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

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

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(); }
  double get_y() throws RemoteException { return loc.y(); }
  void   set_y(int y) throws RemoteException { loc.set_y(); }
  double get_width() throws RemoteException { return dim.width(); }
  void   set_width(int w) throws RemoteException { dim.set_w(); }
  double get_height() throws RemoteException { return dim.height(); }
  void   set_height(int h) throws RemoteException { dim.set_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

```
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();
  }
}
```

```
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};
}
```

```
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() {};
}
```

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.

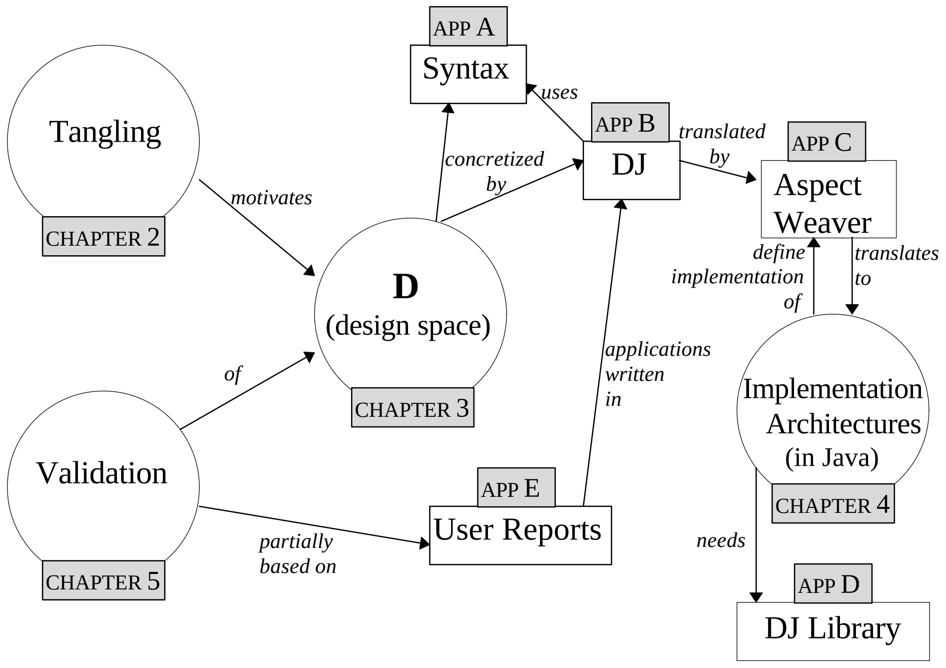


Figure 3. Synopsis of the dissertation.

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.

Chapter 2

Code Tangling

"A principal goal of all contemporary language design is to support the results of methodological research, that is, to provide language features that permit and even encourage the use of "good" program structures."

William Wulf, in "*Trends in the Design and Implementation of Programming Languages*" [75]

2. Code Tangling

By the late 1960's, a small revolution started with the intent of defining programming styles and programming language constructs for producing well-organized programs. This new wave was called *structured programming* [16, 19], and it came in the sequence of the software crisis that resulted from having to write relatively large, complex programs with low-level programming languages. Central to this debate were the `go to` statements, and the higher-level constructs that would eliminate them. Many people were claiming that `go to`'s were harmful, and programs using them usually resulted in "spaghetti code" which was difficult to understand, reason about and maintain. The conservatives claimed that `go to`'s were more efficient than all other higher-level constructs, and therefore should not be eliminated from the languages. In [38], Knuth suggested that `go to`'s were not the cause of the "spaghetti code;" the real problem was the unruled ways in which programmers used them.

Almost thirty years and a few innovations later, `go to`'s are part of the history of programming languages. The new generation of programming languages provides a number of powerful abstractions, such as procedures, functions, recursive data types, classes and objects, that allow the development of software systems so large and complex that they probably could not have been written (and, especially, maintained) with the languages of thirty years ago. Central to the modern programming technologies is the notion of "functional component", that is, a sub-part of a large system that provides a certain functionality that is intuitively appealing, is more or less independent from the other parts, and can be composed with other parts by some form of "uses" relation. Components are nicely captured by function, procedure or class definitions; and they compose nicely through function/procedure calls, and method invocations.

However, just as thirty years ago, the applications are pushing the limits of the current programming technologies. In today's software systems, many issues that must be programmed relate to other parts of the system in sophisticated ways that the "uses" relation doesn't quite capture in their entirety. As a consequence, a lot of useful information is lost when programming these issues with the existing composition mechanisms. Because these issues must be programmed, they end up being inserted in the components' code in more or less arbitrary ways.

The result is a more sophisticated recipe of spaghetti code, where, instead of `go to`'s that distract from the main flow of control in a segment of code, lines of relatively high-level code distract the main flow of control in a component's implementation. This phenomenon has been called *code tangling* [37]. This chapter gives a detailed analysis of code tangling in distributed systems, how it arises, and how it has been addressed by programming guidelines and programming languages.

2.1. How Programs Become Tangled

In order to analyze the code tangling problem, this problem will first be presented with an illustrative example. Suppose we want to implement a book locator which manages an association between books and their physical locations within a company. The functionality of such objects consists of three services: (i) a `register` service that takes one book and one location, and registers the pair in some internal database; (ii) an `unregister` service that takes one book and eliminates it from the registry; and (iii) a `locate` service that takes a key string, searches the string fields of the books for possible matches with the key, and returns the location of the first book that matches it. Besides this basic functionality, we want book locators to be accessible from other sites in the network, and to process requests concurrently.

The following three subsections present one possible implementation of this example using Java. At this point, nothing will be said about the role that the particular programming language and style have in the development of this small application; that will be the topic of the next sections.

2.1.1. Implementing the Functionality

Figure 4 shows one possible implementation of the basic components of this small application. It consists of three classes, namely BookLocator, Book and Location. The BookLocator class implements three methods that correspond to the specified services; the Book and Location classes are basically data structures.

```
public class BookLocator
{
  // This is just one possible implementation.
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  private int      nbooks = 0;
  // the constructor
  public BookLocator (int dbsize) {
    books = new Book[dbsize];
    locations = new Location[dbsize];
  }
  public void register (Book b, Location l)
  throws LocatorFull {

    if (nbooks > books.length)
      throw new LocatorFull();
    else {
      // Just put it at the end
      books[nbooks] = b;
      locations[nbooks++] = l;
    }
  }
  public void unregister (Book b) {
    Book abook = books[0]; int i = 0;
    while (i < nbooks &&
           abook.get_isbn() != b.get_isbn())
      abook = books[++i];
    if (i == nbooks)
      return;
    // simply shift down the rest
    while (i < nbooks - 1) {
      books[i]= books[i+1];
      locations[i]= locations[++i];
    }
    --nbooks;
  }
  public Location locate (String str)
  throws BookNotFound {
    Book abook = books[0];
    int i = 0; boolean found = false;
    while (i < nbooks && found == false) {
      if (abook.get_title().compareTo(str)==0||
          abook.get_author().compareTo(str)==0)
        found = true;
      else abook = books[++i];
    }
    if (found == false)
      throw new BookNotFound (str);

    return locations[i];
  }
}
```

```
public class Book {
  // possible implementation of Books
  String title, author;
  int isbn;
  Project owner;
  OCRImage firstpage;

  public Book (String t, String a,
                   int n) {
    title = t; author = a; isbn = n;
  }
  public String get_title() {
    return title;
  }
  public String get_author() {
    return author;
  }
  public int get_isbn() {
    return isbn;
  }
}
public class Location {
  //possible implementation of Locations
  public int building, room;
  public Location (int bn, int rn) {
    building = bn;
    room = rn;
  }
}
```

Figure 4. Simple implementation of book locators, books and locations.

2.1.2. Adding Synchronization Constraints

The specification also states that book locators should be able to process several requests concurrently. Having written the classes in Java, concurrency exists in the form of threads that execute the objects without any default synchronization. Since the three methods of the BookLocator class use and update the same instance variables, additional code is necessary in order to avoid inconsistencies.

With respect to concurrency, book locators present a pattern that is quite common: “read” accesses (in this case, the `locate` service) can be done concurrently while temporarily blocking all “write” accesses; “write” accesses (in this case, `register`, and `unregister`) should not be done concurrently and should block all other services.

The extended code is shown in Figure 5. Only the BookLocator class is shown, since the other two classes remain unchanged. The new code for coordination is marked in black. In order to avoid multiple conflicting accesses, there is the need to insert lines of code at specific points of the origi-

```java
public class BookLocator
{
  private Book     books[];
  private Location locations[];
  private int      nbooks = 0;
  protected int activeReaders = 0;
  protected int activeWriters = 0;

  // the constructor
  public BookLocator (int dbsize) {
    books = new Book[dbsize];
    locations = new Location[dbsize];
  }
  public void register(Book b, Location l)
  throws LocatorFull {
    beforeWrite();
    if (nbooks > books.length) {
      afterWrite();
      throw new LocatorFull();
    }
    else {
      // Just put it at the end
      books[nbooks] = b;
      locations[nbooks++] = l;
    }
    afterWrite();
  }
  public void unregister (Book b) {
    Book abook = books[0]; int i = 0;
    beforeWrite();
    while (i < nbooks &&
           abook.get_isbn()!=b.get_isbn())
      abook = books[++i];
    if (i == nbooks) {
      afterWrite();
      return;
    }
    // simply shift down the rest
    while (i < nbooks - 1) {
      books[i]= books[i+1];
      locations[i]= locations[++i];
    }
    --nbooks;
    afterWrite();
  }
```

```java
// class BookLocator continued
 public Location locate (String str)
 throws BookNotFound {
   Book abook = books[0];
   int i = 0; boolean found = false;
   Location l;
   synchronized (this) {
     while (activeWriters > 0) {
       try { wait(); }
       catch (InterruptedException e) {}
     }
     ++activeReaders;
   }
   while (i < nbooks && found == false){
    if(abook.get_title().compareTo(str)==0 ||
       abook.get_author().compareTo(str)==0)
      found = true;
    else abook = books[++i];
   }
   if (found == false) {
     synchronized (this) {
       --activeReaders;  notifyAll();
     }
     throw new BookNotFound (str);
   }
   l = locations[i];
   synchronized (this) {
     --activeReaders;  notifyAll();
   }
   return l;
 }
 protected synchronized void beforeWrite() {
   while (activeReaders > 0 ||
          activeWriters > 0) {
     try { wait(); }
     catch (InterruptedException e) {}
   }
   ++activeWriters;
 }
 protected synchronized void afterWrite() {
   --activeWriters;  notifyAll();
 }
}
```

Figure 5. Coordinating the concurrent access to the BookLocator class.

nal component implementation. In this case, we need to (1) define extra instance variables; (2) make a number of checks and affect the state of the object on entering the methods; and (3) change the state in every exit point of the methods. These bits of code inside the body of the methods can be structured calls to other methods (e.g. in `beforeWrite` and `afterWrite`) or simply a number of extra statements (as, for example in the `locate` method). Special attention must be given for methods that return objects, since the return expression may affect the state of the object, and therefore should also be guarded (see the `locate` method).

It should be noticed that the particular coordination code shown in Figure 5 is only one among many possibilities for implementing the coordination of concurrent accesses. A number of refinements could be done in order to minimize the locking periods, and a number of other programming styles could have been used. In short, the programmer, had a relatively large degree of freedom for translating the coordination intentions into pieces of code and to intertwine them within the component implementation.

The complexity of the program is necessarily higher when the coordination issue is taken into account. Also, the particular coordination strategy depends, to a certain degree, on the particular component's implementation. However, by having to intertwine by hand the implementation of the coordination with the implementation of the component, much of the coordination information, as well as the component's basic functionality, is diluted. These two concerns — that were initially described separately — become only one block of code that is hard to understand and reason about.

2.1.3. Providing for Remote Access

The last piece of the specification is that book locators can be accessed from other sites in the network. Suppose, for example, that they are part of a much larger document management application, and that the complete book database is managed by some other server(s). In order to speed up the searches, book locators should cache information about the books, namely their titles, authors and isbn.

The new version of the code is shown in Figure 6, where the new pieces of code are marked in black. In general, the code for implementing distribution is highly dependent on the particular platform used, and, unlike the coordination issue, there isn't even a common understanding of what is the best way to do it. In this case, the Java RMI [27] facility was used.

```
public interface Locator
       extends rmi.Remote {
  void register (BookI b, LocationI l)
       throws rmi.RemoteException,
              LocatorFull;
  void unregister (BookI b)
       throws rmi.RemoteException;
  LocationI locate (BookI b)
       throws rmi.RemoteException,
              BookNotFound;
}
public interface BookI
       extends Serializable {
  String get_title();
  String get_author();
  int get_isbn();
}
public interface LocationI
       extends Serializable {
}

public class BookLocator
       extends UnicastRemoteObject
       implements Locator
{
  private Book     books[];
  private Location locations[];
  private int      nbooks = 0;
  protected int activeReaders = 0;
  protected int activeWriters = 0;

  // the constructor
  public BookLocator (int dbsize) {
    books = new Book[dbsize];
    locations = new Location[dbsize];
  }
  public void register(Book b, Location l)
  throws LocatorFull {
    beforeWrite();
    if (nbooks > books.length) {
      afterWrite();
      throw new LocatorFull();
    }
    else {
      // Just put it at the end
      books[nbooks] = b;
      locations[nbooks++] = l;
    }
    afterWrite();
  }
  public void unregister (Book b) {
    Book abook = books[0]; int i = 0;
    beforeWrite();
    while (i < nbooks &&
           abook.get_isbn()!=b.get_isbn())
      abook = books[++i];
    if (i == nbooks) {
      afterWrite();
      return;
    }
    // simply shift down the rest
    while (i < nbooks - 1) {
      books[i]= books[i+1];
      locations[i]= locations[++i];
    }
    --nbooks;
    afterWrite();
  }
```

```
// class BookLocator continued
  public Location locate (String str)
  throws BookNotFound {
    Book abook = books[0];
    int i = 0; boolean found = false;
    Location l;
    synchronized (this) {
      while (activeWriters > 0)
        try { wait(); }
        catch (InterruptedException e) {}
      ++activeReaders;
    }
    while (i < nbooks && found == false){
     if(abook.get_title().compareTo(str)==0 ||
        abook.get_author().compareTo(str)==0)
      found = true;
     else abook = books[++i];
    }
    if (found == false) {
      synchronized (this) {
        --activeReaders; notifyAll();
      }
      throw new BookNotFound (str);
    }
    l = locations[i];
    synchronized (this) {
      --activeReaders; notifyAll();
    }
    return l;
  }
  protected synchronized void beforeWrite() {
    // as before
  }
  protected synchronized void afterWrite() {
    // as before
  }
}
public class Book implements BookI {
  // possible implementation of Books
  String title, author;
  int isbn;  Project owner;
  OCRImage firstpage;

  public Book (String t, String a, int n){
    title = t; author = a; isbn = n;
  }
  public String get_title() { return title; }
  public String get_author() { eturn author;}
  public int get_isbn() { return isbn; }
  private void writeObject(ObjectOutputStream
                           s)
  throws NotSerializableException,IOException{
    s.writeObject(title);
    s.writeObject(author);
    s.writeInt(isbn);
  }
  private void readObject(ObjectInputStream
                          s)
  throws NotSerializableException,IOException{
    title = (String)s.readObject();
    author = (String)s.readObject();
    isbn = s.readInt();
  }
}
public class Location implements LocationI {
  //the same as in Figure 4.
}
```

Figure 6. Providing remote access to book locators.

In order to use Java RMI, the classes must implement interfaces that extend the Remote, Serializable[1] or Externalizable interfaces. Instances of Remote classes are never copied on remote calls, whereas instances of Serializable or Externalizable classes are always passed by copy on remote calls. Classes that implement the Serializable interface may redefine the `writeObject` and `readObject` methods, for customizing the marshaling policies. That is the case for the Book class, where `writeObject` and `readObject` only deal with three fields of the class, avoiding the marshaling of `owner` and `firstPage`.

A client of the book locator service can invoke it from anywhere in the network, in the following way:

```
// Get the reference to the book locator,
// for example, from the name server
Locator BL = (Locator)Naming.lookup("//localhost/BookLocator");
// …
// invoking the register service
Location aLoc = new Location (35, 1631);
Book aBook = new Book (new String("Title 1"), new String("Author A"), 33);
BL.register (aBook, aLoc);
// …
// invoking the locate service
Location theLoc;
String title2;
// …
theLoc = BL.locate (title2);
```

When an invocation occurs, for example `BL.locate(title2)`, the RMI system takes care of translating the object reference `BL` into a network address and executing a communication protocol between the current space and the space where object `BL` really is. Part of the protocol consists in transferring the parameters of the call, and, for that, upcalls are made to the `writeObject` and `readObject` methods of the Serializable parameter objects.

Although simple for simple cases, programming with RMI soon becomes a non-trivial exercise. Suppose that books are also used by a service that manages projects. This new component is implemented by the code in Figure 7. Only the relevant interface and classes are shown; the Book class refers to the implementation shown in the figures before.

[1] There is one unfortunate collision of terminology, namely the word "Serializable". In concurrent systems serialization usually denotes objects that don't allow internal concurrency, i.e. that handle one request at a time. In distributed systems serialization has a completely different meaning, namely the process of marshaling and unmarshaling objects into/from streams. The use of this word will be avoided, unless when referring to Java's Serialization interface.

```
public interface PManager
       extends rmi.Remote {
  boolean newBook (BookI b, PriceI p)
          throws rmi.RemoteException,
                 ProjectNotFound;
  // other services ommitted
}


public class ProjectManager
       extends UnicastRemoteObject
       implements Pmanager {
  ProjectList projects;

  public boolean newBook (Book b, Price p)
  throws rmi.RemoteException,ProjectNotFound
  {
    Project prj = b.get_owner();
    if (!projects.contains(prj))
      throw ProjectNotFound;
    return prj.newBook (b, p);

  }
  // other methods ommited
}
```

```
public class Project implements ProjectI{
  ProjId projectId;
  Person manager;
  PersonList workers;
  Budget bdgtCenter;
  ComputerList computers;
  BookList books;

  boolean newBook (Book b, Price p) {
    if (bdgtCenter.approvePurchase(p)) {
      books.append (b);
      return true;
    }
    return false;
  }
  // other methods omitted
}
```

Figure 7. Adding a new component to the application.

The problem that this new component introduces is the marshaling of the `Book` parameter for the `newBook` service. The `writeObject` and `readObject` methods defined in Figure 6 are not appropriate for this new service, since they don't marshal the `Project` field of the books — which is not necessary for the book locator service, but which is used here. In general, and for non-trivial examples, the way objects are copied around depends on the service that is being invoked. Implementing this requires some form of marshaling in context. One way of doing it in the Java environment is adding a context variable to class Book, and setting it before making the calls. The class Book and the client code will now look like the code shown in Figure 7, where the additional code for handling the context is underlined.

Notice that the code suffered another layer of tangling, with auxiliary classes being introduced. But worst of all, the client calls are no longer simple method invocations, as they are made aware of the remote communication by explicitly hard-coding the names of the services that are being invoked. Conceptually, this information is redundant; but the implementation must have it. This is in clear violation of the nice abstraction provided by RMI.

Marshaling propagates from the parameter object to its variables, recursively. When it is done in context, the context must also propagate in some way. In this case, a book is marshaled differently for two services because it involves a different marshaling of the `owner` field. Therefore, the class Project must implement a marshaling routine for this particular situation. (Those methods, `pack1` and `unpack1` for class `Project`, are not shown).

```
public class Book implements BookI {
  // possible implementation of Books
  String title, author;
  int isbn;
  Project owner;
  OCRImage firstpage;
  Context ctx;

  public Book (String t,String a, int n){
    title = t; author = a; isbn = n;
  }
  public String get_title() {
    return title; }
  public String get_author() {
    return author;}
  public int get_isbn() {
    return isbn; }
  public Project get_owner() {
    return owner; }

  public void setContext(String srv,
                         String service){
    ctx.set(srv, service);
  }
}
```

```
 // class Book continued
  private void writeObject(ObjectOutputStream
                                   s)
  throws NotSerializableException,IOException{
    ctx.pack(s);
    s.writeObject(title);
    s.writeObject(author);
    s.writeInt(isbn);
    if (ctx.service(“PManager”, “newBook”))
      owner.pack1(s);
  }
  private void readObject(ObjectInputStream s)
  throws NotSerializableException,IOException{
    ctx = Context.unpack(s);
    title = (String)s.readObject();
    author = (String)s.readObject();
    isbn = s.readInt();
    if (ctx.service(“Pmanager”, “newBook”))
      owner = Project.unpack1(s);
  }
}
```

```
  // Client of a book locator service
  aBook.setContext(new String(“Locator”),
                   new String(“register”));
  BL.register (aBook, aLoc);
```

Figure 8. Marshaling parameters depending on the service.

The implementation is already sufficiently complex, but in order for it to be safe there is still one issue that must be taken care of. This has to do with coordinating possibly concurrent marshalings of the same book. Since we’re setting and using a context object in different parts of the code, we must guarantee that no changes to that context will occur between its setting before the call and its use in the `writeObject` method. The coordination can be implemented in the context object itself. But at this point, the code is already too long to serve as an illustrative example.

In short, because different services have different needs with respect to the information that is copied from one space to the other, the code that implements the components suffers an overhead of complexity that makes it much more confusing that what it should be. The tangled code is extremely difficult to maintain, since small changes to the functionality require mentally untangling and then re-tangling it.

After the coordination and distribution issues are programmed in this way, the different concerns are hardly identifiable: the source code has become a tangled mess of hidden intentions, hidden dependencies, and redundant information.

2.2. The Source of Tangling

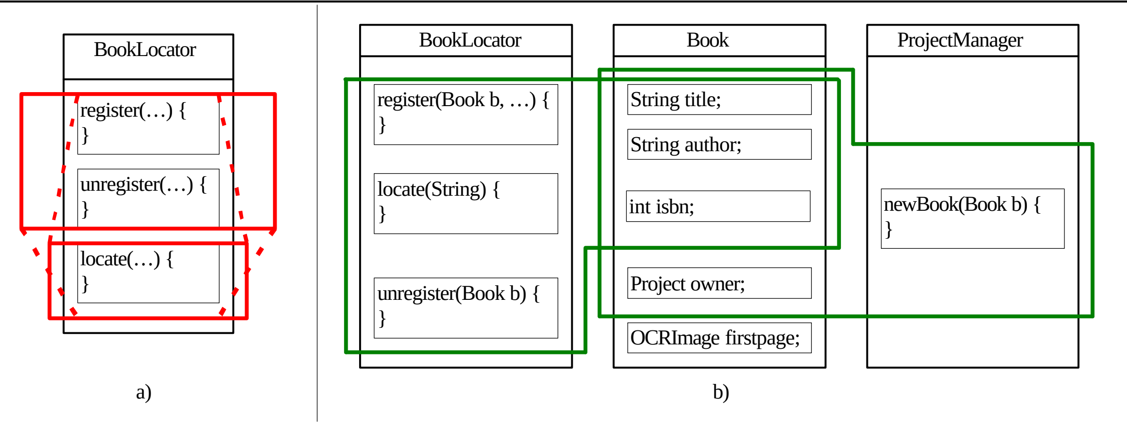


Figure 9. Cross-cutting issues: a) functionality (methods) and coordination schemes, different for different groups of methods; b) functionality (methods, instance variables) and copying schemes, different for different remote services.

Figure 9 provides a basis for understanding the tangling in the example of the previous section. On the left (a) there is a representation of the tangling between the methods and the coordination constraints described in §2.1.2. In this simple example, two of the methods, `register` and `unregister`, share the same kind of coordination — that comes from the fact that they are both "writers" of the same variables — whereas another method, `locate`, is coordinated by another scheme — because it is only a "reader" of those same variables. On the right (b) there is a snapshot of the tangling between components and the copying strategies on remote calls described in §2.1.3. In this case, the `BookLocator` class and the `ProjectManager` class need different parts of book objects, namely `ProjectManager` needs the `owner` part of the books.

These cross-cutting effects can be represented in a more generic form, as shown in Figure 10 and Figure 11. The functionality composes hierarchically and through the "uses" relationship, in the traditional way. But the coordination composes by combining sets of classes and methods of those classes that share coordination constraints and that end up in wait/notify relationships at run-time. And the copying composes by combining paths in the data graph that end up in more global "uses" relationships at run-time that fundamentally cross-cut the implementation of the classes.

This cross-cutting phenomena is directly responsible for the tangling in the code. The composition mechanisms the language provides us — method calling and inheritance — are well suited to building up the functional units. But they are not so good for composing the functional units with cross-cutting issues, because they follow such different composition rules and yet must co-compose. This breakdown forces us to combine the properties entirely by hand — that's what happened in the tangled code presented in §2.1.

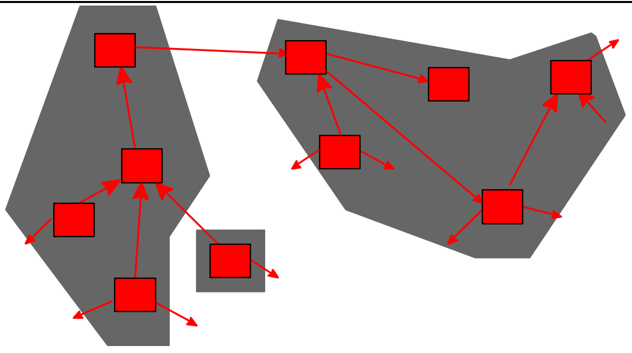


Figure 10. Cross-cutting between classes and coordination. Groups of classes (the squares) compose hierarchically (thick arrows) and with "uses" relationships (thin arrows), but they may be related in different ways from the coordination perspective (background shapes).

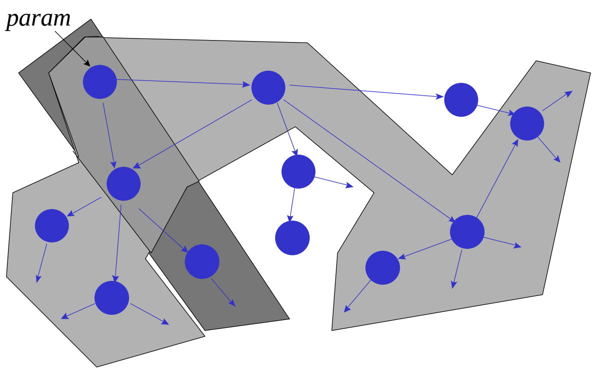


Figure 11. Cross-cutting between objects (the circles) and copying schemes (background shapes) when some root object is passed as parameter to different services.

The tangling phenomena that occurs between these cross-cutting issues can be more or less magnified in the source code depending on two factors: the programming language and the programming practices and styles. Next, these two factors will be analyzed.

2.3. Tangling and Programming Practices

The complexity of programming cross-cutting issues can be decreased by imposing a number of coding rules or by applying well-known design patterns, and documenting them. This section presents a number of such programming practices.

The language used to illustrate the arguments tries to capture one of the major trends in distributed programming, namely one that defines components, composes them by some form of generic procedure call (e.g. method invocation), uses interface definitions to produce the infrastructure for remote calls, and provides some embedded primitives for coordination. Java and Java's RMI capture this trend well. But the arguments and observations made in this section, though mostly illustrated with Java, also apply to many other language frameworks that follow the same trend, for example CORBA and C++ or Lisp with access to a thread library.

2.3.1. Concurrency

2.3.1.1 Units of Synchronization

Synchronizing threads in a multithreaded language environment that doesn't guarantee thread-safety, is known to be one of the most error-prone and time-consuming programming tasks. This is, in part, due to the inherent conflict between the desired amount of concurrency and the safety properties that guarantee that nothing bad will happen.

One of the basic issues in concurrency control has to do with the units of synchronization. Having access to locking and unlocking primitives that control the access to arbitrary critical sections of the program, gives the necessary flexibility for optimizing locking times. Consider the example in Figure 12 (adapted from [40], page 297), where class `BoundedBuffer` implements the classical bounded buffer example with a circular array of elements. In this implementation, elements are put at position `putPtr` (the front of the array), and removed from position `takePtr` (the tail), and these two indices are reset as they reach the maximum capacity of the buffer. In order to identify the issues, the code for coordination is underlined.

