(*this);
   }
}
```

The trick with this application catch-all method is to defer the commitment to an application-specific Visitor interface until it's really needed, *i.e.* in the `AppElement` sub-hierarchy. The catch-all achieves

this by checking the type of the element argument (passed as an `Element *`)) to see if it is suited for its application-specific Visitor interface. If it is the case then the `AppVisitor`'s interface is put into use, starting from the `AppElement::accept(AppVisitor&)` operation [67]:

```
class AppElement : public Element {
public:
   virtual void accept(Visitor &v) {
      v.visit(this);
   }
   virtual void accept(AppVisitor &appv) {
      appv.visit(this);
   }
};
```

Application-specific concrete elements are defined in the usual visitor way:

```
class AppElementA : public AppElement {
public:
   virtual void accept(AppVisitor &v) {
      v.visit(this);
   }
};
```

In order for new application-specific elements (not present in the visitor interface) to be handled without the need to modify the interface of the `AppVisitor`, an *extension catch-all* is introduced:

```
void
AppVisitor::visit(AppElement *appe) {
   defaultVisit(appe);
}
void
AppVisitor::defaultVisit(AppElement *appe) {
   // Implementation
}
// visit operation for known application-specific elements
void
AppVisitor::visit(AppElementA *e) {
   // Implementation
}
// Similarly for other known concrete elements...
```

Then newly introduced elements like:

```
// Application-specific concrete element
class NewlyDerivedAppElement : public AppElement {
public:
  virtual void accept(AppVisitor &v) {
     v.visit(this);
  }
};
```

will be handled, *at least* with a general default behavior. Concrete visitors are simply overriding their base class default visit operation:

```
class AppVisitor1 : public AppVisitor {
public:
   virtual void visit(AppElementA *e) {
      // Implementation
   }
   virtual void visit(AppElementB *e) {
     // Implementation
   }
};
```

Finally, the whole thing is put into motion in the VISITOR way. In a mainline, it looks like this:

```
int main() {
   // Framework knows only about base types
   Visitor *v;
   Element *e;

   // Creation of application-specific visitor and element
   v = new AppVisitor1;
   e = new AppElementA;

   // Trigger the execution
   e->accept(*v);
};
```

Now, let us look at another approach to implement the VISITOR, which is also more suitable than the original VISITOR in a framework context. It is called the ACYCLIC VISITOR.

### 4.4.4 The Acyclic Visitor

The ACYCLIC VISITOR was developed to overcome the problem of cyclic dependency inherent to the original GOF visitor pattern [36, 3]. In the context of framework however, we find it to be helpful for three other reasons:

1. Its implementation is simpler, cleaner, shorter than the STAGGERED VISITOR.
2. Like the STAGGERED VISITOR and unlike the conventional VISITOR, no mention of application-specific subclasses is made in the main visitor base class (since it is degenerate).
3. It can simplify the application developer's responsibilities by providing small visiting classes (as we will see below) which make it clear which methods application-specific concrete subclasses have to implement.

The ACYCLIC VISITOR is similar to the STAGGERED VISITOR except that dynamic casting occurs in the `accept` method rather than the `visit` operation[2].

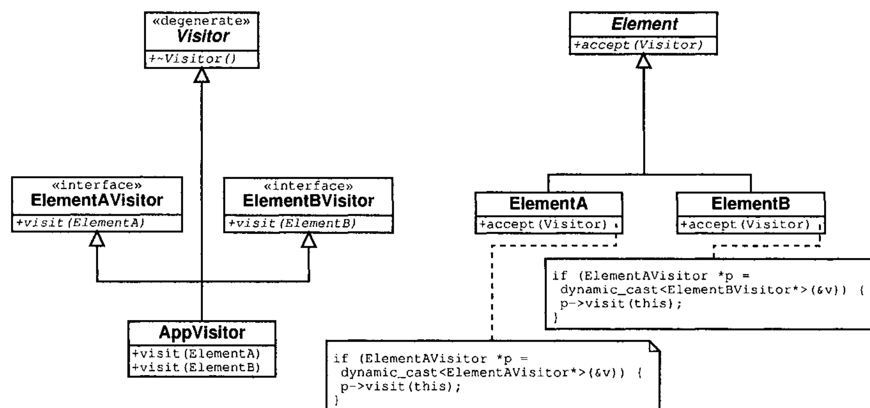



Figure 31: The ACYCLIC VISITOR design pattern.

#### 4.4.4.1 Acyclic Visitor Commented Code Sample

Below is a commented example of the simplest expression of the ACYCLIC VISITOR pattern. The example given here is based on the one on [3].

[2]As is pointed out in [63], there are some undesirable consequences on maintainability when putting the dynamic casting in the `accept` method. The idea is that if, over time, a number of new element subclasses have been defined and we want to update the framework and add corresponding `visit` operations in the `Visitor` base class, then both hierarchies have to be modified: each `accept` in the Element hierarchy must be changed (to remove the casting) together with the base `Visitor` (to add the new `visit` method). And avoiding modification to the Element hierarchy was our main motivation to use VISITOR in the first place. However, this criticism applies to the location of the dynamic casting in a "standard" visitor, but not to the ACYCLIC VISITOR since in this pattern, *a)* the visitor base class is degenerate, and *b)* the dynamic casting remains in the `accept` methods, where it belongs

First, an archetypal base visitor class (a degenerate class) is declared in a class library or framework as a carrier of type information:

```
class Processor {
public:
   // A virtual destructor giving RTTI capabilities being virtual and
   // ensuring correct polymorphic destruction.
   virtual ~Processor() {}
};
```

The base class for the visited hierarchy (also in a library or framework) declares the `accept` method:

```
class Data {
public:
   virtual void accept(Processor &v) = 0;
};
```

And the concrete elements implement the dispatching:

```
class Data1D : public Data {
public:
   virtual void accept(Processor &v) {
      if (Data1DProcessor *p = dynamic_cast<Data1DProcessor *>(&v)) {
         p->visit(*this);
   } else {
      // optionally call a catch-all function
      cout << "ERROR: Other processor with data 1D" << endl;
   }
};
class Data2D : public Data {
public:
   virtual void accept(Processor &v){
      if (Data2DProcessor *p = dynamic_cast<Data2DProcessor *>(&v)) {
         p->visit(*this);
      } else {
         // optionally call a catch-all function
         cout << "ERROR: Other processor with data 2D" << endl;
      }
};
```

A difference with the STAGGERED VISITOR is that dynamic cast occurs in the data hierarchy instead as in the visitor hierarchy.
Also, small visiting classes for each derived class in the visited hierarchy are declaring the appropriate `visit` method:

```
class Data1DProcessor {
public:
   virtual void visit(Data1D &e) = 0;
};
class Data2DProcessor {
public:
   virtual void visit(Data2D &e) = 0;
};
```

to be implemented in a mixin style by the concrete visitors deriving from `Processor` archetypal base class and visiting classes of interest (for the ones that it wants to visit):

```
class ConcreteProcessor1 : public Processor,         // Required
                           public Data1DProcessor { // To visit Data1D
public:
   void visit(Data1D &e) {
      cout << "ConcreteProcessor1::visit(Data1D &)" << endl;
   }
};
class ConcreteProcessor2 : public Processor, // Required
                           public Data2DProcessor { // To visit Data2D
public:
   void visit(Data2D &e) {
      cout << "ConcreteProcessor2::visit(Data2D &)" << endl;
   }
};
```

Finally, the whole thing is put into motion in the usual visitor way. In a mainline, it looks like this:

```
int main() {
   Data *e;
   Processor *v;
   e = new Data1D;
   v = new ConcreteProcessor1;
   e->accept(*v);
```

```
    e = new Data2D;
    v = new ConcreteProcessor1;
    e->accept(*v);

    e = new Data1D;
    v = new ConcreteProcessor2;
    e->accept(*v);

    e = new Data2D;
    v = new ConcreteProcessor2;
    e->accept(*v);
}
```

The output of this program is:

```
ConcreteProcessor1::visit(Data1D &)
ERROR: Other processor with data 2D
ERROR: Other processor with data 1D
ConcreteProcessor2::visit(Data2D &)
```

And we see double-dispatch at work!

### 4.4.5 A Staggered Visitor-Based Framework Candidate Solution

Patterns involved in this candidate solution include DECORATOR, COMPOSITE and STAGGERED VISITOR.
(See Figure 32). The participants are[3]:

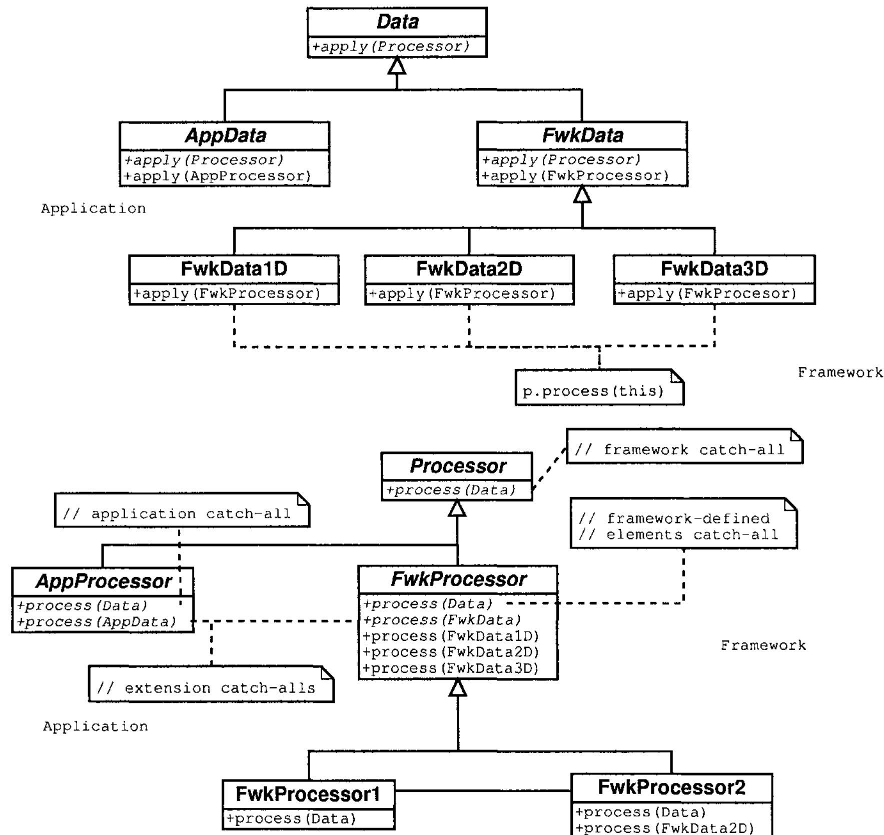



Figure 32: STAGGERED VISITOR-based Data Processing Framework

- VISITOR-**Visitor** (Processor)
  - Declares a `process` operation the base data class (only).
- VISITOR-**Visitor** (AppProcessor, FwkProcessor)
  - Declare a `process` operation for each class of concrete data it processes (or "operates on", i.e. `process(Data1D &)`, `process(Data2D &)`, `process(Data3D &)`).

[3]To identify participants in a (compound) pattern, we are using the following notation inspired by UML-F Profile notation and the "pattern:role annotation" of Erich Gamma: for each design pattern a participant belongs to, there is a pair of the form DESIGNPATTERN-**Participant/Role**, where the first term (in small capitals) indicates the design patterns in which the class participates, and the second term indicates the participant itself (or role). The class name in parenthesis following the pair is the name of the class playing the role in a specific instance of the pattern.

- VISITOR-**Element** (Data)
  - Defines an `apply(Processor &)` operation that takes a processor as an argument.
  - Declares the interface for objects in the object structure.
- VISITOR-**ConcreteElement** (FwkData, AppData, Data1D, Data2D, Data3D)
  - Represents a concrete data object in the object structure.
- VISITOR-**ConcreteVisitor** (FwkProcessor1, FwkProcessor2, AppProcessor1, AppProcessor2, etc...)
  - Implements each operation declared by Visitor (Processor). Each operation implements an algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm.
  - Manipulates objects in the object structure (*i.e.* data).

Not shown here are the roles that can be played by concrete visitors (this will be described in the next chapter) which are:

- Decorator (or wrapper), implementing the DECORATOR design pattern.
- Composite, implementing the COMPOSITE design pattern.

- VISITOR-**ConcreteVisitor**, DECORATOR-**Component**
  - Defines an interface for objects that can have responsibilities dynamically added.
- VISITOR-**ConcreteVisitor**, COMPOSITE-**Component**
  - Declares an interface for objects in the composition.

| DATA PROCESSING FRAMELET |
|---|
| ***Design Patterns*** |
| STAGGERED VISITOR [24]. |
| ***Framelet Interfaces and Abstract Base Classes*** |
| Abstract base classes:<br>`Data`: Abstract base class for data to be transformed and processed.<br>`Processor`: Abstract base class for data processors. |
| ***Framelet Core Components*** |
| `FwkData`, `AppData`, `FwkProcessor`, `AppProcessor`. |
| ***Framelet Default Components*** |
| `FwkData1D`, `FwkData2D`, `FwkData3D` |

### 4.4.6 An Acyclic Visitor-Based Framework Candidate Solution

The participants in this candidate solution are:

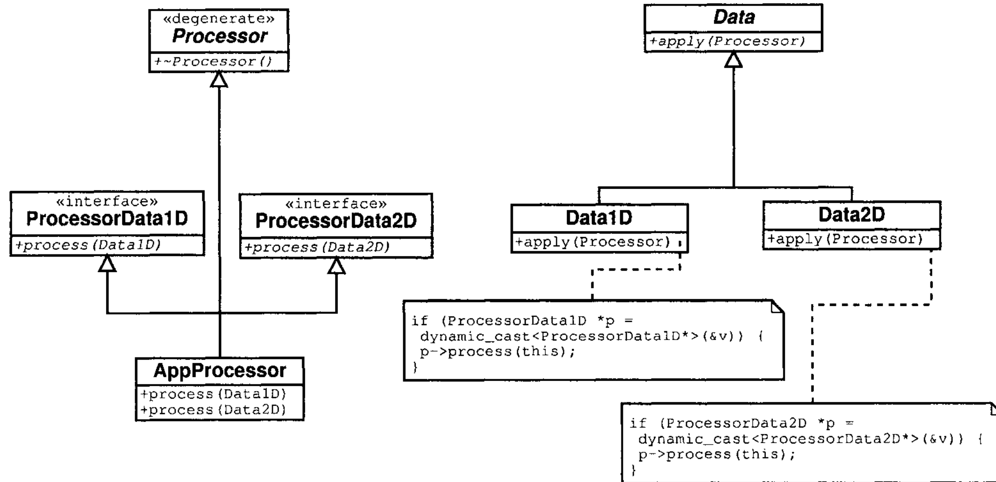



Figure 33: The Wave ImAgE ACYCLIC VISITOR-based Data Processor Framework.

- VISITOR-**Visitor** (Processor, ProcessorDatalD, ProcessorData2D, ProcessorData3D)
  - Declares a degenerate interface with a virtual destructor just to act as a type carrier (`Processor`).
  - Declares interfaces with a `process` operation for each type of concrete data (`ProcessorData1D`, `ProcessorData2D`, etc.).
- VISITOR-**Element** (Data)
  - Declares an `apply(Processor &)` operation that takes a processor as an argument.
- VISITOR-**ConcreteElement** (DatalD, Data2D, Data3D)
  - Represents a concrete object in the data hierarchy (DatalD, Data2D, Data3D).
- VISITOR-**ConcreteVisitor** (AppProcessor)
  - Implements each operation declared by small visitor classes (`ProcessorData1D`, etc.). Each operation implements a fragment of the algorithm defined for the corresponding class of object in the structure. ConcreteVisitor provides the context for the algorithm.
  - Manipulates objects in the composition.

| DATA PROCESSING FRAMELET |
| --- |
| ***Design Patterns*** |
| ACYCLIC VISITOR.[3] |
| ***Framelet Interfaces and Abstract Base Classes***<br>Abstract base classes:<br>`Data`: Base class for data hierarchy.<br>Interfaces:<br>`ProcessorData1D`, `ProcessorData2D`, `ProcessorData3D`.<br>`Processor`: Degenerate interface. |
| ***Framelet Core Components*** |
| `Data1D`, `Data1D`, `Data1D`. |

## 4.5 Liabilities of a Visitor-based Approach.

Encapsulating requests into visitor objects as prescribed by the VISITOR pattern has some drawbacks when compared with having a design offering richer data class interfaces:

- It reduces the *expressivity* of applications.
- The *granularity of change* is at the *whole algorithm* level since implementations are to be found in the `visit` operation of concrete visitor.

These two points will be discussed below.

#### 4.5.0.1 Expressiveness of Applications

Consider for example the following application class:

```
class Application {
public:
    void run(Data *data, Processor *processor) {
        data->apply(processor);
    }
};
```

Clients of that class can use any processor on any data. If we are looking only at the `Application` class to figure out what can be done to the data, we are set for major disappointment: it does not

tell us much on what can actually be done except for the fact that some data is accepting to apply some processing on itself [4]. If we compare this with the application below:

```
class Application {
public:
    virtual void run(Data *data) {
        data->add_noise();
        data->forward_transform(wavelet);
        data->threshold(thr);
        data->inverse_transform();
        data->compare(data2);
    }
};
```

An application using a data class having such a *rich* interface is obviously more informative on what is actually done in this aplication. At least, application developers used to procedural (or non-frameworked application development) feel more comfortable to see such a flow of control or sequence of steps at this place (*i.e.* in the `Application` class implementation). The price to pay at this level is that an application is not flexible (how can we change the sequence of steps) not easily extendable (adding a new operation on data, say `compress` would require modifying the data hierarchy *and* recompiling the application. It is a big price to pay to feel more comfortable, especially knowing that in the first application, processors could be configured (hypothetically), at a proper place, like this:

```
processor->add(new NoiseAdder);
processor->add(new FwdTransform(wavelet));
processor->add(new Denoising);
processor->add(new InvTransform(wavelet));
processor->add(new Evaluation);
```

which is as expressive as what we have in the `Application::run()` method above. This question of evaluating the tradeoffs between settling for a narrow, less expressive but stronger interface, versus opting for a richer, more expressive but more fragile interface becomes especially evident if we want to implement a suite of "applications". What happens is that these applications will all look the same, and in fact be the same. Only their *configuration* (or initialization) will differ, which defies the purpose of defining a suite of applications in the first place.

[4]The implications of that *lack of expressiveness* are mostly that an application developer may feel not at ease to write applications because of that lack of *control* over what is done is actually done. This is a common problem with framework-based development (see Chapter 1).

#### 4.5.0.2 Granularity of Change

If we compare the VISITOR pattern with the TEMPLATE METHOD pattern for example, the element that varies has a different granularity. While *fine-grained steps* of an algorithm are kept flexible in the TEMPLATE METHOD case, *whole* algorithms are added via new subclasses in the VISITOR case [5].

Now that we have defined the mechanism through which is determined which operation will fulfill a request (depending on two receivers, the Processor's and the Data's type - via *double-dispatch*), it's time to look at the kind of support the framework is offering for the *implementation* of those operations.
For example, when a request is made in the framework to a Data object by calling the *apply* (*i.e.* accept) method,

```
Data *data;
Processor *processor;
// ...
data->apply(processor);
```

an application-specific Processor subclass's operation is executed.

