

D: A LANGUAGE FRAMEWORK FOR DISTRIBUTED PROGRAMMING

A Thesis

Presented to the Faculty of the Graduate School

of the College of Computer Science

of Northeastern University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Cristina Isabel Videira Lopes

November 1997

© 1996, 1997 XEROX Corporation. All rights reserved.

To my mother and the memory of my father.

Abstract

Two of the most important issues in distributed systems are the synchronization of concurrent threads and the application-level data transfers between execution spaces. At the design level, addressing these issues typically requires analyzing the components under a different perspective than is required to analyze the functionality. Very often, it also involves analyzing several components at the same time, because of the way those two issues cross-cut the units of functionality. At the implementation level, existing programming languages fail to provide adequate support for programming in terms of these different and cross-cutting perspectives. The result is that the programming of synchronization and remote data transfers ends up being tangled throughout the components code in more or less arbitrary ways.

This thesis presents a language framework called D that untangles the implementation of synchronization schemes and remote data transfers from the implementation of the components. In the D framework there are three kinds of modules: (1) classes, which are used to implement functional components, and are clear of code dealing with the aspects; (2) coordinators, which concentrate the code for dealing with the thread synchronization aspect; and (3) portals which concentrate the code for dealing with the aspect of application-level data transfers over remote method invocations.

To support this separation, D provides two aspect-specific languages: COOL, for programming the coordinators, and RIDL, for programming the portals. COOL and RIDL were designed to address the specific needs of the two kinds of aspects. COOL and RIDL can be integrated with existing object-oriented languages like Java, with little or no modifications to that language. COOL's coordinators and RIDL's portals compose with the classes through the classes' "aspect interfaces." Aspect interfaces are quite different than normal client interfaces but have some of the flavor of specialization interfaces.

D leads to programs whose modules are more focused and where the separation of concerns is more clear than it would be using traditional object-oriented languages. Often, D programs are smaller as well. D programs can be efficient — the performance penalty of the framework is very low. In alpha-user experiments, programmers reported not only that they understood the aspect interfaces and the aspect languages well, but also that, having classes, coordinators and portals, helped them to focus on different issues at different times, and that this was of great help in the development of applications.

Acknowledgments

This work would not have been possible without the guidance and friendship of my two advisors, Gregor Kiczales and Karl Lieberherr. They gave me the freedom and the means to mature my ideas while always pointing me in the right direction. Karl's persistent search for similarities taught me that a lot of good ideas come from simply connecting unlikely pieces of work. Gregor's profound understanding of science, engineering and human nature makes him a unique Master that, undoubtedly, will be the standard to which I will always strive. Furthermore, in arranging all the details of my close collaboration with the AOP group at Xerox PARC during the last two years, they made the improbable actually happen. To Gregor and Karl, my deepest gratitude.

I have also been fortunate in that all members of my proposal and thesis committee gave me continuous feedback during the writing of this dissertation. From Mitch Wand, Boaz Patt-Shamir and, earlier, from Will Clinger, I got precious comments that greatly improved the quality of my thesis. Mitch's messages in many email discussions that went on between PARC and CCS were always sharp and inspiring.

The AOP group at PARC contributed to the development of this work in all possible ways. John Lamping's intellectual brilliance and indestructible positive attitude contributed directly to clarifying many parts of D's semantics and implementation, as well as indirectly to shape my attitude towards my own and other people's work. The many discussions with Anurag Mendhekar about the formal semantics of programming languages were the closest I have been to that part of Computer Science. Anurag and Jean-Marc Loingtier were the two people that initially took my specifications of D, my prototype implementation and my informal explanations and implemented a usable version of the framework. Later, John, Gregor and Venkatesh Choppella joined the implementation effort. While I was busy finishing the dissertation, the whole team produced a shippable language framework that is going to be in alpha-use for the next few months. Gail Murphy conducted the summer experiment and gave me invaluable feedback on chapter 5. I must also acknowledge John Irwin and Chris Maeda, two former members of the AOP group, who, although not directly involved in the D project, contributed to many interesting discussions. Last but not least, the four summer students, Beth Seamans, Tatiana Shpeisman, Mark Marchukov and Jared Smith-Mickelson did a fantastic job in writing applications in DJava when it was still an experimental programming language environment.