This is one of the most efficient implementations of concurrent bounded buffers, since the locking is specialized for each critical section, and it is performed within the object. But this style of programming leads to unruly code that can be very hard to understand and maintain.

```
public class BoundedBuffer {
  private Object[] array;                  // the elements
  private int putPtr = 0, takePtr = 0;    // circular indices
  private int, emptySlots, usedSlots = 0; // slot counts
  private int waitingPuts = 0, waitingTakes = 0;// counters of waiting threads
  private Object putLock, takeLock;      // and synchronization objects

  public BoundedBuffer (int capacity) {
    array = new Object[capacity];
    emptySlots = capacity;
    putLock = new Object(); takeLock = new Object();
  }

  public void put(Object o) {
    synchronized (putLock) {
      while (emptySlots <= 0) { // buffer is full => wait
        ++waitingPuts;
        try { putLock.wait(); } // the wait statement
        catch (InterruptedException e) {};
        --waitingPuts;
      }
      --emptySlots;
      array[putPtr] = o;  // insertion code
      putPtr = (putPtr + 1) % array.length;
    }
    synchronized (takeLock){
      ++usedSlots;
      if (waitingTakes > 0)
        takeLock.notify();   // signal a thread waiting on this lock
    }
  }

  public Object take() {
    Object old = null;
    synchronized (takeLock) {
      while (usedSlots <= 0) {   // buffer is empty => wait
        ++waitingTakes;
        try { takeLock.wait(); } // the wait statement
        catch (InterruptedException e) {};
        --waitingTakes;
      }
      --usedSlots;
      old = array[takePtr];  // removal code
      takePtr = (takePtr + 1) % array.length;
    }
    synchronized (putLock) {
      ++emptySlots;
      if (waitingPuts > 0)
        putLock.notify();       // signal a thread waiting on this lock
    }
    return old;
  }
}
```

Figure 12. Synchronization primitives for dealing with waiting conditions and critical sections.

In a way, this style of programming is similar to programming with `go to` statements. `go to`’s control the sequential execution flow, whereas the low-level synchronization primitives control the relative time of execution of the threads. But from a language design point of view, the idea is the same: give programmers the most basic mechanism for controlling the execution, and let them build the program directly on top of that.

A slightly better style consists in having semaphore objects, and to use them in the `P()` and `V()` traditional way for testing the waiting conditions, while still using locks for critical sections. Figure 13 shows this second version of the bounded buffer.

This code is potentially less efficient that the previous one, since it contains calls to semaphore objects. But its tangleness is clearly less serious than that of Figure 12, as it separates the issue of

```
public class BoundedBuffer {
  private Object[] array;               // the elements
  private int putPtr = 0, takePtr = 0; // circular indices
  private Object putLock, takeLock;    // for critical sections
  private Semaphore putSem, takeSem;   // semaphore objects

  public BoundedBuffer (int capacity) {
    array = new Object[capacity];
    putLock = new Object(); takeLock = new Object();
    putSem = new Semaphore(capacity);
    takeSem = new Semaphore(0);
  }

  public void put(Object o) {
    putSem.P();              // wait if full
    synchronized (putLock) {  // critical section for puts only
      array[putPtr] = o;
      putPtr = (putPtr + 1) % array.length;
    }
    takeSem.V();              // enable takes
  }

  public Object take() {
    Object old = null;
    takeSem.P();               // wait if empty
    synchronized (takeLock) {  // critical section for takes only
      old = array[takePtr];
      takePtr = (takePtr + 1) % array.length;
    }
    putSem.V();                // enable puts
    return old;
  }
}

public class Semaphore {
  private count = 0;
  private waiting = 0;
  public Semaphore (int initialCount) {
      count = initialCount;
  }
  public synchronized void P() {
    while (count <= 0) {
      ++waiting;
      try { wait(); } catch (InterruptedException e) {};
      --waiting;
    }
    --count;
  }
  public synchronized void V() {
    ++count;
    if (waiting > 0) notify();
  }
}
```

Figure 13. Semaphores as waiting conditions. Low-level synchronization for critical sections.

waiting (when the buffer is empty or full) from the issue of having critical sections in the implementation of `put` and `take`. The abstraction of a semaphore is powerful enough to also be used for mutual exclusion, by having binary semaphores (i.e. semaphores initialized to the value 1). Instead of using the direct synchronization on lock objects for guaranteeing mutual exclusion in the critical sections of the code, we can `P()` and `V()` on binary semaphores before entering and after leaving each of the two critical sections. The new version is partially shown in Figure 14.

```
public class BoundedBuffer {
  private Object[] array;               // the elements
  private int putPtr = 0, takePtr = 0; // circular indices
  private Semaphore putExclusion, takeExclusion;    // for critical sections
  private Semaphore putSem, takeSem;   // semaphore objects

  public BoundedBuffer (int capacity) {
    array = new Object[capacity];
    putExclusion = new Semaphore(1);
    takeExclusion = new Semaphore(1);
    putSem = new Semaphore(capacity);
    takeSem = new Semaphore(0);
  }

  public void put(Object o) {
    putSem.P();               // wait if full
    putExclusion.P();         // begin critical section for put
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    putExclusion.V();         // end critical section
    takeSem.V();              // enable takes
  }
  // similar for take
}

// same class Semaphore
```

Figure 14. Using semaphores for handling all waiting conditions, including mutual exclusion on critical sections.

But this style is still relatively low-level, and prone to programming errors, since the programmer must ensure that `P()`'s and `V()`'s to the proper semaphores are placed in the right positions within the implementation of the methods.

A safer style of programming is to follow the rule that says that the units of synchronization should be the objects themselves, and that guards should be placed as the first instructions of the methods (pre-conditions) and notifications should be issued as the last instructions of the methods. This comes in the tradition of monitors [24], and it is safer for two different reasons: first, programmers insert waits and notifications only in the beginning and the end of the methods; second, it guarantees that the internal object consistency is always preserved. Figure 15 shows the new version of the same bounded buffer example, using Java's synchronized methods.

```
public class BoundedBuffer {
  private Object[] array;               // the elements
  private int putPtr = 0, takePtr = 0; // circular indices
  private int usedSlots = 0; // counter
  public BoundedBuffer (int capacity) {
    array = new Object[capacity];
  }

  public synchronized void put(Object o) { // mutual exclusion guaranteed
    // check pre-condition
    while (usedSlots == array.length)
      try { wait(); } catch (InterruptedException e) {};

    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;

    // change state for notification; notify other threads that something changed
    ++usedSlots;
    notifyAll();
  }

  public synchronized Object take() {  // mutual exclusion guaranteed
    Object old;
    // check pre-condition
    while (usedSlots == 0)
      try { wait(); } catch (InterruptedException e) {};

    old = array[takePtr];
    takePtr = (takePtr + 1) % array.length;

    // change state for notification; notify other threads that something changed
    --usedSlots;
    notifyAll();
    return old;
  }
}
```

Figure 15. Synchronization through monitors, one monitor per object.

By always following the coordination pattern of the code in Figure 15, the functionality code is now clearly visible. But it comes with a price: the amount of concurrency on bounded buffers has decreased. `put` and `take` are now mutually exclusive, and, in general, they wouldn't need to be, because the insertion and removal indices are different. The tangleness of the previous implementations was there precisely because of this detail – that the coordination cross-cuts the implementation of those two methods.

In conclusion, there is a tradeoff between the chosen units of synchronization and the tangleness of the code. As a general rule, the more arbitrary those units are, the more efficient the coordination may be, but the more tangled the code may become. Having access to low-level synchronization primitives, designers may, however, chose to use a number of different styles with respect to the units of synchronization, that will lead to possibly less efficient, but certainly less tangled, code.

The next sub-sections present a few more points in this design space. The next few programming styles for concurrency were adapted from Doug Lea's book [40].

2.3.1.2 Splitting Classes

When the implementation of a class can be partitioned into independent, non-interacting subsets of methods, we can refactor the class to use finer-granularity helper objects whose actions are delegated by the host. This is a rule of thumb that generally holds in object-oriented programming, but is of greater importance in concurrent systems. Consider, for example, the following generic class:

```
public class TheClass {
  // set of variables S1
  // set of variables S2
  public synchronized void methodA () {
    // uses and changes variables in S1
  }
  public synchronized void methodB () {
    // uses and changes variables in S1
  }
  public synchronized void methodC() {
    // uses and changes variables in S2
  }
  public synchronized void methodD() {
    // uses and changes variables in S2
  }
}
```

Using a monitor, as above, is too restrictive, since `methodA` and `methodB` don't conflict with `methodC` and `methodD`. Then, by applying a straightforward refactoring procedure, we can program the coordination in the following way:

```
public class TheClass {
  private S1Class s1 = new S1Class();
  private S2Class s2 = new S2Class();
// none of the methods is synchronized
  public void methodA() {
    s1.methodA();
  }
  public void methodB() {
    s1.methodB();
  }
  public void methodC() {
    s2.methodC();
  }
  public void methodD() {
    s2.methodD();
  }
}
```

```
public class S1Class {
  // set of variables S1
  public synchronized methodA(){
    // uses and changes variables in S1
  }
  public synchronized methodB(){
    // uses and changes variables in S1
  }
}
public class S2Class {
  // set of variables S2
  public synchronized methodC(){
    // uses and changes variables in S2
  }
  public synchronized methodD(){
    // uses and changes variables in S2
  }
}
```

This pattern can be applied, with many variations, whenever sets of methods don't conflict and don't interact. The `synchronized` qualifier for the methods of `S1Class` and `S2Class` can eventually be replaced by more sophisticated waiting conditions and notifications, if necessary.

With appropriate documentation, the refactored design produces relatively well-structured code and provides more concurrency than the monitor design.

2.3.1.3 Splitting Locks

Another pattern for achieving the same behavior is to define lock variables for each of the sets of non-interacting methods, and get the respective locks in the beginning of each method. The result is as follows:

```
public class TheClass {
  // set of variables S1
  // set of variables S2

  Object lockS1 = new Object();
  Object lockS1 = new Object();

  public void methodA () {
    synchronized (lockS1) {
      // uses and changes variables in S1
    }
  }
  public void methodB () {
    synchronized (lockS1) {
      // uses and changes variables in S1
    }
  }
  public void methodC() {
    synchronized (lockS2) {
      // uses and changes variables in S2
    }
  }
  public void methodD() {
    synchronized (lockS2) {
      // uses and changes variables in S2
    }
  }
}
```

Again, with some documentation this design is relatively solid. It is more low-level than the refactoring design, since it uses direct locking on objects.

2.3.1.4 Coordination State

The design of concurrent systems usually involves identifying the states in which threads are suspended and the states in which they can proceed. The state space of the objects is usually very large, but only a small subset is important for purposes of action control — the coordination state. For example, in the book locator class Figure 5 the arrays of books and locations are completely ignored for purposes of synchronization; only the instance variables `activeReaders` and `activeWriters` matter. In this particular case there is an obvious separation of the instance variables that hold the synchronization state (`activeReaders` and `activeWriters`) from the ones that don't (books and locations), and this is captured by the fact that these variables were added to the original implementation in Figure 4.

This separation, however, is not enforced by languages like Java. As a consequence, programmers must make sure that suspensions and notifications happen at the right points in the code, and for the right values of the instance variables. Defining and tracking the synchronization state is one of the most critical points of concurrent systems. Doing it in the objects' complete state space can be confusing and error-prone.

For example, when implementing the bounded buffer (Figure 12 and Figure 15), we used ordinary instance variables for holding the coordination state, namely `usedSlots` and `emptySlots` (the latter only used in Figure 12). A better style is to use an explicit state variable that takes the values `EMPTY`, `FULL` and `MIDDLE`, and to write down a method that implements state changes on the bounded buffer. The result is shown in Figure 16.

```
public class BoundedBuffer {
  static final int EMPTY = -1; // the three values of the coordination state
  static final int MIDDLE = 0;
  static final int FULL = 1;
  protected int state = EMPTY; // state variable, initialized to the proper state

  private Object[] array;               // the elements
  private int putPtr = 0, takePtr = 0; // circular indices
  private int usedSlots = 0; // counter

  public BoundedBuffer (int capacity) {
    array = new Object[capacity];
  }
  // this method implements the state transitions
  protected synchronized void changeState() {
    int oldstate = state;
    if (usedSlots == 0) state = EMPTY;
    else if (usedSlots == array.length) state = FULL;
    else state = MIDDLE;

    if (state != oldstate && (oldstate == EMPTY || oldstate == FULL))
      notifyAll();
  }
  public synchronized void put(Object o) { // mutual exclusion guaranteed
    // check the coordination state
    while (state == FULL)
      try { wait(); } catch (InterruptedException e) {};

    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;

    ++usedSlots;
    changeState();  // execute the state machine
  }
  // similar for take. (it checks for EMPTY)
}
```

Figure 16.Tracking state variables.

Another option is to apply the design pattern *State as Objects* [21]. Rather than coding state as a value, we can code it as a set of classes with specific behaviors. The class being coordinated contains a reference that is always bound to the appropriate state object, and to which it delegates all the actions.

2.3.1.5 Coordination by Subclassing

One of the best ways for systematically separating the coordination code from the functionality code is to use the inheritance composition mechanism in object-oriented languages. The basic idea of this design pattern is that the superclass is the repository of method implementations, whereas the subclass, possibly more than one, implements the coordination of those methods using a *before* and *after* style.

Figure 17 shows the code that results from applying this pattern to the book locator class. Class `CoreBookLocator` is just the repository of the methods; its subclass `BookLocator` adds the implementation of the coordination strategy by including wrappers of code before and after calling the method on the superclass. (The `finally` clause is there to guarantee that, even if the body of `try` exits with an exception, the *after* method is executed.)

More generally, by making the variables of the base class available to the subclasses (using Java's C++'s `protected` qualifier) the subclasses have all the power to define fine-grain concurrency policies that involve the object's state (i.e. the implementation of the base class). Although design guidelines discourage this practice — that is, to make all variables of the base class accessible to the subclasses — [14] it can be a powerful way to untangle concurrency control from functionality, as long as the invariants are well documented, and programmers refrain from doing ad-hoc enhancements to the subclasses that may modify the original intentions of the base class.

But coordination by subclassing has its limitations. The problems with this approach are generally denoted by *inheritance anomalies*, and they have caught the attention of a large number of researchers. Basically, these anomalies consist of the following: when coding a base class, if a number of precautions are not followed, the code related to concurrency control cannot be effectively inherited and/or redefined in a subclass without non-trivial redefinitions of the methods' implementations. This situation hampers the reuse of code by inheritance, and, therefore, weakens the usefulness of object-oriented languages in programming distributed applications.

These problems have been presented in the literature and are now well-understood: they are related to the particular programming language in use, and, when using languages like Java, C++ or Smalltalk, they are even more strongly related to the adopted programming styles. In [54] and [40] there are several illustrative examples. Lea summarizes the inheritance anomalies in the following way:

```java
public class CoreBookLocator
{
  // This is just one possible implementation.
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  private int      nbooks = 0;
  // the constructor
  public CoreBookLocator (int dbsize) {
    books = new Book[dbsize];
    locations = new Location[dbsize];
  }
  protected void register_(Book b, Location l)
  throws LocatorFull {

    if (nbooks > books.length)
      throw new LocatorFull();
    else {
      // Just put it at the end
      books[nbooks] = b;
      locations[nbooks++] = l;
    }
  }
  protected void unregister_(Book b) {
    Book abook = books[0]; int i = 0;
    while (i < nbooks &&
           abook.get_isbn() != b.get_isbn())
      abook = books[++i];
    if (i == nbooks)
      return;
    // simply shift down the rest
    while (i < nbooks - 1) {
      books[i]= books[i+1];
      locations[i]= locations[++i];
    }
    --nbooks;
  }
  protected Location locate_(String str)
  throws BookNotFound {
    Book abook = books[0];
    int i = 0; boolean found = false;
    while (i < nbooks && found == false) {
      if (abook.get_title().compareTo(str)==0||
          abook.get_author().compareTo(str)==0)
        found = true;
      else abook = books[++i];
    }
    if (found == false)
      throw new BookNotFound (str);

    return locations[i];
  }
}
```

```java
public class BookLocator
       extends CoreBookLocator {
  private int activeReaders = 0;
  private int activeWriters = 0;

  public void register(Book b,Location l)
    beforeWrite();
    try {
      register_(b, l); // core action
    } finally {
      afterWrite();
    }
  }
  public void unregister(Book b) {
    beforeWrite();
    try {
      unregister_(b);  // core action
    } finally {
      afterWrite();
    }
  }
  public Location locate(String str) {
    synchronized (this) {
      while (activeWriters > 0) {
        try { wait(); }
        catch (InterruptedException e) {}
      }
      ++activeReaders;
    }
    try {
      return locate_(str); // core action
    } finally {
      synchronized (this) {
        --activeReaders;
        notifyAll();
    }
  }
  private synchronized void beforeWrite()
  {
    while (activeReaders > 0 ||
           activeWriters > 0) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
    ++activeWriters;
  }
  private synchronized void afterWrite(){
    --activeWriters;
    notifyAll();
  }
}
```

Figure 17. Coordinating the book locator class by subclassing.

- If the base class lacks explicit representation and tracking of that state on which coordination depends, then those methods of the base class that affect that state must be recoded in the subclass.
- If a subclass partitions the coordination state in a different way than represented by a base class state variable, base class methods that refer to that state variable must be recoded.

•If a subclass includes guarded waits on conditions that base class methods do not provide notifications about, then these methods of the base class must be recoded.

• If the base class uses fine-grain notification (i.e. notification of only one thread) and a subclass adds features that cause the conditions for fine-grain notification to no longer hold, then all methods of the base class performing fine-grain notifications must be recoded.

• If an instance variable is treated as immutable in the base class but is assigned to in the subclass, then all methods taking advantage of immutability must be recoded. Similar problems occur with assumptions about uniqueness.

The inheritance anomalies are not specific to concurrency; they also occur in sequential programs whenever the design decisions have not been properly encapsulated in methods and instance variables. However, they are magnified in concurrent programs: since there are no formal rules for coding coordination, it is too easy to write concurrent classes that, because of code tangling, cannot be easily extended. That is the case of all the examples given above, with the exception of the implementation in Figure 17, which followed the strict coding rules of coordination by subclassing. But even when applying this pattern, inheritance anomalies may occur when subclassing the coordination class.

2.3.2. Communication

2.3.2.1 Splitting Parts

One way of fixing the problem of passing different parts of the objects for different services is to split the object into its parts, and pass them instead. This implies having to adapt the interfaces of the methods to the appropriate smaller object types. For example, for the project manager component previously shown in Figure 7, we could modify its interface, sending it the individual parts of books and projects. The result is shown in Figure 18.

This strategy looses one important invariant, namely that `n`, `t` and `a` are fields of the same book. This style was introduced *only* because of remote communication, but it ends up affecting the whole design of the application. Moreover, programmers are faced with having to split and reconstruct objects in arbitrary places of the code, without any rules or guidelines that encapsulate what is going on. This is a form of low-level marshaling in disguise. Although this style solves the problem at hand, it is a dangerous source of tangling.

```
public interface PManager
       extends rmi.Remote {
  boolean newBook(ProjId pid, int n,
                  String t, String a,
                  Price p)
          throws rmi.RemoteException,
                 ProjectNotFound;
  // other services ommitted
}

public class ProjectManager
       extends UnicastRemoteObject
       implements Pmanager {
  ProjectList projects;

  public boolean newBook(ProjId pid,
                         int n, String t,
                         String a, Price p)
  throws rmi.RemoteException,ProjectNotFound
  {
    //Project prj = b.get_owner();
    Project prj = projects.get_prj(pid);
    if (!projects.contains(prj))
      throw ProjectNotFound;
    return prj.newBook(n, t, a, p);
  }
  // other methods omitted
}
```

```
public class Project implements ProjectI{
  ProjId projectId;
  Person manager;
  PersonList workers;
  Budget bdgtCenter;
  ComputerList computers;
  BookList books;

  boolean newBook(int n, String t,
                  String a, Price p) {
    Book b = new Book(n, t, a);
    if (bdgtCenter.approvePurchase(p)) {
      books.append (b);
      return true;
    }
    return false;
  }
  // other methods omitted
}
```

Figure 18. Splitting the objects into their smaller parts.

2.3.2.2 *Class Transformations*

A better style is to encapsulate the previous procedure into class transformations. That is, define a special class for each situation that needs a special cut on the parameter objects, and implement class conversion methods. So, for the example above, we could implemented the BookI type in more than one class, as shown in Figure 19.

```
public class BookBookLocator
       implements BookI {
  String title, author;
  int isbn;
  public BookBookLocator(String t,String a,
                         int n) {
    title = t; author = a; isbn = n;
  }
  public String get_title(){return title;}
  public String get_author(){return author;}
  public int get_isbn(){return isbn;}
  public Book convert() {
    return new Book (title, author, isbn);
  }
}
```

```
public class BookPManager
       implements BookI {
  String title, author;
  int isbn; Project owner;
  public BookPManager(String t,String a,
                      int n, Project p) {
    title = t; author = a; isbn = n;
    owner = p;
  }
  public String get_title(){return title;}
  public String get_author(){
    return author;}
  public int get_isbn() {return isbn;}
  public Book convert() {
    return new Book (title, author, isbn,
                     owner);
  }
}
```

Figure 19. Implementation of classes related to Book. They all implement a basic interface, but they extend it with conversion methods for constructing “real” book objects.

For interfacing the different components (i.e. `BookLocator` and `Pmanager`), instances of these auxiliary classes are used, instead of instances of class `Book`. The programmer must make the appropriate conversion before the calls.

This style is more modular than the previous one, but it ends up creating a number of intermediate classes that make the program structure confusing.

2.3.2.3 The Serializer Design Pattern

Riehle [64] proposes a design pattern to stream objects into data structures and create objects from those data structures. It can be used whenever objects are written to or read from flat files, network transport buffers, etc. This pattern is more general than Java's serialization API or CORBA's externalization service [58], since it handles the partitioning of arbitrarily complex object graphs and it handles different data representation formats. Using this pattern requires a number of new classes to support its protocol and, as the authors warn, it weakens encapsulation, since some of these new classes must access the objects' internal state. Nevertheless, it is a useful pattern that addresses the problem in a well-structured way and that may produce less tangled code.

2.3.3. Summary

This section analyzed the code tangling problem with respect to programming styles and design patterns. The overall conclusion is that the tangling that occurs from programming cross-cutting issues can be made less severe if programmers follow a number of coding rules that introduce an additional meaning to the pieces of code. With appropriate documentation that includes reference to well-known patterns and informal identification of particular designs, programs can overcome the lack of expressiveness of general-purpose programming languages. The code, more or less tangled, becomes more comprehensible, because programmers understand, at least, the intentions behind the lines.

However, this approach has its limitations. Its main drawback is that there is neither automation nor formal enforcement to the process of coding the designs. Programming is still an exercise in manually weaving different concerns within the components. And, as such, programmers can still do all kinds of mistakes and dubious optimizations that produce confusing, if not buggy, code. Also, these design patterns introduce "noise" in the code. That is, because of concurrency and distribution programmers must hard-code a number of auxiliary components and relationships between them that are anything but obvious.

2.4. Tangling and Programming Languages

Programming practices are informal frameworks for capturing intentions into code. Programming languages are the executable notational mechanisms with which those intentions are described. Much of the work in programming languages comes from trying to capture those intentions more clearly. That was the case with the elimination of `go to`'s and its replacement with higher-level constructs such as `while` loops and the procedure abstraction.

As seen in the previous section, the complexity of the program texts can be reduced by carefully designing the applications and by, somehow, encapsulating the design decisions into modular units of code (that can be classes, functions or just lines of code). But the complexity of the program texts also depends on the particular programming language being used. When it comes to programming concurrency control and distribution, some language frameworks are better than others, in the sense that they provide formal support to encapsulate important design decisions. This section analyses the code tangling problem with respect to programming languages.

2.4.1. Basic Linguistic Support for Distributed Programming

2.4.1.1 Synchronization

One of the first linguistic concepts to reflect synchronization of concurrent threads was the *semaphore*, introduced by Dijkstra [17]. Figure 13 showed an emulation of this concept using a Java class. A semaphore provides indivisible operations for testing and modifying an integer value, and an associated queuing mechanism to block threads until a notification is issued. Semaphores are powerful enough to handle all synchronization scenarios. But because the threads wishing to access shared data are responsible for calling the semaphores in the correct order, semaphores are, as shown before, tricky to use and lead to unreliable code. The erroneous use of a semaphore compromises the integrity of the shared resource, and may deadlock the entire system.

In order to overcome the difficulties of programming with semaphores, Hoare introduced the concept of *monitors* [24]. Monitors shift the responsibility of the synchronization from the clients to the service providers. A monitor encapsulates a piece of shared data with the procedures that directly access and modify that data. Those procedures are responsible for handling mutual exclusion and for the integrity of the data; therefore the clients no longer need to synchronize before accessing that data, but they simply request the service to the monitor. Figure 15 showed the bounded buffer implemented as a monitor.

Monitors are simply constructs for mutual exclusion, and they lack the ability of performing guarded suspensions. Languages that have adopted the monitor concept usually have to incorporate an additional mechanism for handling conditional waits (see bounded buffer example). One approach is to use the concept of *condition variables* [74], which are syntactically similar to semaphores and participate in the queuing policy of the monitor. A second approach is to generalize the concept of monitor by the introduction of *guards*, which not only manage mutual exclusion, but also select the possible calls according to the state of the monitor.

2.4.1.2 Communication

The most basic primitive for communication between execution spaces is *message passing*. An execution space transfers data to another execution space by *sending* it a message, which the other space *receives*, accepting it or not. There many detailed variants of the basic message passing communication scheme, namely synchronous vs. asynchronous modes, blocking vs. non-blocking, unicast vs. multicast, naming conventions, etc. In order to cope with the unreliable media and limited buffering resources, many communication protocols have been defined on top of this basic primitive, the most popular ones being UDP and TCP. These protocols implement a layered architecture in which the unreliability decreases bottom-up.

Because communication over an unreliable network is such a complex problem, the communication protocol stacks are usually part of the operating system, and are made available to the applications through generic library routines that interface with the layers. In other words, what the programmer sees — typically the interface to the transport layer — is just the top of the iceberg, since most of the complexity is transparently handled by the lower layers of the protocol stack. This interface contains, at least, the services *send* and *receive* for passing either datagrams or streams of data.

On the application layer, the communication decisions are mostly related to the semantics of the application’s data (as opposed to being concerned with the reliability of the connection or the ordering of the packages). That is, how to partition the application’s components, what data to send to remote spaces, and when.

Specifically for object systems, some new layers have been defined on top of the transport layer, and below the application objects layer. But that will be the subject of section §2.4.3.

2.4.2. Concurrency in Object-Oriented Languages

The integration of concurrency into the object-oriented model has followed three basic strategies: 1) to add concurrency constructs which are orthogonal to the object-oriented programming features; 2) to achieve full integration at the same level; and 3) to separate the classes from the descriptions of their concurrent behavior.

2.4.2.1 Orthogonal Approach

The orthogonal approach was adopted by languages such as Smalltalk, Trellis/Owl [65], several C++ environments [5, 68] and Eiffel environments [31], all of which were originally sequential languages, as well as by new languages such as Obliq [12] and Java [23]. These environments provide some kind of semaphore objects and/or conditional critical regions, and it's up to the programmer to use these properly. The design philosophy is depicted in Figure 20.

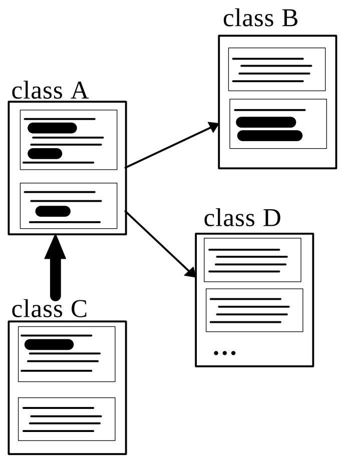


Figure 20. Orthogonal approach: no relation between the object-oriented abstractions (i.e. classes, methods and inheritance) and concurrency control. Methods use synchronization primitives (thick lines) in arbitrary places.

This strategy is certainly very flexible, and environments like these are widely used today. However, it has the disadvantage that concurrency control cross-cuts the language abstractions, making it very easy to write complex, tangled code. This was discussed in section §2.3.1.