```
AppProcessorX::process(AppData2 &) {
   // Implementation...
}
```

Getting to that point is the framework responsibility, but from there on, what are the tools available to an application developer to assist him in writing the implementation? The answer is: `AppData2` interface. This begs two questions:

1. Is this interface rich enough to allow algorithms to do their work, *i.e.* operate on the data?
2. Does the framework offer any help at that level to application developer?

These are very legitimate questions. We provide elements of answers to those questions in the section on generic programming at the end of this chapter, where we look at the usefulness of considering generic programming as an alternative, or complementary paradigm to classic polymorphism.
The answer to the first question is: probably not. We will see why soon. To answer the second question, this prototypical framework, implements the following three data access and traversal mechanisms (which will be discussed further in Chapter 5 - Wavelet Transform Framework):

[5] This is the case at least for the implementation of a visiting operation for a specific kind of data elements in a visitor subclass. The STRATEGY pattern also allows for change to occur at the same *whole-algorithm* level.

- *Indexing*: The data interface is allowing direct access to data elements via indexing with operations like `index(int i)`, `index(int i, int j)`, and `index(int i, int j, int k)` for 1D, 2D and 3D data respectively). Another type of indexing is also available in the form of *accessor methods* for elements in the *transformed data* hierarchical composition consisting in subbands containing subbands (discussed in chapter 5).

- *Iterators*: Iterators allowing for sequential access to raw data elements independently of the aggregate underlying implementation (vectors, arrays, etc.) and data structure type (1D, 2D and 3D data) as well as encapsulating different kinds of traversal and sub-data type access (like rows and columns).

- *Processors*: It is possible for processors (concrete visitor classes) to rely on other processors for their implementation. For example, a 2D algorithm might be implemented using 1D processors.

Some examples of these three mechanisms are given in the next chapter.

These mechanisms are however by no means the only possible ones. Although those are very interesting questions and challenging ones, it is outside the current territory of this thesis (because of time constraints mostly), and we have to relegate any work on that to the future. Possible avenues that would be worth exploring are:

- Processors as *internal iterators.* In this approach, some processors encapsulating and driving an iteration are used as sub-processors by other higher-level processors.

- *Index manipulators* (cursor-based iteration) to support different data traversal strategies.

- Class *wrappers* (adapters) for data coefficients, where wrapping an object amounts to adding new data access operations to it.

- *Decorated iterators and visitors as positions* for access of layered data elements in aggregate.

- Application of the EXTENSION OBJECT patterns for the addition of multiple interfaces to operate on data.

## 4.6 Design Notes

In this section we present some interesting design notes related to the design of the Wave ImAgE framework, and more specifically the data processing framework. These notes contribute to illustrating design problems we were involved with during the design of this prototype framework. They contribute to show the richness of the problems and solutions framework development is involved with. The main theme here however is the place that the generic programming paradigm can occupy with respect to classic polymorphism in the design of data elements and processors.

### 4.6.1 Generic Programming as an Alternative to Classic Polymorphism

Consider for example a template-based framework model like the following:

```
template <class Data, class Processor>
void process(Data &d, Processor &p) {
    p.apply(d);
}
```

Such a template-based approach offers the following **benefits**:

- Compile-time type checking.
- No need for Data and Processor base classes.
- Efficiency in terms of execution time.

This template-based approach however suffers from the following **liabilities**:

- We loose the capability of postponing the decision of which classes to instantiate until run time: each application requires recompiling and run-time configurability is impossible (this is assuming we are not relying on classic polymorphism at all).

There is however possibly a happier use of generic programming, which is outlined in the next section.

### 4.6.2 External Polymorphism and Processor Adapters

A library of generic component can be used to facilitate the development of application components. For example, lets consider the case of an already implemented class containing the implementation of some kind of processor, say `MyProcessor`:

```
class MyProcessor {
public:
   void doit(Data *d) {
      // Implementation
      d->operation();
   }
};
```

The interface of `MyProcessor` has only one method called `doit(Data *d)` in its interface. It would be interesting to be able to *reuse* that class in the framework. However, the processors in the framework have a different interface than `MyProcessor`. The ADAPTER design pattern provides an element of solution to such a problem. The intent section of the pattern says:

ADAPTER

> Convert the interface of a class into another interface clients expect. ADAPTER lets classes work together that couldn't otherwise because of incompatible interfaces. [24]

The processor interface offered by the framework looks like:

```
class Processor {
public:
   virtual void process(Data *d) = 0;
};
```

How can `MyProcessor` be treated as a new Processor subclasses? The solution proposed in Wave ImAgE uses a generic programming approach called *external polymorphism* [14]. We apply external polymorphism to create a templatized adapter that looks like this:

```
template <class ADAPTEE>
class ProcessorAdapter : public Processor {
public:
   ProcessorAdapter(ADAPTEE* o, void (ADAPTEE::*m)(Data *d) ) {
      object = o;
      method = m;
   }
   ~ProcessorAdapter() {
      delete object;
   }
   void process(Data *d) {
      (object->*method)(d);
   }
private:
   ADAPTEE* object;
   void (ADAPTEE::*method)(Data *d);
};
```

The parameter type ADAPTEE can then be treated as a subclass of `Processor`, *as if* `MyProcessor` had a virtual method `process(Data *d)`. Such an adapter can then be used in the following manner:

```
int main() {
   Data *d = new Data;

   // Creation of my application-specific processor
   MyProcessor *myp = new MyProcessor;

   // No need to subclass (and have the same interface) to
   // take benefit of polymorphism. The adapter can be reused
   // to do it for us.
   Processor* p = new ProcessorAdapter<MyProcessor>(myp, &MyProcessor::doit);
   p->process(d);
}
```

Such a use of templates is very useful for application developers to reuse existing classes as the implementation of some processors.

### 4.6.3 Even More Adaptation using the Extension Object Pattern

We present here a "smarter" adapter that we built using external polymorphism and the EXTENSION OBJECT pattern. The participants are the same as above: processors, data and adapter. The addition is an extension hierarchy used to augment the processor's class interface. The intent of the EXTENSION OBJECT pattern is:

> Attach additional methods to a class. Whereas DECORATOR requires that the core class's interface remain fixed as successive "wrappers" are applied, EXTENSION OBJECT allow the class's interface to grow incrementally and dynamically.

So we start with a degenerate base class for the extensions:

```
class DataExtension {
public:
   virtual ~DataExtension() {}
};
```

We add operations `addExt` and `getExt` to the `Data` base class interface to support management of extensions.

```
class Data {
public:
   // Extension management interface
   void addExt(string extName, DataExtension *ext) {
      _extensions[extName] = ext;
   }
   DataExtension *getExt(string extName) {
      return _extensions[extName];
   }
   // Visitor's interface
   virtual void accept(Processor *p) = 0;
protected:
   map<string, DataExtension *> _extensions;
};
```

We provide a templatized concrete extension class with an indexed access to the data.

```
template <class DATA_STRUCTURE>
class Data2DIndexExt : public DataExtension {
public:
   Data2DIndexExt(DATA_STRUCTURE *d):_d(d) {}
   void index2d(int i, int j) {
      // Use data subclass interface...
      (*_d)->index(0);
   }
private:
   DATA_STRUCTURE *_d;
};
```

A generic data subclass can also be defined for reuse, to hold data on any dimension.

```
template <class T>
class TData : public Data {
public:
   typedef T DataType;
   TData(T *t) : _d(t) {}
   void accept(Processor *p) { p->visit(this); }
   // Default interface
   T* index(int i) { return _d; }
private:
```

```
    T *_d;
};
```

We still have our base class for processors of all kinds:

```
class Processor {
public:
   virtual void visit(Data *d) {}
};
```

And the heart of the matter is the templatized adapter, which has now the added responsibility to adapt not only the interface of `MyProcessor`, but also the data interface used in the implementation of the adapted method

```
template <class ADAPTEE, class EXTTYPE>
class ImplAdapter : public Processor {
public:
   typedef EXTTYPE DSTYPE;
   ImplAdapter(ADAPTEE* o, void (ADAPTEE::*m)(DSTYPE *d) ) {
      object = o;
      method = m;
   }
   ImplAdapter(ADAPTEE* o, void (ADAPTEE::*m)(DSTYPE *d),
               string extName ) {
      object = o;
      method = m;
      _ext = extName;
   }
   ~ImplAdapter() {
      delete object;
   }
   void visit(Data *d) {
      DataExtension *ext = d->getExt(_ext);
      DSTYPE *dsExt;
      if (dsExt = dynamic_cast<DSTYPE *>(ext)) {
         (object->*method)(dsExt);
      }
   }
private:
   ADAPTEE* object;
```

```
      void (ADAPTEE::*method)(DSTYPE *d);
      string _ext;
   };
```

Again, the idea is to adapt (reuse) MyProcessor which is *not* a subclass of Processor and it has a quite different interface:

```
   class MyProcessor {
   public:
      void myOperation(Data2DIndexExt<TData<int> > *d) {
         // Use the extension's iterface
         d->index2d(1,2);
      }
   };
```

In a mainline, putting things to work looks like this:

```
   int main() {
      int i = 5;
      TData<int> *dInt = new TData<int>(&i);
      // Create extension
      Data2DIndexExt<TData<int> > *ext1 = new Data2DIndexExt<TData<int> >(dInt);
      // Register the extension with the data
      dInt->addExt("2Dindex", ext1);
      Data *d = dInt;
      // Create processor to reuse
      MyProcessor *myp = new MyProcessor;
      // Configure the adapter with the extension (and the data)
      Processor *p2 =
        new ImplAdapter<MyProcessor, Data2DIndexExt<TData<int> > >
                        (myp, &MyProcessor::myOperation, "2Dindex");
      // Ask the visitor to do its visitor thing
      d->accept(p2);
   };
```

### 4.6.4 Mixing Paradigms: A Common Base Class for Different Parametrizations

A common misconception is that template classes with different parameters are related by inheritance. For example, the class `foo<int>` is *not* related by inheritance to `foo<float>` through a base

class called `foo`. If we want a base class for our template classes, we need to create one. But why would it be useful in the first place to have such a base class? One reason could be that we wish to dynamically create templated objects but decide the parameters at run time.

There are two problems that one may wish to solve using virtual template methods:

- We want to dispatch an overloaded virtual function to a template function because the implementation of the overloaded functions are the same.
- We do not want to make a commitment about what types are supported and we use template as a way to dispatch *any* types. In other words, we want a *flexible* virtual function.

Let's say we wish to use double-dispatch to solve the second problem via a variation of the VISITOR pattern. We then have a base class:

```
class Data {
public:
    virtual void accept(Algo &algo) = 0;
};
```

and templatized derived classes. For two dimensional data, for example, we might have something like this:

```
template <class T>
class Data2D : public Data {
public:
    void accept(Algo &algo) {
        algo.visit(*this);
    }
};
```

Then a visitor class may be defined using the templatized derived class as a parameter:

```
// Visitor
class Algorithm {
public:
    // ...
    virtual void visit(Data2D<int> &data) {
        // Default implementation
    }
```

```
        // Similarly for and other combinations of data
        // structure and data types supported...
    };
```

Concrete visitors can then be defined as below

```
    class Algo1D : public Algorithm {
    public:
        void visit(Data1D<int> &data) {
            // Implementation of 1D algorithm on 1D data of ints.
        }
        void visit(Data1D<float> &data) {
            // Implementation of 1D algorithm on 1D data of floats.
        }

        void visit(Data1D<complex> &data) {
            // Implementation of 1D algorithm on 1D data of complex.
        }
        // ...
    };
```

Such a design will allow framework classes (here Framework) to rely on abstract coupling to implement a process method:

```
    class Framework {
    public:
        void process(Data &data, Algo &algo) {
            data.accept(algo);
        }
    };
    main() {
        // Creation of a framework class
        Framework f;
        // Creation of application-specific data and algorithm
        Algo1D algo;
        Data1D<int> data;
        // Rely on double dispatch to obtain proper behavior
        f.process(data, algo);
    };
```

A problem with this approach is that we have to specify *all possible template parameters in many*

*places*, *i.e.* in every visitor (algorithm) classes. Having many `visit(...)` methods in the visitor's classes is standard in the VISITOR's design[6]. What is less usual is the fact that there are operations for different data types (*i.e.* int, float, complex,...) in addition to the operations for each data structure (*i.e.* the classes of the element structure in the VISITOR pattern: `Data1D<T>`, `Data2D<T>` and `Data3D<T>`).
The data type supported is more likely to change than the dimensionality of the data supported. This stability of the data structure type supported is what prompted us in the first place to consider the VISITOR pattern in our solution. So this design is bad, since the addition of a new data to support (for 1D data for example) would result in the pattern to fall down. It would require a change in the framework, *i.e.* the addition of (at least) one new virtual member function, `visit(Data1D<new type>` in the Visitor's base class for the new type. This is far from ideal.
Another approach would be to make use of the base data class in the visitor classes and adopt the brute-force approach, *i.e.* `dynamic_cast`:

```
// Visitor
class Algo {
public:
    virtual void visit(Data &data) {
        // Default implementation
    }
};
// Concrete visitors
class Algo1D : public Algo {
public:
    void visit(Data &data) {
        Data1D<int> *d1Dint;
        Data1D<int> *d1Dfloat;
        Data1D<int> *d1Dcomplex;

        if (d1Dint = dynamic_cast<Data1D<int>*>(&data)) {
            // int-specific implementation
        } else if (d1Dfloat = dynamic_cast<Data1D<float>*>(&data)) {
            generic_impl(d1Dfloat)
        } else if (d1Dcomplex = dynamic_cast<Data1D<complex>*>(&data)) {
            generic_impl(d1Dcomplex);
        } else { // 2D and 3D algos
             // exception
        }
```

[6] We opted for virtual visit operations in the Visitor base class (`Algo`) instead of pure virtual ones in order to simplify the client classes (`Algo1D`, `Algo2D`, and `Algo3D`) and to rely on proper exception mechanism in the default base class implementation of the `visit` operations.

```
    }
private:
    template <class T>
    void generic_impl(Data1D<T> &data) {
        // generic implementation
    }
};
```

While this approach simplifies the Algo *interface*, it makes the subclasses, *i.e.* the application specific classes, more complicated. It also seems to defy the purpose of the VISITOR pattern in adding type test and conditional branches which amounts to a second level of method dispatch.

In light of this diversity, we are asking the following question: is it still posible to have a common interface for data that is expressive enough to be useful (since *common* tends to mean *narrow* and *minimal*)? We will look at this problem by first considering the **first point of variation** above, the type of element contained in the data.
Knowing that processors ultimately need to *access* and manipulate the data encapsulated within the data objects, what kind of interface allows that? For example, consider transformed images. They can either be represented as an array of integers or real values depending on the application-specific transform that was performed on the original image data. The C++ Template mechanism comes to mind as a way to provide that *parametric polymorphism*[7] we are looking for:

```
template <class PIXTYPE>
class Data {
public:
    // Public interface
    //..
private:
    PIXTYPE *data;
};
```

Since the pixel type is application-specific (in the sense that application-specific (sub)classes only know about the pixel type), how can the framework create data objects, when instantiation of a template class requires the specification of the actual type? The actual type of the contained objects cannot (or should not...) be known at compile-time. For example, in a framework class, the following

```
Data<int> data;
```

[7]Also called *compile-time polymorphism* as oppose to the *run-time polymorphism provided by virtual functions.*

is just not possible, since only an application knows that the data to be handled is represented using integers[8]. The framework has to be able to manipulate data objects in the following manner:

```
void framework(Data *data) {
    data->operation();
};
```

*i.e.* without having to specify a pixel type (which it does not know about). Possible solutions include (1) using inheritance instead of templates, and (2) define a non-template abstract base class providing an interface for template derived classes. We will briefly examine those two alternatives.

1. *Defining a hierarchy of data.*

```
class Data {
public:
    // ...
    virtual void operation() = 0;
};
class DataInt : public Data {
public:
    // ...
    void operation() {
        // ...
    }
private:
    int *data;
};
class DataComplex : public Data {
public:
    // ...
    void operation() {
        // ...
    }
private:
    complex *data;
};
```

[8]And similarly, only application-specific code knows *how* to handle such data. Our problem here is not so much the *creation* of application-specific data–which can always be abstracted with some sort of factory, abstract or not, in fact, any one of several creational pattern in [24] could do– as to how to define a base class that will allow the framework to *do its job*.

and Data can be used like this:

```
main() {
    DataInt *data = new DataInt;
    framework(data);
}
```

2. *Non-generic base class and generic derived class.*[9]

```
class Data {
public:
    virtual void operation() = 0;
};

template class<PIXTYPE>
class DataImp<PIXTYPE> : public Data {
public:
    void operation() {
        // ...
    }
private:
    PIXTYPE *data;
};
```

and Data can be used like this:

```
main() {
    Data<int> *data = new Data<int>;
    framework(data);
}
```

So far, so good. Both approaches seem to provide an acceptable solution to the problem of representing data containers for coefficients of verying types. We have not said anything about the *behavior* of those classes represented here with the apparently quite generic member function `operation()`. If we assume that the type `PIXTYPE` does not affect the *behavior* of the class, then we might prefer the template class approach for its simplicity and economy of code. Let us now look at the **second variability requirement**, the dimensionality of the data.

[9] We are thinking about *public* inheritance for *interface reuse* and not *private* inheritance for code sharing (implementation reuse).

The framework is intended to support the development of applications dealing with 1D, 2D and 3D data processing. In light of what we said in the previous paragraph, it seems that if we want to design classes representing data of different dimensionality, then the use of inheritance seems indicated. In fact, we cannot write a single `operation()` function to handle data of differing dimensionality. Operations performed on 1D data are quite different, in terms of their algorithmic structure, than operations performed on 2D or 3D data. The interface is the same, but the behavior, the implementation differs. Thus, we can consider a class hierarchy like the one below which, by encapsuling the concept that varies:

```
class Data {
public:
    // ...
    virtual void operation() = 0;
    // ...
};
class Data1D : public Data {
public:
    // ...
    void operation();
    // ...
};
class Data2D : public Data {
public:
    // ...
    void operation();
    // ...
};
class Data3D : public Data {
public:
    // ...
    void operation();
    // ...
};
```

With such a hierarchy, the framework can manipulate the abstract base class only:

```
main() {
    Data1D *data = new Data1D;
    framework(data);
}
```

and within `framework` the statement

```
data->operation();
```

will invoke `Data1D::operation()` as desired. In order to handle our first two variability requirements, *i.e.* pixel type and dimensionality, we could combine inheritance and generic classes in the following way:

```
class Data {
public:
    virtual void operation() = 0;
};
template class<PIXTYPE>
class Data1D<PIXTYPE> : public Data {
public:
    void operation() {
        // Implementation of an operation on a 1D object.
    }
private:
    PIXTYPE *data;
};
template class<PIXTYPE>
class Data2D<PIXTYPE> : public Data {
public:
    void operation() {
        // Implementation of an operation on a 2D object.
    }
private:
    PIXTYPE *data;
};
```

This approach mixing inheritance and templates, inspired by the presentation found in chapter 11 of [4] is the one we retain to implement the multidimensional data arrays in our prototype.

### 4.6.5 Genericity of Image Processing Algorithms

The goal of this section two-fold. First, we want to briefly present different techniques that can be used to obtain genericity of image processing algorithms, and second, we want to express the fact that genericity can be usefully and successfully applied when an important condition is met:

commonality of *algorithmic structure* across data structure types. The presentation in this section is inspired by the work found in [25].

Genericity for image processing algorithms means providing a generic writing for image processing algorithms, *i.e.* independent of both *data type* (int, float, complex,...) and *data structure type* (1D signals, 2D images, geometries, etc...). We will briefly look at three approaches used to achieve genericity:

- Generic programming
- Classic polymorphism
- Generic polymorphism

### 4.6.6 Generic programming

Generic programming uses the C++ template mechanism to write data containers and algorithms that are independent of *data types*. For example, consider a class for 2D images:

```
template <class DataType>
class Image2D {
    int nRows;
    int nCols;
    DataType** data;
    // ...
};
```

It is then possible to write algorithms that are independent of the data type in the following way:

```
template <class DataType>
void add(Image2D<DataType> &image, DataType val) {
    for (int i=0; i<image.nRows; i++) {
        for (int i=0; i<image.nRows; i++) {
            // The algorithm is implemented using
            // the public interface of Image2D.
            image.data[i][j] += val;
        }
    }
}
```

Using classic polymormhism, it is possible to write algorithms that are both data type and data structure type independent.

### 4.6.7 Classic polymorphism

Data *structure* type independence is achieve through the use of template abstract base classes: a data base class declaring a factory method for the creation of an iterator, and an iterator base class, to be used by the clients of data subclasses, which are the algorithms operating on the data. the base classes may look like this:

```
// Abstract Data class
template< class DataType >
class Data {
    virtual Iterator<DataType> &createIterator() = 0;
};

// Abstract Iterator class
template< class DataType >
class Iterator {
    virtual first() = 0;
    virtual void next() = 0;
    virtual bool isDone() const = 0;
    virtual DataType &value() =0;
};
```

And the subclasses declarations, like this:

```
template< class DataType >
class Image2D : public Data< DataType >;

template< class DataType >
class IteratorImage2D : public Iterator< DataType >;
```

A generic algorithm can then be written in the following way:

```
template <class DataType>
void add(Data<DataType> &data, DataType val) {
    Iterator<DataType> &it = data.createIterator();
    for (it.first(); !it.isDone; it.next()) {
```

```
            it.value() += val;
        }
    }
```

The important thing is that once written, this algorithm can be reused for all the newly introduced data and iterator subclasses. When performance is important (which is often the case for some kinds of image processing algorithms), then a more efficient implementation can be written using templates. The thing we would like to point out though is that for the above algorithm to be truly data type *and* data *structure* type independent, the *algorithmic structure* of that specific kind of algorithm (*i.e.* what it semantically accomplishes) has to be also data structure independent, at least in the sense that the commonality across data structure types can be captured by an iterator base class while the variability should be encapsulated into an iterator subclass. This may not always be possible.

Generic polymorphism refers to the parametrization of operations for specific types of algorithms with a data structure type, as we will see below.

#### 4.6.7.1 Generic polymorphism

By relying on type definitions in the data structure classes, it is possible to get rid of the need for iterator and data structure type base classes:

```
template< class Data >
class IteratorImage2D;

template< class Data >
class Image2D {
public:
    typedef Data                     DataType;
    typedef IteratorImage2D<Data> Iterator;
    // ...
};
```

This approach is the generic equivalent of the previous approch. The algorithms are now parametrized directly with the data structure type:

```
template<class DataStructure>
void add(DataStructure& data, typename DataStructure::DataType val ) {
    typename DataStructure::Iterator it(data);
    for(it.first(); !it.isDone(); it.next()) {
        iter.value() += val;
```