Also at PARC, the Embedded Computation Area, of which the AOP group is only one part, provided me a highly stimulating working environment, where the conversations flew freely from object-oriented languages to micro-electromechanical systems. The Systems and Practices Laboratory and the whole PARC atmosphere made my last two years a continuous and intense learning experience.

Back at Northeastern, the group of graduate students of three years ago was also a very dynamic group. Walter Hürsch, Salil Pradhan, Linda Seiter, Nacho Silva-Lepe, Greg Sullivan and Cun Xiao made the Demeter seminars interesting forums of discussion. Today, Karl's graduate students seem to continue that dynamics; besides evolving Demeter, they have also implemented D based on early drafts of this dissertation.

The members of the administrative staff both at PARC and at CCS also contributed for making this work possible.

The five years of the PhD program were not only rich in technical learning, but they were crucial for my personal growth. The isolation from everything and everyone I always knew, and a sequence of events, of which the death of my father in April of 94 was the first and most devastating, threw me into very tough times of soul searching. I might not have stayed sane if it wasn't for the new friends I made along the way: Linda Seiter and her husband Jim, Walter Hürsch, Martin Spit, Jeff McAffer, Henrique and Isabel Martins, Ivan Krsul and Jean-Marc Loingtier. Olivier Coudert, an unlikely companion, was with me through the joys and frustrations of the final year; besides the emotional support, he also gave me lots of good ideas for some parts of the dissertation.

My long time friends Júlia Allen and Miguel Silveira were responsible for attracting me to Boston in the first place. But I would not have come if it wasn't for the support from Zé Carlos Monteiro and our mutual interest in engaging in new adventures.

My mother, late father, sisters, brother and nephews, in a total of seventeen people, are the close family with whom I stayed connected by the telephone and through Christmas get-togethers. My determination in getting a PhD is due, in part, to the challenge of being part of this dynamic, outgoing family.

Financial support for this research was provided in part by a grant from the Portuguese Junta Nacional de Investigação Científica e Tecnológica, through its programs Ciência and Praxis, and, in part, from a grant from the College of Computer Science at Northeastern University. This work has also been partially supported by the National Science Foundation under grant numbers CDA-9015692 (Research Instrumentation) and CCR-9402486 (Software Engineering).

Table of Contents

1. INTRODUCTION	2
1.1. The Problem	4
1.2. The Approach	5
1.3. The Thesis	6
1.4. Synopsis of the Dissertation	7
2. CODE TANGLING	10
2.1. How Programs Become Tangled	12
2.1.1. Implementing the Functionality	12
2.1.2. Adding Synchronization Constraints	13
2.1.3. Providing for Remote Access	15
2.2. The Source of Tangling	20
2.3. Tangling and Programming Practices	22
2.3.1. Concurrency	22
2.3.1.1 Units of Synchronization.....	22
2.3.1.2 Splitting Classes.....	27
2.3.1.3 Splitting Locks.....	28
2.3.1.4 Coordination State	28
2.3.1.5 Coordination by Subclassing	30
2.3.2. Communication.....	32
2.3.2.1 Splitting Parts	32
2.3.2.2 Class Transformations.....	33
2.3.2.3 The Serializer Design Pattern.....	34
2.3.3. Summary.....	34
2.4. Tangling and Programming Languages	35
2.4.1. Basic Linguistic Support for Distributed Programming.....	35
2.4.1.1 Synchronization	35
2.4.1.2 Communication.....	36
2.4.2. Concurrency in Object-Oriented Languages.....	37
2.4.2.1 Orthogonal Approach.....	37
2.4.2.2 Full Integration	39
2.4.2.3 Separation of Coordination and Functionality.....	42
2.4.3. Communication in Object-Oriented Languages.....	45
2.4.3.1 Orthogonal Approach.....	45