Some of these languages provide a more varied set of abstractions than others. Obliq, for example, includes 11 concurrency-related constructs, built on top of the Modula-3 threads primitives. Figure 21 shows an implementation of the bounded buffer written in Obliq that is equivalent to the

```
let BoundedBuffer =
  (let nonEmpty = condition();
   let nonFull = condition();
   var takePtr = 0; var putPtr = 0; var usedSlots = 0
   var array = [100]; (* the array, size 100*)

   {serialized,    (* this means that this object is a monitor *)
   (* next, the methods *)
    put =>
      meth (self, obj)
        watch nonFull     (* wait, if it's full. This is a loop, and in each *)
          until usedSlots < 100  (* wake up, this condition is checked again *)
        end;
        array[putPtr] := obj;
        putPtr := (putPtr + 1) % #(array); (* #(array) is the size of the array *)
        usedSlots := usedSlots + 1;
        signal(nonEmpty); (* wake up one thread waiting on this condition *)
      end;
    take =>
      meth (self)
        watch nonEmpty    (* wait, if it's empty. This is a loop, and in each *)
          until usedSlots > 0     (* wake up, this condition is checked again *)
        end;
        let obj = array[takePtr];
        takePtr := (takePtr + 1) % #(array);
        usedSlots := usedSlots - 1;
        signal(nonFull); (* wake up one thread waiting on this condition *)
        obj;
      end;
});
```

Figure 21. The bounded buffer written in Obliq.

one in Figure 15. Besides the obvious syntactic differences, there are some subtle, and more interesting, differences between the code in Figure 15 and this one. Obliq includes the notion of *condition variable*, here illustrated by the identifiers `nonEmpty` and `nonFull`. Waiting and signaling is done on condition variables. This allows more fine-grained waiting and signaling strategies, than that of Java — which does it, at a lower-level, on the object's lock. As it is usually the case for lower-level mechanisms, higher-level constructs can be implemented on top of them; that is, we can easily implement a `ConditionVariable` class in Java, and use it appropriately. But, more important than that, condition variables capture the concept of coordination state, discussed in §2.3.1.4, making it much easier to track the coordination strategy of the objects than in languages like Java in which the coordination state space can be the whole object state space.

Obliq also includes the notion of *guard*, mentioned before, and implemented in this language by the `watch` statement. Guards check the object's state and wait on a certain condition variable if the object is in such a state that the method cannot be executed. Every time the condition variable is signaled, the guard test is checked again.

The code tangling problem also occurs in the Obliq implementation, since the code for coordination is embedded in the implementation of the methods. However, Obliq provides more built-in constructs for concurrency control than Java, and by using them the programmer makes a formal identification of the design decisions, which, potentially, makes the program less prone to programming errors and easier to understand. For example, in the Java implementation of Figure 15, a careless programmer could have omitted the `while` loop that contains the `wait` in the beginning of the methods, replacing it by a simple `if` statement. Such replacement would make the implementation unsafe, since some threads could execute the insertion and removal of objects even when the buffer is full or empty. By using the language's built-in guard construct, such errors will not occur.

2.4.2.2 Full Integration

The full integration approach usually unify the concept of object with the concept of monitor. These languages were designed for the purpose of supporting concurrency. The idea is that method activation does not take place as soon as an invocation is received, but rather when the receiver object decides that it can actually execute the method.

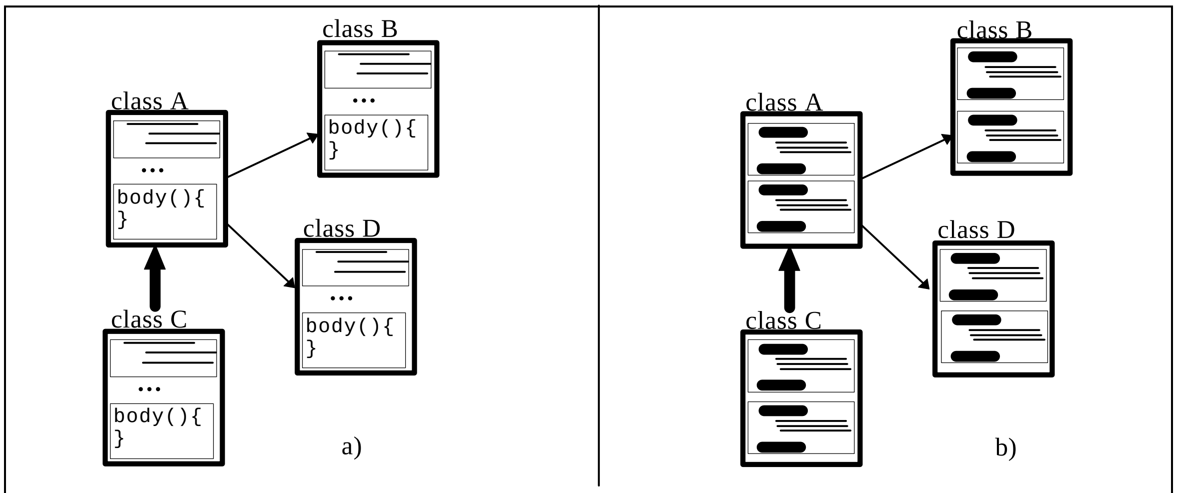


Figure 22. Full integration approach: concurrency control is tightly coupled with the object-oriented model, either: a) by associating with each class a special method for object activation and synchronization; or b) by having each method explicitly define its preconditions and effects on the coordination state.

There are two main ways of achieving this (see Figure 22). Some languages adopt a centralized design, by which the coordination strategy (i.e. the allowed method interleavings) of the class-as-monitor is specified in one special method — called the "body" — that dispatches the method invocations. Some languages taking this approach are POOL [3], ABCL/1 [77] and ACT++ [30]. Other languages adopt a more decentralized design, by which the coordination strategy of the class-as-monitor is dispersed among the methods themselves, in the form of guards and post-conditions. Some Actor languages [1, 29] and Hybrid [56] took this approach.

Some of these languages achieve yet another unification, which is the concept of object with the concept of thread of execution. That is, an object, besides being a monitor, contains its own thread of execution — the active objects model. Concurrency in these systems is of large granularity, since each object is potentially a new thread. Examples are Emerald [10], POOL, Concurrent Smalltalk [76], andABCL/1.

The full integration strategy provides, in principle, a simpler and safer framework for programming concurrent systems than the orthogonal approach, but it is associated with a major drawback that made it unpopular. Because of the philosophy under which these languages were designed, they imposed that the coding of concurrency control *should* be embedded in the source code of the methods; therefore, the inheritance anomalies were an immediate consequence, whether using a centralized or decentralized approach to coordination.

To illustrate these languages, and their related inheritance anomalies, Figure 24 and Figure 23 show two other versions of the bounded buffer, one using a body-like concurrent Actor language and the other using an Actor language supporting *behavioral abstractions* in the style proposed by Kafura [29]. The syntax used here is an hypothetical extension of Java that captures the important constructs of those languages. Languages of the type shown in Figure 24 define the concurrency scheme in only one place that can also include other arbitrary code. Languages of the type shown in Figure 23 impose that each method *must* explicitly specify the new state at the very end.

It's easy to see that these language designs present some problems when extending the class with other methods that partition the state space in different ways. The body-like languages concentrate the redefinitions in only one place (Figure 24), but for languages that decentralize the concurrency control (Figure 23) the redefinitions may be extensive.

The term "inheritance anomaly" was coined by Matsuoka in [51] (and later in [54]). But the problem had been mentioned before [3, 11, 29, 60, 72]. The source of the problem is that these language designs, having concentrated on supporting concurrency entirely within the object-oriented framework, imposed too much of a strong coupling between functionality and synchronization — so much so, that a fully general inheritance mechanism was not even useful. Because of that, some of these languages (e.g. Emerald, POOL, ABCL/1) have chosen to eliminate inheritance. Later designs [13, 20] were more careful about this issue. Matsuoka's own proposal in [54]

```
  pseudo-class BoundedBuffer : implements Actor { // monitor
    int putPtr = 0, takePtr = 0;
    int usedSlots = 0;
    Object array[];
    BoundedBuffer (int capacity) {
      array = new Object[capacity];
    }
    void putThread(Object o) { // the service method
      array[putPtr] = o;
      putPtr = (putPtr + 1) % array.length;
      ++usedSlots;
    }
    Object takeThread() {      // the service method
      Object o = array[takePtr];
      takePtr = (takePtr + 1) % array.length;
      --usedSlots;
      return o;
    }
    void body() {  // the body, which implements the coordination
     // this method can have lots of other code
      loop {
        select {
          accept put(Object o)
            when (usedSlots < array.length)
            start putThread(o);
          or
          accept take()
            when (usedSlots > 0)
            start takeThread();
        }
      }
    }
  }
```

Figure 24. Active objects with an explicit body.

```
pseudo-class BoundedBuffer : implements Actor { // monitor
    int putPtr = 0, takePtr = 0;
    int usedSlots = 0;
    Object array[];
  behavior: // this is the behavioral part that defines the coordination states
            // and the method sets that are enabled in those states
    empty = {put};
    middle = {put, get};
    full = {get};
    BoundedBuffer (int capacity) {
      array = new Object[capacity];
      become empty;
    }
    void put(Object o) {
      array[putPtr] = o;
      putPtr = (putPtr + 1) % array.length;

      if (++usedSlots == array.length) become full;    // state changes
      else                             become middle;
    }
    Object take() {
      Object o = array[takePtr];
      takePtr = (takePtr + 1) % array.length;
      return o;

      if (--usedSlots == 0) become empty;             // state changes
      else                  become middle;
    }
}
```

Figure 23. Behavioral abstractions.

is an interesting language design that includes the concepts of *synchronizers* and *transitions*. But these later designs clearly break from the full integration approach, being the beginning of the separation between functionality and concurrency control, which will be described next.

In conclusion, and from the perspective of the code tangling problem, full integration of concurrency has the advantage that concurrency is treated as a first-class issue, for which there are a number of powerful language constructs, associated with the objects themselves. However, concurrency and functionality are so strongly coupled together that it is impossible to isolate one from the other. This strong coupling is not accidental, but rather a consequence of the design principle of unification of concepts.

2.4.2.3 Separation of Coordination and Functionality

The third approach to the coexistence of classes and concurrency is to separate the classes from concurrency specifications. The basic idea of this approach is depicted in Figure 25. Classes are repositories of implementation, and the concurrent behavior is specified elsewhere.

Although this simple idea can be the basis for many, very different, language designs, the figure suggests several advantages of this approach over the other two approaches. First, there is no code tangling as such; programmers can concentrate on one issue at a time without being distracted with the "noise" introduced by the other issue. Second, there is no inheritance anomaly associated with concurrency, since classes are free of concurrency code. Third, dependent on the particular language, it may also be possible to reuse coordination schemes for different classes. And finally, also

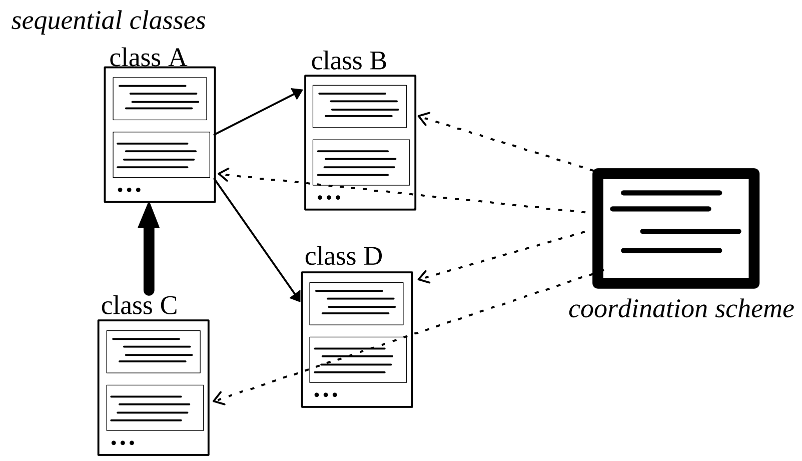


Figure 25. Separation of coordination and functionality: classes are written without synchronization code; the coordination scheme is defined in a separate module.

dependent on the particular language, concurrency control may be programmed on a more global basis, involving sets of collaborating classes.

This approach seems to be a good starting point for “separation of concerns.” As described in the next chapter, the D framework follows this approach. But other languages have followed it before, and there are many different ways by which the separation of functionality and concurrency control can be achieved.

Matsuoka [54] proposes the extension of the class abstraction with a separate part for dealing only with synchronization. That separate part is written in its own little language that is similar to the concurrency annotations of the language shown in Figure 23. His version of the bounded buffer can be seen in Figure 26. Sina [2, 7] follows a similar approach, although it uses a more generic *filtering* mechanism that can be used for purposes other than concurrency control. But synchronization is also defined on a separate part of the class, using a small language that associates enabled sets of methods to conditions on the coordination state of the objects.

```
pseudo-class BoundedBuffer : implements Actor { // monitor
    int putPtr = 0, takePtr = 0;
    int usedSlots = 0;
    Object array[];
  methodSets: // this part defines the coordination states
              // and the method sets that are enabled in those states
    mset EMPTY  #{put}
    mset FULL   #{get}
    mset MIDDLE EMPTY | FULL
  transitions:  // this part defines the coordination state machine
    transition default {
      become EMPTY when (size == 0);
      become FULL  when (usedSlots == array.length);
      become MIDDLE otherwise;
    }
  methods:   // finally, the ordinary methods
    // constructor missing (it’s the usual constructor)
    void put(Object o) {
      array[putPtr] = o;
      putPtr = (putPtr + 1) % array.length;
      usedSlots++;
    }
    Object take() {
      Object o = array[takePtr];
      takePtr = (takePtr + 1) % array.length;
      usedSlots--;
      return o;
    }
}
```

Figure 26. Extending the class abstraction with separate parts for synchronization (the `methodSets` and `transitions`).

DRAGOON [6] takes a different approach, and partitions the world in two kinds of classes: “functional” and “behavioral” classes, the latter being responsible for coordination.[2] “Functional” classes compose in the ordinary object-oriented way, that is through *uses* and inheritance. “Behavioral” classes are written using a completely different language than that of the “functional” classes, they can’t be instantiated, and they don’t compose through inheritance like the “functional” classes do. “Functional” classes compose with “behavioral” classes through the new relation *ruled-by*. “Behavioral” classes are written using a language based on logic assertions over abstract method invocation histories. The version of the bounded buffer written in this language is shown in Figure 27. The pseudo-Java language that has been used throughout this chapter replaces the original Ada-like syntax, since the latter shows a number of particular features DRAGOON that are irrelevant for purposes of this discussion.

```
pseudo-class BoundedBuffer {    // the “functional” class
    int putPtr = 0, takePtr = 0; int usedSlots = 0; Object array[];
    // constructor missing (it’s the usual constructor)
    void put(Object o) throws IsFull {
      if (usedSlots == array.length) throw new IsFull();
      array[putPtr] = o;
      putPtr = (putPtr + 1) % array.length;
      usedSlots++;
    }
    Object take() throws IsEmpty {
      if (usedSlots == 0) throw new IsEmpty();
      Object o = array[takePtr];
      takePtr = (takePtr + 1) % array.length;
      usedSlots--;
      return o;
    }
    boolean isFull() { return (usedSlots == array.length); }
}

behavioral pseudo-class WaitBuffer {    // the coordination class
  ruled PutOps, TakeOps, FullGuard;  // abstract sets of methods
  // next, the rules for when these sets can be executed
  // the “><” means complete exclusion with the other sets
  permission(PutOps) ⇔ (><) and (not FullGuard);
  permission(TakeOs) ⇔ (><) and (activations(PutOps)-activations(TakeOps) > 0);
  permission(FullGuard) ⇔ (><);
}
// finally, THE class
pseudo-class WaitingBoundedBuffer
  extends BoundedBuffer  // the regular inheritance relation.
  ruled by Wait          // the connection to the behavioral class.
    where put => PutOps,        // renaming rules
          take => TakeOps,
          isFull => FullGuard { // empty class  }
```

Figure 27. DRAGOON’s version of the separation between functionality and coordination.

[2] The words “functional” and “behavioral” are written inside double quotes because they are DRAGOON’s own terminology.

DRAGOON's "functional" class and "behavioral" class are completely unaware of each other; they only come together on the concrete class, through a renaming mechanism that maps the abstract sets of the "behavioral" class into the methods of the "functional" class. Because of this, the "behavioral" class must rely on the "functional" class to provide the necessary methods for inspecting the object's state (see method `isFull`, which did not exist in all other implementations). This creates an implicit dependence between the "functional" and the "behavioral" classes that is everything but obvious, and raises questions about how general the compositionality between the abstractions of the language really is. Second, since the "functional" class may be used on its own, it becomes necessary to test the error conditions (see the exception thrown in the methods). This is a consequence (not necessarily bad) of the total independence between the methods and the coordination.

Separation between functionality and synchronization has been proposed with other flavors, such as using reflection [15, 52, 55, 73], concurrency annotations [46] and other extensions to an object-oriented language [22].

In conclusion, this third approach to integrate concurrency in object-oriented languages is very promising: it achieves the goal of drastically reducing the code tangling between class implementations and the coordination of threads on concurrent environments; it has the potential to make full use of the sequential object-oriented model that has become so popular; and it has the potential to isolate the coordination schemes, so that a certain, maybe informal, reasoning can be done over those schemes. But, as seen, there is a large space for designing these kinds of languages, and some points in that space are possibly better than others.

2.4.3. Communication in Object-Oriented Languages

Not by coincidence, the integration of remote communication into the object-oriented model has followed the same two first strategies: 1) to add communication constructs which are orthogonal to the object-oriented programming features; and 2) to achieve full integration at the same level. Due to the many drawbacks of both approaches, most languages have followed an hybrid approach.

2.4.3.1 Orthogonal Approach

One way of integrating remote communication in the object model is by using the low-level communication primitives that the operating system provides. This has been called by Atkinson [6] the "operating system" approach. One way of doing it is by wrapping the communication primitives in classes, giving them object-oriented interfaces, say a Socket class, an IPAddress class, etc. A good example is Java's network API [26]. The resulting programs look very much like Figure 20.

While this approach permits the most efficient implementations of inter-machine communication, having to manually convert the object-oriented entities into the representations expected by those low-level communication primitives is an overhead and a source of tangling. This approach hard-codes the distribution in the classes, making it impossible to use those classes in different architectures, and making it difficult to understand the functionality independent of the data transfer strategies (and vice-versa) due to the unavoidable code tangling.

In short, this approach reduces the reliability of the application's code because remote communication is an uncontrolled issue that escapes the language's rules.

2.4.3.2 Full Integration

A different approach is to integrate the remote communication with the object model. The concepts being unified are *objects* and *execution spaces*, as well as *method invocation* and *message passing*. In its more extreme formulation, the programmer has no knowledge of the target distributed configuration, and develops the application as if it were to be executed in one single machine. This approach delegates to the compiler most, if not all, of the responsibility of splitting the program into network components; the language run-time will be responsible for providing all the necessary mechanisms that support the semantics of the language. In this scenario, the code tangling due to communication issues is almost non-existent, since distribution is made transparent.

At first, it would seem that classes and objects from the programming languages world are ideal constructs for acting as network components in a distributed system. They are separately 'compilable' entities that interact by means of method invocations to well-defined interfaces, and that name each other indirectly. There are, however, some fundamental issues of distributed computing that makes this image less than perfect.

First of all, remote method invocations are more costly than local invocations, and the cost varies with the type of the networks (i.e. LAN, WAN, etc.). A uniform semantics for method invocations — the one that preserves object identity and integrity as if the network was only one execution space — has severe consequences on the performance of the applications. Objects cross-reference each other intensively, they pass other object references around in method invocations, and they create new objects whose references they return as the result of invocations. Having a unique, global, object reference space as the basis for network interaction is unfeasible for practical purposes, because the number of cross-space method invocations increases drastically as the application executes, with uncontrollable, negative effects on the application's performance.

Second, distributed systems, being about sharing, are also about protecting the data. And this is very different from the notion of "encapsulation" provided by the languages, which basically guarantees that the object's state will only be altered by the object itself. This guarantee is too weak for distributed systems. Having unique interfaces to the objects violates the necessary protection boundaries, because as soon as an execution space gets a reference to a remote object (say, a bank account), it can invoke any of the methods of its interface. There is a level of protection on space boundaries that is not properly captured by the centralized object-oriented model.

Due to these difficulties, most distributed object-oriented languages have followed mixed paradigms. Some languages, however, achieved a relatively good integration, most notably Emerald [28] and Obliq [12]. Emerald was one of the first distributed object-oriented languages designed within the full integration approach. It provides a uniform model, where everything is an object (just like Smalltalk), and where object invocation is location-transparent and has the same semantics whether it's local or remote. In order to cope with the performance penalty issue, Emerald also includes a set of primitives for controlling object location: `locate`, `move`, `fix`, `unfix` and `refix,` and the static qualifier `attached` (for grouping objects that should move together). `move` is the migration primitive that recursively transfers parts of the object graph and their associated threads of execution from one space to another. Programmers can use these primitives anywhere in the code. Additionally, these primitives have been embedded with method invocation, so that programmers can chose a number of different parameter passing modes. This means that the distribution strategies are hard-coded in the implementation of the methods, producing some "noise" on the functionality; but this is not so bad, because these are identifiable primitives with very well defined side-effects. Also, since Emerald does not support inheritance, there aren't any inheritance anomalies associated with location control. Some performance tradeoffs were discussed in [28].

Obliq provides a fully transparent object reference space across the network, where objects are always bound to the location where they were created. Objects, other than of primitive types, are never copied. The language does not provide any primitives for location control. Of all the distributed object-oriented languages, Obliq is the one that most closely achieved the full integration between objects and communication (in one simple way: restricting the data sent across spaces to being object references and primitive data values). Therefore, the code tangling due to issues of remote communication is basically null. But because of the full integration, the language suffers from the two drawbacks that were identified before (i.e. performance penalties and lack of protection). If programmers need to tune these issues, they need to decompose the objects in their basic data types and send the pieces, instead of the objects, in method calls (as discussed in §2.3.2.1). Obliq has been integrated with a graphical user interface development environment, which was used to develop simple distributed applications [8].

2.4.3.3 Hybrid Approaches

Most language environments for developing distributed applications have followed a hybrid approach that uses some features of the language as the basis for modeling communication, while still allowing a fair amount of low-level practices that address the difficult issues of distribution.

With respect to the model of communication, these approaches can be grouped in two categories:

- asynchronous interaction. These languages support method invocations as independent sends and receives of messages. There aren't that many object-oriented languages that followed this approach, since it clearly establishes an inconsistent relationship between sequential, local invocations (which are 'synchronous' method calls) and remote invocations. Exception are ABCL and Erlang [4].
- synchronous interaction. These languages support remote communication through some remote method invocation mechanism, where the caller blocks waiting for the result of the method invocation. Many language environments followed this approach: Emerald [9], DRAGOON [6], IK [68], Obliq [12], Java RMI [27].

With respect to the data that is allowed to be transferred between spaces, these approaches can be grouped, roughly, as follows:

- dual object model. These languages are based on an object model that assumes two kinds of entities: the virtual nodes and the data objects. The former are the "server" entities that manage the data objects, and that communicate with each other sending those data objects. There is no communication, as such, between data objects. Argus [45] is a good example of this approach.
- impure object model. These languages support class types and conventional data types (e.g. records or structs). Instances of classes (i.e. objects) have unique references and are never

copied, whereas data structures are always copied. ABCL/M [71], DRAGOON [6] follow this approach.

- uniform object model, dual parameter passing modes. Some languages (e.g. C++, Smalltalk, Eiffel) support pass-by-reference and pass-by-copy on local method invocations. Distributed implementations of these languages could take advantage of this.
- uniform object model, dual types. Parameter passing modes are external to the language, but they are introduced indirectly by the types. That is, the communication run-time decides to pass-by-copy or pass-by-reference on a type basis. Examples are IK [68], Java RMI [27], and CORBA [57].

Of all these hybrid approaches, the ones that fit more naturally into the object-oriented paradigm are the ones that have synchronous remote method invocation and that chose the parameter passing mode on a type-basis. The type-based approach to parameter passing avoids the need to manually convert objects to/from other kinds of data types. However, it is still insufficient, since the data to be sent across spaces may depend on the particular method that is being invoked (this issue was already mentioned in §2.3.2). Emerald's location primitives, for example, address this issue in a more general way.

2.5. Final Remarks

This chapter analyzed the problem of code tangling that occurs when distributed applications are written using current programming languages and practices. First, the problem of code tangling was presented. Then it was analyzed with respect to programming practices. Java and its RMI API, the language environment used to illustrate programming practices, captures two major trends in languages for programming distributed applications: (1) an orthogonal approach to dealing with concurrency control and (2) a type-based approach to dealing with remote communication. Finally, code tangling was analyzed with respect to programming languages. A number of other languages were presented that have taken approaches different from Java's.

The key points of this chapter can be summarized as follows:

- Current object-oriented programming languages provide good abstraction and composition mechanisms for programming functional components.

- Synchronization of concurrent threads and communication between execution spaces relate to the application's functional components in ways that the current composition mechanisms don't capture well. Because these two issues must be programmed, they end up being inserted in the components' code in more or less arbitrary ways, resulting in programs that are *tangled*.
- The complexity of programming these cross-cutting issues can be lessened by applying well-known design patterns or imposing a number of coding rules. However, this approach is not very reliable, because it is based on rules of thumb, lacking automation and formal enforcement.
- The complexity of programming cross-cutting issues can also be lessened by using programming languages that provide abstraction and composition mechanisms specifically designed for addressing these issues. This approach results in programs that are more reliable, since the coding of these issues is made within the rules of the language.
- There is a large design space for distributed object-oriented programming languages. One region that looks promising is the one that separates the coding of functional components from the coding of coordination and communication. This allows taking advantage of the sequential object-oriented model for programming components, programming concurrency and distribution as separate aspects. It fits well in the basic engineering principle of "separation of concerns."

Chapter 3

The D Framework

“**20. Keep related words together.**

The position of the words in a sentence is the principal means of showing their relationship. Confusion and ambiguity result when words are badly placed. The writer must, therefore, bring together the words and groups of words that are related in thought and keep apart those that are not so related.”

William Strunk Jr. and E.B. White, in *The Elements of Style* [70]

3. The D Framework

The previous chapter showed how synchronization of concurrent threads and communication between execution spaces relate to the applications' units of functionality in ways that the composition mechanisms of current object-oriented languages don't capture well. At the same time, it set up the motivation for programming languages that provide abstraction and composition mechanisms specifically designed for addressing those issues. A number of such experimental programming languages were shown. However, all of those proposals have drawbacks that outweigh their advantages, ultimately making their languages unpopular.

This chapter describes another experiment in language design for distributed programming. Based on the analysis made in the previous chapter, a language framework called D has been designed. D follows the principle of "separation of concerns," to the extent that such separation is natural and intuitively appealing. The reason for calling it a "language framework" rather than a "language" is that it really consists of two languages: (1) COOL, for controlling thread synchronization over the execution of the components; and (2) RIDL, for programming interactions between remote components. These two languages were designed without committing to any particular language for programming functional components (from here on, "component language"). They do, however, establish a set of assumptions about the component language. The overall structure of D programs is depicted in Figure 28.

This chapter describes and discusses the design of the aspect languages, their assumptions with respect to the component language, and the composition mechanisms of D. Section §3.1 states the three principles under which most design decisions were taken. The specifications of the languages of D, in section §3.2, are given without committing to any particular implementation. The specification of each of the languages in sections §3.2.4 and §3.2.5 is given in terms of: (1) grammar productions, (2) one or more informal definitions, (3) informal semantic rules, and (4) illustrative examples. The discussion of design decisions and alternatives is concentrated in section §3.3.

D, as specified here, has been implemented using Java as the component language, and this concrete implementation is called DJ. Appendix A contains the syntax, and Appendix B presents an introduction to programming in DJ.