```
    }
}
```

Type deduction is done from the data structure type from type definitions in the structure classes. Again, it is possible to write only one function per type of processing (here addition), *i.e.* independently of data structure and data types, *provided* that the algorithmic structure (the `for` loop in the above examples) is common across data structures, and that that commonality can be captured within an iterator class.

# Chapter 5

# Wavelet Transform Framework

## 5.1 Overview

This chapter describes the wavelet transform framelet for the Wave ImAgE framework. The framework is described at both the framelet design level and at the framelet architectural level.

This framework proposes a design solution to the problem of performing wavelet transforms on data signals.

The framework enhances reusability mainly because of the following reasons:

- It decouples the *data* from the *transform algorithms* that are performed on it. This is achieved by refining the Data Processing Framework presented in the previous chapter. Here, a Wavelet Transform is a specific type of processor.
- It captures the stable part of wavelet transform algorithms and encapsulates the variable parts into well-identified variation-points (also called "hot-spots"). The design of these variation-points, which were identified during the domain analysis (see Chapter 3), will be described in this chapter.

## 5.2 Context

The context for the design of the framework is described in Chapter 3, Wavelets, Wavelet Transforms and Image Processing Applications. This chapter assumes that the reader is familiar with

Design Patterns [24][1] and with the processing framework in Chapter 4 and the concepts that were introduced in Chapter 1.

## 5.3 Problem

The design solution proposed should be influenced by the following *forces*:

- The use of the term *transform* in the literature often refers to two different things: the transform *process* and the transform *product* (or *representation*). The transform construction process and the resulting transform representation (or *transformed data*) *should* be kept separate so that the two can vary independently. This allows, for example, that the same construction *process* can create different *representations.* Such a decoupling should give us a finer control over the transform process and let us vary the transform internal representation. The clients *should not* need to know anything about the internal transform representation.
- A wavelet transform is a *hierarchical structure* produced by a *recursive process.* The product of that process are subbands containing subbands. The data structure used to represent a wavelet transform *should* capture the hierarchical structure of a transform. The question to determine whether a flat data structure (an array for example) is expressive and convenient enough to represent a wavelet transform or if on the other hand a recursive data structure would be more appropriate, should be considered. The kind of structure found in a wavelet transform is a *semantic* one: coefficients in different locations *mean* something different (for different clients, which are algorithms operating on transformed data).
- The number and kind of wavelet transform operations will most likely proliferate as more applications are developed. It *should* be possible to add new wavelet transform processing operations without changing the classes of the elements on which they (the operations) operate, as well as to easily extend and modify any existing operations. Basically, a primary challenge when designing the Data interface is to come up with a minimal set of operations that lets clients build open-ended functionality. Performing "surgery" on Data and its subclasses for *each* new capability is both invasive and error-prone. The risk is that the Data interface evolving as a hodgepodge of operations eventually obscuring the essential properties of Data objects. When that happens, the classes just get harder to understand, to extend and to use.
- A transform (process) should be *configurable* by the clients. This means that a transform's variable elements can be composed into a transform process as interchangeable building blocks. Parameters could also be used for that matter. Examples of configuration parameters are *depth*

[1] Particularly the VISITOR, TEMPLATE METHOD, STRATEGY, DECORATOR, COMPOSITE and PROXY

of transform, *basis function* (wavelet), *prefiltering* schemes, *shape* of the transform (rectangular or square or other combinations), *translation invariance*, *dimensionality* of the transform algorithm (depending on the dimensionality of the input data it operates on), etc...

- A transform (process) should be *extensible.* It should be possible for a user to build more complex, compound (or composite) transforms from a kernel transform by using basic transform building blocks. Examples of such compound transforms are the ridgelet and curvelet transforms. They are composed of a sequence of steps, each step being a building block of the whole transform. These particular transforms will not be considered here further.

## 5.4 Solution

The elements of solution outlined in this section will be presented as candidate solutions for the requirements of a series of variation-points. These requirements are:

- Need for the ability to support transform of data signals of *different dimensions.*
- Need for the possibility to vary wavelet *transform steps algorithm implementation.*
- Need to vary the *prefiltering* performed on data signals.
- Need to vary the *wavelet filters* used in a transform.

In the next few sections, we will present some design elements in support of the following variation-points:

- Transform steps in a multi-level wavelet transform.
- Support for multiple preprocessing and prefiltering schemes.

### 5.4.1 Wavelet Transform Steps

In this section, we present elements of design for a candidate implementation of the wavelet transform step variation-point. The main requirement consists in allowing application developers to modify the behavior of a multi-level wavelet transform by providing application-specific transform *steps.* The patterns involved in this solution are, by order of importance with respect the variation-point semantics (*i.e.* what varies) are TEMPLATE METHOD, STRATEGY, DECORATOR, COMPOSITE, ITERATOR, MEMENTO, FACTORY METHOD and PROXY. Their role in this specific context are briefly described below:

1. TEMPLATE METHOD: The `WaveletTransform::transform()` method encapsulates the stable part of the transform algorithm, which is the recursive structure of the transform. The hooks are the transform steps, *i.e.* the processing applied on a subband at each level of a multi-level transform.

2. STRATEGY: The transform steps are concrete strategies. They implement the actual transform performed on the data.

3. DECORATOR, COMPOSITE: The transform steps for a multi-level wavelet transform are represented by a composite transform step, which is composed of concrete transform steps. Composite processors and transforms can also be supported though not discussed here (due to space constraints). Decorators (or wrappers) are used to add responsibility to transform-step objects, called preprocessing, which include units like prefiltering and signal extension (not discussed here).

4. ITERATOR: An iterator is used to allow a multi-level wavelet transform to traverse the composite transform steps.

5. MEMENTO: Memento is used to hold the state of the traversal during a wavelet transform. Memento allows for a recursive descent into the transform steps so that concrete transform steps are encountered in the order they are meant to be processed.

6. PROXY: A proxy is used for internal purposes to allow automatic destruction of an iterator so that a client (a wavelet transform) does not have to worry about deleting it.

7. FACTORY METHOD: The iterator is created using a factory method.

#### 5.4.1.1 Wavelet Transform

The wavelet transform class presented here in the context of this specific variation-point has a very simple interface (`transform()`) reflecting just what the users are interested in, *i.e.* performing a wavelet transform on data [2]:

```
class WaveletTransform {
public:
   WaveletTransform(TransformStep *step);
   // Template Method taking care of the recursion.
   bool transform();
private:
   bool transform(DescriptorHandle &it);
   DescriptorHandle _transformStepDescriptor; };
```

[2]Since our focus is on the algorithmic structure of the transform, we can omit mentioning the data to unclutter the code samples.

Before we go on, just a few words on DescriptorHandle. DescriptorHandle is an example of a proxy (see Figure 34) (for the real iterator DescriptionTraverser, to be discussed below). The wavelet transform is using this proxy (which overloaded -> and * operators) so that _transformStepDescriptor has to be used like a pointer even if it is *not* really a pointer. The "real" iterator DescriptorTraverser is allocated on the stack, therefore C++ automatically takes care of calling the destructor when it goes out of scope (which in turn deletes the real DescriptorTraverser), so that the "client" is not responsible for deleting it. This usage of a proxy is particularly useful in application-specific code, to simplify the life of application developers (by taking some responsibilities out of their busy hands).

A WaveletTransform is created by passing a TransformStep as argument to the constructor. In do-

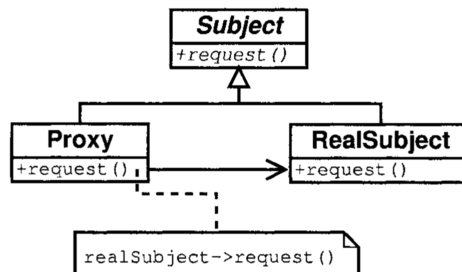



Figure 34: The PROXY design pattern.

ing so we are configuring a wavelet transform with a concrete transform step strategy (see Figure 35). We are actually modifying the behavior of the template method (see Figure 36) transform() by providing implementation to a hook method (transform()) located in a separate class (TransformStep) (this is an example of the *separation construction principle*).

```
WaveletTransform::WaveletTransform(TransformStep *step)
   :_transformStepDescriptor(step->descriptorTraverser()) {}
```

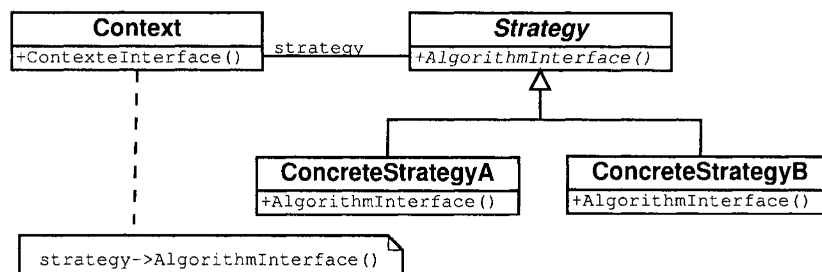



Figure 35: The STRATEGY design pattern.

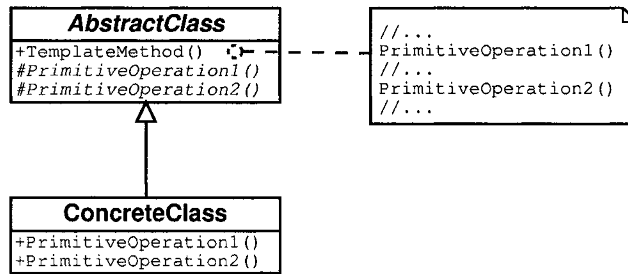



Figure 36: The TEMPLATE METHOD design pattern.

#### 5.4.1.2 Performing a Multi-Level Transform via Iteration

In order to perform a multi-level wavelet transform, the wavelet transform `transform()` method has to take care of the recursive nature of the algorithm. It does so by iterating through the elements contained in a multi-level transform step (called a `description` of a multi-level transform) using an iterator (see Figure 37) that we call `DescriptionTraverser`. This iterator allows for a simple, sequential iteration through the composite structure giving access, at each level of the transform, to the transform step to be used at that level. The public interface of this iterator is given below:

```
class DescriptorTraverser {
public:
   DescriptorTraverser( TransformStep* first );
   void firstStep();
   void nextStep();
   TransformStep* currentStep();
   bool isCompleted();
   // ...
};
```

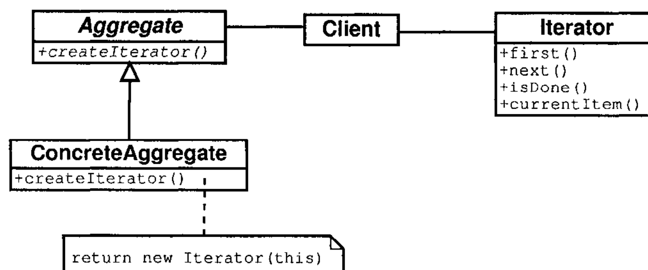



Figure 37: The ITERATOR design pattern.

When the `transform()` request is actually made, the descriptor of the transform is initialized with the transform step for the first level of the transform (using the iterator), and then the transform process is started:

```
bool
WaveletTransform::transform() {
   _transformStepDescriptor->firstStep();
   recursiveTransform();
}
```

The heart of the `WaveletTransform` class is in fact the `recursiveTransform()` *template method* which really captures what is common across wavelet transforms: the recursive structure.

```
void WaveletTransform::recursiveTransform() {
   // Get current transform step and
   // perform the transform step by delegating to the
   // TransformStep strategy.
   _transformStepDescriptor->currentStep()->transform();

   // If transform has a next level
   _transformStepDescriptor->nextStep();
   if (!_transformStepDescriptor->isCompleted())
   // Recursive call on lower subband
   // Note: data manipulation not shown here
   // since the focus is on the structure
   // of the algorithm.
   recursiveTransform(); }
```

What is not shown here is that the next level of the transform is performed on the low-level subband of the newly transformed signal.

Now, let us look at how a transform step is actually performed. Figure 44 illustrates the class structure of this variation-point:

The wavelet transform algorithm only knows about abstract transform steps `TransformStep`.

#### 5.4.1.3 Transform Steps

The base class for transform steps looks like:

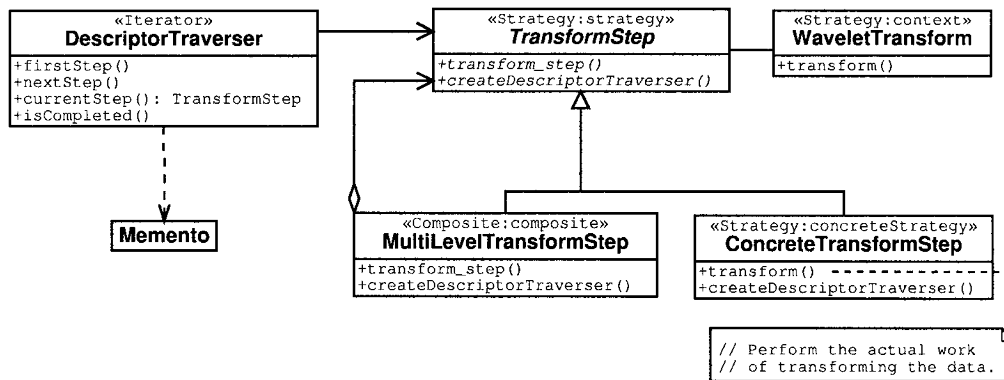



Figure 38: Design of the wavelet transform variation-point.

```
class TransformStep {
public:
   virtual DescriptorTraverser* descriptorTraverser() = 0;
   virtual void transform() { // Default empty implementation }
   // ...
};
```

Concrete transform steps only have to implement the factory method `descriptorTraverser()` and provide an implementation for the transform step.

```
class ConcreteTransformStep : public TransformStep {
public:
   ConcreteTransformStep() { //...}
   DescriptorTraverser* descriptorTraverser() {
      return new DescriptorTraverser( this ); }
   void transform() {
      // Implementation of the transform step
   }
   // ...
};
```

The sequence of transform steps to be performed in a multi-level transform are represented by a composite transform step called `MultiLevelTransformSteps`:

```
class MultiLevelTransformStep : public TransformStep {
public:
   friend class DescriptorTraverser;
```

```
    DescriptorTraverser* descriptorTraverser() {
        return new DescriptorTraverser( this ); }
    // Methods for setting up a multi-level transform
    void addLevel( TransformStep* step ) { _steps.push_back( step ); }
    // ...
private:
    // Transform steps for all levels
    vector<TransformStep*> _steps;
};
```

A wavelet transform algorithm only knows about abstract transform step (`TransformStep`) and their iterator proxy (`DescriptionHandle`). So, whether the transform (step) is single or multi-level, this is transparent to the wavelet transform. Hence the simplicity of the `transform` template method.

#### 5.4.1.4 Configuring and Executing a Multi-Level Wavelet Transform Application

The main driver program for this wavelet transform framelet implementing the transform step variation-point is illustrated in the following code fragment:

```
int main() {
    // 1- Creating a 2D multi-level transform
    MultiLevelTransformStep *descriptor = new MultiLevelTransformStep;
    // Level 1: on rows then columns
    descriptor->addLevel(new ConcreteTransformStep("RC"));
    // Level 2: on rows only
    descriptor->addLevel(new ConcreteTransformStep("R"));
    // Level 3 - on columns only
    descriptor->addLevel(new ConcreteTransformStep("C"));

    // 2- Configuring the transform
    WaveletTransform multi_level_wt( descriptor );

    // 3- Performing the transform
    multi_level_wt.transform();

    // 1- Building a single-level transform
    ConcreteTransformStep *step = new ConcreteTransformStep("RC");

    // 2- Executing the transform
    WaveletTransform single_level_wt( step );
```

```
    single_level_wt.transform();
}
```

This approach allows for great flexibility in configuring a transform since each level of a transform can be configured independently with an open-ended number of extensible algorithms.

### 5.4.2 Supporting Different Prefiltering Schemes

Prefiltering schemes were introduced in Chapter 3 as means to prepare data signals for a transform step processing unit when the wavelet coefficients and the signal coefficients are of different types (or dimensions). In this section, we present some elements of design in support for the prefiltering variation-point, *i.e.* the need for different applications to add, remove and customize preprocessing steps, and more particularly, pre-*filtering* steps.

#### 5.4.2.1 Transform

In this context, the framework provides a `Transform` processor with default behavior. The `process` method is adapted in order to provide the application developer a semantically meaningful interface (the `process` operations are only of interest to the framework taking care of double-dispatching).

```
// Predefined framework
class Transform : public Processor {
public:
   // Adapter methods for transform on different
   // framework predefined data
   Data1D* process(Data1D *d) {
      return transform(d);
   }
   Data2D* process(Data2D *d) {
      return transform(d);
   }
private:
   // Hook methods for subclasses to override
   virtual Data1D* transform(Data1D *d1) {
      // Default ``empty'' implementation
      return d1;
   }
   virtual Data2D* transform(Data2D *d2) {
      // Default ``empty'' implementation
      return d2;
   }
};
```

#### 5.4.2.2 General Processor Decorator

The general processor decorator, as its name indicates, is based on the DECORATOR pattern. The intent section of this pattern says:
DECORATOR

> Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.[24]

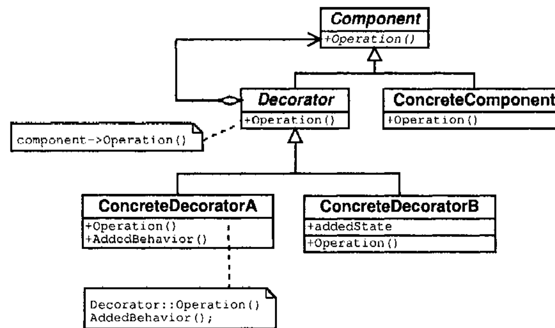



Figure 39: DECORATOR design pattern [24].

To filter the data *before* actually processing it is to extend the functionality of the processor object with the additional responsibility to pre-filter the data. So we define a general decorator to support various kinds of extensions:

```
// Framework
class ProcessorDecorator : public Processor {
public:
   ProcessorDecorator(Processor *processor)
      :_processor(processor) {}
   virtual Data* process(Data *d) {
      return _processor->process(d);
   }
private:
   Processor *_processor;
};
```

#### 5.4.2.3 Preprocessing Wrapper

A subclass of the `PreprocessorDecorator` specializes the decoration for pre-*processing*:

```
// Framework
class Preprocessor : public ProcessorDecorator {
public:
   Preprocessor(Processor *p, Processor *supplier)
      : ProcessorDecorator(p), _supplier(supplier) {}
   Data* process(D *d) {
      // Notice how the data is filtered before
      // it is being processed.
      return ProcessorDecorator::process( prefilter(d) );
   }
private:
   virtual Data* prefilter(Data *d) {
      return d->apply(_supplier);
   }
   Processor *_supplier;
};
```

Hence, a preprocessor is configured with a `Processor` supplier.

#### 5.4.2.4 Prefilter Supplier Processor

In order to guide the development of applications interested in prefiltering as a specific kind of preprocessing, a `Prefilter` class plays the role of an interface for the `prefilter` operation (it also plays the role of a carrier of type information and an adapter for the `Processor::process` method.

```
// Predefined framework
class Prefilter : public Processor {
public:
   // Adapter method
   Data1D* process(Data1D *d1) {
      return prefilter(d1);
   }
private:
   virtual Data1D* prefilter(Data1D *d1) = 0;
};
```

#### 5.4.2.5 Application-Specific Prefilter

Applications subclass the prefilter and provide an implementation to the `prefilter` operation. Basically, a subclass represents a place to put an implementation:

```
// Application: subclass of framework-provided Prefilter
class AppPrefilter : public Prefilter {
public:
   Data1D* prefilter(Data1D *d1) {
     // Prefiltering implementation
   }
};
```

A concrete transform is configured with a wavelet (here seen as a container of wavelet coefficients)

```
// Application
class ConcreteTransform : public Transform {
public:
   MyTransform(Wavelet1 *w1)
      :_w(w1){}
   Data1D* transform(Data1D *d1) {
      // Perform the transform and return transformed signal
      // Details of the transform are ignored here since
      // they are not important in the context of this
      // variation-point.
return d1;
   }
private:
   Wavelet1 *_w;
};
```

#### 5.4.2.6 Creating a Prefiltering Application

The following code illustrates the configuration and execution of an application that performs prefiltering on a data signal as a preprocessing step to a wavelet transform:

```
int main() {
   // 1- Creation of input data signal
   Data *input;
   // ...
   // 2- Configure a transform processor with
   // a wavelet and a corresponding prefilter.
   Processor *p =
      new Preprocessor(new ConcreteTransform (new Wavelet1),
       new AppPrefilter);
```

```
    // 3- Initiate the processing
    Data *output = p->process( input );

    return 0;
}
```

## 5.5 Data Representation

We refer the reader to the Design Notes sections from this chapter and the previous one, especially the section 4.6.4 (p. 85), which illustrates the problem of defining the interface of a data class hierarchy in the face of diversity. In this section, we simply present a candidate design for the representation of data in the context of wavelet transforms.

The interface proposed for the data hierarchy is composed of three kinds of operations:

- *Data element index methods*: The data interface is allowing direct access to data elements via indexing with operations like `index(int i)`, `index(int i, int j)`, and `index(int i, int j, int k)` for 1D, 2D and 3D data respectively.
- *Subband access methods*: Another type of indexing is also available in the form of *accessor methods* for elements in the *transformed data* hierarchical composition consisting in subbands containing subbands.
- *Factory methods*: Iterators allowing sequential access to raw data elements independently of the underlying implementation (vectors, arrays, etc.) and data structure type (1D, 2D and 3D data) as well as encapsulating different kinds of traversal and sub-data type access (like rows and columns) are created using a factory method.

### 5.5.1 Base Data

Let us start our tour of this design with the base class for data signals:

```
class Data {
public:
   virtual void accept(DataProcessor *ts) = 0;
};
```

declaring only the visitor's `accept` interface.

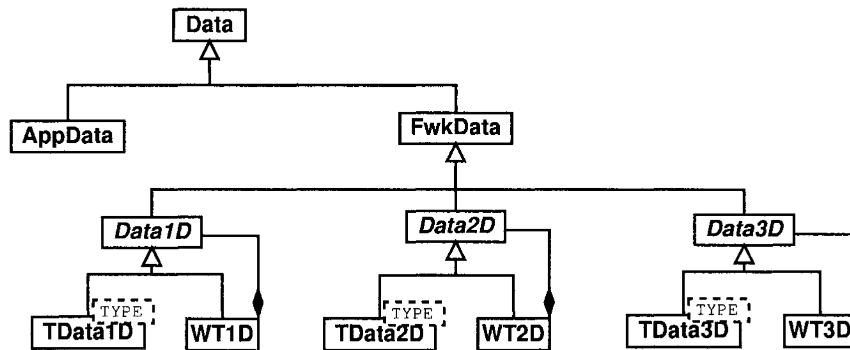



Figure 40: Data hierarchy in Wave ImAgE.

### 5.5.2 Data Subclass for Non-Transformed 1D Signals

This subclass declares a general interface for subband access and to support the VISITOR pattern.