2.4.3.2 Full Integration	46
2.4.3.3 Hybrid Approaches.....	48
2.5. Final Remarks	49
3. THE D FRAMEWORK.....	52
3.1. Design Principles.....	54
3.1.1. Separation of Concerns: Identification of Aspects	54
3.1.2. Control over the Separation	55
3.1.3. Integration with Existing Languages	56
3.2. Specification of the Languages.....	57
3.2.1. Conventions and Notation	57
3.2.2. The Component Language.....	58
3.2.2.1 Types, Values and Variables	58
3.2.2.2 Classes	59
3.2.2.3 Creation of Threads.....	59
3.2.2.4 The Meaning of Objects, Threads and Execution Spaces	60
3.2.3. The Visible Elements of Components	60
3.2.4. The Coordination Aspect Language.....	61
3.2.4.1 Visible Elements of Classes.....	63
3.2.4.2 Coordinator Declaration.....	63
3.2.4.3 Coordinator Body.....	65
3.2.4.4 Condition Variables	66
3.2.4.5 Ordinary Variables.....	67
3.2.4.6 Self-Exclusive Methods.....	67
3.2.4.7 Mutual Exclusion Declarations.....	68
3.2.4.8 Method Managers	71
3.2.4.9 Guarded Suspension.....	71
3.2.4.10 On Entry and On Exit Statements	73
3.2.4.11 Some Examples of Coordinators.....	74
3.2.5. The Remote Interface Aspect Language	76
3.2.5.1 Visible Elements of Classes.....	77
3.2.5.2 Portal Declaration	77
3.2.5.3 Portal Body	78
3.2.5.4 Remote Operations.....	78
3.2.5.5 Object Transfer Specifications.....	81

3.2.5.6	Type Transfer Specifications	82
3.2.5.7	Passing Global References.....	82
3.2.5.8	Passing Copies	83
3.2.5.9	Copying Directives.....	83
3.2.5.10	Classes that Must Have a Portal.....	88
3.2.5.11	Some Examples of Portals	88
3.2.6.	Interaction between Aspects and Class Inheritance	89
3.3.	Design Decisions and Alternatives	91
3.3.1.	General	91
3.3.1.1	On Modules, Components, Aspects and Interfaces.....	91
3.3.1.2	The Need for Special Abstractions and Composition Mechanisms.....	95
3.3.1.3	Who Drives Who	96
3.3.2.	The Component Language.....	96
3.3.2.1	Class-based Language vs. Prototype-based Language	96
3.3.2.2	Uniform Reference Semantics	97
3.3.2.3	Threads.....	97
3.3.2.4	The Default Synchronization.....	98
3.3.2.5	The Default Communication	99
3.3.3.	COOL	99
3.3.3.1	Coordination: Classes vs. Abstract Method Sets	99
3.3.3.2	Granularity of Synchronization	101
3.3.3.3	Synchronization: Local vs. Distributed	103
3.3.3.4	Exclusion Constraints	103
3.3.3.5	Assignments	103
3.3.3.6	Current Limitations.....	104
3.3.4.	RIDL.....	104
3.3.4.1	Remote Interaction: Classes vs. Abstract Types	104
3.3.4.2	Granularity of Remote Interaction	105
3.3.4.3	On the Semantics of <code>gref</code>	107
3.3.4.4	The Copying Directives.....	107
3.3.4.5	The Missing Parts	108
3.3.4.6	Current Limitations.....	109
3.4.	Final Remarks	109
4.	IMPLEMENTATION	112