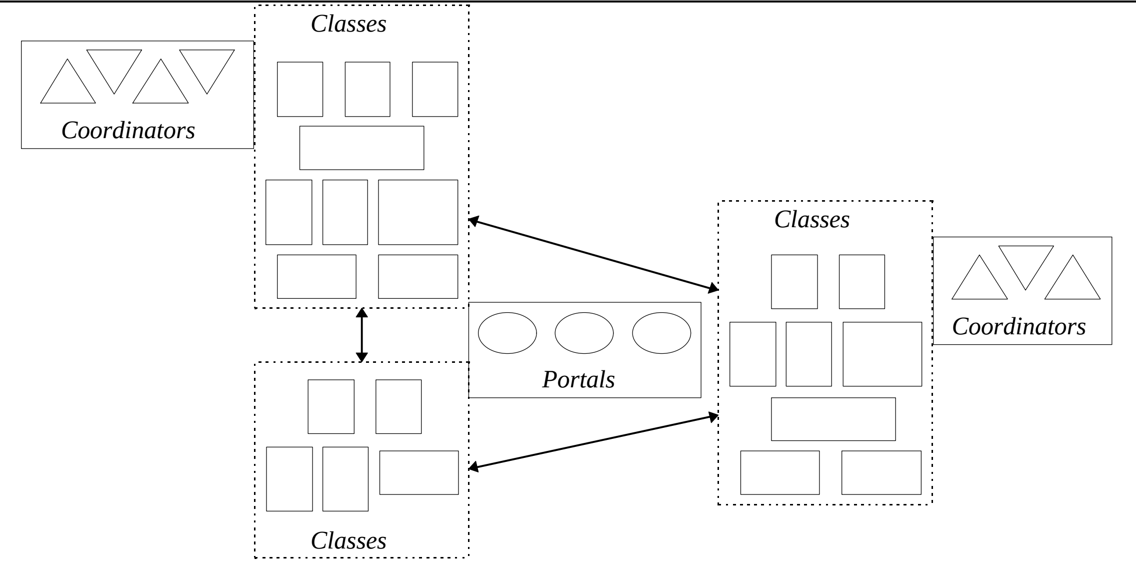


Figure 28. The D language framework: components (rectangles) are compose using the ordinary object-oriented composition mechanisms (i.e. method calls and inheritance); thread coordination is concentrated in special constructs called coordinators (triangles); remote interaction between components is declared in special constructs called portals (circles) that are true “remote interfacing” programs. Both coordinators and portals are in close relation with the component implementations .

3.1. Design Principles

This section describes the principles under which most of the design decisions were taken.

3.1.1. Separation of Concerns: Identification of Aspects

The ultimate goal of D is to help programmers achieve a clear separation of concerns throughout the development of distributed applications [25]. Therefore, the most basic design principle of D is “separation of concerns.” What this principle states is that if a problem can be analyzed under different — overlapping, orthogonal, complementary, or hierarchical — views of smaller complexity then we should analyze each of those views in order to be able to solve the whole problem. This is, of course, the old engineering principle of divide and conquer, and it is abstract enough that it can be concretized in many different ways.

In this thesis, this principle is applied as follows. If there are *issues* in the domain such that

a) it is natural for programmers to think about them in relative separation from the implementation of the functional components, and

b) their implementation using current language technology results in having to artificially modify the coding of the components, and to arbitrarily insert code within the implementation of the components,

then we should pursue the goal to maintain that separation in the source code itself by designing abstraction and composition mechanisms that address those issues in separate. Doing so, the source code will be closer to the programmer's intentions, avoiding the need to manually tangle and mentally untangle code for the many concerns of the implementations.

We have called these issues *aspects* [37], to differentiate them from components (which fail on b)). As the previous chapter suggests, there is no crisp boundary between components and aspects. They lay in the gray zone between application design and language design. Nevertheless, this definition of aspect gives at least an awareness for identifying issues that cause code tangling and that should be handled with special care. Depending on how important the issue is, it may be desirable to handle it through a set of specific rules — i.e. language constructs or even its own language.

That is what happens in D. Two important issues of distributed systems were identified that show indications of being aspects: thread synchronization and remote access. Because these issues are in the core of distributed systems, it seems worthwhile to try to capture them in separate from each other and from the components, and to carry that separation throughout the many phases of the application development, including the implementation phase.

Separation, however, does not mean complete separation. There are many interpretations of this word, different from its connotation with the black-box abstraction. A separation mechanism may simply allow us to temporarily forget the details of parts of the system that are irrelevant for the part at hand, and yet provide information about the internals of the other parts that is useful for the part at hand. The aspect programs in D know much more about the components than just the operations and variables they export: they know a lot about the components' implementation. That's the reason why they are *aspect* programs in the first place.

3.1.2. Control over the Separation

A second design principle is "encapsulation of responsibility." Once the aspects have been identified, it is necessary to define their regions of influence and the protocols between them and the components.

This is a critical issue in the design of any programming language. First, it involves limiting what the languages can do, defining the "right" cuts on generality. As Hoare has put it in his now famous quote, “the most difficult problem of language design is deciding what to leave out.” This is particularly important for aspect languages: they should be kept under a tight control, so that they can’t introduce chaos in the components, and their presence shouldn’t be overwhelming or intrusive. Secondly, control over the separation involves defining new abstraction and composition mechanisms for addressing those protocols appropriately.

This last point is particularly important for distributed systems. In Chapter 2 it is suggested that ordinary classes of sequential, non-distributed systems do not provide appropriate abstractions for programming distributed systems. Their role as repositories of implementation of functionality does not align well with the special needs of concurrency control and remote interaction, giving rise to the cross-cutting effects in the code. Something else is necessary.

3.1.3. Integration with Existing Languages

Another design principle for the aspect languages was “incremental innovation rather than revolution.” While it is too tempting to design new languages from scratch, it is also the surest thing to do to condemn them to oblivion. As Wulf points out [75], there is an exceptionally long design-acceptance-use cycle in programming languages; programmers and managers are understandably reluctant to change languages because of the large personal and financial investment involved in learning a new language and writing a high-quality compiler for it. That is even more true 17 years after Wulf’s observation.[3]

But, in the case of D, cost is not the only reason for preferring smooth transitions. The existing object-oriented programming languages are reasonably good tools for programming components. Therefore, there is no need to re-invent the wheel or even to impose artificial restrictions on the existing languages for the sake of programming distribution. On the contrary, the challenge is to design add-ons that address the aspects without interfering with the language used to program sequential, non-distributed components.

One consequence of this design principle is that the aspect languages of D were designed having a specific model of computation in mind that seems to be the preferred one, according to the major object-oriented languages, C++, Smalltalk, CLOS, Java and Eiffel. All of these have been integrated in concurrent and distributed environments using orthogonal approaches (see Chapter 2). Even Java, the newest of the four, integrated concurrency and distribution in a non-intrusive way.

[3] Java being the exception that proves this rule. Java is C++ intersected with many interesting languages that have failed to being accepted in general (CLOS, Dylan, Emerald, Modula-3, Self, etc.)

There seems to be a perfectly understandable preference for keeping concurrency and distribution away from the core of the languages. After all, most of the effort in the design of an application goes to the components; having to deal with the component's concurrent and distributed behavior all the time is an unnecessary overhead.

3.2. Specification of the Languages

This section concentrates on the specification of the languages of D, without committing to a particular component language or to a particular implementation. It serves as the framework's reference manual, and contains no justifications for why the languages were designed as described. All justifications and discussion of alternatives are given in section §3.3.

3.2.1. Conventions and Notation

The specification of D is given in an informal way, using English and illustrative examples, as opposed to using a formal notation. There are, however, some pieces of formal notation, conventions and inter-dependencies that need some explanation.

Cross-references

- All cross-references among language constructs are appropriately documented with "(§*Number*)", where *Number* is a (sub-)subsection number where the construct is defined.
- Cross-references from the language specification to the design decision(s) involved in a particular construct are given at the end of that construct specification as a list of "DD§*Number*", where *Number* is a sub-subsection number.
- Cross-references from the design decisions back to the language constructs they affect are marked in the beginning of each design decision (sub-)subsection, with the sub-section number where that design decision was referenced.

Notation

Terminal symbols are shown in `fixed width` font, and keywords are shown in **`bold fixed width`** font. Non terminal symbols are shown in *Italic* type. The definition of a non terminal is introduced by the name of the non terminal being defined followed by a colon. One or more alternative right-hand sides for the non terminal then follow; the alternatives are separated by the character "|". New lines and indentation are meaningless. The subscripted suffix "*opt*", which may appear after a terminal or non-terminal, indicates an optional part. Cross-references to a forward definition of a non terminal may appear further right of the occurrence of the symbol.

To make the grammar more concise, the productions for list of symbols are omitted. There are, however, two kinds of list of symbols that appear frequently: a simple list, which consists of a sequence of symbols, and a comma list, which contains commas between the symbols. The convention is as follows. A simple list of *Foo* symbols, concatenated with the suffix "*_List*", is defined as

```
Foo_List:
        Foo |
        Foo_List Foo
```

A comma list of *Foo* symbols, concatenated with "*_CommaList*", is defined as

```
Foo_CommaList:
        Foo |
        Foo_List  ,  Foo
```

3.2.2. The Component Language

The design of D is mostly independent of the component language. The most important assumption is that it is an object-oriented language. This subsection describes the general requirements for such language. These specifications come from how the aspect languages were designed, and what they expect from the component language; they define the least common denominator for an object-oriented language that can be integrated with the aspect languages of D.

Following these specifications, D has been integrated with Java, and the result is called DJ (Appendix B). Although not part of the claims of this thesis, it is speculated that it should be possible to integrate D with many other object-oriented languages that comply with the requirements described in this subsection.

3.2.2.1 Types, Values and Variables

There are two categories of types: primitive types and user-defined types. The particular primitive types depend on the specific language, but they are typically boolean, numeric and character types. The user-defined types include at least the class types (see §3.2.2.2 below). (DD§3.3.2.1)

There are two categories of data values: primitive and reference values. A primitive data value is a value of a primitive type. A reference value holds the reference to an object. An object is a dynamically created instance of a class, and it has a unique identifier (i.e. its reference). In other words, objects are always handled indirectly through their references. (DD§3.3.2.2)

A variable is a storage location that holds primitive or reference values. Variables are typed, that is, the type of the values that a variable holds is known at compile time. Strong typing helps detecting errors at compile time, since it limits the operations supported on values and helps determine the meaning of those operations. A variable of a class type C can hold a null reference, a reference to an instance of class C or a reference to an instance of any subclass of C.

3.2.2.2 Classes

The term "class" has been used to capture three distinct concepts. First, a class is module that contains a *repository of implementations*, i.e. a set of variables and operations defined within a lexical scope. Secondly, a class is a *template* for the generation of structurally and behaviorally identical objects (the instances). Finally, a class is also a *type*, in that it identifies objects which respond to the same set of operation requests.

Classes may *use* other classes. That is, the implementation of the operations may involve invocations to instances of other classes. Such instances may be stored in class or instance variables of the class, or may be passed as parameters to method invocations. This is the object-oriented version of the conventional composition mechanism between software modules. It is assumed that classes do not contain inner classes.

Classes may *inherit* from other classes. Inheritance is, first of all, the mechanism with which a class includes, and possibly modifies, variables and methods defined in another class (the base class). Subclasses can define new variables and methods, and they can also redefine (overwrite) the implementations of the operations of the base class.

3.2.2.3 Creation of Threads

Threads are sequential virtual processors that are created in order to execute concurrent activities in the application. The component language environment must provide the necessary support to create new threads. This can be done either by constructs of the language or by interfacing a thread library. (DD§3.3.2.3)

3.2.2.4 The Meaning of Objects, Threads and Execution Spaces

The meaning of these abstractions in D is the meaning they have in Java [23]. Some clarifications need to be made, however, with respect to concurrency and remote access.

A program consisting only of classes is a valid, executable program. That is, if concurrency and distribution are not necessary, then the framework is completely transparent, and the program is an ordinary program written in the sequential component language.

Synchronization is an issue that D tries to capture as a separate aspect. Therefore, classes are devoid of code for concurrency control. But a program may have several concurrent threads (see §3.2.2.3). The default synchronization strategy, without COOL's coordinators, is that there is none: in the presence of multiple threads, all methods of all objects can be executed concurrently (DD§3.3.2.4).

Remote interaction is the other issue that D tries to capture as a separate aspect. Therefore, classes are also devoid of code for remote communication. A program, without RIDL's portals, is not a distributed program — that is, the default communication strategy is that there is none. A program becomes a distributed program only if portals are defined, using RIDL. (DD§3.3.2.5)

3.2.3. The Visible Elements of Components

As already mentioned in §3.2.2.2, aspect modules (i.e. coordinators and portals) can access the components (i.e. classes) they are associated with. This access, however, is ruled by a precise protocol, which is at the very core of the concept of aspect. Such protocol may be different for each aspect, since the different concerns dealt with may require different "cuts" on the components. This subsection explains the concept of visible elements of components, which is the basis for the aspect/component protocols.

The portions of the components that can be used by the aspect modules are called the visible elements of components. Visible elements form a special kind of environment that aspect modules can use. At the same time, they establish a dependency context between components and aspect modules.

COOL and RIDL share one kind of visible elements of classes, but differ on others. More precisely, in D, the visible elements of a class *C* are, at least, the following:

- all method signatures, public, protected and private, of *C*;
- all non-private method signatures of the superclasses of *C*.

If method *m* declared in a superclass of *C* is overridden along the inheritance hierarchy, then there is only one visible method *m* in C, namely the closest to *C* in the class hierarchy.

Sections §3.2.4.1 and §3.2.5.1 indicate the complete set of visible elements for each aspect language.

The handles for these visible elements are the names they have in the classes. For example, if a class has a method named *f*, then the class's aspect modules can access this part using the name *f*; if a superclass has a non-private method named *g*, then the class's aspect modules can access it using the name *g*. Viewing an aspect module as a different representation of the classes it is associated with, the rule above follows the one in object-oriented programming languages.

The access rights for visible elements are identical for COOL and RIDL. Coordinators and portals have only inspection rights (in reflective systems this is called introspection). That is, aspect modules can neither modify the state of the objects nor invoke methods of the objects.

3.2.4. The Coordination Aspect Language

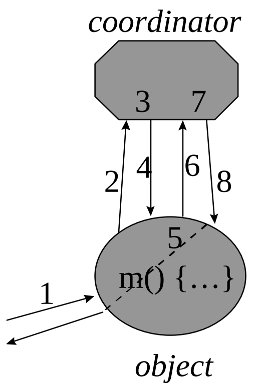
COOL provides means for dealing with mutual exclusion of threads, synchronization state, guarded suspension and notification, in relative separation from the classes. A COOL program consists of a set of coordinator modules:

```
COOLProgram:
        CoordinatorDeclaration_List                              (§3.2.4.2)
```

Coordinator modules (coordinators, for short) are associated with the classes on a name basis. A single coordinator may coordinate more than one class. Coordinators are helpers with respect to the implementation of the classes: they take care of thread synchronization over the execution of the methods. The smallest units for synchronization are the methods. Coordinator declarations (§3.2.4.2) describe those coordination strategies.

Coordinators are not classes: they use a different language, they cannot be directly instantiated, and they serve a very specific purpose. Coordinators are automatically associated with the instances of the classes they coordinate at instantiation time, and throughout the life of the objects this relation has a well-defined protocol, which is depicted in Figure 29.

Coordinators are written with knowledge of the classes they coordinate, and the implementation of a coordinator can and should be aware of the implementation of those classes, so that the best coordination strategies can be defined. Classes are unaware of coordinators, i.e. it is not possible



1: Within thread T, there is a method invocation to method m of the object, say obj.m()
2: The request is first presented to the object's coordinator
3: The coordinator checks exclusion constraints and pre-conditions for method m. If any of those constraints is not met, T is suspended. When all constraints are met, T has the right to execute method m. Just before it does so, the coordinator executes its on_entry statements for method m.
4: The request proceeds to the object.
5: Thread T executes method m in the object.
6: As the method invocation returns, the return is presented to the coordinator.
7: The coordinator executes its on_exit statements for method m.
8: The method invocation finally returns.

Figure 29. Protocol between an object and its coordinator.

for a class to name a coordinator. The association between a class and a coordinator is driven by the coordinator, not by the class.

At run-time, and by default, the association between objects and coordinators is one-to-one, and it is called coordination "per object." However, a coordinator may also be associated with all objects of one or more classes, and that is called coordination "per class." A coordinator may be declared `per_class` (§3.2.4.2) and must be declared `per_class` if it applies to more than one class.

The body of a coordinator may have condition variable declarations (§3.2.4.4), ordinary variable declarations (§3.2.4.5), one self-exclusion method set (§3.2.4.6), several mutual-exclusion method sets (§3.2.4.7), and method managers (§3.2.4.8). The methods referred to in the coordinator's body must be valid methods of the coordinated classes.

The exclusion sets capture the strategies for mutual exclusion of threads over the execution of the methods; these strategies are expressed in a declarative form. The condition variables capture the synchronization state; synchronization actions, i.e., suspension and notification of threads, are performed based on the synchronization state. Ordinary variables keep track of the rest the coordinator's state that doesn't lead directly to synchronization actions, but that may affect the synchronization state (typically, it is used for keeping track of method invocation histories). The method managers operate on the coordinator's variables (both condition and ordinary variables), declaring pre-conditions and modifying the values of the coordinator's variables. Changing the value of a condition variable results in issuing automatic notifications to threads that are waiting on pre-conditions including those variables. Coordinators are atomic entities, that is, all the computation made by a coordinator itself (pre-condition checks and state changes) is guaranteed to be thread-safe and free of race conditions.

3.2.4.1 Visible Elements of Classes

The complete set of visible elements for COOL is: (1) the visible elements described in §3.2.3; (2) all variables, private, protected and public, of the classes the coordinator is directly associated with; and (3) all non-private variables of the superclasses of the classes the coordinator is directly associated with.

Because COOL's coordinator may be associated with several classes, the handles for the visible elements in COOL may be qualified names of the form:

> *QualifiedName:*
> > *ClassName* **.** *VisibleElementName*
>
> *ClassName:*
> > *Identifier*
>
> *VisibleElementName:*
> > *Identifier* | `*`

When the context in unambiguous the *ClassName* (as well as the dot) may be dropped. The symbol '*' denotes the wild card (i.e. all parts).

***Related Design Decisions*:** DD§3.3.1.1.

3.2.4.2 Coordinator Declaration

A coordinator declaration establishes an association between the coordinator being declared and a set of classes within execution spaces:

> *CoordinatorDeclaration:*
> > *Granularity*$_{opt}$ **`coordinator`** *ClassName_CommaList*
> > *CoordinatorBody*
>
> *Granularity:* **`per_class`**

The *ClassName_CommaList* is a list of class names. Each class can be associated with at most one coordinator.

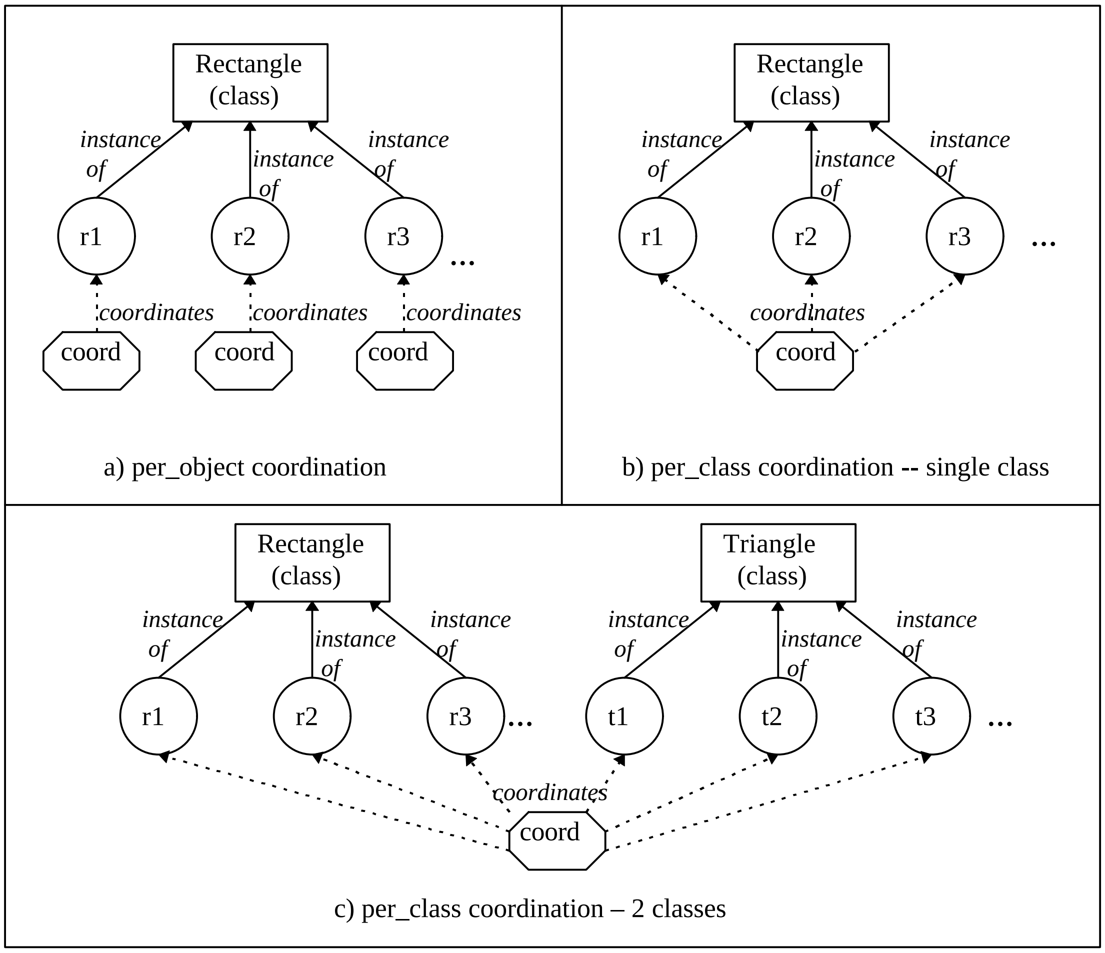

Rectangle (class)
instance of
r1
r2
r3
...
coordinates
coord
a) per_object coordination
b) per_class coordination -- single class
Triangle (class)
t1
t2
t3
c) per_class coordination – 2 classes

Figure 30. per_object and per_class coordination.

The *Granularity* of a coordinator defines whether it coordinates each instance of a class or entire classes (see Figure 30). If the granularity qualifier is omitted, then it is assumed to be per object, and only one class name must be specified (Figure 30.a). For example,

```
coordinator Rectangle {  // declaration for Figure 30.a
  // coordinator body
}
```

In this case each instance has its own coordinator, with its own coordination state (§3.2.4.4, §3.2.4.5); each of those coordinators uses the same coordination strategy — as defined in the coordinator's body.

If the coordinator is declared `per_class`, all instances of the coordinated classes that exist in an execution space share the same coordinator. For example, the following declaration defines a coordinator that is shared among all instances of class Rectangle (see Figure 30.b):

```
per_class coordinator Rectangle {  // declaration for Figure 30.b
  // coordinator body
}
```

The following declaration defines a coordinator that is shared among all instances of the classes Rectangle and Triangle (see Figure 30.c):

```
per_class coordinator Rectangle, Triangle { // declaration for Figure 30.c
  // coordinator body
}
```

The granularity is either per object or per class; there is no means to associate coordinators, by declaration, to particular instances of particular classes. This is a consequence of using JCore, a class-based language. In class-based languages the behavior of instances of a class is similar; objects may have instance fields and class fields, but there is no means to express, by declaration, behavior for particular instances of a class.

Particular objects may, however, be coordinated differently in the coordinator body (but not in the coordinator declaration). §3.2.4.11 presents one example of this.

It is an error to declare a per object multi-class coordinator. The following declaration results in a weave-time error:

```
coordinator Rectangle, Triangle {  // Error: must be declared per_class
  // coordinator body
}
```

Related Design Decisions: DD§3.3.1.1, DD§3.3.1.2, DD§3.3.1.3, DD§3.3.3.1, DD§3.3.3.2, DD§3.3.3.3

3.2.4.3 Coordinator Body

The coordinator body encapsulates the synchronization state and the constraints on the concurrent execution of the methods of the coordinated classes:

CoordinatorBody:
{
CondVarDeclaration_List$_{opt}$ (§3.2.4.4)
VariableDeclaration_List$_{opt}$ (§3.2.4.5)
SelfExclusiveMethods$_{opt}$ (§3.2.4.6)
MutuallyExclusiveMethodSet_List$_{opt}$ (§3.2.4.7)
MethodManager_List$_{opt}$ (§3.2.4.8)
}

The coordinator body defines the coordination strategy for the given classes, on a method basis. That is, the smallest units of synchronization are the methods. When a thread T is trying to execute a method M on an instance of a coordinated class, two things can happen:

1) if M is not mentioned in the coordinator, then T executes M immediately.
2) if M is mentioned in the coordinator, then T may be suspended. There are two circumstances under which a thread may be suspended:
   - exclusion constraints, as given by the self- and mutual- exclusion declarations (§3.2.4.6, §3.2.4.7), are not met.
   - the pre-condition defined in the method manager (§3.2.4.9) is false.

   T waits until all the exclusion constraints, if any, are met and the pre-condition, if any, becomes true. Only then T may execute M.

Related Design Decisions: DD§3.3.3.2

3.2.4.4 Condition Variables

Condition variables (conditions, for short) hold the part of the coordinator's state that is used for purposes of guarded suspension and notification of threads on the execution of the methods. This state is called the synchronization state. Condition variable declarations identify conditions to be used within the lexical scope of the coordinator:

CondVarDecl:
    `condition` *VariableDeclarator_CommaList* `;`

VariableDeclarator:
    Identifier `=` *CondVarInitializer* |
    Identifier `[]` = *CondArrayInitializer*

CondVarInitializer:
    `true` | **`false`**

CondArrayInitializer:
    `{` *CondVarInitializer_CommaList* `}`

Condition variables can only hold the values `true` or `false` (i.e. they are booleans). The use of conditions is explained in §3.2.4.8.

3.2.4.5 Ordinary Variables

Ordinary variables hold the part of the coordinator's state that doesn't directly lead to suspension and notification of threads, but that may affect the synchronization state. It is used as auxiliary state, typically for keeping track of method invocation histories. Ordinary variable declarations identify such variables, to be used within the lexical scope of the coordinator:

VarDeclaration:
    PrimitiveType VariableDeclarator_CommaList ;

VariableDeclarator:
    Identifier = *VarInitializer*
    Identifier [ ] = *ArrayInitializer*

VarInitializer:
    Expression

ArrayInitializer:
    { *VarInitializer_CommaList* }

Auxiliary variables are only of primitive types. They serve as counters and other basic data. A broader notion of type (say, class types) would make a less crisp boundary between The component language and COOL. At this point it is not clear if such a generalization will be useful enough to justify the intrusion.

3.2.4.6 Self-Exclusive Methods

The self-exclusion declaration (selfex set, for short) identifies the methods that can be executed by at most one thread at a time:

SelfExclusiveMethods:
    `selfex` *QualifiedName_CommaList* ; (§3.2.4.1)

In each *QualifiedName*, the *VisibleElementt* must a visible method of the *ClassName*.

Self-exclusion is re-entrant. That is, if the methods are directly or indirectly recursive, self-exclusion does not deadlock the thread. This is coherent with the concept of thread as a sequential virtual processor: a thread does not block on its recursive calls, unless something else prevents it from proceeding. For example, consider the following implementation of class Fact:

```
public class Fact {
  int counter = 1; // counter… to inspect where the computation is
  public int f(int n) { // factorial
    counter = n;
```

```
      if (n <= 1) return 1;
      else return n*f(n-1);
    }
    public int inspect() { return counter; }
  }
```

In a concurrent environment, we can inspect where the factorial computation is by concurrently invoking the method `inspect` on the Fact object. However, we shouldn't allow two threads to run the method `f` on the same Fact object, because the counter is being destructively assigned. So, a possible coordination for these objects can be:

```
  coordinator Fact {
    selfex f;
  }
```

The recursive call in method in `f` does not block the thread that starts executing `f`. Self-exclusion simply prevents other threads from executing `f` on the same object at the same time.

There isn't any mutual exclusion relation between the referred methods. Consider, for example, the following coordinator:

```
 per_class coordinator A, B {
   selfex A.f, A.g, B.f;
 }
```

If two different threads try to execute `A.f` at the same time, one of them waits until the other finishes executing that method (the same for `A.g` and `B.f`). But different threads may execute concurrently these three methods. The only guarantee made by this coordination program is that, at any time, there will be at most one thread executing `A.f`, at most one thread executing `A.g` and at most one thread executing `B.f`.

Related Design Decisions: DD§3.3.3.4

3.2.4.7 Mutual Exclusion Declarations

A mutual exclusion declaration (mutex set, for short) identifies a set of methods that cannot be executed concurrently by different threads; that is, the execution by thread T of one of the methods in the set prevents the execution by other threads T' ≠ T of all other methods in the same set. There may be several mutex sets in a coordinator.