```
class Data1D : public Data {
public:
   Data1D() {}
   Data1D(Data1D *parent)
      :_parent(parent) {}
   // Visitor's accept method
   virtual void accept(DataProcessor *p);
   // Wavelet transform related method (as base class)
   virtual WT1D* GetWT() { return 0; }
   // Return subband <band> at current level
   virtual Data1D *subband(const enum WT_Subband &band) { return 0; }
   virtual Data1D *average_subband() = 0;
   // Return subband <band> at level <level>
   Data1D *subband(const enum WT_Subband &band,
      const int level) { return 0; }
   virtual const int size() const = 0;
protected:
   Data1D *_parent;
};
```

### 5.5.3 Data Subclass for Non-Transformed 2D Signals

This subclass for two-dimensional data provides additional methods for rows and columns access.

```
class Data2D : public Data {
public:
   Data2D() {}
   Data2D(Data2D *parent)
      :_parent(parent) {}
   // Visitor's accept method
   virtual void accept(DataProcessor *p);

   // Wavelet transform related method (as base class)
   virtual WT2D* GetWT() { return 0; }
   // Return subband <band> at current level
   virtual Data2D *subband(const enum WT_Subband &band) { return 0; }
   virtual Data2D *average_subband() = 0;
   // Return subband <band> at level <level>
   Data2D *subband(const enum WT_Subband &band, const int level) { return 0; }
   // Rows and columns access
   virtual const Data1D* row(int iRow) const;
   virtual Data1D *row(int iRow);
   virtual void setRow(int iRow, Data1D *row) {}
   // ...
protected:
   Data2D *_parent;
};
```

Notice the absence of data element access methods. At this level in the hierarchy, the type of the data coefficients is not known, and does not need to be known.

### 5.5.4 Templatized Data Subclasses

These classes declare the index-based interface for element access plus the factory methods for the creation of iterators. These classes play the role of the envelop in the ENVELOP AND LETTER idiom.

```
template<class Element>
class TData2D : public Data2D {
public:
   typedef Impl2DA<Element> Impl_t;
   typedef Element CoefType;
   // Constrution
   TData2D(int nRows, int nCols)
      :_impl(new Impl_t(nRows, nCols)) {}
```

```
   TData2D(const Element *coefficients, int nRows, int nCols)
      :_impl(new Impl_t(coefficients, nRows, nCols)) {}
   TData2D(const TData2D &rhs) { // ...}

   // Iterators factory methods
   virtual Iterator<Element> *CreateIterator() {
      return _impl->CreateIterator();
   }
   virtual RowIterator<Element> *CreateRowIterator() {
      return _impl->CreateRowIterator();
   }
   const int size() const { return _impl->Count(); }
   virtual int nRows() const { return _impl->nRows(); }
   virtual int nCols() const { return _impl->nCols(); }

   // Access elements
   const Element& operator()(int i, int j) const { return _impl->Get(i, j); }
   Element& operator()(int i, int j) { return _impl->GetRef(i, j); }

   // Access rows and columns
   // ...

   // Subband access
   virtual Data2D *average_subband() { return this; }

   // Acyclic Visitor's accept
   void accept(DataProcessor *ts) {
     if (Data2DGenericProcessor *p =
        dynamic_cast<Data2DGenericProcessor *>(ts)) {
        p->process(this);
     } else {
        // optionally call a catch-all function
        // ...
     }
   }
private:
   struct Impl_t;
   Impl_t *_impl;
};
```

### 5.5.5 Concrete Letters

Concrete templatized classes are defining storage layout and implementation's data structure, playing the role of the letter in the ENVELOP AND LETTER idiom. This class is application-specific.

```
template<class Element>
class Impl2DA : public Impl2D<Element> {
public:
   Impl2DA(int nRows, int nCols);
   Impl2DA(const Element *coefficients, int nRows, int nCols);

   // Factory methods for iterators creation
   Iterator<Element> *CreateIterator();
   RowIterator<Element> *CreateRowIterator();

   // Interface used by the iterator
   int Count() const;
   int nRows() const;
   int nCols() const;
   const Element& Get(int i, int j) const;

   // Element and substructure access
   Element& GetRef(int iRow, int iCol);
   const Data1D *row(int iRow) const;
   TData1D<Element> rowRef(int iRow);
   Data1D *row(int iRow);
   void setRow(int iRow, Data1D *data);
private:
   int _count;
   int _nRows;
   int _nCols;
   Element *_elements;
};
```

### 5.5.6 Iterators for Element Access

Iterators are used by clients (aplication algorithms) to access data sequentially. The external iterator `Impl2DA_Iterator` is written by an application developer to give access to application-specific data.

```
template<class Element>
```

```
class Impl2DA_Iterator : public Iterator<Element> {
public:
   Impl2DA_Iterator(Impl2DA<Element> *impl);
   virtual void First();
   virtual void Next();
   virtual Element &CurrentElementRef();
   virtual bool IsDone() const;
private:
   Impl2DA<Element>* _impl;
   int _iRow;
   int _iCol;
};
```

### 5.5.7 Row and Colums Iterators for 2D Data

For 2D data, it is also possible to iterate through the rows and columns of an image using iterators.

```
template<class Element>
class Impl2DA_RowIterator : public RowIterator<Element> {
public:
   Impl2DA_RowIterator(Impl2DA<Element> *data);
   virtual void First();
   virtual void Next();
   virtual Data1D *getCurrentRowPtr();
   virtual void setCurrentRowPtr(Data1D *row);
   virtual Data1D &CurrentRowRef();
   virtual TData1D<Element> CurrentRow();
   virtual bool IsDone() const;
private:
   Impl2DA<Element>* _data;
   int _iRow;
};
```

### 5.5.8 Transformed Data

Wavelet transforms are composed of subchannels, as illustrated in Figure 41. These subchannels can be of two kinds: transformed or non-transformed (data). The transformed channels are composed of subchannels which can either be *transformed* channels themselves or *data* channels, *i.e.* channels holding data.

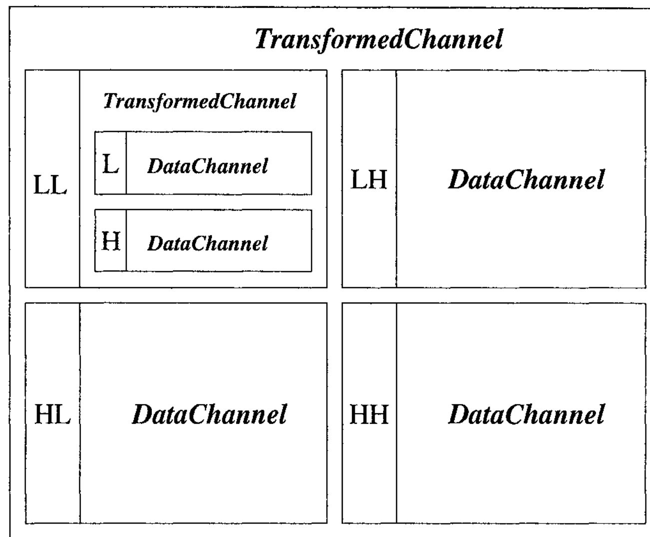



Figure 41: 2D Wavelet decomposition into subbands.

For example, a non-standard (square) wavelet decomposition on a image can be represented using a quadtree: every transformed channel has 4 children, the data channels (see Figure 42). As for the

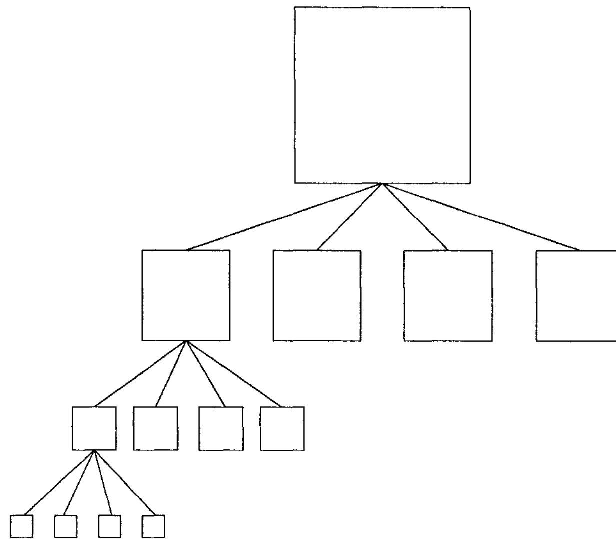


Figure 42: Quadtree representation of a two-level non-standard wavelet decomposition.

hierarchical representation, at first sight, the COMPOSITE design pattern (see Figure 50) seems to provide a possible solution, where Leaf is a DataChannel and Composite a TransformedChannel (or WaveletTransform). The Intent section says:
COMPOSITE

> Compose objects into tree structures to represent part-whole hierarchies. COMPOSITE lets clients treat individual objects and compositions of objects uniformly [24].

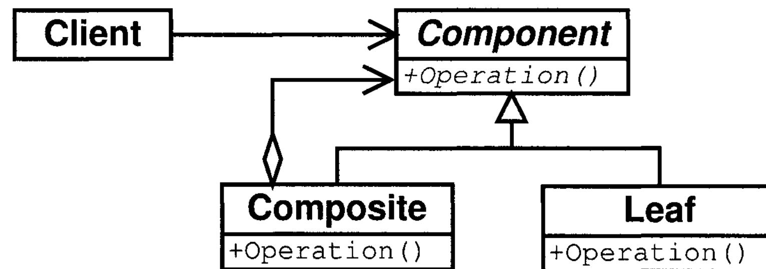



Figure 43: COMPOSITE design pattern [24].

Thus, in order to allow access to those subbands, the interface for 2D wavelet transform should contain operations to manage subbands across levels of a multi-level transform.

```
class WT2D : public Data2D {
public:
   typedef map<enum WT_Subband, Data2D *>::value_type valType;
   map<enum WT_Subband, Data2D *>::iterator iter;
  // ...
  // Interface for 2D transformed data
  void add(Data2D *aSubband, const enum WT_Subband &aSubbandName) {
     _subbands.insert(valType(aSubbandName, aSubband));
  }
  virtual WT2D* GetWT() { return this; }
  // Return subband <band>
  Data2D *subband(const enum WT_Subband &band) {
     return _subbands[band];
  }
  Data2D *average_subband() {
     return _subbands[Subband_L]->average_subband();
  }
  Data2D *low() { return _subbands[Subband_LL]; }
  void low(Data2D *subband) { _subbands[Subband_LL] = subband; }
  // Return subband <band> at level <level>
  Data2D *subband(const enum WT_Subband &band, const int level) {
     Data2D *ch = 0;
     if (level == _level) {
        ch = subband(band);
     } else if (level > _level) {
         for (iter = _subbands.begin(); iter != _subbands.end(); iter++) {
```

```
    // If the subchannel is a WTChannel, recursive call
    if ((*iter).second->GetWT() != 0)
       ch = (*iter).second->subband(band, level);
          }
       }
       return ch;
    }
private:
    int _level;
    int _offset_x, _nRows, _offset_y, _nCols;
    map<enum WT_Subband, Data2D *> _subbands;
};
```

This sample code simply illustrates the kind of operations present in a wavelet transformed data class.

### 5.5.9 Refining the Data Processing Framework

The Wavelet Transform Framework is a sub-framework created by adapting the Data Processing Framework discussed in the previous chapter. In this section we outline this adaptation of a general VISITOR-based framework. for a STAGGERED VISITOR-based framework, see figure 44.

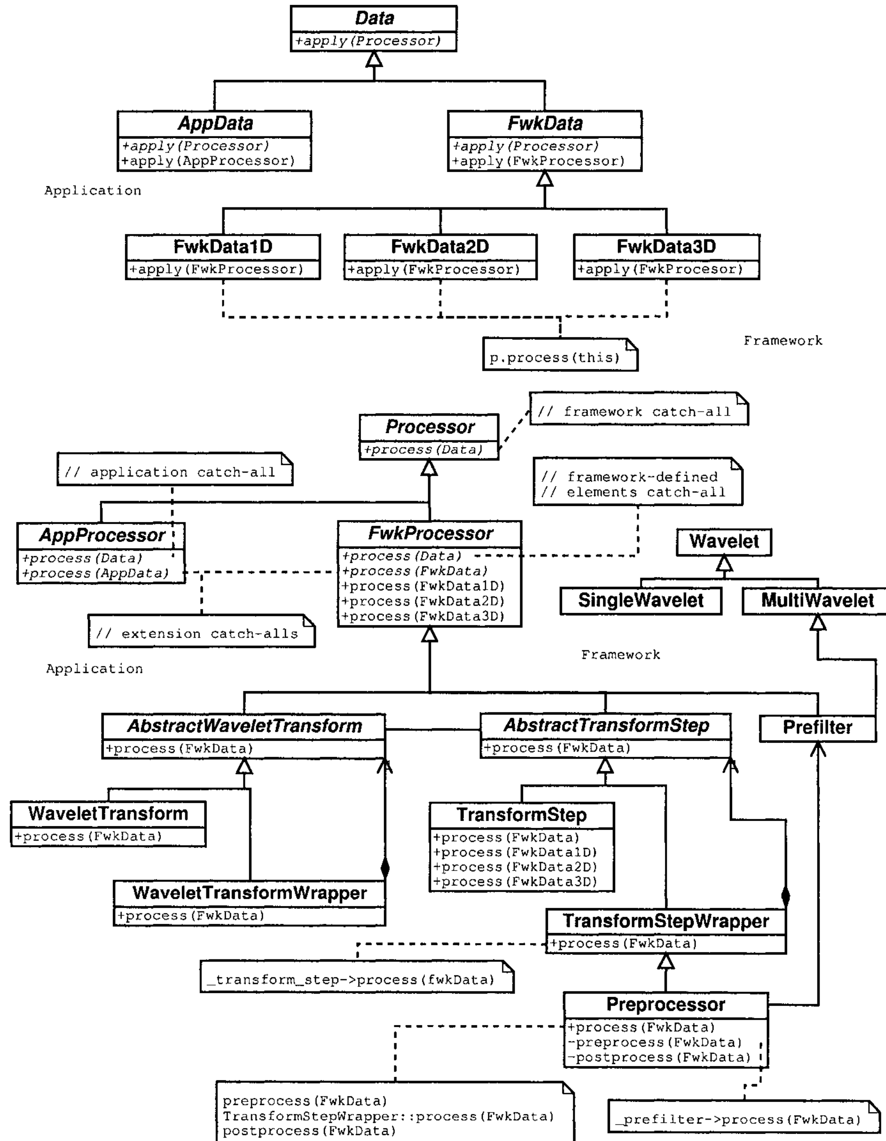



Figure 44: Overview of a STAGGERED VISITOR-based Wavelet Transform Framework.

#### 5.5.9.1 Framework Defined and Predefined Processor and Data Classes

The basis for the framework are the classic visitor base classes for data and processors:

```
// Main framework data class
class Data {
public:
   virtual void apply(Processor&) = 0;
};
// Main framework processor class
class Processor {
public:
   virtual void process(Data *e) {
      // Default empty implementation
   }
   virtual void process(Data1D* d) { // ...}
   virtual void process(Data2D* d) { // ...}
   // and so on for other concrete data...
   //...
};
```

#### 5.5.9.2 Transform Steps

Abstract wavelet transform steps are just another kind of processor.[3] An abstract wavelet transform step contains the hook method (and is therefore the hook class in the context of a wavelet transform):

```
// <<WaveImage>>, <<Sep-H:Transform>>, <<Strat-Strategy>>,
// <<WavTrfmStep-TransformStep>>
// <<Visitor-ConcreteVisitor>>, <<Decorator-Component>>
class AbstractWaveletTransformStep : public Processor {
public:
   virtual void process(Data* e) = 0;
};
```

Concrete wavelet transform steps implement the hook methods. This class is adaptable either via composition (*i.e.* an implementation supplier can be provided) or subclassing.

```
// <<WaveImage>>, <<Sep-H:WaveletTransform>>,
```

[3]In the code samples to follow, we use UML-F profile notation to express the role of the the classes.

```
// <<Unif-T:TransformStep>>, <<Strat-ConcreteStrategy>>,
// <<WavTrfmStep-TransformStep>>
// <<Visitor-ConcreteVisitor>>, <<Decorator-ConcreteComponent>>
class WaveletTransformStep : public AbstractWaveletTransformStep {
public:
    WaveletTransformStep(Processor* supplier = 0)
      :_supplier(supplier) {}
    void process(Data* e) {
       if (_supplier) {
          e->apply(*_supplier);
       } else {
          e->apply(*this);
       }
    }
   // Initialization method indicating the shape of a transform step
   void shape(const enum Shape &shape) { _shape = shape; }
private:
   // Adapter method: so that subclasses can implement
   // a method that is more meaningful.
   void process(Data2D* e) { // <<Unif-t:TransformStep>>
      transform_step(e);
   }
   // Hook/template method
   // <<Unif-h:TransformStep>>, <<Unif-t:2DTransformStep>>
   virtual void transform_step(Data2D* e) {
      switch(_shape) {
         case R:
            transformStepR(data);
         break;
         case C:
            transformStepC(data);
         break;
         case RC:
            transformStepRC(data);
         break;
         default:
            transformStepRC(data);
         break;
     }
  }
```

```
   // Hook (finest level)/ template (finest level of granularity)
   // <<Unif-h:2DTransformStep>>,
   // <<Sep-t:2DTransformStepAlgo>>
   virtual void transformStepR(Data2D* e) {
      // Default implementation
   }
   virtual void transformStepC(Data2D* e) {
      // Default implementation
   }
   virtual void transformStepRC(Data2D* e) {
      // Default implementation
   }
private:
   // Generic processor that can be used as a
   // service supplier
   Processor* _supplier;
   enum Shape _shape;
};
```

#### 5.5.9.3 Framework Predefined Default Transform Step

The framework is providing a default Concrete wavelet transform step.

```
// <<Application>>, <<Strat-ConcreteStrategy>>,
// <<WavTrfmStep-ConcreteTransformStep>>
// <<Visitor-ConcreteVisitor>>, <<Decorator-ConcreteComponent>>
class DefaultWaveletTransformStep
   : public WaveletTransformStep { // Framework defined default transform step
public:
   DefaultWaveletTransformStep() { // ...}
   virtual void process(Data2D *a) { // Default impl. }
   virtual void process(Data1D *a) { // default impl. }
};
```

#### 5.5.9.4 Transform Step Wrapper

A transform step can be decorated at run-time, *i.e.* new functionality can be added or removed to/from a transform step object. The added behavior can be provided by any kind of processor.

```
// <<WaveImage>>, <<Sep-H:Transform>>, <<Strat-ConcreteStrategy>>,
```

```
// <<WavTrfmStep-TransformStep>>
// <<Visitor-ConcreteVisitor>>, <<Decorator-Decorator>>
class WaveletTransformStepDecorator : public AbstractWaveletTransformStep {
public:
   WaveletTransformStepDecorator(Processor* preprocessor,
                                 AbstractWaveletTransformStep *step,
                                 Processor* postprocessor)
           :_preprocessor(preprocessor),
            _decoratedWaveletTransformStep(step),
            _postprocessor(postprocessor) {}
   // Other constructors...
   virtual void process(Data* e) { // ... }
      Preprocess(e);
      _decoratedWaveletTransformStep->process(e);
      Postprocess(e);
   }
protected:
   virtual void Preprocess(Data* e) {
      if (_preprocessor) {
         e->apply(*_preprocessor);
      } else {
         // Default behavior - NOTHING
      }
   }
   virtual void Postprocess(Data* e) {
      if (_postprocessor) {
         e->apply(*_postprocessor);
      } else {
      // Default behavior - NOTHING
      }
   }
private:
   AbstractWaveletTransformStep* _decoratedWaveletTransformStep;
   Processor* _preprocessor;
   Processor* _postprocessor;
};
```

#### 5.5.9.5 Application-Specific Transform Step Wrappers

It is also possible for application developpers to add application-specific wrappers (to extend the functionality of transform steps) by *subclassing* the framework generic transform steps wrapper. Extension is achieved by implementing the `preprocess` and/or `postprocess` hook methods as illustrated below: transform

```
class AppPreprocessor : public WaveletTransformStepWrapper {
public:
   AppPreProcessor(AbstractWaveletTransformStep* step)
     : WaveletTransformStepWrapper(step) {}
   virtual void preprocess(Data* e) {
      // Application-specific implementation
   }
};
```

#### 5.5.9.6 Application-Specific Transform Steps

It is interesting to note that application developers can focus on implementating only the hook method corresponding to the data dimension of their application. Below, only `transform_step(Data2D *)` is implemented.