4.1. Engineering the Implementation Space	114
4.2. Target Architectures for Implementing COOL	115
4.2.1. Coordinator Objects.....	116
4.2.2. Mutual Exclusion and Re-entrance	117
4.2.3. Requires Clause.....	119
4.2.4. Access to Variables of the Coordinated Objects.....	119
4.2.5. Per Class Coordination	120
4.2.6. Inheritance of Aspect Code.....	120
4.2.7. Examples.....	121
4.2.7.1 Simplest Case: one class, per_object coordination, no inheritance	121
4.2.7.2 Inheritance of Coordination.....	126
4.2.7.3 Overriding of Coordination	128
4.2.7.4 Multi-class Coordination (per_class)	132
4.3. Target Architectures for Implementing RIDL.....	134
4.3.1. Run-time Architecture	135
4.3.1.1 Application-level Proxies for D's Remote Objects	136
4.3.1.2 Portal Objects and their Proxies.....	137
4.3.1.3 Traversals and Traversal Classes.....	138
4.3.2. The Name Service	139
4.3.3. RIDL's Data Transfer Protocols.....	139
4.3.3.1 Passing Primitive Data	139
4.3.3.2 Passing Global References.....	140
4.3.3.3 Passing Copies	141
4.3.4. Examples.....	142
4.3.4.1 Arguments of Primitive Types and gref.....	142
4.3.4.2 Passing Copies	148
4.3.4.3 Inheritance of Portals	154
4.4. Integrating COOL and RIDL	155
4.5. Summary	156
5. VALIDATION	158
5.1. Case-Studies	160
5.1.1. The Bounded Buffer	162
5.1.2. The Dining Philosophers	163
5.1.3. The Shape	164

5.1.4. Concurrent Matrix Multiplication.....	165
5.1.5. Concurrent Graph Traversal.....	166
5.1.6. Assembly Line.....	168
5.1.7. Distributed BookLocator/PrintService.....	170
5.1.8. Distributed Text Editor.....	171
5.1.9. Distributed Document Service.....	173
5.1.10. Message Queue.....	176
5.1.11. Analysis.....	177
5.1.11.1 LOC.....	177
5.1.11.2 Aspectual Bloat.....	178
5.1.11.3 Methods Affected by Aspect Code.....	178
5.1.11.4 Tangling Ratio.....	179
5.1.11.5 Analysis of the Results.....	180
5.2. Performance.....	182
5.3. Preliminary User-Studies.....	184
5.3.1. The Summer Experiment.....	185
5.3.2. The Applications.....	185
5.3.2.1 The Space War.....	186
5.3.2.2 The Space War - Java and Sockets.....	189
5.3.2.3 Distributed Library System.....	189
5.3.3. Alpha-Users' Reports.....	190
5.4. Final Remarks.....	191
6. CONCLUSIONS.....	194
6.1. Summary.....	195
6.2. Contributions.....	196
6.3. Future Work.....	198
6.4. Conclusion.....	198
APPENDIX A. SYNTAX.....	201
APPENDIX B. DJ PRIMER.....	207
APPENDIX C. THE ASPECT WEAVER.....	229
APPENDIX D. DJ LIBRARY CLASSES.....	255
APPENDIX E. USER REPORTS.....	269
BIBLIOGRAPHY.....	277

Chapter 1

Introduction

“To my taste the main characteristic of intelligent thinking is that one is willing and able to study in depth an aspect of one’s subject matter in isolation, for the sake of its own consistency, all the time knowing that one is occupying oneself with only one of the aspects. The other aspects have to wait their turn, because our heads are so small that we cannot deal with them simultaneously without getting confused.”

Edsger Dijkstra in “*A discipline of programming*” [18]

1. Introduction

With the advances in hardware and communication technology, concurrency and distribution have come naturally into a large spectrum of applications, as they try to cope with a world where people and information are geographically distributed and where several things can happen simultaneously. Concurrency and distribution introduce a set of problems that greatly increase the complexity of the software.

First, and most importantly, concurrent and distributed systems are inherently more complex than non-distributed systems. By “inherently” I mean that, independent of any particular software realization, modeling situations involving several active, concurrent components that can communicate with each other is harder than modeling sequential, centralized systems.

Secondly, existing programming models and languages that are appropriate for sequential, centralized systems don’t necessarily provide the appropriate mechanisms for effectively expressing concurrent and distributed scenarios. As a consequence, the inherent complexity of these systems is severely magnified in the program texts themselves.