MutuallyExclusiveMethodSet:
  `mutex` `{` *QualifiedName_CommaList* `};` (§3.2.4.1)

In each *QualifiedName*, the *VisibleElement* must a visible method of the *ClassName*.

Each mutual exclusion set must have at least two different elements – that is, it must name at least two methods that are mutually exclusive. The following does not establish self-exclusion, and results in a weave-time warning:

```
coordinator Rectangle {
  mutex {adjustLocation}; // Warning: mutex must have at least two
}                         // different elements; mutex declaration is
                          // ignored.
```

Repeating method names in the same mutex set also does not establish self-exclusion of the method, and results in weave-time warnings:

```
coordinator Rectangle {
  mutex {adjustLocation, adjustLocation};
        // Warning: mutex must have at least two different elements;
        // mutex declaration is ignored.
}

coordinator Rectangle {
  mutex {adjustLocation, adjustLocation, set_x};
                    // Warning: repeated names in mutex; one occurrence
}                   // is ignored, but the mutex declaration is valid.
```

Mutual exclusion does not establish self-exclusion of each of the methods in the mutex set. That is, a method M that belongs to some mutex is not selfex, unless it is explicitly declared as such in the selfex declaration. Consider, for example, the book locator class defined in Chapter 2 (§2.1.1). This class has three methods: `register`, `unregister` and `locate`. The first two update internal variables of the object, while the third only accesses those variables without modifying them — a typical readers/writers synchronization. We can define its coordinator simply as

```
coordinator BookLocator {
  selfex register, unregister;
  mutex {register, unregister, locate};
}
```

The `locate` method doesn't need to be self-exclusive: it only reads the state of the book locator, so it's safe to have several concurrent activations of it; however, neither `register` nor `unregister` can proceed when some thread is executing `locate`. And executions of `register` and `unregister` also exclude each other. Hence, the three methods are declared to be mutually exclusive.

In a coordinator, there may be more than one mutex set. Consider, for example, the following class:

```
public class Rectangle {
  int width, height;
  Bitmap pixels;
  public Rectangle(int w, int h) {
    width = w; height = h;
    pixels = new Bitmap(w, h);
  }
  public void set_width(int newvalue) {
    width = newvalue;
    pixels.set_width(newvalue);
  }
  public void set_height(int newvalue) {
    height = newvalue;
    pixels.set_height(newvalue);
  }
  public int area() {
    return width*height;
  }
  public void fill(Color c) {
    pixels.fill( c );
  }
}
```

The coordination scheme for instances of this class can be as follows:

```
coordinator Rectangle {
  selfex set_width, set_height, fill; // area is not selfex
  mutex {set_width, area};
  mutex {set_height, area};
  mutex {set_width, fill};
  mutex {set_height, fill};
}
```

We want to ensure that the dimensions of rectangles remain consistent while some thread is executing a method involving those values. Therefore, the "set" methods are declared mutually exclusive with methods `area` and `fill`. But the two "set" methods themselves don't need to exclude each other, since they set different variables. Also the methods `area` and `fill` don't conflict with each other, and don't need to be mutually exclusive. Hence, the four mutex sets.

Note that the four separate mutex sets have a completely different semantics than the set:

```
mutex {set_width, set_height, area, fill};
```

In this case, the four methods are declared to be mutually exclusive. For example, while some thread is executing method `fill`, no other thread can execute `set_width`, `set_height` or `area`. This is a more constrained synchronization scheme than the previous one.

Related Design Decisions: DD§3.3.3.4

3.2.4.8 Method Managers

Method managers handle guarded suspension and notification of threads, using a style of pre-conditions, on entry statements and on exit statements for methods:

```
MethodManager:
        QualifiedName_CommaList :                    (§3.2.4.1)
        Requires_opt                                 (§3.2.4.9)
        OnEntry_opt                                  (§3.2.4.10)
        OnExit_opt                                   (§3.2.4.10)
```

In each *QualifiedName*, the *VisibleElement* must a visible method of the *ClassName*.

A particular method manager may manage more than one method at once, as implied by the list of method names; this is convenient for those methods that share the same constraints for suspension and have the same effects on the coordination state, since it avoids the repetition of code.

The same method name may appear in more than one method manager. This is another notation convenience, which is typically used when a method shares different pieces of on entry and on exit statements with other methods. In that case, those statements are cumulative for each method, and they take the order in which they appear in the coordinator.

However, it is a weave-time error for a method name to appear in two or more method managers which define a *Requires* clause (§3.2.4.9). Since pre-conditions are given in terms of a boolean expression, the logical function to apply among the different *Requires* clauses would be ambiguous. Therefore, the pre-conditions associated with a method name should appear in the *Requires* clause of at most one method manager.

3.2.4.9 Guarded Suspension

Guarded suspension of threads is expressed in terms of a boolean expression of conditions:

```
Requires:
        requires CondVarExpression ;

CondVarExpression:
        VarRef                  |
        Not CondVarExpression   |
        ( CondVarExpression )   |
        CondVarExpression ConditionalOp CondVarExpression

VarRef:
        Identifier              |
        ArrayRef                |
```

ArrayRef:
  Identifier[ArrayIndex]

ArrayIndex:
  Identifier |
  IntegerLiteral |

Not:
  `!`

ConditionalOp:
  `&&` | `||`

The *Identifier* in the *VarRef* and *ArrayRef* rules must be a condition variable declared in this coordinator. The following results in weave-time errors:

```
coordinator Rectangle {
  boolean after_decreaseSize = false;
  guard increaseSize:
    // maxSizeFlag is an instance variable declared in the Rectangle class
    // (see §3.2.4.1)
    requires !maxSizeFlag ; // Error: cannot reference external variables
    on_exit {
      after_decreaseSize = false;
    }
  guard decreaseSize:
    requires !after_decreaseSize; //Error: cannot reference ordinary variables
    on_exit {
      after_decreaseSize = true;
    }
}
```

Guarded suspension must be done over conditions, and those conditions should be clearly traced in the coordination program. In the example above, the second error can be avoided by declaring `after_decreaseSize` as a condition variable.

The semantics of guarded suspension is as follows. In addition to the exclusion constraints (§3.2.4.6, §3.2.4.7), if any, on the methods, the *Requires* clause defines the pre-conditions over the state of the coordinator that must be met in order for threads to proceed executing the methods managed by this method manager. When a thread T wants to execute a method M that has a pre-condition, as defined in the *Requires* clause of the method manager, and the exclusion constraints are met, one of the following occurs:

1) If the *CondVarExpression* evaluates to true, then the thread has the right to execute M.

2) Otherwise the thread does not have the right to execute M, and it is suspended. The thread stays suspended until the pre-condition becomes true; when that happens, T is notified, and then one of the following occurs:
    - if the exclusion constraints still hold, T has the right to execute M.
    - if the exclusion constraints no longer hold, T is suspended due to constraints not met (see §3.2.4.3).

3.2.4.10 On Entry and On Exit Statements

As soon as a thread has the right to execute a method, but just before it does so, the coordinator may update its internal state. This is done using the "on entry" statements of the method managers:

> *OnEntry:*
> **`on_entry`** { *Statement_List* }

Similarly, as soon as a thread finishes executing a method, the coordinator may also update its internal state. This is done using the "on exit" statements of the method managers:

> *OnExit:*
> **`on_exit`** { *Statement_List* }

The statements consist of a sequence of conditionals and assignments:

> *Statement:*
> *IfStatement* |
> *AssignStatement*
>
> *IfStatement:*
> **`if`** *Expression* {*Statement_List* } |
> **`if`** *Expression* {*Statement_List* } **`else`** {*Statement_List* }
>
> *AssignStatement:*
> *Identifier* = *Expression* `;` (See Appendix A for definition of *Expression*)

There are a number of validity rules applying to these grammar productions. First, expressions are typed, and relations between expressions must conform to the typing rules. Typing rules are the same as the component language typing rules. Secondly, there are two rules related to assignments and expressions:

- With respect to the *AssignStatement* production, the *Identifier* must be a variable (condition or ordinary) previously declared in the coordinator, and the *Expression* must be of boolean type.
- With respect to the *Expression* production, the *Identifier* must be one of the following:
  1) a variable previously declared in the coordinator; or
  2) a visible variable of the class(es) to which the method(s) being managed belong.

In other words, method managers may inspect the state of the coordinated objects (given by the variables declared in the classes), and use the result of that inspection to decide on their own state changes. However, method managers cannot modify the state of the coordinated objects, since the only valid assignment statements are those involving the coordinator's own variables.

An assignment to a condition variable may result in the notification of suspended threads. An assignment to an ordinary variable doesn't have any side-effects other than the assignment itself.

Related Design Decisions: DD§3.3.3.5

3.2.4.11 Some Examples of Coordinators

The following three examples illustrate the use of coordinators. In order to concentrate on COOL, the classes and clients of the classes are not shown (see Appendix B for the complete versions of these examples). The thread synchronization strategies shown here depend on the implementation of the classes. However, independently of the classes, coordinators disclose all the necessary information for understanding those strategies.

Coordinator for the classical bounded buffer:

```
coordinator BoundedBuffer {
  selfex put, take;
  mutex {put, take};
  condition empty = true, full = false;

  put: requires !full;
       on_exit {
         if (empty) empty = false;
         if (usedSlots == capacity) full = true;
       }
  take: requires !empty;
       on_exit {
         if (full) full = false;
         if (usedSlots == 0) empty = true;
       }
}
```

Coordinator for the dinning philosophers (monitor solution):

```
per_class coordinator Philosopher {
  condition OKToEat[] = {true, true, true, true, true};
  boolean eating[] = {false, false, false, false, false};

  eat: requires OKToEat[mynumber];
       on_entry {
         OKToEat[(mynumber+1) % max] = false;
         OKToEat[(mynumber-1) % max] = false;
         eating[mynumber] = true;
       }
       on_exit {
         if (eating[(mynumber+2) % max] == false)
           OKToEat[(mynumber+1) % max] = true;
         if (eating[(mynumber-2) % max] == false)
           OKToEat[(mynumber-1) % max] = true;
         eating[mynumber] = false;
       }
}
```

Coordinator for an assembly line:

This application consists of a number of concurrent agents, some of them operating in parallel and others in sequence. Candy Makers produce one candy at a time, which they feed, concurrently, to a Packer; the Packer fills a packet with a maximum number of candy and passes the packet to a Finalizer agent; the Finalizer takes one packet from the Packer and one label from a Label Maker, glues the latter in the former, and produces the final candy packet. (see Appendix B for an illustration of the agents)

```
coordinator Packer, Finalizer {
  selfex Packer.newCandy;
  condition packFull = false, gotPack = false, gotLabel = false;

  Packer.newCandy: requires !packFull;
      on_exit { if (nCandy == nCandyPerPack) packFull = true; }

  Packer.processPack: requires packFull;

  Finalizer.newPack: requires !gotPack;
      on_entry { gotPack = true; }
      on_exit { packFull = false; }

  Finalizer.newLabel: requires !gotLabel;
      on_entry { gotLabel = true; }

  Finalizer.glueLabelToPack: requires (gotPack && gotLabel);

  Finalizer.newDJCandyPack:
      on_exit {gotPack = false; gotLabel = false;}
}
```

3.2.5. The Remote Interface Aspect Language

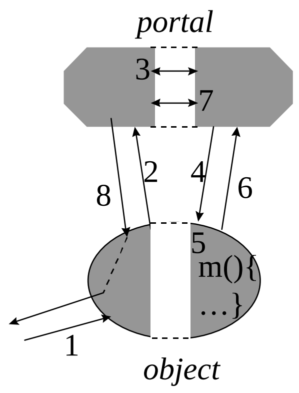
RIDL provides means for dealing with data transfers between different execution spaces in relative separation from the classes. A RIDL program consists of a set of portal modules:

RIDLProgram:
    PortalDeclaration_List (§3.2.5.2)

Portal modules (portals, for short) are associated with the classes on a name basis. There is at most one portal associated with each class. Portals are helpers with respect to the implementation of the classes: they take care of data transfers across space boundaries.

Portal declarations (§3.2.5.2) identify classes whose instances may be referenced from remote spaces. Such instances are called remote objects. The portal declaration identifies which methods of the class are exported over the network. In the portal, such methods are called remote operations. For each of those operations, the portal declaration describes what data the remote objects are expecting and what data they send back to the callers.

Portals are not classes: they use a different language, they cannot be instantiated, and they serve a very specific purpose. Nor are they types in the strict sense of the word. A portal is automatically associated with an instance of the class to which it applies as soon as a reference to that instance is exported outside the space where the instance was created. Throughout the life of the instance this relation has a well-defined protocol, depicted in Figure 31.



1: From some remote space, there is an invocation to method m of the object.
2: The request is first presented to the object's portal.
3: The method is processed according to its declaration as remote operation in the portal: parameters are extracted from the remote space into the local space according to the passing modes and copying directives declared for the remote operation.
4: The request proceeds to the object.
5: Method m is executed. (This execution may be ruled by the object's coordinator, if it exists.)
6: As the method invocation returns, the return is presented to the object's portal.
7: The return value is processed according to its declaration in the remote operation.
8: The method invocation finally returns, and the return value is passed back to the remote space.

Figure 31. Protocol between an object and its portal. Both the object and the portal are shown as two pieces to denote their remote and local handles.

Classes are unaware of portals, i.e. it is not possible for a class to name a portal. The association between a class and a portal is driven by the portal, not by the class. Portals are fully aware of the classes to which they apply, and the implementation of a portal can and should be aware of the implementation of the classes, so that the best transfer strategies can be defined.

At run-time, the association between objects and portals is one-to-one. That is, if a class is associated with a portal, then each of its instances will have a portal of its own.

The body of a portal (§3.2.5.3) defines the remote operations for the corresponding class (§3.2.5.4) and the transfers modes of the arguments and return value for each of those operations (§3.2.5.5). The objects may be sent by global reference (§3.2.5.7) or by copy (§3.2.5.8). In case of pass-by-copy, an optional copying directive may be included to define which parts of the object graph actually get copied (§3.2.5.9).

3.2.5.1 Visible Elements of Classes

The complete set of visible elements for RIDL is: (1) the visible elements described in §3.2.3, except the static methods; and (2) all variables, private, protected and public, of all the classes of a D application.

Note that 2 establishes an explicit dependency between the portal modules and the overall structural relationships of the classes. This dependency exposes the need for controlling the data transfers across execution spaces.

Related Design Decisions: DD§3.3.1.1.

3.2.5.2 Portal Declaration

A portal declaration establishes an association between the portal being declared and a class:

PortalDeclaration:
    `portal` *ClassName PortalBody*

Portals don't have proper names. They simply refer to the *ClassName* class. It is an error to declare two portals for the same class. Not all classes need to have portals (see §3.2.5.10). When a portal is defined for a given class, remote interactions with instances of that class are ruled by the portal declaration. That is, the rules for remote interactions are specified only by the receiver objects, not by the senders.

Related Design Decisions: DD§3.3.1.1, DD§3.3.1.2, DD§3.3.1.3, DD§3.3.4.1, DD§3.3.4.2.

3.2.5.3 Portal Body

The portal body declares which methods of the class can be invoked remotely. Optionally, it may declare default parameter passing modes for the arguments and return values of the operations declared:

```
PortalBody:
        {
         RemoteOperation_List                                  (§3.2.5.4)
         DefaultTransfers_opt
        }

DefaultTransfers:
        default: TransferableType_List                         (§3.2.5.6)
```

The set of remote operations must be a subset of the visible methods of the class whose portal is being declared.

When an invocation occurs to an instance of this class, the following happens:

1) if the instance is local to the space where the call occurs, then a local invocation happens, and the portal is ignored.
2) if the instance is remote, then a remote method invocation *may* occur:
   - ⇒ if the method being invoked is a valid method of the class and a valid remote operation of the class's portal, then a remote method invocation occurs.
   - ⇒ if the method being invoked is a valid method of the class but not a valid remote operation in the class's portal, then an error occurs.

If the remote method invocation does occur, then the arguments to the method and the return value are passed according to the remote operation declaration (§3.2.5.4).

Related Design Decisions: DD§3.3.4.2.

3.2.5.4 Remote Operations

The remote operations define which methods can be invoked on remote objects which are instances of the class for which the portal is being declared:

```
RemoteOperation:
        ReturnType MethodName ( Parameter_CommaList_opt )
        RemoteOperationBody_opt ;

ReturnType:
        Type |
        void
```

Parameter:
  Type ParamName

Type: (same as Java’s *Type* production)

ParamName:
  Identifier |
  ParamName [ ]

RemoteOperationBody:
  { *ObjectTransferSpec_List* } (§3.2.5.5)

The *MethodName* must be a visible method of the class for which the portal is being declared (§3.2.5.1).

The *Type* both of the return value and the parameters of a remote operation declaration must be exactly the same as the one declared in the corresponding method of the class. For example:

```
public class subA extends A { /* some fields */ } // subA is subclass of A

public class B {
  public void f(subA a) { /* implementation of f */ }
  public A g(subA a)    { /* implementation of g */ }
  public int h()        { /* implementation of h */ }
}

portal B {
  void f(A a);   // Weave-time error: parameter must be of type subA
  A g(subA a);   // OK: types are exactly the same
  // h not declared; only f and g are made available to remote spaces
}
```

When a remote method invocation occurs, the arguments and the return value are transferred from one space to the other, according to the following rule:

1) An argument of a primitive type is always copied.
2) An argument *o* of a reference type (i.e. an object) may be copied or not:
   - if there is an object transfer specification (§3.2.5.5) for *o* then *o* is transferred according to that specification, else
   - if there is a type transfer specification (§3.2.5.6) for the type of *o* then *o* is transferred according to that specification, else
   - the default data transfer is a complete, recursive copy of the object graph that has *o* as root.

Not all visible methods of the class must be declared as remote operations. That is, the availability of a method over the network is orthogonal to the protection qualifiers of JCore, which rule the accessibility of methods between classes. Consider the class Rectangle and its portal:

```
portal Rectangle {
  void set_width(int newvalue);
  int area();
}

public class Rectangle {
  int width, height;
  Bitmap pixels;
  public Rectangle(int w, int h) {
    width = w; height = h;
    pixels = new Bitmap(w, h);
  }
  public void set_width(int newvalue) {
    width = newvalue;
    pixels.set_width(newvalue);
  }
  public void set_height(int newvalue) {
    height = newvalue;
    pixels.set_height(newvalue);
  }
  public int area() {
    return width*height;
  }
  public void fill(Color c) {
    pixels.fill( c );
  }
}
```

In the example above, although all methods of the class Rectangle are public, only two of them, `set_width` and `area`, are declared as remote operations. This means that clients of remote rectangle objects can only invoke these two methods.

Since the remoteness of objects is known only at run-time, run-time errors – DInvalidRemoteOperation error– may occur. Consider, for example, a client of class Rectangle:

```
class SomeClass {
  void doSomething(Rectangle r) {
    int a = r.area(); // OK for any Rectangle r, local or remote
    r.fill(); // OK if Rectangle r is local;
              // Run-time error in the client if Rectangle r is remote
  }
}
```

These run-time errors can be avoided by declaring all the methods of the class as remote operations. However, RIDL does not impose such alignment, making it possible to define protections that are different for local and remote clients.

Visible methods include private and protected methods; that is, it is possible for a class to export private and protected operations over the network. This feature simply extends the semantics of the protection qualifiers to remote invocations: instances of a class can invoke private and protected methods on other instances of the same class remotely. However, this feature maintains the semantics of the protection qualifiers with respect to other classes. For example,

```
portal Rectangle {
  void print1();
  void print2();
}

class Rectangle {
  //… variables …
  private void print1() {
    System.out.println(“Print private:” +this.toString());
  }
  protected void print2() {
    System.out.println(“Print protected:” +this.toString());
  }
  public void print3() {
    System.out.println(“Print public:” +this.toString());
  }

  public void print(Rectangle r) {
    r.print1(); // OK for any Rectangle r, this or other, local or remote
    r.print2(); // OK for any Rectangle r, this or other, local or remote
    r.print3(); // OK for local Rectangle r;
                // run-time error in the client for remote Rectangle r
  }
}

public class Triangle {
  public void print(Rectangle r) {
    r.print1(); // Compile-time error; cannot access private method
    r.print2(); // OK for any Rectangle r, local or remote, if Triangle is
                // in the same package as Rectangle;
                // Compile-time error otherwise
    r.print3(); // OK for local Rectangle r;
                // run-time error in the client for remote Rectangle r
  }
}
```

3.2.5.5 *Object Transfer Specifications*

It is possible to define particular transfer modes for arguments and return values which are objects (i.e., of reference types):

ObjectTransferSpec:
    ObjectName `:` *Mode* `;`

ObjectName:
    Identifier |
    `return`

Mode:
  `gref` |
  `copy` *CopyDirective*$_{opt}$ (§3.2.5.9)

If the *ObjectName* is an *Identifier*, the identifier must be a parameter name of the remote operation. The return value is denoted by the keyword `return`.

Parameters of primitive types are always passed by copy. Therefore, in the body of a remote operation, the only valid object names are the ones whose types are reference types.

Related Design Decisions: DD§3.3.4.1.

3.2.5.6 Type Transfer Specifications

The portal body (§3.2.5.3) may include default modes for transferring reference types:

TypeTransferSpec:
  ReferenceType `:` *Mode* `;`

The scope of the default type transfer specifications is the lexical scope of the portal.

Related Design Decisions: DD§3.3.4.1.

3.2.5.7 Passing Global References

If the *Mode* in the transfer specification is **`gref`**, then the corresponding argument or return value in the remote operation is passed by global reference. That is, the object is not copied, and only a unique, global reference to it is passed. If the gref argument or return object is invoked in the remote space, then a rebounding method invocation occurs to the space where the object exists. The remote interaction with a gref object is determined by the portal for the class of the object. Hence, remote invocations to an object denoted by a global reference are:

- protected on space boundaries by the object's portal. Remote calls to the object are guaranteed to be confined to the operations declared in the portal, which are a subset of the operations declared in the class.
- guaranteed to be performed within the object itself.

Related Design Decisions: DD§3.3.4.1, §3.3.4.3.

3.2.5.8 Passing Copies

If the *Mode* in the transfer specification is **`copy`**, then the corresponding argument or return value is cloned during the remote call. That is, a (possibly incomplete) replica of the object is passed from the sender to the receiver spaces. Replicas contain a snapshot of:

1) the object's primitive values; and
2) recursive replicas of the object's reference values. (The implementation of the replication mechanism should detect and resolve cycles in the object graph.)

Replicas of objects passed in remote calls are ordinary objects of the same classes as their originals. If, in the receiver space, there is an invocation to an object which has been transferred by copy, the invocation is a local invocation to the replica. There is no notion of group; replicas and original don't have any relationship. Hence, objects passed in the copy mode are affected by the following consequences:

- No guarantees are made with respect to the coherence of the replicas. Once they are passed to remote spaces:
  - ⇒ if the local space modifies the state of the original object, the modification is not propagated to the replicas.
  - ⇒ if a remote space modifies the state of a replica, the modification is not propagated to the original object or to other replicas of the object.
- Spaces containing replicas have all the rights over them. The interface to replicas is given by their classes, not by their class's portal.

This guarantees that the original objects are fully protected from wrongful modifications, while allowing the remote spaces to perform all the necessary operations on the replicas.

The copy mode may be used to improve the performance and/or to ensure complete separation for purposes of protection of the objects.

3.2.5.9 Copying Directives

When objects are to be passed by copy, an optional copying directive may be given in their transfer specifications:

```
CopyDirective:
        {
         SelectionDirective_List_opt
        }
```

SelectionDirective:
  ClassSelector SelectionPrimitive VariableSelector_CommaList;

SelectionPrimitive:
  `only` |
  `bypass`

ClassSelector:
  ClassName

VariableSelector:
  VariableName |
  `all.`*TypeName*

ClassName must be a valid class name of the application; *VariableName* must be a valid variable name in the class referred to in the left part of the selection directive. **`all`**.*TypeName* is a special field selector that is used to select fields according to their type (**`all`**`.int` means all fields of type int).

Transferring an object *o* to a remote space results in recursively traversing the object graph that has *o* as root, and packing all the primitive data that is found along the traversal. This is what RPC systems do. The generic object transfer facility for an object-oriented language can be given as:

```
function transfer(object)
  foreach class in (Class(object) and superclasses of Class(object))
    foreach field in class
      if field is of primitive type
        send the value of the field
      else
        transfer(field)  // recursive call
```

For the existing RMI systems, this facility is generic, in that all objects are always recursively traversed, independently of their classes and of which operation is being invoked. All object along the traversal are marshaled. With respect to the generic object transfer facility shown above, RIDL allows programmers to associate different transfers with different predicates on the fields of the classes, so that a decision can be made on whether to send/traverse them or not, if the traversal is about to reach them. Therefore it becomes possible to specify cuts of entire subgraphs of objects. The classes involved in a copying directive are not confined to the class of the argument/return value for which the directive is being defined, but can be any class of the application. The named fields must be fields of those classes or of their superclasses.

Copying directives are a subset of Demeter's graph traversal language [42, 44]. In RIDL there are only two constructs:

- `bypass` identifies the fields of a given class that are to be excluded, whenever the traversal reaches instances of that class.
- `only` identifies the fields of a given class that are to be copied, whenever the traversal reaches instances of that class, and excludes all the fields of that class that are not mentioned.

One is the complement of the other. For example, if class `A` has three fields, f_1, f_2 and f_3, the specification “`A` **`bypass`** `f1`” is equivalent to the specification “`A` **`only`** f_2, f_3”. Both forms are made available, because sometimes positive constraints are more expressive than negative constraints, and vice-versa.

For an example, consider the application depicted in Figure 32, where the boxes represent class types, the ovals represent primitive types, and the edges represent variables defined in the class (possibly in a one-to-many relationship, denoted here by ‘*’ for short):

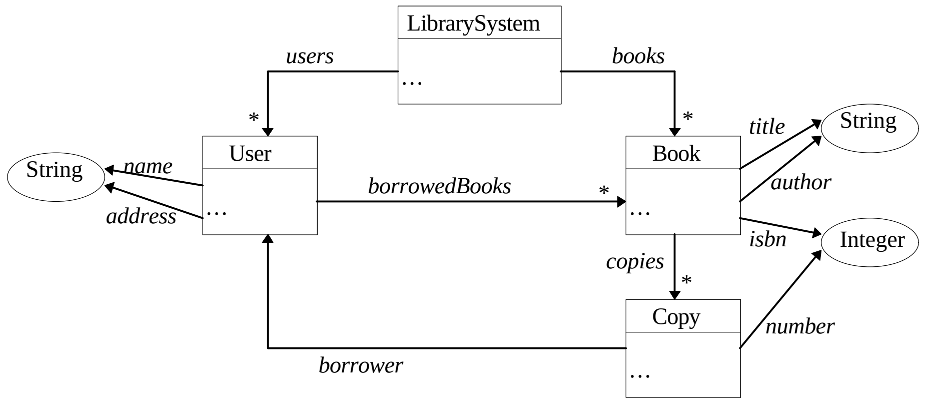


Figure 32. Class graph of a library application.

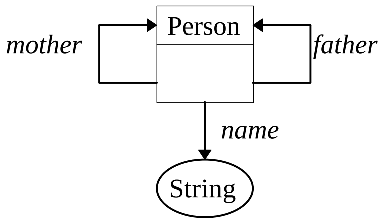
A possible portal to the class `LibrarySystem` can be:

```
portal LibrarySystem {
  boolean registerUser(User user) {
    //Only strings. Everything else of User is excluded.
    user: copy {User only all.String;}
  };

  Book getBook(int isbn){
    //for return object, exclude this edge; this excludes the copies
    // and breaks nasty cycle.
    return: copy {Book bypass copies;}
  };

  BookList borrowedBooks(User user) {
    //for return object, exclude this edge; this breaks nasty cycle
    return: copy {Book bypass copies;}
    // for User, bring only the name
    user: copy {User only name;}
  };
}
```

In this example, instances of `User` are transferred differently for the remote operations `registerUser` and `borrowedBooks`. For the Book objects returning both from `getBook` and `borrowedBooks`, the `copies` are not passed. If we wanted to pass the copies but still break the cycle back to `User`, we should specify `{Copy` **`bypass`** `borrower}` instead.

The language is not powerful enough to specify copying directives for arbitrary objects. For example, when transferring the list of books that `borrowedBooks` returns, it is not possible to transfer only some of the books.