```
// <<Application>>, <<Strat-ConcreteStrategy>>,
// <<WavTrfmStep-ConcreteTransformStep>>
// <<Visitor-ConcreteVisitor>>, <<Decorator-ConcreteComponent>>
class AppWaveletTransformStep
   : public WaveletTransformStep {
public:
   void transform_step(Data2D *a) {
      // Application-specific implementation
   }
};
```

Such a thing is made possible because of two methods provided by the transform step base class:

- *Adapter method*: A `process(Data2D *)` method performs adaptation for the `transform_step(Data2D *)` method.
- *Catch-all methods*: A catch all `process(Data *)` method taking care of *dispatching* to the correct method operating on data of interest to the application.

It is also possible for application-developers to implement only one or more fine-grained shape-specific transform methods. This is illustrated below.

```
class SingleShapeWaveletTransformStep
   : public WaveletTransformStep {
public:
   void transformStepR(Data2D *a) {
      // Application-specific implementation
      // for a transform step on the rows only.
   }
private:
   enum Shape shape() { return Shape::R; }
};
```

or in case multiple shapes are supported:

```
class MultiShapeWaveletTransformStep
   : public WaveletTransformStep {
public:
   void transformStepR(Data2D *a) {
      // Application-specific implementation
      // for a transform step on the rows only.
   }
   void transformStepC(Data2D *a) {
      // Application-specific implementation
      // for a transform step on the rows only.
   }
};
```

In order to have the desired method executed however, the application developer is responsible for indicating to the calling method in the framework which shape-specific method is to be executed. This can be done in two ways:

1. *In the transform step subclass*: By implementing the hook returning the desired shape. The framework class will ask the application class for the shape of the transform step. This is useful when the subclass is dedicated to a single type of transform. Once the class is written, no further initialization is necessary. This is shown in the above code sample.

2. *During the configuration phase*: During the initialization phase of the transform configuration, the application will place a request on the transform step to set a flag indicating to the template method which hook method to execute. The initialization is shown below.

These two situations are illustrated in the code fragment below.
In the first case, since the transform step is dedicated to a single shape, no initialization is necessary:

```
SingleShapeWaveletTransformStep *s = new App3WaveletTransformStep;
```

In the second case, the application has to specify which method should be called:

```
MultiShapeWaveletTransformStep *step = new App4WaveletTransformStep;
step->shape(C);
```

#### 5.5.9.7 The Abstract Wavelet Transform

In the previous sections, we have described the strategy side of the collaboration between a wavelet transform (as a context in the sense of the STRATEGY pattern) and its numerous possible strategies. Now we look at the context side of the transform, *i.e.* the class containing the template method capturing the commonality between many wavelet transforms. The abstract wavelet transform looks like:

```
// <<WaveImage>>, <<Sep-T:Transform>>, <<Strat-Ctxt>>, <<WavTrfm-Transform>>
// <<Visitor-ConcreteVisitor>>, <<Decorator-Component>>
class AbstractWaveletTransform : public Processor {
public:
   virtual void process(Data *e) = 0;
};
```

Notice how the process method is accepting the base data class as argument. This indicates the data structure independence of the wavelet transform algorithm.

#### 5.5.9.8 The Concrete Wavelet Transform

The WaveletTransform class is containing the process template method which captures the recursive nature of the transform algorithm.

```
// <<WaveImage>>, <<Sep-T:Transform>>, <<Strat-Ctxt>>, <<WavTrfm-Transform>>
// <<Visitor-ConcreteVisitor>>, <<Decorator-ConcreteComponent>>
class WaveletTransform : public AbstractWaveletTransform {
public:
   WaveletTransform(AbstractWaveletTransformStep *step = 0)
      : AbstractWaveletTransform(),
```

```
        _step( step ? step : new DefaultWaveletTransformStep) {}
    // Template method
    void process(Data* e) {
        // Stable structure of the wavelet
        // transform algorithm is captured here
        // (details omitted).
        _step->process(e);
    }
private:
    AbstractWaveletTransformStep *_step;
};
```

#### 5.5.9.9 Supporting Legacy Wavelet Transforms

We use the term "legacy" loosely to refer not necessarily to pre-existing wavelet transform code, but rather to refer to the kind of wavelet transform implementation that is found in many other systems. Thus, by *legacy wavelet transforms* we mean a wavelet transform for which extensibility is limited to what a minimal descriptor can hold *i.e.*, for each level of a multi-level transform, a *wavelet filter*, and, in the 2D case illustrated here, a *direction* (or shape):

```
enum DIRECTION {R, C, RC};
class Wavelet { // ...};
class Wavelet1 : public Wavelet { // ...};
class Wavelet2 : public Wavelet { // ...};

class LegacyDescriptor {
public:
    struct step {
    Wavelet *wavelet;
    DIRECTION dir; };
    step *_steps;
};
```

#### 5.5.9.10 Legacy 2D Wavelet Transform

A base class for legacy wavelet transform may look like this

```
class LegacyWaveletTransform : public AbstractWaveletTransform {
public:
```

```
    LegacyWaveletTransform()
        : AbstractWaveletTransform() {}
    void process(Data* d) {
        d->apply(*this);
    }
};
```

and a subclass for a 2D wavelet transform implements an `init(string config_info)` operation, which is using the configuration information passed as a parameter to initialize a transform descriptor. The format is irrelevant, as long as it contains the necessary information, which is a *wavelet type* and a *direction* for each level.

```
class LegacyWaveletTransform2D : public LegacyWaveletTransform {
public:
    LegacyWaveletTransform2D()
        : LegacyWaveletTransform() {}
    void init(string config_info) {
        _descriptor = config_info; /* For ex.: "W1 RC W2 C W1 R" */
    }
    // Template transform method
    void process(Data2D* d) {
        Wavelet *w1; Wavelet *w2;
        // ...
        transform_RC(d, w1);
        // ...
        transform_C(d, w2);
        transform_R(d, w1);
        // ...
    }
private:
    // Hook methods to be overridden in subclasses
    virtual void transform_R(Data2D *d, Wavelet *w) { // ...}
    virtual void transform_C(Data2D *d, Wavelet *w) { // ...}
    virtual void transform_RC(Data2D *d, Wavelet *w) { // ... }
    string _descriptor;
};
```

where the `process` method contains the switch logic and looping taking care executing a sequence of calls according to the information contained in the descriptor (details omitted in the above code sample).

#### 5.5.9.11 Creating Wavelet Transform Applications

In this section, we give three examples of how this prototypical framework can be adapted to create wavelet transform applications. A simple 1-level wavelet transform on 2D data can be build the following way:

```
// 1- Creation or acquisition of a 2D input signal
Data *data = new Data2D;
// 2- Creation and configuration of a wavelet transform
Processor *processor = new WaveletTransform(new AppWaveletTransformStep);
// 3- Launch the execution
processor->process(data);
```

A wavelet transform with pre- and post-processing applied to the transform step can be written like this:

```
Data *data = new Data2D;
// Creation of a wavelet transform processor
// where the transform step is extended
// to perform preprocessing.
Processor *processor =
 new WaveletTransform(
   new WaveletTransformStepDecorator(new Preprocessor, // Preprocessing
                                     new AppWaveletTransformStep,
                                     new Postprocessor));// Postprocessing
processor->process(data);
```

Creating and configuring a "legacy" wavelet transform application can be achieved in the following way:

```
Data *data = new Data2D;
string config_info = "W1 RC W2 R W1 C";
LegacyWaveletTransform2D *wt = new LegacyWaveletTransform2D();
wt->init( config_info );
Processor *processor = wt;
processor->process(data);
```

## 5.6 Design Notes

### 5.6.1 From the Ground Up: Towards More Flexibility and Extensibility.

In this section, we expose the requirements for the design of a wavelet transform and image processing system by following a systematic generalization approach. This means that we start with a simple and unflexible example (a specific, but overly simple application) and work our way towards a flexible framework.

Consider for example the basic paradigm involved in the task at hand: *transforming an image.* In object-oriented terms (or at least in C++), such a requirement could be captured by a single class `Image` having a single method `transform()` as an interface.

```
class Image {
public:
   void transform() {
      // Implementation of the
      // transform algorithm.
   }
};
```

It can hardly be any simpler than that. It can also hardly be less flexible and extensible than that. Adding a new operation (other than a transform) would require *changing* `Image`'s interface. Providing a different transform algorithm's implementation would require *changing* the current implementation:

```
class Image {
public:
   void transform() {
      // Modified implementation to provide
      // a more efficient transform algorithm.
   }
   void another_operation() {
      // Operation's implementation
   }
};
```

In both case, this is forcing a recompilation of the client code and increasing the risk of introducing errors.

A step towards more flexibility would be to take advantage of *polymorphism*, by making the operations already present in the interface virtual:

```
class Image {
public:
   virtual void transform() {
      // Base class default implementation of the
      // transform algorithm.
   }
   virtual void another_operation() {
      // Base class default implementation.
   }
};
```

This approach at least violates *less* the *Open-Closed* principle saying that module should be *open for extension* but *closed for modification*: Adding a new implementation for any of the methods in the image interface can now be achieved by subclassing, *i.e.* without touching the code in `Image` at all.

```
class MyImage : public Image {
public:
   void transform() {
      // My new implementation of the
      // transform algorithm.
   }
   void another_operation() {
      // My new implementation.
   }
};
```

But there are still two big problems with this approach:

1. Modifying *operations* incurs providing a different *image* subclass, even though the image representation itself may not be different at all.
2. Adding a new operation (semantically) *still* requires changing the base class interface.

The second problem is magnified if we want to add operations to a true hierarchy of images.

Since one of the flexibility requirement is to keep the transform algorithms flexible, it seems like a good idea to encapsulate the transform algorithm in its own class, `Transform`. If we do not do so,

then, in order to change the transform routine, one would need to modify the code in the `Image` class. This approach would not only lead to code duplication, but it would also contribute to increase the chance of error. In that context (*i.e.* a single class hierarchy), if one wants to keep flexible the transform algorithm, then two main avenues can be looked at:

- **Modify the code:**
  Modifying the `Image` class `transform()` method by adding conditional statements for the different transform algorithms supported. This approach is clearly prone to error. Why touch well working code (the `Image` class) when the transform and only the transform should change? The problem here is really that the *data representation* and the *transform process* are the same thing (at least, are in the same location: the `Image` class).

- **Subclassing:**
  Subclassing the `Image` class and overriding the `transform()` method. This amounts to creating different kinds of images when what one really wants are different transform algorithms. This is clearly not satisfactory. Another drawback is code duplication: the whole `transform()` method has to be overridden when only part of it should be changed. This is also error prone.

Extensibility requires that it should be easy to *add* functionality, to *add* new image processing algorithms. In the above example, this can be achieved by adding *new methods* to the `Image` interface. Why this can quickly become difficult and ugly can easily be seen if we have a *hierarchy* of images to account for the types of images supported by the framework.

A first design decision that has to be made is the separation of the *image representation* from the *transform algorithm*:

```
class Image {
public:
   void transform() {
      _transform->transform();
   }
private:
   Transform* _transform;
};

// A separate class encapsulating
// the transform algorithm
// (i.e. the concept that varies)
class Transform {
public:
   void transform() {
```

```
            // Implementation of
            // transform algorithm
        }
    };
```

This design is better with regards to isolating the concept that varies. However, if one wants to perform the transform differently, *i.e.* to use a different implementation, then the whole method `transform()` would need to be changed, even if the required change is only a well localized single *step* of the whole algorithm.
The change to the code would not be local anymore (in the `transform` method context). One way to ensure *locality of change* would be to (1) clearly identify the parts that vary, and (2) provide a design solution that support those flexibility requirements by encapsulating what varies.

Our goal is to keep a transform step (or sub-routine) flexible. The step is the part that varies. A possible way to support that flexibility (or *adaptability*) would be to follow the unification principle through the use of *template* and *hook* methods. Class `Transform` could contain a template `transform()` method and a `transform_step()` hook method. Then, if the transform step of the `Transform` was to be changed without touching the existing source code, one would do so by defining a subclass of `Transform` that simply overrides the `transform_step()` method. This elegantly changes the behavior of the `transform` method without duplicating or altering any source code.

This is an example of the TEMPLATE METHOD design pattern [24]. Basically the intent of the Template Method design pattern is to define the *skeleton* of an algorithm in an operation (here, `transform(...)`) in a base class, letting subclasses redefine some steps (here, `transform_step(...)`) of an algorithm, without changing the algorithm's structure.
In our domain analysis, we have identified, for example, two possible resulting shapes for a two-dimensional transform algorithm: square (non-standard) and rectangular (standard) transforms. We could consider having two *template* methods in the `Transform` class: `square_transform(...)` and `rect_transform(...)`. The *hook* method, tentatively named `transform_step(...)` could be a prefiltering operation for example. And since a prefiltering operation is needed only for a certain type of transform (the ones using multifilters), the default implementation could be left empty.

Consider a class `Data` representing the data to be transformed. The operations `transformA()`, `transformB()` and `transformC()` represent different transform algorithms. The problem here is that distributing all these operations across various classes leads to a system that is hard to understand, and hard to maintain. Adding a new transform algorithm requires recompiling all of these classes and their clients. It would be much better if each new transform algorithm could be added separatly, and if the data classes were independent of the operations that are applied on them, as we acknowledged earlier. This can be achieve by packaging transform operations in separate objects, in a separate class hierarchy (See Figure 46). This simplistic decomposition of the problem is based on the recognition

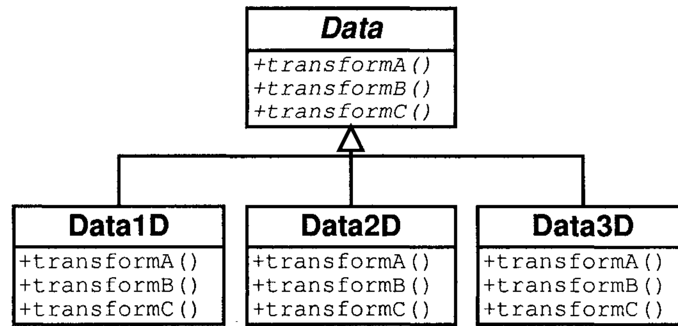



Figure 45: A data class hierarchy representing the data to be transformed with various transform algorithms.

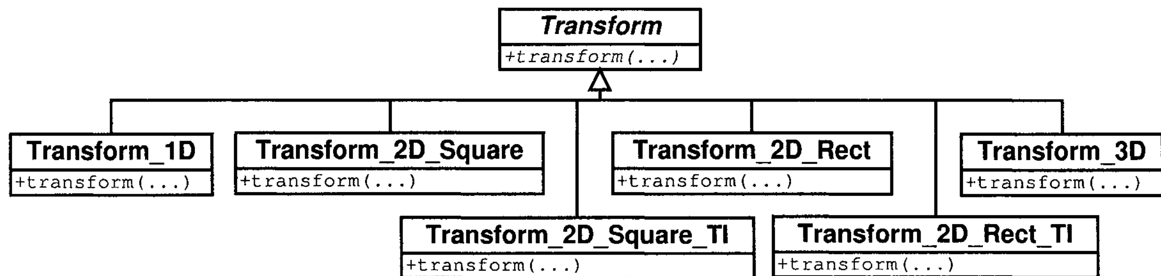



Figure 46: Transform hierarchy. Such a shallow and wide hierarchy indicates that more abstractions are still to be found.

of the following requirement: different applications use different wavelet transforms. A variety of applications differ in, among other things, the fact that they use different transforms. We now have two tasks to perform:

1. Study closely how the different transform algorithms differ, *i.e.* clearly identify what is their commonality and where they vary (variation-points).

2. Find a design solution supporting the variability requirements identified above.

Here we are concerned with the dimensionality of the signal on which the transform algorithm operates as well as with the dimensionality of the algorithm itself (*i.e.* the dependency of the algorithmic structure on the data structure type). We qualify transform algorithms operating on one-dimensional data as one-dimensional algorithms, and so on with other dimensions.
Notice that in Figure 46 we assist at an explosion of subclasses, since there is one subclass per transform algorithm. Such a shallow and wide class hierarchy, even though it gives us the flexibility of define new algorithm at will through subclassing, is hard to understand and to maintain.
A shallow and wide hierarchy may indicate that:

- More *abstractions* still have to be found, and
- Some classes could be removed through a clever use of *parametrization.*

Parametrization will be considered in the next section, where we discuss the shape (standard, or non-standard) that a transform can take. Now we will focus on refining the class hierarchy in Figure 46 following the guideline:

| **Guideline** | Class hierarchies should be fairly deep and narrow. Shallow and wide inheritance hierarchies indicate that abstractions still are to be found in the hierarchy |
|---|---|

In doing so however, we should be careful of *preserving* the abstractions that were identified in the domain analysis, or at least not violating them. Applications and application developers are often interested in algorithms and signals of only a single specific dimension at a time. It would therefore make sense to *group* the transform algorithms of same dimensionality together. Such a grouping can be achieved in object-oriented terms with base classes, one per dimension (see Figure 47). Doing so results in the hierarchy depicted in Figure 47. This approach has the advantage that adding a new algorithm's implementation for a specific dimension requires subclassing the base class of the appropriate dimension.

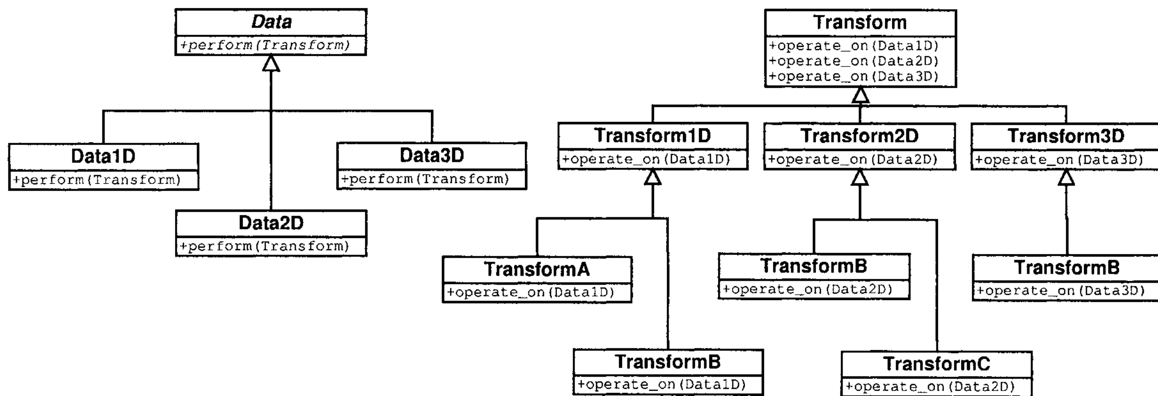



Figure 47: A data class hierarchy representing the data to be transformed with various transform algorithms. The new classes `Transform1D`, `Transform2D` and `Transform3D` capture an intuitive (and easy to learn) classification for application developers.

### 5.6.2 Standard and Non-Standard Wavelet Decomposition

Having narrowed down our hierarchy like that, haven't we get rid of a few categories that were identified during analysis, namely the *shape* of a transform (square and rectangular transforms)? Whenever parametrization can express the variability of a variation-point, we should use a parametrized

class instead of providing new alternative classes, since this is simpler. In doing so, we apply the following guideline:

| **Guideline** | Eliminate difference by parametrizing. Consider replacing classes or methods providing semantically equivalent behavior by common parametrized classes or methods. |
|---|---|

A closer analysis of a two-dimensional wavelet transform, shows us that parametrization can be used to get rid of classes encapsulating the shape of a transform. The parameter introduced describes the shape or type of the transform for *every level* of a multi-level transform. This approach results in greater flexibility for the transform since at each level, the transform can be of three types: square (transformation of columns and then rows), rectangle (rows only) and rectangle (columns only). This parametrization takes care of the *transform shape variation-point.* For example, consider the following:

```
class Transform2D {
public:
   // Instead of subclasses, we use a parameter to
   // select the desired implementation.
   void transform(ShapeParameter *sp) {
      // Switch logic to call the operation
      // corresponding to the <ShapeParameter> sp.
   }
private:
   // Different operations for different shapes,
   // instead of subclasses.
   virtual void transformRows() { //... }
   virtual void transformCols() { //... }
   virtual void transformRowsCols() { //... }
}
```

And the gain here is even more than we may think, since the resulting shape of a *multi-level* transform is the product of a recursive process for which the basic three operations above (`transformRows()`, `transformCols` and `transformRowsCols()`) are only the building blocks, at a single level. Wanting to represent each possible resulting shapes with a subclass is simply not possible since it depends on the shape of each single level of the transform and the composition of those in a multi-level transform.

### 5.6.3 Basis Functions, Wavelets and Filters

Another variation-point is that different basis functions, or wavelets (filter coefficients) may be used in different applications. Not only can the filter coefficients differ, but we can also have different *types* of wavelet coefficients: integer coefficients (lifting scheme), single wavelets (scalar coefficients), multiwavelets (matrix coefficients) and complex wavelet (complex coefficients).

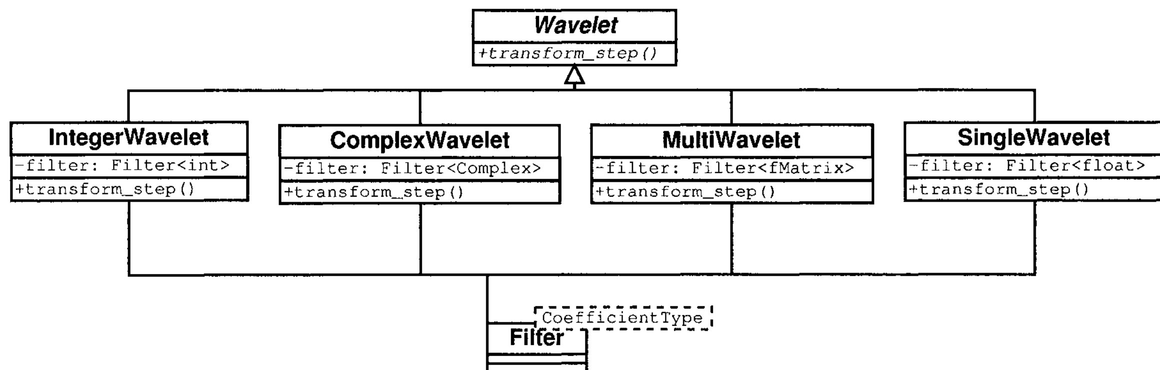



Figure 48: `Wavelet` class provides an interface for low-level transform operations only, *i.e.* those operations where the actual work is done with the filter: convolution with a signal (signals details omitted here). The different wavelet subclasses implement those operations since their implementation depends on the type of filter used. `Filter` is a template class acting as a container for various types of coefficients. The framework is also providing "helper classes" where coefficients types like matrix, complex numbers are defined as well as low-level default operations like `convolution` between a signal and a filter sequence.

### 5.6.4 Wavelet Transform Process

Performing a multi-level wavelet transform is a recursive process (see Figure 49). The recursive wavelet transform algorithm can be described in terms of his overall structure and actual processing steps.