When developing a distributed application, and on top of the functional concerns of the applications (i.e. the features that are made available), designers and programmers must deal with issues of partitioning the components through the network, make them communicate appropriately, define and synchronize concurrent activities, handle partial failures, and provide acceptable performance. Being intrinsic to distributed systems, these other issues cannot be ignored; they must not only be thought of in the design, but they must ultimately be dealt with in the program that runs in the network of computers. While very little can be done to decrease the natural complexity of distributed applications, there is space to improve the quality and reliability of the development and maintenance processes of the corresponding programs.

This thesis builds on the software engineering problems involved in merging the distribution issues together with the functionality to obtain the desired distributed behavior. In particular, it focuses on the role of programming languages as tools for programming distribution issues.

1.1. The Problem

Figure 1 captures the problem of code tangling that is the base of this thesis. On the left, there is the code for a class when it is used in a non-distributed environment. On the right, there is the “same” class with capabilities for handling concurrent and remote invocations. The first observation is that the code on the right is much bigger. That is not surprising, given that a distributed scenario is inherently more complex than a non-distributed one. However, after a careful analysis of the code on the right, we come to the conclusion that such code (1) lost the functional encapsulation of the non-distributed version, (2) is a confusing intermingling of lines of code for different purposes, and (3) is full of redundant information.

Existing programming languages, in particular object-oriented languages, have relatively powerful capabilities for capturing the functionality of the application’s components. However, existing programming languages, including object-oriented languages, have relatively poor capabilities for

```
public class Shape {
    protected double x_= 0.0, y_ = 0.0;
    protected double width_=0.0, height_=0.0;

    double get_x() { return x_(); }
    void set_x(int x) { x_ = x; }
    double get_y() { return y_(); }
    void set_y(int y) { y_ = y; }
    double get_width(){ return width_(); }
    void set_width(int w) { width_ = w; }
    double get_height(){ return height_(); }
    void set_height(int h) { height_ = h; }
    void adjustLocation() {
        x_ = longCalculation1();
        y_ = longCalculation2();
    }
    void adjustDimensions() {
        width_ = longCalculation3();
        height_ = longCalculation4();
    }
}
```

Code for Shape objects in a non-distributed environment.

```
public class Shape implements ShapeI {
    protected AdjustableLocation loc;
    protected AdjustableDimension dim;
    public Shape() {
        loc = new AdjustableLocation(0, 0);
        dim = new AdjustableDimension(0, 0);
    }
    double get_x() throws RemoteException { return loc.x(); }
    void set_x(int x) throws RemoteException { loc.set_x(x); }
    double get_y() throws RemoteException { return loc.y(); }
    void set_y(int y) throws RemoteException { loc.set_y(y); }
    double get_width() throws RemoteException { return dim.width(); }
    void set_width(int w) throws RemoteException { dim.set_w(w); }
    double get_height() throws RemoteException { return dim.height(); }
    void set_height(int h) throws RemoteException { dim.set_h(h); }
    void adjustLocation() throws RemoteException {
        loc.adjust();
    }
    void adjustDimensions() throws RemoteException {
        dim.adjust();
    }
}

class AdjustableLocation {
    protected double x_, y_;
    public AdjustableLocation(double x, double y) {
        x_ = x; y_ = y;
    }
    synchronized double get_x() { return x_; }
    synchronized void set_x(int x) { x_ = x; }
    synchronized double get_y() { return y_; }
    synchronized void set_y(int y) { y_ = y; }
    synchronized void adjust() {
        x_ = longCalculation1();
        y_ = longCalculation2();
    }
}

class AdjustableDimension {
    protected double width_=0.0, height_=0.0;
    public AdjustableDimension(double h, double w) {
        height_ = h; width_ = w;
    }
    synchronized double get_width() { return width_; }
    synchronized void set_w(int w) { width_ = w; }
    synchronized double get_height() { return height_; }
    synchronized void set_h(int h) { height_ = h; }
    synchronized void adjust() {
        width_ = longCalculation3();
        height_ = longCalculation4();
    }
}

interface ShapeI extends Remote {
    double get_x() throws RemoteException ;
    void set_x(int x) throws RemoteException ;
    double get_y() throws RemoteException ;
    void set_y(int y) throws RemoteException ;
    double get_width() throws RemoteException ;
    void set_width(int w) throws RemoteException ;
    double get_height() throws RemoteException ;
    void set_height(int h) throws RemoteException ;
    void adjustLocation() throws RemoteException ;
    void adjustDimensions() throws RemoteException ;
}
```