For each transfer specification, the specification holds for any instance of the involved classes. Consider, for example, the following class structure, where *mother* and *father* can be null:



Passing a Person object with the directive `{Person` **`only`** `mother, name;}` is equivalent to passing it with the directive `{Person` **`bypass`** `father;}`. All instances of Person are transferred in the same way. In this case, this results in transferring only the names of the female subset of ancestors.

A class may be affected by several specifications. The result of having different specifications *for the same class* is given by the following rules:

- for `bypass` constraints, the resulting set of fields is the union of the sets of fields of each constraint.

  Example 1: `{A` **`bypass`** `f`$_1$`; A` **`bypass`** `f`$_2$`;}` $\Leftrightarrow$ `{A` **`bypass`** `f`$_1$`,f`$_2$`;}`
  Example 2: `{A` **`bypass`** `f`$_1$`; A` **`bypass all`**`.B;}` $\Leftrightarrow$ `{A` **`bypass`** `f`$_1$`,`**`all`**`.B;}`

- for `only` constraints, the resulting set of fields is the intersection of the sets of fields of each constraint.

  Example 3: `{A` **`only`** `f`$_1$`,f`$_3$`; A` **`only`** `f`$_2$`,f`$_3$`;}` $\Leftrightarrow$ `{A` **`only`** `f`$_3$`;}`
  Example 4: `{A` **`only`** `f`$_1$`; A` **`only all`**`.B;}` $\Leftrightarrow$ `{A` **`only`** `f`$_1$`;}` if `f`$_1$ is of type `B`
  `{A` **`only`** $\varnothing$`;}` otherwise

- when `bypass` and `only` constraints are given simultaneously, the resulting set of fields F is the intersection given by F = $\bigcap$ *sets of fields of* only *constraints* , $\bigcup$*sets of fields of* bypass *constraints*

  Example 5: `{A` **`only`** `f`$_1$`;A` **`bypass`** `f`$_1$`,f`$_2$`;A` **`bypass`** `f`$_3$`;}` $\Leftrightarrow$ `{A` **`only`** `f`$_1$`;}`

Figure 33 illustrates the meaning of these primitives for a generic class graph.

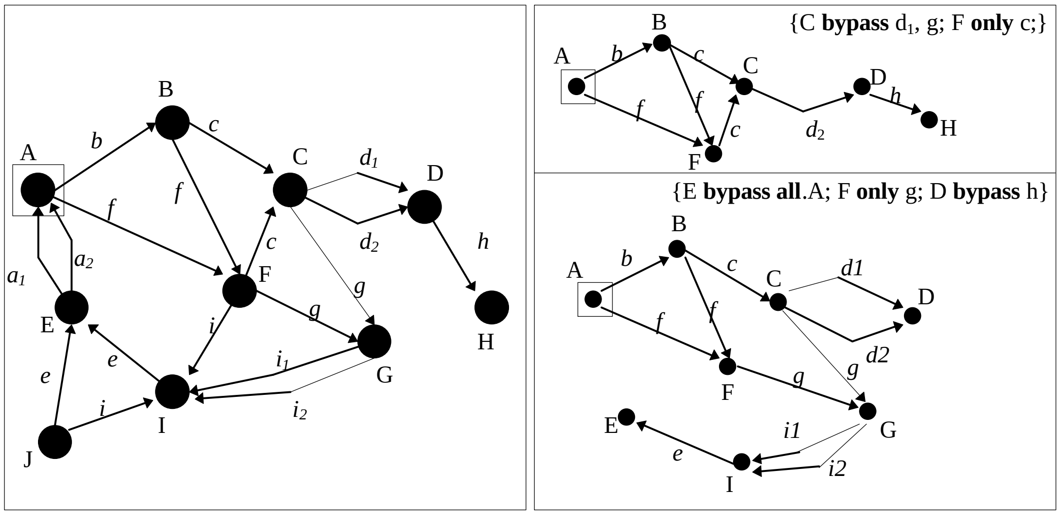

{C **bypass** d₁, g; F **only** c;}
{E **bypass all**.A; F **only** g; D **bypass** h}
A
B
C
D
E
F
G
H
I
J
b
c
f
d1
d2
h
g
i
i1
i2
e
a1
a2

Figure 33. Two examples of copying directives. The circles are classes, and the edges are variables declared in them (e.g. class A contains a variable named *b* of type B and a variable named *f* of type F). The two graphs on the right are the result of applying the given copying directive to the graph on the left, having class A as root.

Compatibility of copying directives:

The class graph of the application is given by the visible variables of the classes (§3.2.5.1). Copying directives assume that there are certain traversal paths in the class graph, namely that there is at least one path from the class of the argument/return value to each of the classes referred in the directive. A copying directive is compatible with the class graph when that assumption is true. For example, with respect to the graph shown in Figure 33, the following directive is not compatible: having an argument of type A, `{J only i}`; given the class graph on the left, from an instance of A, the traversal cannot reach instances of J. Compatibility of the copying directives must be checked every time the remote interaction interface of classes change (remember, the visible variables of a class are part of their remote interaction interface). This can be done automatically, using one of several existing algorithms for finding paths in graphs.

Copying directives introduce one problem that does not exist in existing RMI systems: what happens when a remote space tries to access a part that was not copied? The only guarantee is that there is neither automatic fetching of the missing parts nor the "promotion" of those parts to be remote objects.

Related Design Decisions: DD§3.3.4.4, §3.3.4.5.

3.2.5.10 Classes that Must Have a Portal

The following classes must have a portal, because some of their instances may be accessed remotely:

- Classes whose instances may be registered with the name server.
- Classes of arguments and return values of some remote operation of any portal, whose transfer specifications are `gref` .

All other classes don't need to have portal declarations, since instances of those classes will never be accessed remotely.

3.2.5.11 Some Examples of Portals

The following two examples illustrate the use of portals. In order to concentrate on RIDL, the classes and clients of the classes are not shown. The remote interaction strategies shown here depend on the implementation of the classes. However, independently of the classes, portals disclose all the necessary information for understanding those strategies.

A portal for the bounded buffer:

```
portal BoundedBuffer {
  void put(Book o);
  Book take();
 default:
    Book : copy { Book only all.String, all.int; };
}
```

BookLocator/ProjectManager:

The following portals solve the problem of the BookLocator/ProjectManager application described in Chapter 2. The problem, in short, was that books must be transferred differently for the BookLocator and ProjectManager services.

```
portal BookLocator {
  void register(Book b, Location l);
  void unregister(Book b);
  Location locate(Book b);

  default:
    Book : copy { Book only all.String, all.int; };
}

portal ProjectManager {
  boolean newBook(Book b, Price p) {
    Book : copy { Book only owner, all.String, all.int; };
  }
}
```

3.2.6. Interaction between Aspects and Class Inheritance

Aspect modules (i.e. coordinators and portals) relate to class inheritance in very much the same way. This sub-section explains that interaction. Let *C* be class and A_C an aspect module directly associated with *C*; the following rules apply:

- *Field visibility*. As explained in §3.2.4.1 and §3.2.5.1, elements inherited from superclasses of *C* are visible to A_C.
- *No upwards effect*. A_C does not affect any superclass of *C*.
- *Overriding semantics*. A_C completely overrides any aspect module of the same kind defined in the superclasses of *C*. There is no relation whatsoever between A_C and the aspect modules of the superclasses of *C*.
- *Inheritance of coordination*. A_C affects all subclasses of *C* that are not associated with any other aspect module of the same kind. The aspect program does not affect the new methods defined in subclasses of *C* (they cannot be referred to in A_C). If method *m* declared in *C* is overridden by a subclass of *C* that is not associated with any other aspect module of the same kind, then the aspect program defined in A_C for method *m* of that subclass is the same as for method *m* of *C* (see rule for field visibility).

In the current version of the language, it is not possible for an aspect module to refer to another aspect module. As a consequence, it is not possible to establish any relation (e.g., inheritance) between the aspect modules themselves.

Consider, for example, a bounded buffer class and its coordinator implemented as follows:

```
public class BoundedBuffer {
  protected Object[] array;               // the elements
  protected int putPtr = 0, takePtr = 0;  // circular indices
  protected int capacity;
  protected int usedSlots = 0; // counter

  public BoundedBuffer (int size) {
    array = new Object[size]; capacity = size;
  }
  public void put(Object o) {
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    ++usedSlots;
  }
  public Object take() {
    Object old;
    old = array[takePtr];
    takePtr = (takePtr + 1) % array.length;
    --usedSlots;
    return old;
  }
}
```

```
coordinator BoundedBuffer {
  selfex put, take;
  mutex {put, take};
  condition empty = true, full = false;

  put: requires !full;
       on_exit {
         if (empty) empty = false;
         if (usedSlots == capacity) full = true;
       }
  take: requires !empty;
       on_exit {
         if (full) full = false;
         if (usedSlots == 0) empty = true;
       }
}
```

The following subclass of `BoundedBuffer` transforms the buffer from FIFO into LIFO, while inheriting the coordination behavior of its superclass:

```
public class BoundedBufferLIFO extends BoundedBuffer {
  public BoundedBufferLIFO (int size) {
    super(size);
  }
  public Object take() {
    Object old;
    putPtr = putPtr-1;
    old = array[putPtr];
    --usedSlots;
    return old;
  }
}
```

In this case, the synchronization strategy for the subclass is the same as for the superclass, and the coordinator can be reused. However, that may not always be the case, since different implementation of the methods may imply different synchronization schemes.

Inheritance of aspect modules should be seen as the regular inheritance of implementation: some times, no redefinitions are necessary, but sometimes overriding (of methods and aspects) may be necessary.

Related Design Decisions: DD§3.3.1.1, DD§3.3.3.2, DD§3.3.4.2.

3.3. Design Decisions and Alternatives

Earlier versions of COOL and RIDL were described in [49] and [47], respectively. Since then, the languages evolved and changed their names. However, the design principles have been preserved. This section contains a discussion of the design decisions, the features that were removed from earlier versions and the design alternatives that have been considered.

3.3.1. General

3.3.1.1 On Modules, Components, Aspects and Interfaces

[From §3.2.4.1, §3.2.4.2, §3.2.5.1, §3.2.5.2, §3.2.6]

As described in the previous section, the relation aspect/component is different from the relation component/component. This shows up in the visible elements of components and in the semantics of the aspect languages. It may be argued that the concept of "aspect" breaks the notion of modularization that has been proven of uttermost importance in software engineering. Not so. The discussion that follows retrieves the original proposal for "modular programming," and explains (1) why the idea of "aspect" is proposed, (2) the connection between aspects and components, and (3) how the original notion of modularity is preserved.

The idea of modular programming is usually credited to Parnas [61-63]. Modular programming was introduced as a better alternative to the software engineering practices of the time, which made the designs disclose almost everything of how the systems were implemented. Parnas defines modularization as follows [63]: "The system is divided into a number of modules with well-defined interfaces: each one is small enough and simple enough to be thoroughly understood and well programmed." The interface is, then, the key for making modularization work. Although Parnas doesn't define the word "interface" he uses it interchangeably to mean "connection [between modules]" [63] and "the information disclosed in the module description" [62]. In any case, the interface of a module should disclose all the necessary information for using/implementing the module, and no more than that information. In [61], Parnas studies the meaning of the phrase "connection between modules" in the following way: "Many assume that the "connections" are control transfer points, passed parameters, and shared data for software […]. Such a definition of "connection" is a highly dangerous oversimplification which results in misleading structures descriptions. *The con-*

nections between modules are the assumptions which the modules make about each other." (The emphasized is his)

The reason why the concept of module is so good is that (a) it isolates the dependencies between modules in explicit descriptions (interfaces), and (b) it allows the development of implementation techniques (i.e. languages and compilers) that support selective replacement and reassembly of parts without having to reassemble the whole system.

Note, then, that there are two distinct roles for modularization. First, modularization is the design process of breaking the system into modules with well-defined interfaces. Secondly, modularization is the ability to selectively reassemble parts of the system when certain module implementations change.

Virtually every modern programming language includes a module system. In object-oriented languages, for example, modules are usually classes or sets of classes, and a class has two kinds of interfaces: (1) the client interface, for the users of the class, and (2) the specialization interface, for its subclasses. In languages that attach qualifiers to fields, the former is usually given by the public fields of the class, and the latter includes also the protected fields.

However, there seems to have been a shift between Parnas's original notion of "interface of a module" and what today is perceived as "the interface of a class." The latter is close the concept of "type" — Java's "interface" construct is an good example of this — whereas the former included the type as well as a description of the module. In this thesis, the word "interface" is used to denote a type with a description. But independently of which of the two concepts should the word "interface" denote, it is clear that, as Parnas suggested 25 years ago, (a) the connections between modules must be clearly documented, (b) they must include more than the control transfer points, passed parameters and shared data, and (c) they should not disclose more information than necessary. Sometimes, the information flow from one module to another must include partial descriptions of how the module is implemented. That does not violate the concept of modularization, as long as the interface is clean. The work on open implementations [32, 33, 36] focused this point.

Following the previous discussion, it is now possible to understand aspects:

(1) Aspects capture issues of the *implementation* (of the components) that are naturally thought of in relative separation from what the components do. We would like to isolate the coding of these issues in modules, but with current language technology, they end up being tangled in the coding of the components.

(2) Aspect modules isolate the coding of those issues and free the components from code tangles. Besides the client and specialization interfaces, aspect modules introduce new kinds of interfaces for components, which focus *only* on particular subject matters. These interfaces are called "aspect interfaces."

(3) Aspect interfaces use particular sets of visible elements, which have been presented in §3.2.3, §3.2.4.1 and §3.2.5.1. Visible elements establish those dependencies between aspects and components that can be checked automatically. They have their counterparts in types, for example.

With respect to point (2), aspect interfaces are very different from the usual client/provider interface, but they have the same flavor as the specialization interface, in that they need to include information about the implementation of the component module. The following example illustrates the aspect interfaces. Consider the following class, described only by its fields:

```
class BoundedBuffer {
  public void put(DObject o);
  public DObject take();
  protected int size, capacity;
  protected int putPtr, takePtr, usedSlots, emptySlots;
  private DObject array[];
  private do_put(DObject o);
  private DObject do_take();
  private void increment_usedSlots();
  private void increment_emptySlots();
}
```

First, the two ordinary object-oriented interfaces are described. This description follows the general terms of what it is usually accepted as documentation of classes. (See, for example, the documentation for class Object in [23])

Documentation for clients:

In order to be able to use this class, we need to know what it does. A possible client interface can be documented as follows.

> "Each instance of class BoundedBuffer maintains a FIFO queue of objects. `put` inserts an object in the queue; when put returns, the object is guaranteed to be inserted. `take` removes an object from the queue; when `take` returns, the object is guaranteed to be removed and returned to the caller. Clients in remote execution spaces can only call `put`. The BoundedBuffer stores references to objects, local or remote, and distributes those references to local clients."

Documentation for specialization:

In order to be able to extend this class, we need to know some parts of how it is implemented and what we can use. A possible specialization interface can be documented as follows.

> "The general intent of this class is to collect objects from all over the network, through `put`, and distribute them to local clients, thorough `take`. The number of elements in the queue is given by the variable `size`, and the capacity is given by the variable `capacity`. The general intent of `put` is to insert the object in the queue; the method `put` for class BoundedBuffer first calls an internal method for inserting the element at the head of the queue, given by `putPtr`, and then `usedSlots` is incremented. The general intent of `take` is to remove an object from the queue; the method `take` for the class BoundedBuffer first calls an internal method for removing the tail of the queue, given by `takePtr`, and then `emptySlots` is incremented. "

Note that the specialization interface must disclose a lot about the implementation, because the subclasses may need to override the methods. Although the bounded buffer is not the best example for illustrating specialization interfaces, the argument holds (see example in §3.2.6).

Next, the two new interfaces are described. They serve as documentation for implementing the aspect modules. Note that this documentation should not be interpreted as a suggesting that components and aspects can be implemented in separate and by different groups of people, although that can eventually be done. It simply points out that aspects introduce the need for new kinds of component descriptions which disclose how aspect modules should be connected to component modules without disclosing all the details of the implementation of components or aspects. These descriptions may even involve groups of component modules.

Documentation for coordination:

In order to be able to coordinate this class, we need to know some parts of how it is implemented and what we can use. A possible coordination interface can be documented as follows.

> "Clients should be suspended whenever they call `put` and the queue is full and whenever they call `take` and the queue is empty. The capacity is given by `capacity`. The queue is empty when `emptySlots` reaches the capacity, and it's full when `usedSlots` reaches the capacity. `emptySlots` and `usedSlots` are incremented by the `increment_` methods. `do_put` guarantees that the queue is not empty. `do_take` guarantees that the queue is not full. `do_put` and `do_take` read and modify variables, but not the same variables. `do_put` and `increment_emptySlots` read and modify the same variables. `do_take` and `increment_emptySlots` read and modify the same variables."

Note that this description does not disclose whether the queue is implemented by an array or by a list, or what is it that the methods do. Nor does it say anything about the class's behavior in a distributed environment.

Documentation for remote interaction:

In order to be able to access instances of this class over the network, we need to know a little about how the class is used in a larger context. A possible remote interaction interface can be documented as follows:

> "For implementors of the class's portal: `put` is a remote operation; `take` is not. The BoundedBuffer class simply stores and distributes object references: no caching is necessary. For implementors of other portals: in case instances of the BoundedBuffer class are passed by copy in other services, the set of variables in the BoundedBuffer class is the one shown above."

Client interfaces are well-understood. Specialization introduced a new set of implementation-dependent relations between the modules; therefore, specialization interfaces have been much harder to understand. A lot of work has been done in this area — [35, 39]; [69] contains an extensive bibliography related to this issue. The specification of aspect interfaces will benefit from all the work that has been done for clarifying the specification of specialization interfaces.

As a final remark to this discussion, something must be said about how aspects relate to modularization as the ability to selectively reassemble parts of the system when certain module implementations, but not their interfaces, change. The first observation is that this is not a primary goal of Aspect Orientation [37]. The second observation is that the extent to which the system must be reassembled depends on the language implementation techniques. The last observation is that in the implementation of D described in Chapter 4, this property holds.

3.3.1.2 The Need for Special Abstractions and Composition Mechanisms

[From §3.2.4.2, §3.2.5.2]

As Figure 29 and Figure 31 suggest, coordinators and portals can be seen as metaobjects. The protocol between a coordinator or a portal and the objects they are associated with can be seen as a metaobject protocol [34]. This suggests that COOL's coordinators and RIDL's portals could be programmed using the component language itself, using a style of before and after methods. That has been the approach taken by some reflective languages [53, 55, 73].

There is one reason for defining new constructs for the aspects. Whatever these constructs will be (classes or special constructs), they compose with the components in special ways; this can, indeed, be done using reflection (meta-objects) or any other special objects (e.g., context objects [67]). It can even be done without any special composition mechanism, following only design guidelines (i.e. manual weaving). The need for new constructs comes not from *operational* deficiencies of the existing general purpose abstractions, but rather from (1) their lack of expressiveness for capturing the rules that are important for programming the aspects and (2) the appealing possibility of programming the aspects under aspect-specific rules.

If coordinators and portals were ordinary classes, we could not control their implementations. The encapsulation of responsibility would be all but clear. That is, in my opinion, one of the major drawbacks of using the existing general purpose composition mechanisms, including reflection, for programming aspects. So, following the second design principle (§3.1.2), D defines two aspect-specific languages, COOL and RIDL, both for facilitating and controlling aspect programming.

3.3.1.3 Who Drives Who

[From §3.2.4.2, §3.2.5.2]

Classes are totally devoid of explicit connections to their aspect modules. The association is done in the declaration of the aspect modules. There is one simple reason for this: the goal is of not interfering with the component language and with the components themselves. If the association was done by the classes, either (1) the component language would have to be extended, or (2) the inheritance mechanism would be used; in the latter case the components could not stand alone, because they would be attached, by inheritance, to D.

3.3.2. The Component Language

3.3.2.1 Class-based Language vs. Prototype-based Language

[From§3.2.2.1]

The component language is class-based. The alternative would be to have a prototype-based language, such as Self. The reason for choosing a class-based language is pragmatic: one of the design principles of D is to be used with existing languages, and most object-oriented languages in use today are class-based and strongly typed.

3.3.2.2 Uniform Reference Semantics

[From§3.2.2.1]

Some object-oriented languages, most notably C++, do not require all objects to be created dynamically, but they also allow them to be directly declared. In C++, for example, objects may be created automatically on the stack when the program execution enters a new block. This creates a mixed paradigm for handling objects, namely through their references (stored in reference variables) and through the objects themselves.

D's component language can also include such mixed paradigm. What it can't support so well is passing objects by value in method invocations. If the component language supports pass-by-value semantics for local method invocations, there will be some confusion in integrating the language with RIDL's parameter passing mode declarations (§3.2.5.5). In particular, RIDL's gref mode would conflict with the component language's pass-by-copy: what would it mean to pass a global reference to an object that the method's signature declares to be passed by copy?

Although RIDL could be redesigned to work under a mixed paradigm, the uniform reference semantics is simple and powerful enough for prototype purposes. Therefore D assumes the Java-like uniform view of objects.

3.3.2.3 Threads

[From§3.2.2.3]

Strictly speaking, the creation of new threads should not be part of The component language, the object language. The creation of concurrent activities can be seen as an issue that is relatively separate from the class implementation, since ultimately it affects the amount of concurrency, and it could have its own language. Alternatively, The component language could include a syntactically identifiable form for denoting the start of a new thread (e.g., fork).

However, *creating* threads is a relatively untangled procedure, since it consists of interfacing to the thread library with no consequences other than the new thread itself. *Coordinating* them is the difficult part, and that is dealt with by COOL. Therefore, to keep things simple, the creation of new threads is assumed to be made within the components, and not in any special manner. The particular way by which threads are created depends on the language environment; it can be by interfacing the thread library directly or by creating special thread objects (like in Java).

3.3.2.4 The Default Synchronization

[From §3.2.2.4]

An earlier version of the design of D [48] defined a different default strategy for thread coordination. That is, all objects were monitors, and COOL would then relax/modify that behavior. Although that strategy seems more manageable for doing an eventual formal reasoning about concurrent objects (because the object's consistency is guaranteed to be preserved), there is one major reason for not doing it:

Thread synchronization includes two different issues: mutual exclusion and guarded suspension. Both of these issues are *additional constraints* on the unsynchronized execution of methods. We could envision a language framework in which the component language imposes the maximum constraints and the coordination language relaxes those constraints. The problem is that the monitor abstraction captures mutual-exclusion well, but it fails to capture guarded-suspensions (see discussion §2.4.1.1, in Chapter 2). That is, a simple monitor behavior is not enough to capture the synchronization scenarios where threads are suspended waiting for state changes. Three options exist, then: (1) we could eliminate guarded suspensions (method invocations simply fail if preconditions are not met); (2) we could include guarded suspensions in the component language; or (3) we could design a coordination language for expressing both relaxation for mutual exclusion and additional constraints for guarded suspension.

The first option is clearly undesirable: one of the most powerful features of multithreaded systems is precisely the ability for threads to wait for state changes made by other threads without consuming CPU cycles. The second option defeats the goal of aspect separation. The earlier version of D followed the third option. However, under that design, the coordination language was confusing. For example, what was the role of COOL's coordinators (§3.2.4.2)? Would they take complete responsibility of all the synchronization constraints, including mutual exclusion, or would they handle mutual exclusion in terms of additional relaxation to the default monitor behavior? If they would take complete responsibility, then the simple fact that a class had a coordinator, even if empty, would take the objects to the other extreme of the synchronization spectrum, that is, total relaxation. If mutual exclusion was expressed in terms of additional relaxation, then that wouldn't be coherent with the additional constraints for guarded suspension.

After considering these pros and cons, it was decided that the default behavior of objects should be as it is: no synchronization at all. Coordinators have the very clear role of defining *all* additional synchronization constraints, both for mutual exclusion and guarded suspension.

3.3.2.5 The Default Communication

[From §3.2.2.4]

The same earlier version of D [48] also defined a different default strategy for remote communication. That is, all objects could be accessed remotely, even without having RIDL's portals. The implementation of the language would, of course, generate all the necessary infrastructure for remote access directly from the class declarations. A set of components could, therefore, be a distributed program, and all objects could be remote objects whose portals are given by their classes. This is a design similar to that of Obliq [12], for example, establishing a model of objects that is completely location-transparent. But again, there are good reasons for not doing it.

The argument is as follows. Classes are poor abstractions for remote access, and something else is necessary (see discussion in §3.3.4.1). But if the default remote access strategy assumes that all classes also define implicit remote interfaces, then portals themselves become a confusing and intrusive abstraction. What would it mean when some methods of the class were omitted from the portal declaration? Would the portal be just an annotation to the interface defined by the class, or would it identify a subset of methods that can be invoked remotely? Ultimately, this design didn't seem too solid.

3.3.3. COOL

3.3.3.1 Coordination: Classes vs. Abstract Method Sets

[From §3.2.4.2]

Take, for example, the design of the "behavioral" classes of DRAGOON (see 2.4.2.3, page 42). The "behavior" is defined in terms of sets of abstract operations, without any consideration for whether there exists a class that complies with the specifications, or even needs that behavior. "Behavioral" classes are, indeed, abstract descriptions of coordination. It is then the responsibility of the classes to comply with that abstract description, providing not only the necessary method mappings, but also the particular implementations that are consistent with the abstract "behavior."

Although interesting, this approach is artificial and counter-intuitive. In the literature, there isn't one single design methodology that takes abstract coordination as a step in the design of concurrent applications. But that's not all. DRAGOON's "behavioral" classes — which are not associated with any particular implementation — are misleading. Consider, for example, the "behavioral' class shown in **Figure 27** (page 44). It describes the synchronization for only specific implementations of the bounded buffer, not for all implementations. To make this point clear, consider this alternative implementation of bounded buffers, inspired by **Figure 12** (page 23):

```
public class BoundedBuffer {
  int putPtr = 0, takePtr = 0, usedSlots = 0, emptySlots;
  Object array[];

  public BoundedBuffer(int capacity) {
    array = new Object[capacity];
    emptySlots = capacity;
  }
  public void put(Object o) throws IsFull {
    do_put(o);
    increment_usedSlots();
  }
  public Object take() throws IsEmpty {
    Object old = do_take();
    increment_emptySlots();
    return old;
  }
  public boolean isFull() { return (usedSlots == array.length); }
  private void do_put (Object o) {
    if (emptySlots <= 0) // buffer is full      |
      throw new IsFull();                       |  this uses only emptySlots
    --emptySlots;                               |
    array[putPtr] = o;                          |
    putPtr = (putPtr + 1) % array.length;       |
  }
  private Object do_take() {
    Object old;                                 |
    if (usedSlots <= 0) // buffer is empty      |
      throw new IsEmpty();                      |  this uses only usedSlots
    --usedSlots;                                |
    old = array[takePtr];                       |
    takePtr = (takePtr + 1) % array.length;     |
    return old;
  }
  private void increment_usedSlots()  { usedSlots++; }
  private void increment_emptySlots() { emptySlots++; }
}
```

It is relatively easy to see that in this implementation `put` and `take` don't need to be neither mutually exclusive nor self-exclusive. Concurrent accesses to these objects can be safely synchronized as follows:

```
coordinator BoundedBuffer {
  selfex do_take, do_put;
  mutex  {do_take, increment_emptySlots};
  mutex  {do_put, increment_usedSlots};
  do_take: requires !empty;
        on_exit { if (full) full = false; }
  do_put:  requires !full;
        on_exit { if (empty) empty = false; }
  increment_emptySlots:
        on_exit { if (emptySlots == array.length) empty = true; }
  increment_usedSlots:
        on_exit { if (usedSlots == array.length) full = true; }
}
```

This coordination strategy allows more concurrency than the one defined in the “behavioral” class of **Figure 27**; it is possible because of how the bounded buffer is *implemented*, namely having two variables, instead of one, to account for the number of elements in the buffer. If this BoundedBuffer class was a DRAGOON “functional” class, then, although the method mapping is possible, we shouldn’t use the “behavioral” class of **Figure 27**, but rather we should define a new “behavioral” class that coordinates these bounded buffers according to this implementation.

That is, DRAGOON’s abstract behavior descriptions are only abstract to the extent that they *implicitly* comply with the implementations. The fact is that thread synchronization is intrinsically dependent on the *implementation* of the components, and not only on the names of their methods. Therefore, an abstraction mechanism based only in abstract method sets is not only not very useful, but it is also misguiding.