```
class WaveletTransform {
public:
   CH *transform(CH *nt, TD *td) {
   //---------------------------------
   // Recursive structure of wavelet
   // transform algorithm.
   // This structure is common to all
   // wavelet transform algorithms,
   // independently of data structure
```

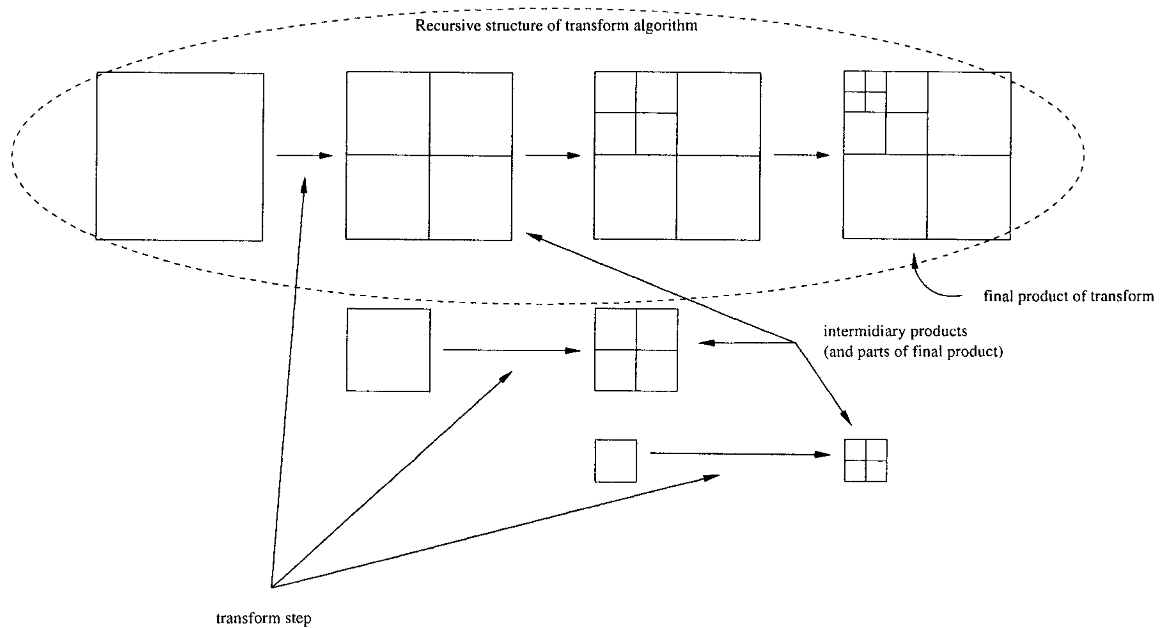



Figure 49: Wavelet transform recursive process yielding a recursive wavelet transform structure.

```
        // and filter used.
        //---------------------------------
        CH *wt = new CH;
        //---------------------------
        // Transform step:
        // Forward request to a
        // TransformStep object whose
        // responsibility is to perform
        // that transformation.
        //---------------------------
        _transformStep->transform_step(nt, td);
        //---------------------------
        // Recursive step
        //---------------------------
        if (td->next)
           wt->low( transform( wt->low() ,td->next) );
        return wt;
     }
     private:
        // How and when _transformStep pointer gets initialized?
        TransformStep *_transformStep;
```

```
};
```

The motivation behind delegating to a `TransformStep` object is simple: how such a transform step is implemented depends on at least two things, the *dimensionality* of the data and the *filter* used. Since the implementation of `WaveletTransform::transform(...)` is *data structure* and *filter* independent, it makes sense to keep it that way by encapsulating the concept that varies, the transform step:

```
class TransformStep {
public:
   CH *transform_step(CH *nt, TD *td) {
      // Implementation ???
   }
};
```

Ultimately, some useful work corresponding to a single-level wavelet transform has to be performed on the data *i.e.* somewhere some functions have to provide implementation that actually manipulates the data:

```
CH1 *transform_step(CH1 *nt, ...) {
   // Transform 1D signal
};
CH2 *transform_step(CH2 *nt, ...) {
   // Transform 2D signal
};
```

Let's focus on the 2D case first.

```
class TransformStep2D {
public:
   CH2 *transform_step(CH2 *nt, ...) {
      // Transform 2D signal:
      // There are three possible shapes
      // for a 2D transform: on the rows only,
      // on the columns only, and on the rows
      // followed by on the columns.
   }
};
```

A possible way to select the correct implementation (see code sample above) is to have three different *functions*:

```
class TransformStep2D {
public:
   CH2 *wt;
   CH2 *transform_step(CH2 *nt, TD *td) {
switch( td->shape ) {
         case WT2D_SHAPE_R :
            wt = transformR(nt, td->filter);
            break;
         case WT2D_SHAPE_C :
            wt = transformC(nt, td->filter);
            break;
         case WT2D_SHAPE_RC :
            wt = transformRC(nt, td->filter);
            break;
      }
      return wt;
}
private:
   // Helper functions performing the actual transform
   CH2 *transformR(CH1 *nt,...) { // ... }
   CH2 *transformC(CH1 *nt,...) { // ... }
   CH2 *transformRC(CH1 *nt,...) { // ... }
};
```

One way to get rid of that switch logic would be to apply the STRATEGY pattern (which replaces functions by classes). The context for a 2D transform step could then look like:

```
class TransformStep2D {
public:
   CH2 *wt;
   CH2 *transform_step(CH2 *nt, TD *td) {
      // No details on that Factory class for now...
      _transformStepAlgo = TransformStep2DAlgoFactory::instance()
     ->makeAlgo(td->shape)
      return _transformStepAlgo->transform_step(nt);
   }
private:
   TransformStep2DAlgo *_transformStepAlgo;
};
```

where we have the new hierarchy for 2D transform step family of algorithms:

```
class TransformStep2DAlgo {
public:
   virtual CH2 *transform_step(CH2 *nt) = 0;
};
class TransformStep2DAlgo_R : public TransformStep2DAlgo {
public:
   virtual CH2 *transform_step(CH2 *nt) {
      // Implementation of transformation on rows
   }
};
class TransformStep2DAlgo_C : public TransformStep2DAlgo {
public:
   virtual CH2 *transform_step(CH2 *nt) {
      // Implementation of transformation on columns
   }
};
class TransformStep2DAlgo_RC : public TransformStep2DAlgo {
public:
   virtual CH2 *transform_step(CH2 *nt) {
      // Implementation of transformation
      // on rows, then columns.
   }
};
```

This begs the question: How to bridge the gap between the `transform_step(...)` method and its implementation *i.e.* how to get the right code to execute when `TransformStep::transform_step(CH *nt, TD *td)` gets called? A possible solution is in via a VISITOR-based approach.

### 5.6.5 Wavelet Transform Data Representations

Different transform *processes* yield different transform *representations* (products) (see Figure 49). Processes and products are different. A wavelet transform consists of hierarchically structured information. A wavelet transform is *constructed* through a recursive process yielding a recursive hierarchical representation. How to represent such a hierarchical structure? With a *flat* array or a *composite* structure? *Do we gain anything in using a representation that reflects this hierarchical structure?* It all depends on the users or clients of wavelet transforms, which are the algorithms or processors operating on the transformed data.

Let us consider two possible options: a simple array of pixels and a composite structure (recursive composition). In the array-based solution, we may have something like this:

```
class WaveletTransform {
    int data[10][10]; // a bi-dimensional array
public:
    // Public interface
};
```

A flat representation like that can be implemented in many ways:

```
int *data;
vector<int> data;
valarray<int> data;
// and many other possible representations...
```

As for the hierarchical representation, at first sight, the COMPOSITE design pattern (see Figure 50) seems to provide a possible solution, where Leaf is a DataChannel and Composite a Transformed-Channel (or WaveletTransform). The Intent section says:
COMPOSITE

> Compose objects into tree structures to represent part-whole hierarchies. COMPOSITE lets clients treat individual objects and compositions of objects uniformly.[24]

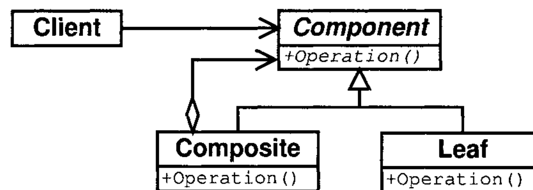



Figure 50: COMPOSITE design pattern [24].

We can think of Component as a generic Channel ([?]). The important question is whether we want clients to ignore the difference between compositions of objects and individual objects, *i.e.* treat them *uniformly*.

Let us consider the clients of wavelet transforms: the algorithms that operate on them.

```
void algorithm(WaveletTransform &wt);
```

Points to consider:

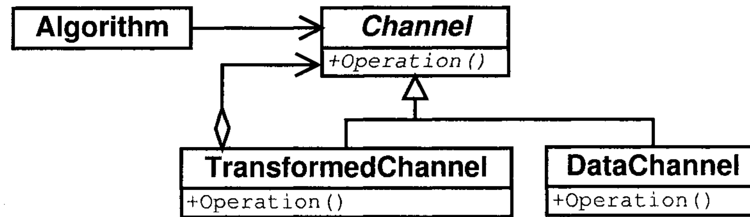



Figure 51: Wavelet transform data implemented using the COMPOSITE design pattern.

- It does not make sense to ask a non-transformed channel to inverse-transform itself. So, do we really want compositions and parts to share a common interface? Probably not.

- There are various particular wavelet transforms with *varying structure.* It is easy to understand that a COMPOSITE-based approach is *powerful* enough to represent all those transforms, but is it *expressive* enough to be able to differenciate compositions (or does it hide too much information)? For example, let's say an algorithm is expecting a non-standard transform as parameter. The signature of such an algorithm could be:

```
void algorithm(Channel &transform);
```

or

```
void algorithm(TransformedChannel &transform);
```

In both cases, there is no way to guaranty that the transform object passed as an argument will have the desired structure, *i.e.* that it will be a composition of a specific *type*, for there is no type information attached to a specific composition, for example, of a 2-level non-standard transform (*i.e.* with 4 subbands at each scale). This is a problem shared with the flat array-based approach. Unless there is a *type* corresponding to each possible decomposition, there is no way to ensure that the transform (composition) passed as an argument to an algorithm is of the right type, at least at compile-time[4].

### 5.6.6 Finding an Interface for Wavelet Transforms

In this section we address the problem of defining a *complete* and *minimal* interface for wavelet-transformed *data* also called wavelet transforms.

A *wavelet transform* can also be called a *hierarchical subband decomposition.* Sometimes, a wavelet transform refers to the *wavelet coefficients* only, excluding the *scaling* coefficients. Also, a wavelet

[4]We refer to compile-time type checking in the strong sense, and not in weak run-time "type" checking provided by *reflective* mechanisms where an object can provide information about itself when asked.

transform is sometimes referred to as the *transform coefficients*, which can include the scaling and wavelet coefficients or only the wavelet coefficients. Two important concepts are used to described a wavelet transform: *scale* and *subband* (or orientation). Each scale of a multilevel wavelet transform is composed of subbands. The orientation refers to the type of subbands across scales which, in the case of a 2D non-standard transform, are labeled $LH$, $HL$, $HH$ and $LL$, arising from the separable application of vertical or horizontal filters on a 2D signal. The latter ($LL$) being the lowest frequency subband, a representation of the information at all coarser scales. Depending on what one wants to achieve, a wavelet transform can be seen as a set of coefficients where each coefficient is treated as a distinct, potentially important piece of data regardless of its scale and subband or it can be seen as composed of subbands where the coefficients from a given subband (at a certain scale) are being grouped together suggesting that statistics computed from a subband are representative of the samples in that subbands [52]. Other concepts of importance, from the point of view of image coding, are the *orientation* and the *spatial location* of wavelet coefficients.

In defining a complete and minimal interface for a wavelet transform class, we have to take into consideration what the clients might reasonably want to do (with a wavelet transform) [see [39], Item 18]. The clients of wavelet transforms are algorithms operating on that type of data (*i.e.* taking a wavelet transform as input).

A typical transform-domain algorithm will want to perform the following:

- Access (coefficients belonging to) specific subbands. Some algorithm sub-routines might want to perform some operations on the coefficients that are specific to the subband from which the coeffcients are extracted. The coefficients themselves in a subband need to be accessed (and manipulated) and traversed in an algorithm-specific order.
- The subbands might need to be accessed and traversed following an algorithm-specific order across scales.

From what we just said, it seems that an important role of a wavelet transform interface is to allow access to its elements. So, regardless of the data structure used to implement a wavelet transform we can agree on a class interface might look like this[5]:

```
template< class T >
class WaveletTransform {
public:
    // Overloaded operator() for access
    // of data elements using indexing
    // for dimensionality supported: 1D, 2D and 3D
```

[5]Here we use a template class with the data type as an argument only to simplify the interface

```
        T operator()(int i);
        T operator()(int row, int col);
        T operator()(int sli, int row, int col);

        // Retrieval of a subband <sub> at scale <scale>
        T* subband(int sub, int scale);

    private:
        T* data;
    };
```

Obviously, in addition of being far from complete, there are many problems or weaknesses with such a minimal interface. One of the most striking problem is the `subband` function returning a pointer (an array) to a `T` object. To illustrate this point, let's consider a "typical" algorithm operating on a wavelet transform. We might have something like this:

```
template< class T >
void algorithm(WaveletTransform &wt) {

    // Assuming the interface provides
    // a way to make sure the wavelet transform
    // is a 2D one.
    int subband_LH = 1;
    int level = 2;

    // Assuming the interface provides
    // a neat way to get bound/size information
    T *subband = new T[size];
    subband = wt.subband(subband_LH, level);

    // Manipulation of subband coefficients
    for (int i=0; j<size; j++) {
        // Some operation on coefficient
        // subband[j] of type T
    }
};
```

It is virtually impossible for the algorithm to be written in a way that takes advantage of the hierarchical nature of the wavelet transform, a shape that is often present in the structure of the algorithm itself: algorithms taking advantage of the structure of the data structure, or, in other words, algorithms operating on structured data (structure) are often described in terms of how they

make use of that structure. It's just another way to say that algorithms are expecting the data they operate on to *be* of a certain structure. The interface proposed above is weak in the sense that it allows a user to lose precious *structural* information, by calling `subband` for example.

A wavelet transform is ultimately just an arrangement of basic *logical* structures such as *subbands* and *groups of coefficients*. These elements capture the total information content of a wavelet transform. The user should be able to refer to a subband as a whole having structure and not as an unstructured mass of coefficients. That helps make the interface simple, clear and *intuitive* for the user. An internal representation that has similar qualities should support the following:

- Maintaining the wavelet transform's logical structure, that is, the arrangement of subbands and groupings of coefficients.
- Mapping positions of coefficients on the transform to elements of the internal logical representation letting users access groups of coefficients belonging to a particular subband and accesing subband elements through coefficients at specific locations.
- Generate and present the transform in a format suitable for storage and visualization.

A common way to represent *hierarchically structured information* is through a technique called *recursive composition*. This technique has two important implications:

1. The objects in the composition need corresponding classes.
2. These classes must have compatible interfaces.

Those two implications require careful analysis since we want to keep the number of classes reasonable (do we want a subclass for all possible "type of" subband for 1D, 2D and 3D data?) and compatible interfaces to be meaningful (if non-transformed and transformed data classes have same base class, issues like being transformable (invertible) and having subbands need to be taken care of).

# Chapter 6

# The Image Processing Experiment Configuration and Scheduling Framework

## 6.1 Overview

This chapter describes the Image Processing Experiment Configuration and Scheduling Framework for the Wave ImAgE framework. The framework is described at both the framelet design level and at the framelet architectural level.

This framework proposes a design solution to the problem of supporting the flexible configuration of wavelet-based image processing experiments.

## 6.2 Context

This chapter assumes that the reader is familiar with Design Patterns [24][1] and with the data processing and wavelet transform frameworks in Chapter 4 and 5.
The Image Processing Experiment Configuration and Scheduling Framework implements a variant of the COMPONENT CONFIGURATOR pattern to support the flexible configuration of wavelet-based

[1] Particularly the COMPONENT CONFIGURATOR, MANAGER, ABSTRACT FACTORY, BUILDER, PROTOTYPE, STRATEGY and WRAPPER FACADE

image processing experiments through the dynamic configuration of abstract processors, data factories and strategies into an experiment. This framework enhances reuse by enabling abstract (and pluggable) factories (and the algorithms they embody) to be treated as interchangeable building blocks in wavelet-based image processing applications.

## 6.3 Problem

The design solution for the Configuration and Scheduling Framework should be influenced by the following *forces*:

- Since as an application matures new algorithms may be discovered and need to be tested, it should be possible to modify processors implementations at any point during an application's development. It should be possible to initiate, exchange, or shutdown a component dynamically within an application at run-time.
- Since a large number of applications are likely to be designed using the framework, a fixed application structure should be introduced to ensure consistency among the different programs. Treating an application as another object in the object model (as opposed to the widespread use of a `main()` function) should allow the sharing of common application logic since all applications in the system would be derived from the `Application` class.
- The system should provide and enforce a *common programing model* for all applications. Failing to provide and enforce such a common programming model may result in developers implementing their own unique versions of an execution model for each application they write, leading to hard to understand, hard to maintain and hard to reuse code. The system should therefore provide a simple, easy to learn but yet flexible programming model that all developers will adhere to. This could be accomplished by letting an Application class defining an interface providing a set of fundamental execution stages for any program.
- Wavelet-based image processing applications are characterized by algorithms or processors *operating on* data (images, transforms). It should therefore be possible for developers to reuse experiments in a way so that they could *operate differently* on the *same data*, or *operate (the same way) on different data.*

## 6.4 Solution

The patterns involved in this solution are, by order of importance with respect the variation-point semantics (*i.e.* what varies) MANAGER, SERVICE CONFIGURATOR, ABSTRACT FACTORY, FACTORY METHOD, PROTOTYPE, ADAPTER, STRATEGY, COMPOSITE, ITERATOR, and SINGLETON. Their role in this specific context are briefly described below:

1. MANAGER and SERVICE CONFIGURATOR: Components (processors and processor factories embodying processors as strategies) are configured dynamically into applications using a variation of the SERVIVE CONFIGURATOR pattern. The component repository of this pattern is implemented as a MANAGER.

2. ABSTRACT FACTORY and FACTORY METHOD: Combined together, they provide a *single* component that builds *related* objects. Concrete factories are dynamically linked into the system and the factory methods they implement allow for the creation of groups of semantically compatible processors.

3. PROTOTYPE: A concrete factory implements the PROTOTYPE pattern to allow for more flexibility in the creation of concrete factories.

4. STRATEGY: The transform steps and other processors are strategies. They implement algorithms performing the actual transform or processing on the data. Also, experiments (as a class, `Experiment`) are also strategies used in a context (role played by the `Experiment_Scheduler`).

5. COMPOSITE: `Experiment_Lineup` is a composite. It is composed of experiments to be executed sequencially.

6. ITERATOR: Iterators are used by the `Experiment_Scheduler` to iterate through the experiment lineup in order to execute them sequencially.

7. ADAPTER: Allow existing implementations not related by inheritance in the processor hierarchy to become reusable implementations.

8. SINGLETON: Many classes in this framework are meant to have only one instance (`Experiment_Scheduler`, `Component_Configurator`, `Options`, and `Environment` for example) because they provide global points of access. They are therefore implemented as singletons.

## 6.5 Configuration and Scheduling Sequence

In this section we present the steps performed during the configuration and scheduling of image processing experiments.

1. An initial configuration source (script file, command line, database, etc.) is created by an application developer. It then becomes the single location that can be managed centrally by applications.

   A sample configuration file, say `exp.conf`, for the configuration of two experiments, may look like this[2]:

[2]The exact syntax of the language for writing directives for the configuration of components is not important here, but it has to be taken care of eventually. Then, the INTERPRETER pattern could be used to implement a mechanism to parse and process these directives.