Code to handle distributed Shape objects.

Figure 1. Code tangling in distributed applications.

capturing the behavior of components when they are distributed across a network and when the modularity of their sequential behavior is cut by the concurrent execution of their services.

Programming concurrency and distribution affects the implementation of each of the components in ways that tend to cross-cut the functionality of those components. It also potentially affects the implementation of groups of components. A simple component/interface division of the world leads to programs that are a confusing tangling of lines of code for different purposes.

1.2. The Approach

The approach taken in this thesis is to partition the knot of requirements and constraints of distributed systems into a number of general concerns, each of which has its own consistency and can be thought of in isolation throughout the lifecycle of the applications. That separation is preserved on the program texts themselves, resulting in programs that are untangled.

For that, a framework called D was designed. D is a language framework that supports new kinds of modules for addressing some of the distribution issues that are hard to capture in classes. These new kinds of modules compose with the classes in special ways. D's version of the example in Figure 1 is shown in Figure 2: the class is clear of code for dealing with distribution issues, and these are localized in special modules.

In designing D it was necessary to identify concerns that are reasonably well dealt within the

<pre>public class Shape { protected double x_= 0.0, y_= 0.0; protected double width_=0.0, height_=0.0; double get_x() { return x_(); } void set_x(int x) { x_ = x; } double get_y() { return y_(); } void set_y(int y) { y_ = y; } double get_width(){ return width_(); } void set_width(int w) { width_ = w; } double get_height(){ return height_(); } void set_height(int h) { height_ = h; } void adjustLocation() { x_ = longCalculation1(); y_ = longCalculation2(); } void adjustDimensions() { width_ = longCalculation3(); height_ = longCalculation4(); } }</pre>	<pre>coordinator Shape { selfex adjustLocation, adjustDimensions; mutex {adjustLocation, get_x, set_x, get_y, set_y}; mutex {adjustDimensions, get_width, get_height, set_width, set_height}; }</pre>
	<pre>portal Shape { double get_x() {} ; void set_x(int x) {} ; double get_y() {} ; void set_y(int y) {} ; double get_width() {} ; void set_width(int w) {} ; double get_height() {} ; void set_height(int h) {} ; void adjustLocation() {} ; void adjustDimensions() {} ; }</pre>

Figure 2. Programming in the D framework.

existing programming language's definitional mechanisms, and concerns that are not. To refer to the former we use the term *components*; to the latter, we have given the technical term of *aspects* [37]. The aspects affect the distributed and concurrent behavior of the components, and therefore they are closely related to their implementation. Yet they have some important characteristics that 1) makes it desirable to think of them in separate; 2) makes it hard to locate their effects on any of the components; and 3) makes it possible to achieve a reasonably effective separation from the components.

Selecting a Representative Class of Distributed Systems

“Distributed system” is a term that applies to a wide range of systems that exist in the whole spectrum of Computer Science. The language framework proposed here focuses only on a subset of applications that seem to be relatively important for the software industry, at least during the next decade. Examples are: applications over the Internet, the Web, corporate-wide applications such as calendar managers, network managers, document management systems (integrated scanning, storing, editing and printing), hospital management systems, front-end applications for database access, interactive multi-user environments, and many others like these. Throughout this dissertation the term “distributed application” or “distributed system” is used to denote this subset of applications.