D avoids this by associating coordinators with classes, not with types. Coordinators have access to parts of the implementation of the classes. The coordination aspect programs are *helpers* with respect to the *implementation* of the components. They don’t intend to be more general than that.

3.3.3.2 Granularity of Synchronization

[From §3.2.4.2, §3.2.4.3, §3.2.6]

Synchronization strategies can be programmed only on a {per method $\times$ per class} basis; those strategies are then associated either with each instance of the class (per object: each object has its own coordinator) or with all instances of the class (per class: all instances share the same coordinator). The decisions involved in the granularity of synchronization in COOL can be summarized as follows:

(1) the provider (i.e. the class) is the one that defines the synchronization (monitor approach);

(2) the smallest unit of synchronization is the method;

(3) there is no middle ground between one instance and all instances of the same class;

(4) the coordination is contained within one coordinator;

(5) the association between an object and its coordinator is static.

The alternatives to these decisions are:

(a) the synchronization would be driven not only by the class, but also by its clients;

(b) the units of synchronization would be not only methods, but also arbitrary blocks of code;

(c) synchronization strategies could affect arbitrary instances of arbitrary classes;

(d) the coordination of one object could be split among several coordinators;

(e) the association could change dynamically.

(1) vs. (a): it may happen that the synchronization scheme affecting an object depends on the context in the caller. For example, the caller may need to ensure mutual exclusion of two objects simultaneously in order to execute safely. The only way of doing this is by using some kind of semaphore approach (e.g. locks). But, as discussed in Chapter 2, semaphores do terrible things to programs, because they establish hidden implementation dependencies between modules. COOL discourages those practices, and favors the monitor approach. If the synchronization is context-dependent, then that context should be abstracted in a class, and that class should then be coordinated. However, COOL does not prevent from programming with semaphores.

(2) vs. (b): while (b) could, eventually, be supported — by, for example, including some kind of pattern matching primitives in COOL — its non-alignment with the object-oriented modularities raises some doubts about its clarity. That kind of unruled synchronization can always be transformed into more solid designs, by encapsulating those blocks in methods. But this is certainly a point to have in mind while evolving COOL.

(3) vs. (c): the decision comes as a natural consequence of using object-oriented languages based on classes. A class defines the same behavior for all of its instances, and it is not possible to define particular behaviors for particular instances. The only way to do that is with tests in the code. This is what happens in COOL, too (see coordinator for dinning philosophers in §3.2.4.11).

(4) vs. (d): splitting the responsibility of coordination would overcome some limitations of COOL. For example, we could have one coordinator for taking care of issues that affect the whole class (methods that affect class variables), and several per object coordinators for taking care of issues that affect each of its instances, individually. This will require

a new design effort for defining how the coordinators relate to each other, and for understanding all the benefits and disadvantages of such approach. The current version of COOL decided for the simplest approach, but this is another idea to have in mind while evolving COOL.

(5) vs. (e): the idea of being able to change an object's coordination scheme at run-time is appealing. However, it is not clear that dynamic associations are useful in practice. Much more evidence is necessary to justify that modification to COOL.

3.3.3.3 Synchronization: Local vs. Distributed

[From §3.2.4.2]

In distributed systems, the issue of which component does what and when may be important. COOL does not capture that issue. It only deals with thread synchronization within each execution space. The reason is simple: the issue didn't arise in the many distributed applications that have been studied. It is not clear if distributed coordination leads to code tangling in the implementation of the components or if it is a logical function of the components themselves. More study needs to be done.

3.3.3.4 Exclusion Constraints

[From §3.2.4.6, §3.2.4.7]

Exclusion of threads over the execution of methods is expressed in a declarative form. An alternative would be using an imperative form. The detection of a method execution and completion could be done in the method managers, which could set and reset variables in the one entry and on exit clauses. However, that seems like a complicated and error-prone alternative to the simple declarative form that is used in COOL.

3.3.3.5 Assignments

[From §3.2.4.10]

Coordinators can access the objects' state, but they can only modify their own state. This comes from design principle number two (§3.1.2): it is extremely important that the aspect programs are well within bounds of their responsibilities. Objects maintain their own invariants; if coordinators could modify the state of the objects, the coordinators might destroy those invariants.

One of the major drawbacks of the reflective approaches to concurrency control is their lack of clear boundaries between the metaobjects and the base-objects: metaobjects can do almost anything. COOL restricts coordinators to deal with synchronization.

3.3.3.6 Current Limitations

The current version of COOL can express relatively sophisticated coordination schemes, but there are some limitations for what it can do. These limitations come not from any fundamental problem with the design principles, but simply from the fact that some issues were not addressed yet.

One of those issues is time-bounded suspensions. In COOL, when threads are suspended due to a pre-condition, there is no means to abort the suspension. An earlier version of COOL [49] addressed this issue by providing an optional timeout clause associated with the requires clause. Although such design is straightforward, it was temporarily removed until there is a better understanding of how to deal with failures.

Another issue that has not been addressed is thread scheduling. In COOL, when threads are suspended and resumed there is no way to control which thread runs first and for how long; COOL uses the default scheduling policies, which are not necessarily the best ones in every case. This limitation can be overcome by doing thread scheduling in the classes, but that is in clear violation with the design principles of the framework.

3.3.4. RIDL

3.3.4.1 Remote Interaction: Classes vs. Abstract Types

[From §3.2.5.2, §3.2.5.5, §3.2.5.6, §3.2.5.7]

Although portals look like abstract types, i.e. a collection of operation signatures without implementation, they are not abstract types. The visible difference is in the portals' copying directives, which can refer to fields of any class of the application. But there is a fundamental difference that will allow the concept of portal to evolve into a sophisticated construct for defining interaction with remote objects: portals were designed to work directly with classes; they rely on the visible elements and on the remote interaction interface, which, currently, they use only in transfer specifications, but which future designs of RIDL can use for other purposes (replication, for example).

However, many languages and systems before D have chosen to use abstract types as the basis for remote interaction. Some examples are Emerald [10], CORBA [57] and Java RMI [27]. The benefits of connecting remote components using abstract types, as opposed to using low level point-to-point connections, can be summarized as follows: (1) a significant part of the interface, namely the valid operations, their parameters and return values, can be checked at compile time; (2) it allows for the possibility of changing the implementation of the services without changing the client code.

While abstract types are powerful abstractions that programming languages, distributed or not, should support, they are not necessarily the best abstraction for capturing remote interaction. Abstract types are limited, precisely because they don't disclose anything about their implementations. Therefore all the issues about remote interfacing that are dependent on the implementations (e.g., selective data transfers, consistency of replicated data, etc.) must be coded around the abstract types and within the classes. When defining the interface to a remote object, a lot more can be said about that remote interface than just which operations the object supports.

D's portals are designed to be used as connectors-between-remote-components. Portals have all the advantages that abstract types have: (1) the valid operations, their parameters and return values can be checked at compile time; (2) changing the implementation without changing the clients can be achieved with the ordinary mechanisms that the component language provides for doing it (e.g., inheritance). On top of that, they allow the specification of transfer strategies whose consistency can be checked at compile-time. Going back to the discussion in §3.3.1.1, transfer strategies are an important part of the assumptions that remote components make about each other, therefore being part of their interface. Abstract types force that part to be implicit, whereas portals make it explicit.

The current version of RIDL is only a sample of what a real remote interaction language can be. Nevertheless, from the language design point of view, RIDL, as it is now, makes the important design decision of shifting from the current abstract type - based remote interaction to a language of true remote interaction that is smoothly integrated with the component language.

3.3.4.2 Granularity of Remote Interaction

[From §3.2.5.2, §3.2.5.3, §3.2.6]

The granularity of remote interaction in RIDL is very similar to the granularity of synchronization in COOL, which was discussed in §3.3.3.2. The alternatives are also similar. In RIDL the decisions were based on prudence. The current version of RIDL is highly influenced by the existing interface definition languages. IDLs have proven to be a good idea, but the existing ones are too much grounded on, and therefore limited by, the concept of abstract type (see discussion in §3.3.4.1). The first step towards a powerful language that expresses many kinds of remote interaction protocols at the application level is to make the shift from the type-based approach to the aspect-based approach. Once that shift is done, the possibilities are immense, as the following list suggests.

The decisions on the granularity of remote interaction can be summarized as follows:

(1) the provider (i.e. the class) is the one that defines the remote interaction;

(2) the smallest unit for remote interaction is the method;

(3) the remote interaction is contained within one portal;

(4) the association between an object and its portal is static;

(5) there are no multi-class portals.

The alternatives to these decisions are:

(a) the remote interaction would be driven not only by the class, but also by its clients;

(b) the units would be not only methods, but also arbitrary blocks of code;

(c) the remote interaction with one object could be split among several portals;

(d) the association could change dynamically;

(e) there could be specialized portals which would specify remote interaction strategies between particular pairs/sets of components.

(1) vs. (a): portals establish a contract between the classes and all the clients in remote spaces, and that contract, in the current version of RIDL, is not flexible. However, that kind of flexibility is important for remote interactions. A point to have in mind when evolving RIDL.

(2) vs. (b): the alternative here is not to align data transfer protocols with service requests, resulting in some kind of lazy data transfers. RIDL chose the simplest protocol. Another point to have in mind while evolving RIDL.

(3) vs. (c): for the current version of RIDL, this alternative would add nothing useful. However, if RIDL evolves towards more sophisticated remote client/provider protocols, it is very likely that splitting a portal into a pair of input/output portals will make those protocols more clear. The current portals are input portals.

(4) vs. (d): as in COOL, it is not clear that dynamic associations in RIDL are useful in practice. Much more evidence is necessary to justify that modification to RIDL.

(5) vs. (e): considering that one component may interact differently with different remote components, (e) seems like a good idea. This can be seen as an extension to alternative (c).

3.3.4.3 On the Semantics of gref

[From §3.2.5.7]

Smart implementations could eventually replicate remote objects and guarantee the consistency among the replicas. But the semantics of gref includes operational specifications, namely that these parameters will not be automatically replicated.

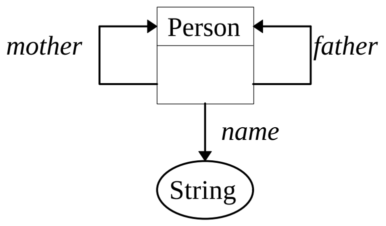
There are operational differences between the implementation with replication and the other implementation without replication, namely differences in performance (potentially, but not necessarily, better in the first case), availability and protection. In a distributed system it is important to be able to chose between different operational options. Even if a consistent replication mechanism is available, using it is not necessarily the best in all cases. For example, clients may not want replicas of remote objects in their spaces, since methods of those objects may carry unwanted overhead (new threads, etc.). For that reason, gref is defined as operating under the most simple remote access mechanism, namely the one that guarantees that no replication occurs and that the invocations will always be performed within the remote object itself.

3.3.4.4 The Copying Directives

[From §3.2.5.9]

Copying directives allow a finer granularity for the transfer facility than that of the existing RMI systems. Their selection primitives, `bypass` and `only`, were taken from Demeter's graph traversal language [42, 43, 59] and [50]. Most of the expressiveness and complexity of Demeter language were left out, because they didn't seem to be necessary for RIDL. The simple directives for cutting fields provide a basis for controlling object transfers. (Several examples are shown in Chapter 5)

While `bypass` has the same meaning both in RIDL and Demeter, `only` is different from its Demeter counterpart `through`. The `only` primitive is context-free: for the same transfer specification, instances of the same class are always transferred in the same way; Demeter's `through` primitive is context-dependent: for the same traversal specification, instances of the same class may be traversed differently. Consider, for example, the following class graph, in which the parts *mother* and *father* may by null:



In RIDL, passing a Person object with the directive `{Person` **`only`** `mother, name;}` is equivalent to passing it with the directive `{Person` **`bypass`** `father}`; the next instance of Person to be transferred (i.e. the person's *mother*) results in bypassing the *father* part again; and so on.

In Demeter, traversing a Person object with the directive **`through`** `Person` $\rightarrow$ `mother` excludes the father part, but only for the first Person object that is traversed; that is, the first person's father is excluded, but all of the first person's grandparents, male and female, will be traversed. While context-dependent directives may seem more powerful, their disadvantages with respect to the "surprise" factor — introduced by the history of the traversed objects — are far greater than their advantages.

An earlier version of RIDL [47, 48] included Demeter's `to` primitive. This primitive was taken out, because it didn't seem very useful for data transfers.

3.3.4.5 The Missing Parts

[From §3.2.5.9]

When objects are passed by copy, and if a copying directive is provided, some parts of the original object graph may be missing in the replica. The question, then, is what to do if the space that contains the replica tries to access one of those missing parts.

There are at least three possibilities: (1) to make those parts remote objects, and pass global references to them; (2) to automatically fetch the missing objects; (3) to signal an error. Both options (1) and (2) violate the principles of encapsulation and protection in distributed systems. Therefore, D follows the third option: a space that contains an incomplete replica is confined to execute only over that portion of the original object graph, since that was what the programmer of the portal defined.

However, errors can be signaled in different ways: (1) trust the programmer, and do nothing about it — this may result in severe run-time errors; (2) detect those invocations and issue run-time warnings; (3) detect those invocations and raise exceptions. The current specification of RIDL does not establish what the right procedure should be (see §3.4).

3.3.4.6 Current Limitations

With respect to parameter passing modes in the existing RMI systems (Java's and some implementations of CORBA), RIDL provides more expressive power than these systems, because it gives the means express object transfers that are more than type-based.

There are, however, many limitations to what RIDL can do with respect to other platforms for distributed programming that have been proposed before. RIDL's two parameter passing semantics (i.e. gref and copy) are not enough to capture many interesting situations in distributed systems. Migration and replication are two examples of parameter passing semantics that are missing from RIDL.

3.4. Final Remarks

This chapter presented D. First, the design principles were stated, then the language was described, and finally the design decisions were discussed. In this presentation, one important issue was purposely left unspecified: error handling. Very little was said about how to deal with errors, both compile- and run-time errors.

The specification of the languages in §3.2 is, hopefully, enough for inferring the errors that a compiler should detect. Unfortunately, the run-time errors are far more important, from the design point of view. For example, how is it that a client detects a remote invocation that doesn't succeed? How does it detect the invocation to a part that was not copied? If COOL is extended with guarded suspension that is bounded in time, for example, how does the client detect the timeout?

The real issue here is not so much the mechanism of error handling — which can be implemented using the existing mechanism of exceptions — but *how to design the integration of that mechanism so that it doesn't violate the design principle of separation of concerns*. If the code in the clients is invaded by exception handlers for detecting aspect failures, that separation is jeopardized. Ideally, the code for exception handling should be specified in separate from the implementation of the clients. But that is a whole new aspect language, which will have to define how aspect errors propagate from provider classes to client classes.

In summary, the reason why error handling was omitted from the specification of D is not that error handling is not a problem, but rather that, if we follow the three design principles, it is too big of a design problem. Much more research needs to be done.

Chapter 4

Implementation

“HACK ATTACK *noun.*

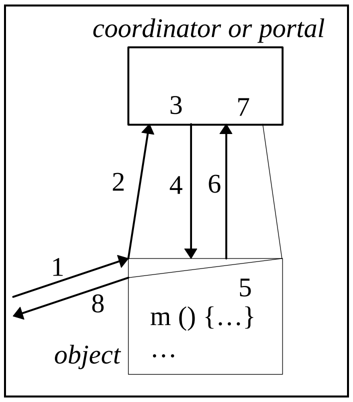
A period of greatly increased hacking activity.”

Guy Steele et al. in *The Hacker’s Dictionary*

4. Implementation

D, as described in Chapter 3, was integrated with Java in a framework called DJ. The subset of Java that is used in DJ is called JCore, and it contains almost everything of Java. The description of JCore and an introduction to programming in DJ is given in Appendix B. This chapter describes the most important points about the implementation of DJ.

In describing the interaction between objects and aspects, and in spite of the differences between COOL and RIDL, the same illustration was used (pages 62 and 76); this illustration is presented below, without the specifics of the protocols. This figure shows that coordinators and portals interact with the objects they are associated with in very much the same way: through trapping method invocations and performing actions before and after method execution.



The figure also suggests a simple implementation of the aspect languages that consists of translating coordinators and portals into Java "aspect objects." Such objects execute the particular aspect run-time. The ordinary JCore objects simply need to be extended (*woven*) with hooks that transfer the control to the aspect objects at the beginning and at the end of all methods. In the presence of a reflective OOPL with capabilities for reifying method invocation (e.g. CLOS), those hooks aren't even necessary, since they are already part of the language. But that is not the case with Java. The implementation described here follows this simple approach.

The protocol between objects and aspects being so simple, most of the effort of the implementation is concentrated in the translation of the aspect modules into "aspect classes." How are COOL's mutual exclusion declarations implemented in Java? What are the run-time structures that implement a remote object? How are RIDL's copying directives implemented? All D constructs are translated into specific patterns of Java code, hereafter called the target architectures. The correctness and performance of D depends on these architectures. Because they are so important, they are also called "the implementation" of D.

The transformation from DJ into the target architectures in Java can be done manually. Manual weaving is convenient for experimenting with different implementations, but not suitable for general use. A tool, called the Aspect Weaver, automates those transformations. An implementation of the Aspect Weaver is given in Appendix C.

There is a large and interesting space of different possibilities for the target architectures. This chapter and its associated appendices C and D show one particular point in that space, and one that values simplicity over performance. The architectures described here are simple and not optimized, so that the transformation algorithms are easy to understand and reproduce.

4.1. Engineering the Implementation Space

In implementing D, the first engineering decision that must be made is on how much information to process at a time. One possibility is to process the whole source code of a DJ program at the same time. So, for example, a superclass would always be processed with all its sub-classes; the classes of the parameters of a method would always be processed together with the class where the method is implemented; the aspect programs would be processed together with the classes they are associated with; etc. Weaving over the global program has the advantage that, at translation time, there is all the necessary information to generate exactly the right code, and, more importantly, no more than that code; therefore, the target architectures can be highly optimized. In this approach, modules cannot be selectively re-assembled according to the modularities of D, because the output code is a mixture of information coming from several modules. That is, if an aspect program changes, the classes it is associated with must be re-woven; if the input class changes there is the need to re-generate the aspect classes.

The other option is separate processing of modules. Separate processing introduces additional constraints in the target architectures, because the information is limited to the interfaces of the modules (as described in §3.3.1.1). In order to cope with limited information, the target architectures need to be prepared for all possible situations. Therefore, much more code needs to be generated. However, because there is not much space for optimizations, the target architectures are relatively simple, and the translation rules are easy to understand. The implementation described in this chapter follows this second approach, mostly because of its simplicity.

As a preview of the architectures that will be presented here, Table 1 shows the relations between input and output modules for a class named `aClass` that is associated with a coordinator and a portal.

Input DJ modules	**Output Java modules**
aClass.jcore (JCore class)	aClass.java (woven class)
aClass.cool (coordinator)	aClassCoord.java (coordination class)
aClass.ridl (portal)	aClassPRI.java (RMI interface of portal) aClassP.java (portal class) aClassPP.java (proxy of the previous) aClassTraversals.java (repository of traversals)

Table 1. Relation between input and output modules.

For multi-class coordination, ".cool" files must follow an additional naming convention (e.g. dashes between names). When a JCore class is not associated with a coordinator, the corresponding output coordinator class is not generated. The same for when the JCore class is not associated with a portal. However, in this implementation all JCore classes, even if not associated with coordinators or portals, are still woven, for purposes of the implementation of RIDL (more specifically, for marshaling).

As the number of output modules already suggests, the implementation of RIDL is considerably more complex than the implementation of COOL. The next two sections explain in detail the target architectures for COOL and RIDL, separately. Section §4.4 explains how to integrate them.

4.2. Target Architectures for Implementing COOL

Coordinators are translated into Java classes, which implement the coordination run-time. The variables of these classes maintain the execution state of the corresponding JCore objects, and the methods of these classes are 'before' and 'after' methods associated with the JCore classes' methods. In turn, the JCore classes are translated into Java by weaving calls to a coordinator object in the beginning and at the end of each of the coordinated methods. This solution is depicted in Figure 34. The "try… finally" block traps the return of the original method body even in the presence of exceptions.

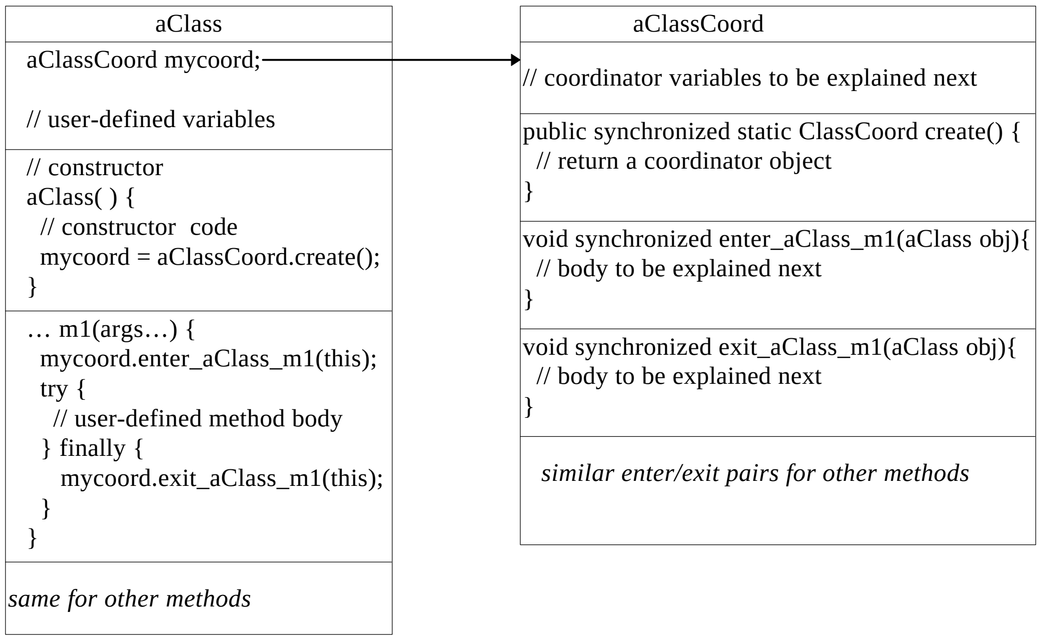

aClass
aClassCoord mycoord;
// user-defined variables
// constructor
aClass( ) {
// constructor code
mycoord = aClassCoord.create();
}
… m1(args…) {
mycoord.enter_aClass_m1(this);
try {
// user-defined method body
} finally {
mycoord.exit_aClass_m1(this);
}
}
same for other methods
aClassCoord
// coordinator variables to be explained next
public synchronized static ClassCoord create() {
// return a coordinator object
}
void synchronized enter_aClass_m1(aClass obj){
// body to be explained next
}
void synchronized exit_aClass_m1(aClass obj){
// body to be explained next
}
similar enter/exit pairs for other methods

Figure 34. Output code architecture for weaving COOL.

4.2.1. Coordinator Objects

Coordinator objects are associated with JCore objects when the JCore objects are instantiated. For that, all constructors in JCore classes are extended with a statement for obtaining their coordinator object. There is only one important detail in obtaining coordinator objects: for multi-class coordination, all instances of the coordinated classes must share the same coordinator object. In order to implement this correctly, the instantiation of coordinator objects is done by a class method in the coordinator class. For per object coordination, this method always returns a new instance of the class; for per class coordination, this method checks whether the class has already been instantiated once; if not, it creates one instance; the method returns the only instance of the class.

PER OBJECT COORDINATION	PER CLASS COORDINATION

```
class aClassCoord {

  static aClassCoord createCoord(){
    return new aClassCoord();
  }
}
```
	```
	class aClassCoord {
	  static boolean one = false;
	  static aClassCoord theCoord;
	  static synchronized
	         aClassCoord createCoord(){
	    if (!one) {
	      theCoord = new aClassCoord();
	      one = true;
	    }
	    return theCoord;
	  }
	}
	```

As explained before, the role of coordinator objects is to keep track of the synchronization state of the JCore objects. In order for this state to be consistent throughout the life of the objects, coordinator objects are fully synchronized. Hence, the `synchronized` qualifier in every method of the class (see Figure 34).This guarantees that all accesses and updates to the synchronization state are performed exclusively by one thread at a time. This architecture is completely different from having the synchronization in the JCore methods themselves. The coordination methods in the coordinator class have only a small amount of computation that corresponds to checking the synchronization state and, eventually, updating that state. The execution of the JCore methods, which can be arbitrarily lengthy, is then performed outside Java's synchronized blocks (methods or statements); instead, it is ruled by the more sophisticated synchronization implemented by the coordinator objects.

The pairs of 'before' and 'after' methods follow the architecture described below in a mixture of Java and English:

```
synchronized void enter_className_methodName (className obj) {
   1. check conditions for waiting; wait while they are not met;
   2. if conditions are met, update internal synchronization state;
   3. execute on_entry statements;
}
synchronized void exit_className_methodName (className obj)  {
  1. execute on_exit statements;
  2. update internal synchronization state;
  3. notify waiting threads, so that they can re-check the conditions;
}
```

4.2.2. Mutual Exclusion and Re-entrance

COOL's most distinct feature is the declarative mutual exclusion of sets of methods. For example,

```
coordinator aClass {
  selfex f, g;
  mutex {f, h};
  mutex {g, m, n};
}
```

A simple implementation of these declarations consists in having run-time structures that allow an almost literal interpretation of the mutual exclusion constraints. That is, for each method of the coordinated classes there is an object that keeps track of the method's execution state: if some thread is executing it, and, if so, the thread's identifier. Implementing the mutual exclusion constraints consists simply of testing these method state objects and acting accordingly. So, for example, the interpretation of the constraints above is: f can only be executed by thread T if no other thread is executing f (selfex constraint) and no other thread is executing h (mutex constraint); g can only be executed by thread T if no other thread is executing g (selfex constraint) and no other thread is executing m and no other thread is executing n (mutex constraints); h can only be executed by thread T if no other thread is executing f (mutex constraint); etc. The translation of the coordinator shown above is a direct application of this interpretation:

```
class aClassCoord {
  MethState f = new MethState(), g = new MethState(), h = new MethState();
  MethState m = new MethState(), n = new MethState();

  synchronized void enter_aClass_f(aClass obj) {
    // conditions for waiting: other thread in f (selfex) or in h (mutex)
    while (f.isBusyByOtherThread() || h.isBusyByOtherThread()) {
      try { wait(); }
      catch (InterruptedException e) {} // notifyAll raises this exception
    }
    // at this point, constraints are met. Thread may proceed.
    f.in(); // update the method state: this thread is executing f
  }
  synchronized void exit_aClass_f(aClass obj) {
    f.out(); // update the method state: this thread finished executing f
    notifyAll(); // before leaving, notify other threads that this thread is
                 // out. Other threads may be waiting to execute f or
                 // other methods
  }
  synchronized void enter_aClass_g(aClass obj) {
    // conditions for waiting: other thread in g (selfex) or in m,n (mutex)
    while (g.isBusyByOtherThread() ||
           m.isBusyByOtherThread() || n.isBusyByOtherThread()) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
    g.in(); // update the method state: this thread is executing g
  }
  synchronized void exit_aClass_g(aClass obj) {
    g.out(); // update the method state: this thread finished executing g
    notifyAll(); // before leaving, notify other threads that this thread is
                 // out. Other threads may be waiting to execute g or
                 // other methods
  }
  synchronized void enter_aClass_h(aClass obj) {
    // conditions for waiting: other thread in f (mutex); h is not selfex
    while (f.isBusyByOtherThread()) {
      try { wait(); }
      catch (InterruptedException e) {}
    }
    h.in(); // update the method state: this thread is executing h
  }
  synchronized void exit_aClass_g(aClass obj) {
    h.out(); // update the method state: this thread finished executing h
    notifyAll(); // before leaving, notify other threads that this thread is
                 // out. Other threads may be waiting to execute
                 // other methods
  }

  // methods for m and n are similar, but mutex with g

}
```

The MethState objects, in a first approach, consist of one boolean variable (a lock) that indicates whether the method is being executed or not. In this first approach, the method `isBusyByOtherThread` tests that boolean; the method `in` sets it to true; and the method `out` sets it to false. However, that approach is not sufficient for implementing the semantics of selfex and mutex in the presence of recursion. As a reminder, selfex and mutex exclude the execution of methods by a thread T with respect to other threads, not to T itself. If, in the example above, f makes a recursive call to itself or makes a call to h, as long as no other threads are executing those methods, the thread must proceed.