```
exp:
   /image_processing/processors/ processor1 ''lenna.pgm 3 ghm''

exp:
   /image_processing/factories/ factory2 ''RC daub R haar''
```

where what is important here the presence on some tag or divider for different experiment (exp), the location of the component (which can be a processor or a factory), an ID or name tag for the component to be identified, and finally, optionally, some extra parameters to be used to initialize the experiment.

2. A generic `main()` program is then in charge of performing the following activities:

   (a) Reading in and parsing the *framework-level* command line parameters providing the following information:

   - *Configuration source type*: text file, xml file, command line, etc.
   - *Configuration source name*: including the location of the source.
   - *Type of Experiment to perform*: lineup of experiment or task (configurable experiment).
   - *Application-specific parameters*: optionally, parameters to be used to configure the experiment(s).

   (b) Configuring the *configuration source manager* singleton with the configuration to be used.

   (c) Accessing the *experiment configurator* single instance and calling the `process_directives()` method on it. Using the `Configuration_Source_Manager`, it will parse and process the directives present in the config source to create and populate configuration information structures (`Experiment_Config_Info`) storing them in a container.

   (d) Launching the experiment scheduler by calling the `Experiment_Scheduler::run()` method.

   The *experiment scheduler* then performs the following actions:

   (a) Access the singleton `Options` to figure out whether to run a lineup or a generic experiment. In case of a lineup:

   - The *scheduler* uses a lineup iterator to iterate through the experiments in the lineup. It use the experiment's id to retrieve the configuration information stored in the configurator. Then the `process_directive()` method is called with the *configuration information* passed as arguments.
   - The *component configurator* then dynamically link the corresponding component, initializes it with the configuration information, and insert it into the *component repository*.

   In case of a generic experiment:

   The *scheduler* iterates through the `Experiment_Config_Info` objects, and for each one:

- It ask the *component configurator* to process the configuration info, which results in having the components loaded, initialized, and added to the *repository*.
- The *scheduler* creates and configures a task with the components found in the *repository*.

(b) The *scheduler* calls the `performExp` on the experiment.

## 6.6 Resolved Example: From the Ground Up

One of the force of the solution is to allow for flexible configuration of components making up image processing experiments. One element responsible for that flexibility is the *dynamic* configurability, *i.e.* the ability to dynamically load components into applications. We first look at the lower-level mechanisms that support run-time configurability.

#### 6.6.0.1 Dynamically Loading a Component

Let's say we have a base class `Component`, and we want to be able to dynamically load subclassed component object in the framework:

```
Component *component = new ...
```

Since the experiment configurator has no way to know the name of the class to load ahead of time, the component to be loaded has to provide to the configurator a way to let it create the *new* class. The component does so via a factory method (as in the FACTORY METHOD design pattern) returning a pointer to the new object (this is possible since this factory is located in the same file as the definition of the subclassed object).

```
Component *createComponent() {
   return new ConcreteComponent;
}
```

The framework still needs a receiver on which to place the `createComponent()` request. This is achieved by defining a global factory keeping information about concrete component via an associative map

```
typedef Component *maker_t();
extern map<string, maker_t *, less<string> > factory;
```

And concrete component can then register their factory method themselves in the global associative container in the following way:

```
Component *myComponent = factory[‘‘my component’’];
```

We ensure that the factory method will register itself automatically (this is called "self-registering objects") when the library is loaded by using a proxy class that will do just that:

```
class Proxy {
public:
   Proxy() {
      factory[‘‘my component’’] = createComponent;
   }
};
```

Since the registration occurs in the constructor, we need to ensure that one (and only one) instance of the proxy is created

```
Proxy p;
```

The rest consists in passing the `RTLD_NOW` flag to `dlopen` (see the DLL Wrapper Facade below) to cause `p` to be instantiated.
So the file containing the concrete component's implementation looks like this:

```
ConcreteComponent()::ConcreteComponent() {}
void
ConcreteComponent::operation() { // Implementation... }
// ...
extern "C" { // To resolve name mangling
   Component *createComponent() {
      return new ConcreteComponent;
   }
   // Proxy class whose sole purpose is to
   // register a maker for this concrete factory.
   class Proxy {
   public:
      Proxy() {
         // register the maker with the factory
         factory["ConcreteComponent"] = createComponent;
      }
   };
   // Instance of the proxy: concrete component will
   // now be registered with global component factory.
```

```
   Proxy p;
}
```

And it is now possible for the component configurator located in the framework to create concrete component

```
string componentName;
// ...
Component *concreteComponent = factory[ componentName ];
```

#### 6.6.0.2 A DLL Wrapper Facade for Dynamically Loading Libraries

The following wrapper facade encapsulates the low-level function calls responsible for the dynamic linking of shared libraries. It thus shields the client code from platform-specific details which results in simpler, more robust client code.

```
class DLL {
public:
   DLL(const string &dll_name) {
      handle_ = dlopen(dll_name.c_str(), RTLD_NOW);
   }
   ~DLL() {
      dlclose(handle_);
   }
   void *symbol(const string &symbol_name) {
      return dlsym(handle_, symbol_name.c_str());
   }
private:
   void *handle_;
};
```

### 6.6.1 Dynamically Loading Concrete Factories into Applications

As we saw from the previous two chapters, Wave ImAgE can become a heavily strategized framework very fast. Processors are strategies of other higher level processors providing the context of an algorithm execution. Such an aggressive use of the STRATEGY pattern can lead to a configuration nightmare for two main reasons:

- Managing a large number of strategies becomes harder as the number goes up.

- It becomes hard to ensure that groups of semantically compatible strategies are configured.

For example, two-dimensional transform step algorithms are not meant to operate on 1D signals. *Denoising* strategies are not meant to configure *compression* algorithms, etc. In order to solve this problem, we can use the ABSTRACT FACTORY pattern (see Figure 52) to consolidate multiple image processing experiment strategies into *compatible configurations.* The number of abstract factories is also more manageable and it allows applications to create their components without explicitely specifying their subclass name. The intent of the ABSTRACT FACTORY patterns says:

> Provide an interface for creating families of related or dependent objects without specifying their concrete classes.[24]

A general example of how to use abstract factories to create objects is given below. In that example, we want to reuse an existing implementation to create a wavelet transform in a 2D denoising image processing experiment:

```
// The implementation we wish to reuse
class MyEfficientWaveletTransform {
public:
   virtual void transform();
};
```

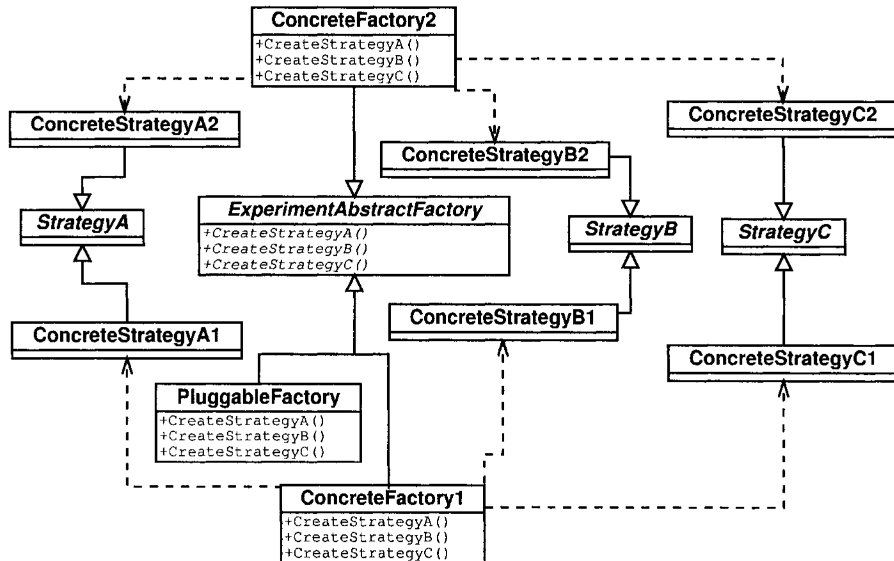



Figure 52: ABSTRACT FACTORY and PLUGGABLE FACTORY (based on the PROTOTYPE patterns to flexibly consolidate multiple strategies into semantically compatible configurations.

#### 6.6.1.1 The Framework-Provided Abstract Factory

The framework provides an ExperimentFactory base class:

```
class ExperimentFactory : public FwkComponent {
public:
   // Initialization and termination interface (not shown)
   // ...
   // Service-specific interface (factory methods)
   virtual Transform* createTransform() const = 0;
   virtual ImageProcessor* createImageProcessor() const = 0;
   // ...
};
```

Then this abstract factory can be subclassed to implement the factory methods for the creation of components for 2D denoising experiments for example:

```
class Denoising2DExperimentFactory : public ExperimentFactory {
public:
   // Initialization and termination interface (not shown)
   // ...
   // Service-specific (Abstract_Factory) functionality
   // declaring the interface of interest to the client
   virtual Data* getData() const;
   virtual Transform* createTransform() const;
   virtual ImageProcessor* createImageProcessor() const;
   // ...
};
```

where Data, Transform and ImageProcessor are considered as three different kinds of abstract products. For example, we can implement the createTransform() factory method like this:

```
Transform *
Denoising2DExperimentFactory::createTransform() const {
   // Creation of my reusable implementation
   MyEfficientWaveletTransform *transform =
      new MyEfficientWaveletTransform;
   // The adapter ''adapts'' the implementation
   ProcessorAdapter<MyEfficientWaveletTransform> *adapted_transform =
      new ProcessorAdapter<MyEfficientWaveletTransform>
```

```
            (transform, &MyEfficientWaveletTransform::transform);
        return adapted_transform;
    }
```

This factory can then be used to configure an application, as shown below:

```
    int main() {
        // 1- Initialization (i.e. creation of concrete factory)
        ExperimentFactory *factory = new Denoising2DExperimentFactory;

        // 2- Creation of semantically compatible component
        Data *data = factory->getData();
        Transform *wt = factory->createTransform();
        ImageProcessor *ip = factory->createImageProcessor();

        // 3- Usage of the components...
        // ...
    }
```

simply and clearly without having to mention the names of subclasses and without having to worry about the compatibility of the transform and the image processor since these are specified in a single, application-defined location: a concrete factory.

However, we want a framework to give us more opportunity for reuse than this simple dynamic linking mechanism. This is where the component configurator comes into play.

### 6.6.2 The Component Configurator

The configuration framework we designed is implementing a variant of the COMPONENT CONFIGURATOR pattern [3]. It supports the flexible configuration of image processing experiments whose processing strategies can be determined at run-time into experiments via a configuration source (command line arguments, configuration files, etc.). As a result, the abstract factories (and the algorithms they embody) can be treated as building blocks (see Figure 53). According to the COMPONENT CONFIGURATOR pattern [51], the configurator's responsibility is to coordinate the configuration of concrete components into an application. Using a component configurator, we can write a generic main program that now look like this:

[3]It is a variant in the sense that the scheduling functionality is reduced to initialization and termination and *re*-configuration functionality is absent.

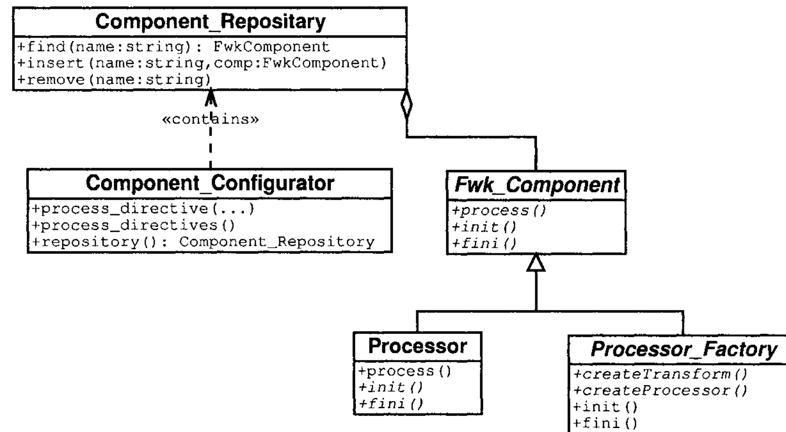



Figure 53: Class diagram for the COMPONENT CONFIGURATOR pattern.

```
int main(int argc, char *argv[]) {
   // Create the component configurator
   Component_Configurator *comp_config =
      Component_Configurator::instance();

   // The factory specified in argv[1] is linked
   // and added to the repository
   comp_config->process_directive(argv[1]);

   // Retrieval of the factory from the repository
   ExperimentFactory *factory =
      Concrete_Component<ExperimentFactory>::instance(argv[1]);

   // Usage of the factory to create the experiment components
   Data *data = factory->getData();
   TransformProcessor *transform = factory->createTransform();
   ImageProcessor *imgProcAlgo = factory->createProcessor();

   // Usage of the processors to operate on the data
   // i.e. to process the transformed data.
   Data *processedData =
      imgProcAlgo->process( transform->process( data ) );
};
```

Such a main program is now *generic* and *reusable* in the sense that, based on an *external* configuration source (a configuration file, or command line arguments, etc.), component factories are *dynamically* configured (loaded into the application). The factories are then used for the creation of data and processors, which are the basic components of image processing applications. Processors *implement* image processing algorithms operating on data.
There is a strong separation between a component implementation and configuration. Component developers are just providing implementations for the components, while the component configurator takes care of accessing the separated configuration information and dynamically configuring the components into applications. An application programmer can develop his code against the component interface without being concerned with the implementations, since it is simply not visible to the application programmer *which implementations* are currently loaded. In our example, a *group* of compatible implementations are loaded via a factory.
If we join the roles of *application* and *component implementation* developers, then what needs to be done is:

- Edition of a configuration source, and definition of concrete factories.
- Implementation of components.

Now, going back to our discussion of the liabilities of a VISITOR-based approach in Chapter 4, and more precisely about the expressiveness of applications, we ask the questions:

- What is and what goes into an application (as a class, *i.e.* an `Experiment`)?
- What is the meaning of an experiment lineup?

Without doing what Robert Martin calls *pattern abuse* in Chapter 14 of [37], we claim that there is a place for an *experiment as a class* to provide to an application developer some extra flexibility and freedom to implement a number of user-defined "helper" routines where they are needed. We see an experiment (as a class) as a *place* where code can be written and tested by a developer, probably before these elements of code are promoted to the rank of application-specific components. We illustrate. Consider a main program like the one below:

```
int main(int argc, char *argv[]) {
   // Create the component configurator
   Component_Configurator *comp_config =
      Component_Configurator::instance();
   // The factory specified in argv[1] is linked
   // and added to the repository
   comp_config->process_directive(argv[1]);
   // My experiment is created and executed
```

```
    Experiment *exp = new ProcessingExperiment;
    exp->performExp(argc, argv);
    return 0;
};
```

where ProcessingExperiment is a class now encapsulating the creation, access and usage of the concrete components to perform some useful work. In our case, Experiment could be a base class offering a template method performExp with init, perform and shutdown hooks that subclasses override and implement [4]

```
class Experiment {
public:
    void performExp(int argc, char *argv[]) {
        init(argc, argv);
        perform(argc, argv);
        shutdown();
    }
protected:
    virtual void init(int argc, char *argv[]) {
        processor = Concrete_Component<Processor>::instance(argv[1]);
    }
    virtual void perform (int argc, char *argv[]) = 0;
    virtual void shutdown() { // ... }
    AbstractFactory *factory;
    Processor *processor;
    TransformProcessor *transProc;
    ImageProcessor *imgProc;
};
```

where Concrete_Component is just a templatized class taking care of retrieving components form the repository and down casting them to application-specific types. Concrete experiments can either be a simple experiment using an unspecified processor:

```
class ProcessingExperiment : public Experiment {
public:
    virtual void perform (int argc, char *argv[]) {
        processor->process();
    }
};
```

[4] An additional string id() method *must* be implemented to return a unique id tag for the experiment. For example, it can simply be the name of the class: string MyApplication::id() const { return "MyApplication";}

or typical wavelet-based applications involving the transformation of a signal followed by the processing of the transformed data:

```
class TransformBasedExperiment : public Experiment {
public:
   virtual void init (int argc, char *argv[]) {
      // Get the components from the component repository
      TransformProcessor *transformPrototype =
         Concrete_Component<TransformProcessor>::instance(argv[1]);
      ImageProcessor *processorPrototype =
         Concrete_Component<ImageProcessor>::instance(argv[2]);

      // Configure a concrete ``pluggable'' concrete factory
      // with prototypical instance loaded earlier
      PluggableFactory plugFac(transformPrototype, processorPrototype);
      factory = &plugFac;
};

class TransformBasedExperiment1 : public TransformBasedExperiment {
public:
   virtual void perform (int argc, char *argv[]) {
      // ...
      factory->createTransform()->process();
      // ...
      factory->createProcessor()->process();
      // ...
   }
};
```

The flexibility (or ease of use for the application developer who does not have to implement a concrete factory subclass) gained from using the prototype factory has a cost: it becomes harder to enforce statically the consistency between processors. It is however possible to check for illegal combinations, see [66], and [64] for possible mechanisms to ensure compatibility.

Now that we have a mechanism to configure experiments, let us look a ways to schedule their orderly execution. In the next sections, we will give an overview of the scheduling subsystem of the Image Processing Experiment Configuration and Scheduling Framework.

### 6.6.3 Experiment Scheduling

Scheduling an experiment in our context refers to the activity of launching the execution of application lineups. The classes involved in this activity are shown in Figure 54. An *experiment scheduler*

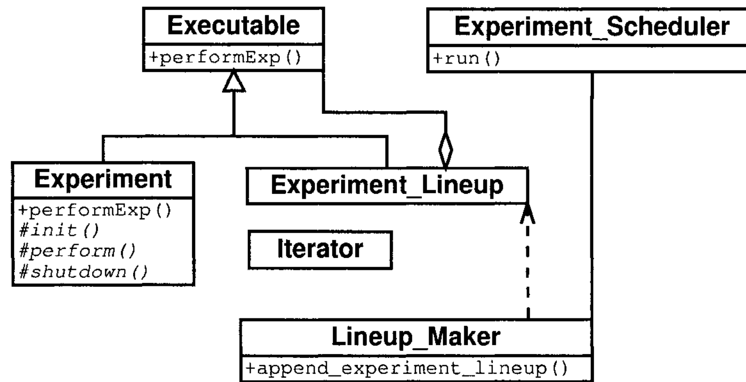



Figure 54: Class diagram for the experiment scheduling subsystem.

is responsible for the execution of image processing experiments.

#### 6.6.3.1 Experiments Lineups

Adding experiments to the experiment lineups amounts to creating a lineup maker class implementing the lineup building process in the `build_experiment_lineup()` method. A simple example is given below where three distinct experiments are created, the first two being part if the same composite experiment:

```
#include <Experiment_Lineup.H>

// Include the experiments to perform
#include <./NewAlgorithms/MyFirstExperiment.H>
#include <./OldAlgorithms/MySecondExperiment.H>
#include <./ImageStatistics/MyThirdExperiment.H>

void
build_experiment_lineup(Experiment_Lineup &aLineup) {
   Experiment_Lineup *subLineup = new Experiment_Application;
   subLineup->add( new MyFirstExperiment );
   subLineup->add( new MySecondExperiment );
   aLineup.append( subLineup );
```

```
    aLineup.append( new MyThirdExperiment );
}
```

where the header files included are containing the implementation of distinct image processing experiments.

## 6.7 The Configuration Framework: Putting It All Together

The UML diagram in Figure 55 contains the main classes involved in the configuration and scheeduling of image processing experiments. With these mechanisms into place, it is now possible to set the framework into motion by writing a generic main program that looks like this:

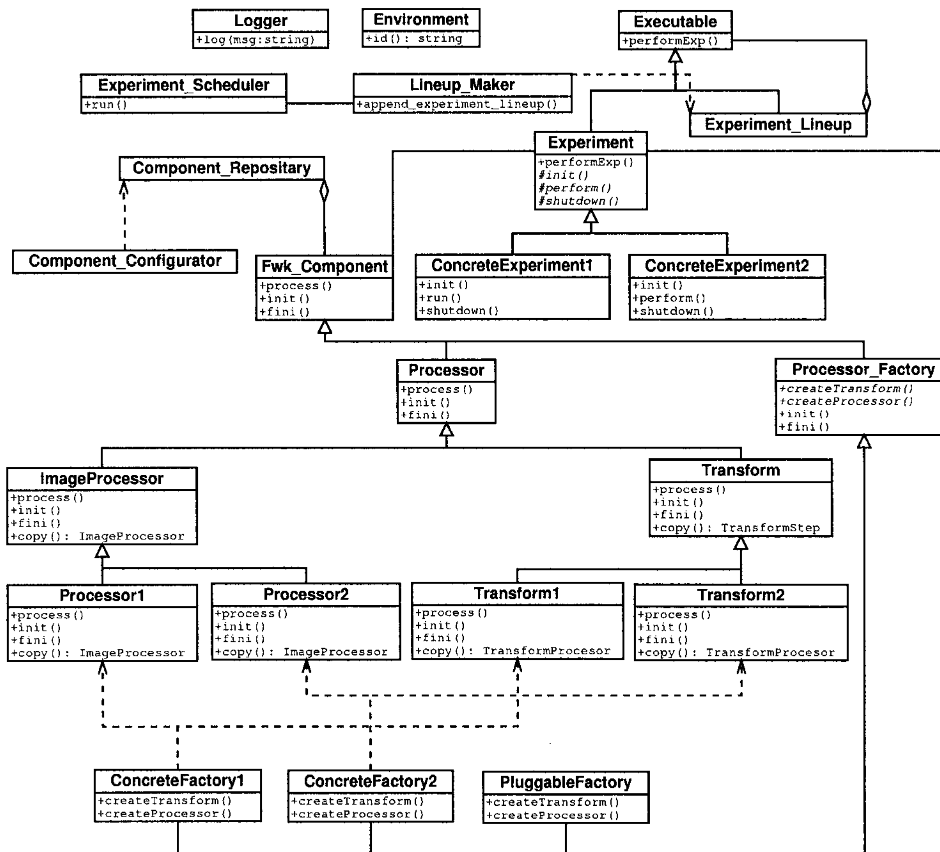



Figure 55: Class diagram for the Image Processing Experiment Configuration and Scheduling Framework.