1.3. The Thesis

This thesis demonstrates that the code for implementing certain distribution issues can be untangled from functionality code by providing new abstraction and composition mechanisms specifically designed for programming those distribution issues. The new mechanisms can be smoothly integrated with an object-oriented language with little modifications to that language and at a very low cost in terms of performance. The new aspect interfaces are easy to understand. D leads to programs whose modules are more focused and where the separation of concerns is more clear than it would be using traditional object-oriented languages.

In order to validate these claims, three different sets of results are used:

- (1) case-studies, which serve as the basis for making a quantitative analysis of the framework in terms of lines of code and metrics for measuring tangling;
- (2) performance measures; and
- (3) a preliminary user-study, in which an implementation of D was given to four alpha-users.

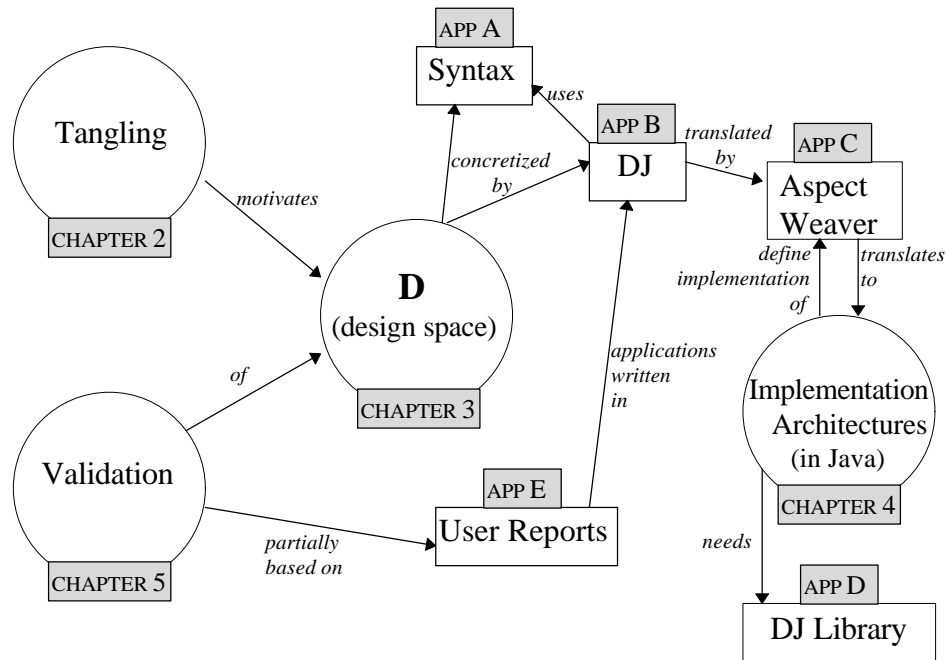


Figure 3. Synopsis of the dissertation.

1.4. Synopsis of the Dissertation

Figure 3 is the road map to the dissertation. The arrows indicate the structure of the argumentation in the thesis. Chapter 2 sets up the motivation for D by analyzing sources of complexity overhead in distributed programs. Chapter 3 describes the design of the D framework, gives a detailed specification of the semantics of the two aspect languages and discusses the design decisions as well as a number of design alternatives. Those specifications were concretized in a language called DJ, which uses Java as the component language. DJ uses the syntax described in Appendix A. An introduction to DJ is given in Appendix B. DJ was implemented by a pre-processor, called the Aspect Weaver, that outputs specific patterns of Java code. Those implementation architectures of the output Java code are the key for correctly implementing the specified semantics of D, and they are described in Chapter 4. Such architectures determine, to some extent, the implementation of the Aspect Weaver itself, presented in Appendix C. They also establish the library support that is needed in order to execute DJ programs; that library is given in Appendix D. Chapter 5 validates the claims and analyses the proposed language framework. Part of the validation comes from the feedback from alpha-users, presented in Appendix E. Finally, Chapter 6 concludes the dissertation.