```
#include <Component_Configurator.H>
```

```
#include <Exception.H>
#include <Experiment_Scheduler.H>
#include <Options.H>
#include <Configuration_Source_Manager.H>
#include <Config_Source_Factory.H>

// Main Program.
// Parse the framework level command line arguments
// and start up the Experiment scheduler.
int main(int argc, char *argv[]) {
   // Set up some default values
   Options::instance()->disable_lineup();
   Options::instance()->config_source_info("file", "config.conf");
   short result = -1;

   // Framework-level command line
   // arguments parsing & options set up
   Options::instance()->parse_args(argc, argv);

   // Create Configuration Source Manager configured
   // with source specified in command line
   // to use different types of configuration sources:
   // text, xml, remote or command line arguments.
   ConfigSourceFactory factory;
   ConfigManager::instance(
      factory.createConfigSource(
         Options::instance()->config_source_type(),
         Options::instance()->config_source_name()));

   // Create the configurator singleton
   Component_Configurator *comp_config = Component_Configurator::
      instance();

   // Launch the experiment scheduler
   try {
      // Launch the Experiment Scheduler
      result = Experiment_Scheduler::instance()->run();
   } catch(Exception::scheduler_failure &sf) {
      // Catch the exceptions...
   }
```

```
        return result;
    }
```

The *scheduler* run() method simply decides what kind of experiments the user wants to perform based on the _is_lineup_enabled option. Either a lineup of experiments or a generic task will be executed.

### 6.7.1 Generic Experiment and Application-Specific Experiment Lineup

The scheduler implements a run method that selects the course of action to take:

```
short
Experiment_Scheduler::run() const {
   short result;
   if (Options::instance()->is_lineup_enabled()) { // Run experiment lineup
      result = run_lineup();
   } else { // Run generic experiment (a semantic task)
      result = run_task();
   }
   return result;
}
```

The *scheduler* implements these two functions, run_lineup() to run the configured lineup, and run_task() to execute a generic (in the sense that it can be reused, because configured differently, to perform many different "applications") experiment.

#### 6.7.1.1 Configuring and Executing a Lineup

```
short
Experiment_Scheduler::run_lineup() const {
   // Get the available lineups
   Experiment_Lineup &lineups =
   Experiment_Lineups_Maker::instance()->lineups();
   // Iterate through the experiments in the lineup
   for(Experiment_Lineup::Experiment_Iterator I = lineups.begin();
       I != lineups.end(); I++) {
      // Set up the environment to current experiment
      // (for configuration and logging purposes -- currently
      // only a file logger is implemented: see future work section.)
```

```
        Environment::instance()->id( (*I)->id() );

        // The component configurator will process the configuration
        // information for this experiment by looking at the stored
        // configuration information created earlier from the config
        // source, initialize the component, and load them into the
        // repository.
        Component_Configurator::
           instance()->process_directive(Environment::instance()->id());

        // Run the application by calling the performExp template method
        (*I)->performExp();

        // Will call fini on components named in <id> configuration info
        // structure, remove them from repository and unlink them.
        Component_Configurator::
           instance()->process_directive(Environment::instance()->id());
     }
     return 0;
  }
```

The next section illustrates how a generic experiment can be configured.

#### 6.7.1.2 Configuring and Executing a Generic Experiment

We call image processing experiments that are generic *tasks*.

```
short
Experiment_Scheduler::run_task() const {
   // Will iterate through experiment configuration info
   ExperimentIterator I = Component_Configurator::
      instance()->createIterator();
   Experiment *task = new ExperimentTask;
   for ( I++) {
      if ( task->init((*I)) ) {
         task->run();
      }
      task->shutdown();

      // Load factory into repository
```

```
            Component_Configurator::instance()->process_directive((*I));
            // Run the experiment
            ExperimentTask::instance()->performExp();
            // Terminate components
            Component_Configurator::instance()->process_directive((*I));
         }
         return 0;
         delete task;
      }
```

where ExperimentTask is a singleton experiment using the component configurator to configure itself.

```
class ExperimentTask {
public:
   bool init(ExperimentConfigInfo &info) {
      // 1- If component referred to by the config info is
      // not already in the repository, load it, and (re-)initialize it
      // and add it to the repository.
      Component_Configurator::instance()->process_directive((*I));
      // 2- Retrieve the component from repository.
      _processor = // ...
      _data = // ...
   }
   void run() {
      _processor->process( data );
   }
private:
   Processor *_processor;
};
```

In the next section, we give an overview of the framework.

## 6.8 Framelet Constructs

The main framework constructs are shown in the table below.

| CONFIGURATION FRAMELET |
|---|
| ***Design Patterns*** |
| COMPONENT CONFIGURATOR, WRAPPER FACADE, ABSTRACT FACTORY, FACTORY METHOD, PLUGGABLE FACTORY, STRATEGY, SINGLETON |
| ***Framelet Interfaces and Base Classes*** |
| **Interfaces**:<br>*Executable*, *FwkComponent*<br>**Base Classes**:<br>*Experiment*, *AbstractProcessorFactory*, *AbstractProcessor* |
| ***Framelet Core Components*** |
| *Experiment_Scheduler*, *Experiment_Lineup*, *Lineup_Maker*<br>*Component_Configurator*, *Component_Repositary* |
| ***Framelet Default Components*** |
| *FileConfigSource* |

The roles played by each of the design pattern involved in the solution elements presented in this chapter are given below.

- A *Component Configurator*: Central administrative unit responsible of configuring concrete components into an experiment.
- A *Component Repository*: A container holding the component currently configured into an experiment.
- An *Abstract Experiment Factory*: Interface for all factories to be used by experiments to create the semantically compatible components they need to perform their image processing tasks.
- *Concrete Experiment Factories*: Implemented by application developer and packaged in DLLs, they implement factory methods for the creation of concrete components (implementation) without specifying the name of the concrete classes.
- A *DLL Wrapper Facade*: Wrap the low-level (OS) function calls to dynamically load shared libraries.
- A *Configuration Source Manager*: Singleton manager class providing a common interface to underlying concrete configuration sources of different types.
- An *Abstract Configuration Source*: Anstract interface defining the operations supported by all concrete sources.

- A *Default Concrete Configuration Source*: Currently, only text file configuration source are provided by default by the framework.
- A *Logger* singleton is responsible for the logging of messages to text files in a consistent and straightforward way.
- An *Environment* singleton is a globally accessible class holding information about the current experiment running. Used mostly for logging purposes (the destination file where logging is made) and to access configuration information for each individual experiment.
- An *Options* singleton responsible of parsing framework-level command line arguments and to offer a globally accessible options ...
- *Lineup Maker*: Class providing an interface for setting up an experiment lineup.
- *Experiment Lineup*: Composite representing a lineup (sequence) of executables.
- *Experiment Scheduler*: Manager class responsible to iterate through a list of experiments and run each of them sequencially after having configured them using the component configurator.
- *Experiment*: Environment where processors and data are configured and where the execution of processors operating on data is launched.
- *Executable*: An interface for all executable objects (experiments, lineups).

This concludes our presentation of the design elements involved in the configuration and scheduling framework.

# Chapter 7

# Conclusion and Future R&D Focus Areas

## 7.1 Conclusion

Before we even thought of developing an object-oriented framework, we were working on specific applications for image (2D) denoising using multiwavelets. Writing specific, well-targeted applications specialized in doing a single thing, down to the low-level details (multiplication of coefficients in a convolution function, thresholding of coefficients, image loading, etc.) is relatively easy (after all, we did it already, as our starting point...) and "down to earth" compared to the challenges represented by the development of a framework.

We then became interested in the development of another type of application, denoising (using a different denoising algorithm) of noisy images (a different kind of noise), using complex wavelet filter (different filter, different transform algorithm), and we were interested in expressing in code the similarity found in theory between those two setting. Modifying the code we were working with to be general enough to support both applications soon appeared to be a not so smart task: it either required duplicating lots of similar code, or rewriting everything from the beginning. In both cases, we would have ended up with more generic code, but still dedicated to only two applications... We found out that a framework solution would be more appropriate, and definitely, more interesting. This is what we did. Developing a framework being much harder than developing a single application (at least, the focus is shifted), there remains lots of areas where further developments are possible and desirable. Below, we name a few.

## 7.2 Future R&D Focus Areas

Given the depth and the breadth to which the development of a framework can lead in terms of domain analysis, design and implementation (let alone documentation), there are some areas that we originally planned to cover more extensively, that we would have liked to analyse, explore and design further but that, due to choices made early in terms of focus, as well as to very real time and space constraints, we were unable to tackle, either at all, or at the level we would have liked. Here they are, organized by category:

### 7.2.1 Extensions

- Adapters for the reuse of Matlab implementations. Due to the widespread use of Matlab in numerical and wavelets computations, it would be interesting to write wrappers for Matlab code for Matlab implementations to be reusable within the framework. Similar efforts have been done by the developers of the AOCS framework [20].

### 7.2.2 Support for processors

- We would like to push further the study of iterators, internal iterators, wrappers in the context of multi-paradigm programming (classic polymorphism and generic programming) in order to develop additional support in the framework for the writing of image processing algorithms.

### 7.2.3 Image Processing Functionality

Development of application-specific functionality like:

- Translation-Invariance: Even though that was originally the area that we were working on before thinking about building a framework, this functionality was ignored in the current framework. A variant of the LIGHTWEIGHT pattern could be used to represent translation-invariant transforms, which exibit redundancy.
- Denoising algorithms. Same as above. We started by working on applications for denoising 2D images using multiwavelets. But because of our focus on the architecture of the framework, we left aside any implementation of specific processor.
- Lifting scheme: integer wavelet transform is a targetted application. It would be interesting for an application developer to customize the framework for such a transform.

### 7.2.4 Sub-Frameworks and Other Subsystems

As mentionned in the core of the text, here are the other frameworks that we worked on and for which designs or protoypes were developed:

- Image handling. We designed a prototype, but it was not discussed nor included.
- Flexible logger that can log to different sources. A skeleton of this small framework is implemented, but we decided to leave aside for space and time constraints.
- Viewing subsystem.
- Memory management system.
- Architect exception handling into the framework.

# Bibliography


[1] M. A. Abidi and R. C. Gonzales, editors. *Data Fusion in Robotics and Machine Intelligence.* Academic Press, 1992.

[2] M. Aksit, B. Tekinerdogan, F. Marcelloni, and L. Bergmans. *Deriving Frameworks from Domain Knowledge*, pages 169–198. In M. E. Fayad [34], 1999.

[3] A. Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied.* Addison-Wesley, 2001.

[4] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++. An Introduction with Advanced Techniques and Examples.* Addison-Wesley, 1994.

[5] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1999.

[6] T. D. Bui, G. Y. Chen, and Y. Roy. Translation-Invariant Multiwavelets for Image Denoising. In *Proceedings of International Conference on Advances in Intelligent Systems and Computer Science*, pages 127–131, July 4-8 1999.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns.* John Wiley & Sons, 1996.

[8] G. Butler and P. Denommee. *Documenting Frameworks*, chapter 21, pages 495–503. In M. E. Fayad [34], 1999.

[9] G. Butler and L. Xu. Cascaded refactoring for framework evolution. *Proceedings of 2001 Symposium on Software Reusability, ACM Press*, 2001.

[10] R. Calderbank, I. Daubechies, W. Sweldens, and B.-L. Yeo. Wavelet transforms that map integers to integers. Technical Report 3, 1998. `http://cm.bell-labs.com/who/wim/papers/integer.ps.gz`.

[11] K. R. Castelman. *Digital Image Processing.* Prentice-Hall, 1996.

[12] K. W.. Cheung and L. M. Po. Low complexity 2-d preprocessing for discrete multiwavelet transform of image signals. Technical report, City University of Hong Kong, 1999. `http://www.image.cityu`.

[13] C. K. Chui. *An Introduction to Wavelets.* Academic Press, 1992.

[14] C. Cleeland, D. C. Schmidt, and T. H. Harrison. External polymorphism. In *Proceedings of the 3rd Pattern Languages of Programming Conference*, Allerton Park, Illinois, September 4-6 1996. `http://www.cs.wustl.edu/~cleeland/papers/`.

[15] I. Daubechies. *Ten Lectures on Wavelets.* SIAM, Society for Industrial and Applied Mathematics, 1992.

[16] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. Technical Report 3, 1998. http://cm.bell-labs.com/who/wim/papers/factor.ps.gz.

[17] G. Davis. Wavelet Image Compression Construction Kit, 1997. `http://www.cs.kuleuven.ac.be/~wavelets/`.

[18] EPITA. Applying Generic Programming to Image Processing. `http://www.lrde.epita.fr/people/thierry.geraud/papers/01-ai-slides.pdf`.

[19] M. Fontoura, W. Pree, and B. Rumpe. *The UML Profile for Framework Architectures.* Addison-Wesley, 2002.

[20] AOCS Framework Project. Design and prototype a component-based software framework for a satellite Attitude and Orbit Control System. `http://control.ee.ethz.ch/~ceg/AocsFramework/index.html`.

[21] F. Khendek G. Butler, P. Grogono. A Reuse Case Perspective on Documenting Frameworks. Taiwan, December 1-4. IEEE Computer Society Press.

[22] L. Gagnon and A. Jouan. Speckle filtering of sar images - a comparative study between complex-wavelet-based and standard filters. volume 3169 of *SPIE Proceedings*, 1997.

[23] L. Gagnon and F. Drissi Smaili. Speckle noise reduction of airborne sar images with symmetric daubechies wavelets. volume 2759 of *SPIE Proceedings*, 1996.

[24] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* John Wiley & Sons, 1995.

[25] T. Geraud, Y. Fabre, D. Papadopoulos-Orfanos, and J.-F. Mangin. Towards a Total Reusability of Image Processing Algorithms. Technical Report 9902, 1999. `http://www.ldre.epita.fr`.

[26] J. S. Geronimo, D. P. Hardin, and P. R. Massopust. Fractal functions and wavelet expansions based on several scaling functions. *Journal of Approximation Theory*, 78:373–401, 1994.

[27] D. P. Hardin and D. W. Roach. Multiwavelets prefilters i: Orthogonal prefilters preserving approximation order $p \leq 2$. *preprint*, 1997.

[28] B. Burke Hubbard. *The World According to Wavelets. The Story of a Mathematical Technique in the Making.* AK Peters, Wellesley, Massachusetts, 1996.

[29] ImageLib. An Image Processing C++ Class Library. `http://www.dip.ee.uct.ac.za/~brendt/srcdist/`.

[30] R. Johnson. Documenting Frameworks using Patterns. In *Proceedings of OOPSLA*, pages 63–76. New York:ACM/SIGPLAN, 1992.

[31] S. R. Jones. *A Framework Recipe*, pages 237–266. In M. E. Fayad [34], 1999.

[32] G. Kaiser. *A Friendly Guide to Wavelets.* Birkhauser, 1994.

[33] J. M. Lina and B. MacGibbon. Complex daubechies wavelets. *App. Comp. Harmonic. Anal.*, 2:219–229, 1995.

[34] R. E. Johnson M. E. Fayad, D. C. Schmidt, editor. *Building Application Frameworks: Object-Oriented Foundations of Framework Design.* Wiley, 1999.

[35] S. Mallat, editor. *A Wavelet Tour of Signal Processing.* Academic Press, 1998.

[36] R. C. Martin. Acyclic visitor. 1996. Available from `http://www.objectmentor.com/ressources/articleIndex`.

[37] R. C. Martin, editor. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall, 2002.

[38] Y. Meyer. *Wavelets – Algorithms and Applications.* SIAM, Society for Industrial and Applied Mathematics, 1993.

[39] S. Meyers. *Effective C++.* Addison-Wesley Professional Computing Series, 1998.

[40] S. G. Nikolov, D. R. Bull, C. N. Canagarajah, M. Halliwell, and P. N. T. Wells. Image Fusion Using a 3-D Wavelet Transform. In A. F. Laine and M. Unser, editors, *Image Processing and Its Applications*, number No. 465 in IEE Conference Publication, pages 235–239. IEE, 1999.

[41] POOMA. Parallel Object-Oriented Methods and Applications. `http://www.codesourcery.com/pooma/pooma`.

[42] W. Pree. *Framework Patterns.* SIGS Books, 1996.

[43] W. Pree. *Hot-Spot-Driven Development*, pages 379–394. In M. E. Fayad [34], 1999.

[44] W. Pree and K. Koskimies. Framelets: Small and Loosely Coupled Frameworks. *ACM Computing Surveys (CSUR)*, Issue 1, Vol 32(6), March 2000.

[45] D. Roberts and R. Johnson. Evolving Frameworks. A Pattern Language for Developing Object-Oriented Frameworks.

[46] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object Oriented Modeling and Design.* Prentice Hall, 1991.

[47] H. A. Schmid. *Framework Design by Systematic Generalization*, chapter 15, pages 353–378. In M. E. Fayad [34], 1999.

[48] D. C. Schmidt and F. Buschmann. Patterns, Frameworks, and Middleware: Their Synergistic Relationships. pages 694–704, Portland, Oregon, May 3-10 2003. IEEE Computer Society. `http://www.cs.wustl.edu/~schmidt/PDF/ICSE-03.pdf`.

[49] D. C. Schmidt and S. D. Huston. *C++ Network Programming: Resolving Complexity Using ACE and Paterns*. Addison-Wesley, Reading, Massachusetts, 2002.

[50] D. C. Schmidt and S. D. Huston. *C++ Network Programming: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, Reading, Massachusetts, 2003.

[51] D. C. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture - Patterns for Concurrent and Networked objects*. John Wiley & Sons, 2000.

[52] J. M. Shapiro. Embedded Image Coding Using Zerotrees of Wavelet Coefficients. *IEEE Transactions on Signal Processing*, 41(12):3445–3462, 1993.

[53] G. Strang and T. Nguyen. *Wavelets and Filter Banks*. Wellesley-Cambridge Press, 1997.

[54] V. Strela, P. N. Heller, G. Strang, P. Topiwala, and C. Heil. The application of multiwavelet filter banks to image processing. Technical report, MIT, USA, 1995.

[55] V. Strela and A. T. Walden. Signal and image denoising via wavelet thresholding: Orthogonal and biorthogonal, scalar and multiple wavelet transforms. Technical report, STATISTICAL SECTION TECHNICAL REPORT TR-98-01, Dept. of Mathematics, Imperial College of Science, Technology & Medecine, 1999.

[56] G. Succi, P. Predonanzi, A. Valerio, and T. Vernazza. *Sidebar 3, Frameworks and Domain Models: Two sides of the same coin*, pages 211–214. In M. E. Fayad [34], 1999.

[57] W. Sweldens. The lifting scheme: A new philosophy in biorthogonal wavelet constructions. In A. F. Laine and M. Unser, editors, *Wavelet Applications in Signal and Image Processing III*, pages 68–79. Proc. SPIE 2569, 1995. `http://cm.bell-labs.com/who/wim/papers/lift2.ps.gz`.

[58] W. Sweldens. The lifting scheme: A custom-design construction of biorthogonal wavelets. *Appl. Comput. Harmon. Anal.*, 3(2):186–200, 1996. `http://cm.bell-labs.com/who/wim/papers/lift1.ps.gz`.

[59] W. Sweldens. The lifting scheme: A construction of second generation wavelets. *SIAM J. Math. Anal.*, 29(2):511–546, 1997. `http://cm.bell-labs.com/who/wim/papers/lift2.ps.gz`.

[60] Taligent. Introducing Object-Oriented Frameworks. Taligent Inc., 1996.

[61] G. Uytterhoeven, F. Van Wulpen, M. Jansen, D. Roose, and A. Bultheel. WAILI: A software library for image processing using integer wavelet transforms. In K.M. Hanson, editor, *Medical Imaging 1998: Image Processing*, volume 3338 of *SPIE Proceedings*, pages 1490–1501. The International Society for Optical Engineering, February 1998.

[62] M. Vetterly and J. Kovacevic. *Wavelets and Subband Coding.* Prentice Hall PTR, Upper Saddle River, New Jersey, 1995.

[63] J. Vlissides, editor. *Pattern Hatching. Design Patterns Applied.* Addison-Wesley, 1998.

[64] J. Vlissides. Pattern Hatching. Pluggable Factory, Part II. *C++ Report*, February 1999.

[65] J. Vlissides. Pattern Hatching. Composite Design Patterns (They Aren't What You Think. *C++ Report*, June 1998.

[66] J. Vlissides. Pattern Hatching. Pluggable Factory, Part I. *C++ Report*, November-December 1998.

[67] J. Vlissides. Pattern Hatching. Visitor in Frameworks. *C++ Report*, November-December 1999.

[68] X. G. Xia, J. Geronimo, D. Hardin, , and B. Suter. Design of prefilters for discrete multiwavelet transforms. *IEEE Trans. on Signal Processing*, 44:25–35, 1996.