Therefore, MethState objects must be slightly more sophisticated than simple booleans. They must keep track of the number of recursive calls, as well as the identifiers of the threads that are executing the methods. In this approach, `in` increments a counter that keeps track of the number of calls to the method and stores the thread identifier; `out` decrements that counter; `isBusyByOtherThread` checks if the method is being executed by threads other than the current one. When the counter is 0, the method is not being executed by any thread; otherwise some thread is executing it. Note that if a method is not selfex, there may be several threads executing the method at the same time; MethState must keep track of all that information. An implementation of MethState is given in Appendix D; this implementation stores the thread identifiers in a list.

4.2.3. Requires Clause

In addition to the exclusion constraints (if any), the condition of a requires clause (§3.2.4.9) is simply another condition for waiting. Therefore, its implementation simply adds another test to the waiting condition shown for the “enter_” methods.

4.2.4. Access to Variables of the Coordinated Objects

The on_entry and on_exit statements (§3.2.4.10) can access variables of the coordinated objects. Because in the implementation architecture devised here the coordinator and the coordinated classes are different classes, and those variables may be private or protected, there is the problem of how to access those variables. The simplest way to implement this is by making all variables of the coordinated classes being public. But that violates the accessibility defined by the programmers, making it possible for other classes not only to read but also to modify those variables. A second option, which was followed here, is to generate accessor methods for all variables of the coordinated classes. This gives the necessary access and prevents from accidental modifications.

4.2.5. Per Class Coordination

The implementation of per class coordination is exactly the same as the implementation of per object coordination. The only difference is the instantiation of the coordinator object, which takes place as explained in §4.2.1. The naming of variables and methods in the coordinator class always includes the names of the class, so that there are no conflicts in method names.

4.2.6. Inheritance of Aspect Code

Implementing inheritance and its interaction with the aspects is one of the less straightforward points of the target architectures. According to the semantics of D, subclasses inherit the aspect programs of the superclasses, while being able to override the implementation of the inherited methods. Or they may override the aspect programs of the superclasses, while inheriting the methods from the superclasses. For example:

`class A {`
`  void f() { … }`
`  void g() { … }`
`}`
	`coordinator A {`
	`  selfex f;`
	`  mutex {f, g};`
	`}`

`class subA extends A{`
`  void h() { … }`
`}`
	`coordinator subA {`
	`  selfex f;`
	`  mutex {f, g, h};`
	`}`

`class B {`
`  void k() { … }`
`}`
	`coordinator B {`
	`  selfex k;`
	`  mutex {k, n};`
	`}`

`class subB extends B{`
`  void k() { … } // overriding`
`  void n() { … }`
`}`	

In subA, method g is inherited from A, but its coordination must check mutual exclusion not only with f but with h too. In subB, the method k, although redefined, must be coordinated in exactly the same way as in B.

There are a number of ways of correctly implementing this semantics. The architecture devised here takes a simple approach: it separates between implementation and coordination code. Each method of the coordinated classes results in two methods in the output woven classes: one method that contains the original method implementation and a second method that wraps the call to the first method in calls to the coordinator. For example, for class B, the output woven class is:

```
class B { //constructor omitted
  BCoord _BCoord; // the coordinator object

  protected void _d_k() { original implementation of k }

  void k() {
    _BCoord.enter_B_k(this);
    try { _d_k(); }
    finally {
      _BCoord.exit_B_k(this);
    }
  }
}
```

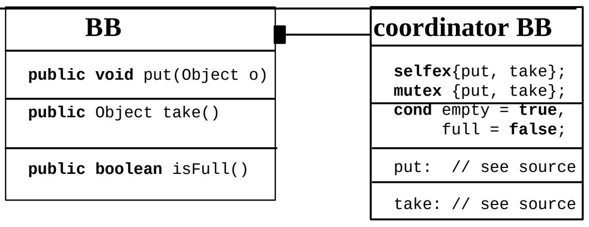
By doing this separation it is trivial to re-use the parts of the superclass that are necessary in the subclass. The implementation of method overriding consists in overriding the first method of the pair. For example, the woven class subB overrides the _d_k (the implementation) but not k (the coordination). Therefore, invocations to method k in subB objects first use the inherited k, which calls the proper _d_k by ordinary method dispatching. The implementation of aspect overriding consists of overriding the second method of the pair.

4.2.7. Examples

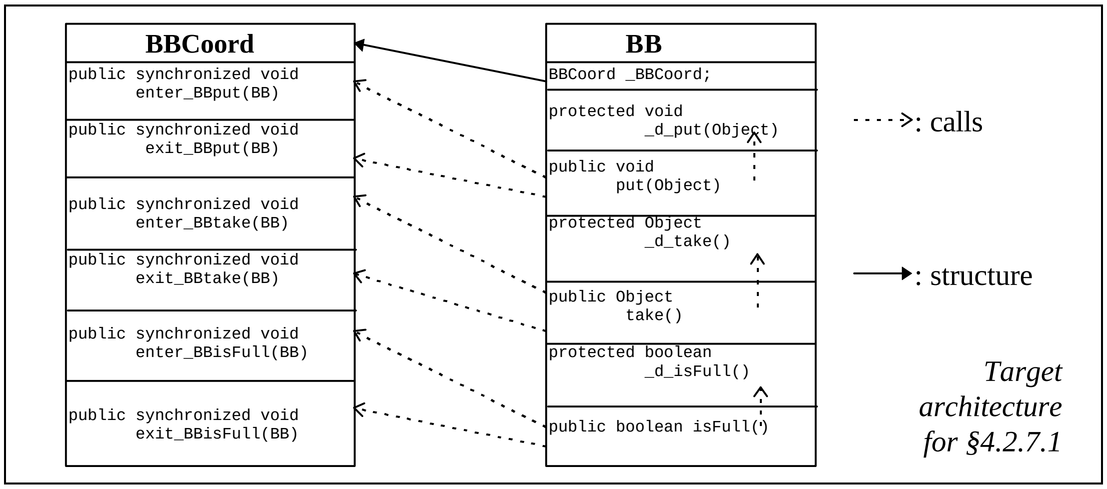
The following 4 examples illustrate the input/output code in a number of situations that cover all the important points in the semantics of COOL.

4.2.7.1 Simplest Case: one class, per_object coordination, no inheritance

Consider the following JCore class and its coordinator:



```
public class BB {
  protected int putPtr = 0,takePtr = 0;
  int usedSlots = 0; int size;
  private Object array[];

  public BB(int capacity) {
    size = capacity;
    array = new Object[capacity];
  }
  public void put(Object o) {
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    usedSlots++;
  }
  public Object take() {
    Object o = array[takePtr];
    takePtr = (takePtr + 1) % array.length;
    usedSlots--;
    return o;
  }
  public boolean isFull() {
    return (usedSlots == size);
 }
}
```


BBCoord
public synchronized void enter_BBput(BB)
public synchronized void exit_BBput(BB)
public synchronized void enter_BBtake(BB)
public synchronized void exit_BBtake(BB)
public synchronized void enter_BBisFull(BB)
public synchronized void exit_BBisFull(BB)
BB
BBCoord _BBCoord;
protected void _d_put(Object)
public void put(Object)
protected Object _d_take()
public Object take()
protected boolean _d_isFull()
public boolean isFull()
: calls
: structure
Target architecture for §4.2.7.1

```
coordinator BB {
  selfex put, take;
  mutex {put, take};
  cond empty = true,
       full = false;

  put:  requires !full;
        on_exit {
          empty = false;
          if (usedSlots == size) full = true;
        }
  take: requires !empty;
        on_exit {
          full = false;
          if (usedSlots == 0) empty = true;
        }
}
```

From this class and this coordinator, the target output code consists of the following two Java classes (in these examples, *italic* is used to mark lines of code that are woven in the original JCore classes):

```
// The woven class resulting from the JCore class BB
public class BB {
  protected BBCoord _BBCoord;                     // See key point A
  protected int putPtr = 0, takePtr = 0; int usedSlots = 0; int size;
  private Object array[];
  public BB(int capacity) {
    size = capacity;
    array = new Object[capacity];
    _BBCoord = BBCoord.createCoord();             // See key point A
  }
  protected void _d_put(Object o) {               // See key point B
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    usedSlots++;
  }
  public void put(Object o)
```

```
    _BBCoord.enter_BBput(this);                   // See key point C
    try {
      this._d_put(o);                             // See key point B
    } finally {
      _BBCoord.exit_BBput(this);                  // See key point C
    }
  }
  protected Object _d_take() {                    // See key point B
    Object o = array[takePtr];
    takePtr = (takePtr + 1) % array.length;
    usedSlots--;
    return o;
  }
  public Object take() {
    _BBCoord.enter_BBtake(this);                  // See key point C
    try {
      return this._d_take();                      // See key point B
    } finally {
      _BBCoord.exit_BBtake(this);                 // See key point C
    }
  }
  protected boolean _d_isFull() {                 // See key point B
    return (usedSlots == size);
  }
  public boolean isFull() {
    _BBCoord.enter_BBisFull(this);                // See key point C
    try {
      return this._d_isFull();                    // See key point B
    } finally {
      _BBCoord.exit_BBisFull(this);               // See key point C
    }
  }
  // accessor methods                                See key point D
  public int _dget_putPtr() { return putPtr; }
  public int _dget_takePtr() { return takePtr; }
  public int _dget_usedSlots() { return usedSlots; }
  public int _dget_size() { return size; }
  public Object _dget_array() { return array; }
}
// From COOL’s coordinator
public class BBCoord {                            // See key point E
  MethState BBput = new MethState();
  MethState BBtake = new MethState();             // See key point F
  MethState BBisFull = new MethState();
  // condition variables
  boolean empty = true;
  boolean full = false;
  public static BBCoord createCoord() {           // See key point G
    return new BBCoord();
  }

  // before method for put
  public synchronized void enter_BBput(BB jcoreobj) { // See key point H
    while (// conditions for waiting                   See key point I
           BBput.isBusyByOtherThread()   /* from selfex */ ||
           BBtake.isBusyByOtherThread() /* from mutex */ ||
           !(!full)      /* from requires */ ) {
```

```
      try { wait(); } catch (InterruptedException e) {};
    }
    // all conditions are false; current thread has the right to execute put
    BBput.in();                                   // See key point J
    // on_entry statements                           See key point K
  }
  // after method for put
  public synchronized void exit_BBput(BB jcoreobj) { // See key point H
    // current thread is leaving put
    BBput.out();                                  // See key point L
    // on_exit statements
    empty = false;                                // See key point K
    if (jcoreobj._dget_usedSlots() == jcoreobj._dget_size()) full = true;
    // notify blocked threads                        See key point M
    if (BBput.depth == 0) notifyAll();
  }

  // before method for take
  public synchronized void enter_BBtake(BB jcoreobj) { //See key point H
    while (// conditions for waiting                 See key point I
           BBtake.isBusyByOtherThread()  /* from selfex */ ||
           BBput.isBusyByOtherThread()  /* from mutex */ ||
           !(!empty)     /* from requires */ ) {
      try { wait(); } catch (InterruptedException e) {};
    }
    // all conditions are false; current thread has the right to execute take
    BBtake.in();                                  // See key point J
    // on_entry statements                           See key point K
  }
  // after method for take
  public synchronized void exit_BBtake(BB jcoreobj) {  // See key point H
    // current thread is leaving take
    BBtake.out();                                 // See key point L
    // on_exit statements                            See key point K
    full = false;
    if (jcoreobj._dget_usedSlots() == 0) empty = true;
    // notify blocked threads                        See key point M
    if (BBtake.depth == 0) notifyAll();
  }

  // before method for isFull
  public synchronized void enter_BBisFull(BB jcoreobj){}// See key point H
  // after method for isFull
  public synchronized void exit_BBisFull(BB jcoreobj){} // See key point H
}
```

Key points:

A. Being associated with a coordinator, class BB gets an additional instance variable that will hold the reference to the coordinator object. The type of this variable is the class type that results from translating the coordinator of COOL into a Java class, in this case class BBCoord (see §4.2.1).

B. The original methods of class BB are renamed. At the same time, new methods that take the original names are generated. These new methods make calls to the original, but renamed, methods. This renaming/duplication scheme ensures the separation between coordination (the public method of the pair) and implementation (the protected method of the pair) along the inheritance hierarchy. (See §4.2.6) Examples 2 and 3 will make this point clear.

C. The new, public, “coordination” methods are responsible for wrapping the calls to their corresponding protected “implementation” methods within before and after calls to the coordinator object.

D. Accessor methods to all variables are also generated. These methods are used by the coordinator object. (See §4.2.4)

E. The COOL coordinator is translated into a Java class whose name is the concatenation of the names of the coordinated classes, plus the suffix “Coord”. In this case, there is only the BB class; hence the name of the Java coordinator class is BBCoord. (See §4.2.1)

F. For each method of the coordinated classes, the coordinator class contains a variable of class type MethState. At run-time these variables hold the state of execution of each method of the JCore objects, and they are the basis for implementing mutual exclusion of threads on the execution of the JCore methods. Class MethState is given in Appendix D. (See §4.2.2)

G. Instantiation of coordinator objects is done though a class method, the “factory method.” (See §4.2.1 and example 4)

H. For every method of the coordinated JCore classes, the coordinator class contains two methods: one to be called before the JCore method is executed and one to be called immediately after the JCore method is executed (see C). These methods take the JCore object as parameter, so that they can access the object’s state in the on_entry and on_exit statements. All these methods are synchronized, to ensure that the coordinator’s state remains consistent. (See §4.2.1)

I. The exclusion constraints in the self-exclusion set and in the mutual exclusion sets of the COOL coordinator, as well as the requires clause in some method manager, define a waiting condition. The waiting condition uses the MethState variables for each of the methods that are mutually exclusive with the method at hand. The wait itself is done through Java’s `wait` method. (See §4.2.2 and §4.2.3)

J. As soon as all constraints are met (exclusion constraints and pre-condition), the thread has the right to execute the method. That fact is signaled to the corresponding MethState variable, by invoking its `in` method.

K. The on_entry and on_exit clauses of the COOL coordinator result in Java statements that are almost the same as the COOL statements. The only difference is that the identifiers that do not correspond to coordinator's variables are assumed to refer to variables of the JCore objects, and therefore are translated into calls to the JCore object's accessor methods (see D).

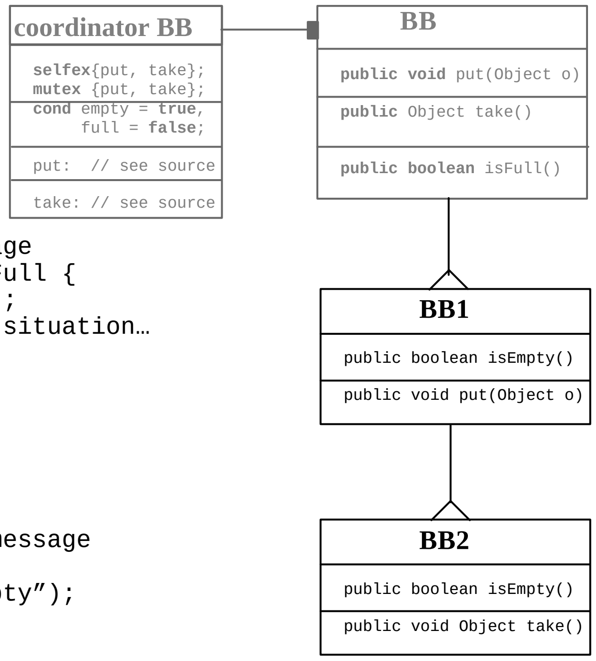
L. Immediately after the JCore method is executed, the corresponding exit_ method in the coordinator is called. The coordinator signals the fact that the thread is leaving by invoking the `out` method in the corresponding MethState variable.

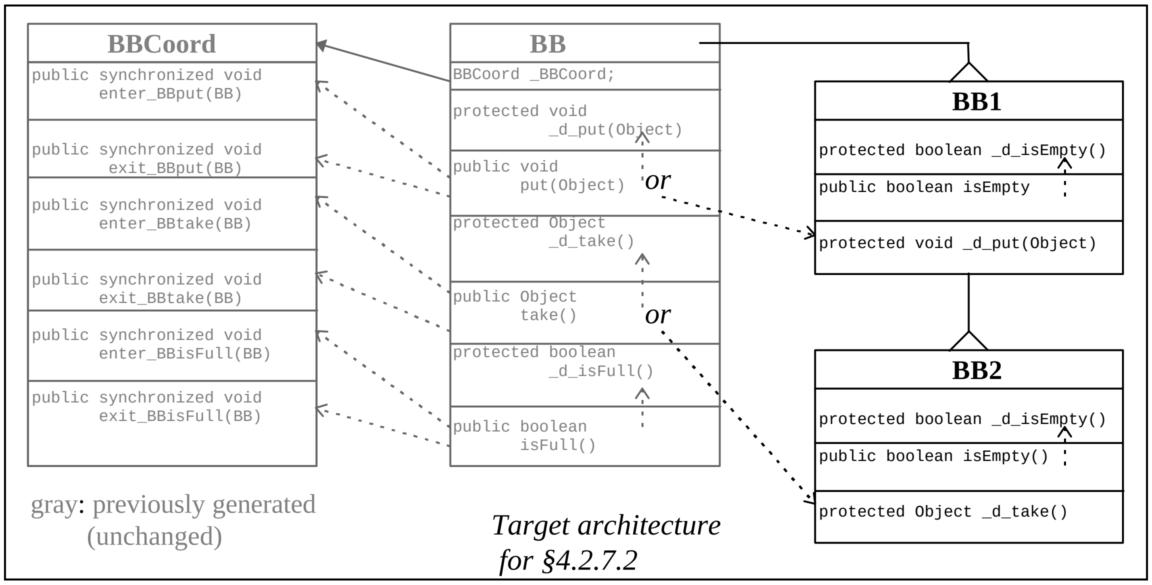
M. The last thread out of the method wakes up blocked threads, if any, so that the waiting conditions can be rechecked.

4.2.7.2 Inheritance of Coordination

Consider the following class hierarchy:

```
public class BB1 extends BB {
  public BB1(int capacity) {
    super(capacity);
  }
  // new method
  public boolean isEmpty() {
    return (usedSlots == 0);
  }
  // override put to print out a message
  public void put(Object o) throws IsFull {
    System.out.println (“This is put”);
    super.put();  // this is a tricky situation…
  }
}

public class BB2 extends BB1 {
  public BB2(int capacity) {
    super(capacity);
  }
  // override isEmpty to print out a message
  public boolean isEmpty() {
    System.out.println (“This is isEmpty”);
    return super.isEmpty();
  }
  // override take to print out a message
  public Object take() {
    System.out.println (“This is take”);
    return super.take();  // same tricky situation.
  }
}
```




BBCoord
public synchronized void enter_BBput(BB)
public synchronized void exit_BBput(BB)
public synchronized void enter_BBtake(BB)
public synchronized void exit_BBtake(BB)
public synchronized void enter_BBisFull(BB)
public synchronized void exit_BBisFull(BB)
BB
BBCoord _BBCoord;
protected void _d_put(Object)
public void put(Object)
protected Object _d_take()
public Object take()
protected boolean _d_isFull()
public boolean isFull()
or
or
BB1
protected boolean _d_isEmpty()
public boolean isEmpty
protected void _d_put(Object)
BB2
protected boolean _d_isEmpty()
public boolean isEmpty()
protected Object _d_take()
gray: previously generated (unchanged)
Target architecture for §4.2.7.2

According to the semantics of D, unless a class is explicitly associated with a coordinator, it inherits the coordinated behavior of its superclass. The output code is the following:

```
public class BB1 extends BB {
  public BB1(int capacity) {
    super(capacity);
  }                                             // See key point N
  protected boolean _d_isEmpty() {
    return (usedSlots == 0);
  }
  public boolean isEmpty() {
    return this._d_isEmpty();                   // See key point O
  }
  protected void _d_put(Object o) {             // See key point P
    System.out.println (“This is put”);
    super._d_put(o);
  }
  // no overriding of put                       // See key point Q
}
public class BB2 extends BB1 {
  public BB2(int capacity) { super(capacity); } // See key point N
  protected boolead _d_isEmpty() {
    System.out.println (“This is isEmpty”);
    return super.isEmpty();
  }
  public boolean isEmpty() {                    // See key point O
    return this._d_isEmpty();
  }
  protected Object _d_take() {                  // See key point P
    System.out.println (“This is take”);
    return super._d_take();
  }
  // no overriding of take                      // See key point Q
}
```

Key points:

N. Since these classes are not directly associated with a COOL coordinator, no coordinator variable is declared in them. Instead, they inherit the coordinator variable declared in BB.

O. For the new method `isEmpty`, the wrapper simply calls the implementation method without any corrdination.

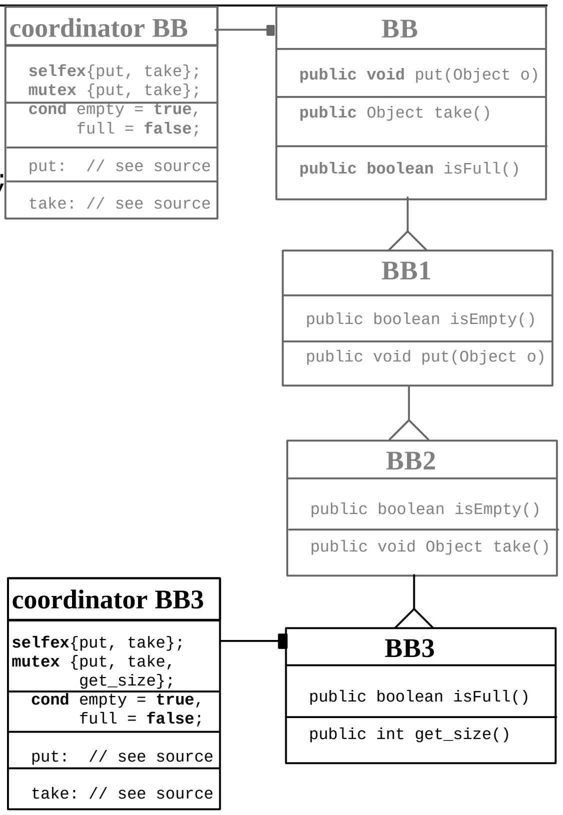
P. The semantics of COOL states that the overriding of `put` in BB1 and the overriding of `take` in BB2 are still affected by the coordination scheme of the superclass's corresponding methods. Since these two methods are originally implemented for BB, which has a coordinator, only the "implementation" methods `_d_put` and `_d_take` need to be overriden.

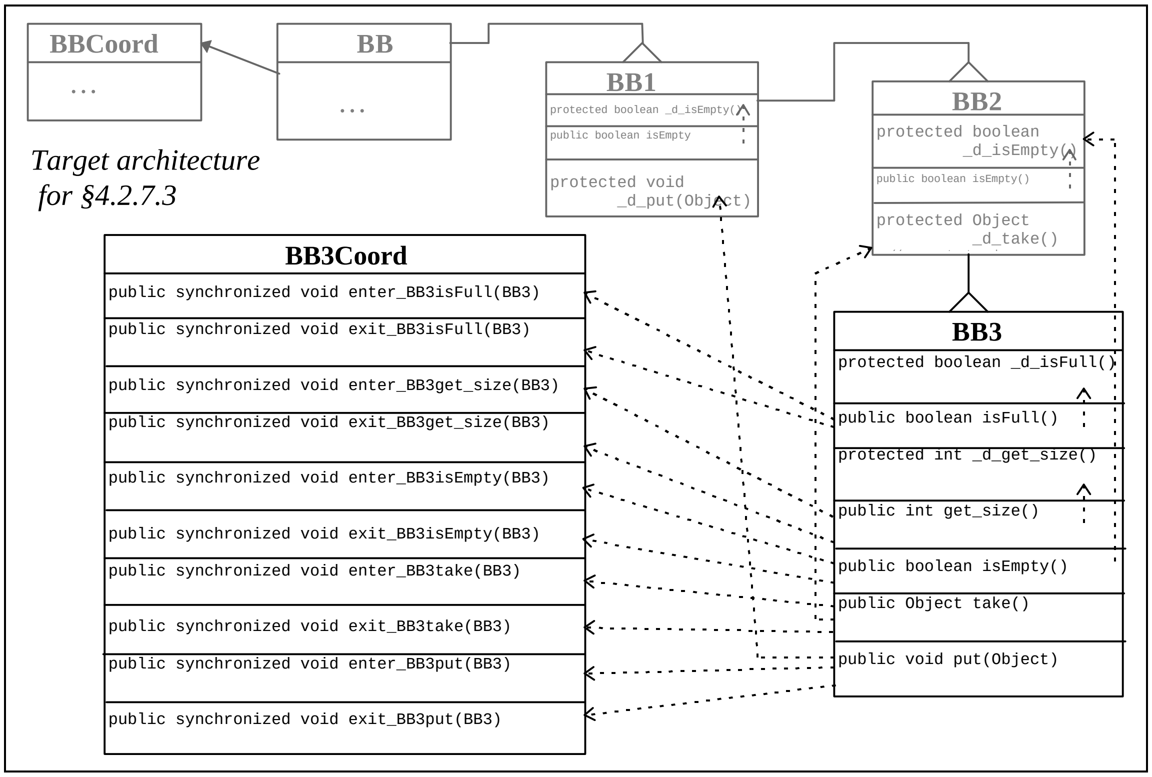
Q. The coordination for methods `put` and `take` in BB2 is done through the inherited methods `put` and `take`, which, for instances of BB2, end up calling the appropriate "implementation" methods `_d_put` and `_d_take` that were redefined in BB2. This ensures that the coordination in the subclasses is the same as defined for the methods of the superclass.

4.2.7.3 Overriding of Coordination

Consider a third level of inheritance, but this time with a new coordinator:

```
public class BB3 extends BB2 {
  public BB3(int capacity) {
    super(capacity);
  }
  // override isFull
  public boolean isFull() {
    System.out.println ("This is isFull");
    return super.isFull();
  }
  // new method
  public int get_size() { return size; }
}
coordinator BB3 {
  selfex put, take;
  mutex {put, take, get_size};
  cond empty = true,
       full = false;
  put:  requires !full;
        on_exit {
          empty = false;
          if (usedSlots == size)
            full = true;
        }
  take: requires !empty;
        on_exit {
          full = false;
          if (usedSlots == 0)
            empty = true;
        }
}
```




BBCoord
...
BB
...
BB1
protected boolean _d_isEmpty()
public boolean isEmpty
protected void _d_put(Object)
BB2
protected boolean _d_isEmpty()
public boolean isEmpty()
protected Object _d_take()
Target architecture for §4.2.7.3
BB3Coord
public synchronized void enter_BB3isFull(BB3)
public synchronized void exit_BB3isFull(BB3)
public synchronized void enter_BB3get_size(BB3)
public synchronized void exit_BB3get_size(BB3)
public synchronized void enter_BB3isEmpty(BB3)
public synchronized void exit_BB3isEmpty(BB3)
public synchronized void enter_BB3take(BB3)
public synchronized void exit_BB3take(BB3)
public synchronized void enter_BB3put(BB3)
public synchronized void exit_BB3put(BB3)
BB3
protected boolean _d_isFull()
public boolean isFull()
protected int _d_get_size()
public int get_size()
public boolean isEmpty()
public Object take()
public void put(Object)

From this class and this coordinator, and according to the given class hierarchy, the output code is as follows:

```
public class BB3 extends BB2 {
  protected BB3Coord _BB3Coord;                         // See key point R
  public BB3(int capacity) {
    super();
    _BB3Coord = BB3Coord.createCoord();                 // See key point R
  }
  public void put(Object o) {                           // See key point S
    _BB3Coord.enter_BB3put(this);                       // See key point T
    try {
      this._d_put(o);                                   // See key point U
    } finally {
      _BB3Coord.exit_BB3put(this);                      // See key point T
    }
  }
  public Object take() {                                // See key point S
    _BB3Coord.enter_BB3take(this);                      // See key point T
    try {
      return this._d_take();                            // See key point U
    } finally {
      _BB3Coord.exit_BB3take(this);                     // See key point T
    }
  }
  public boolean isEmpty() {
    _BB3Coord.enter_BB3isEmpty(this);                  // See key point T
    try {
      return this._d_isEmpty();                        // See key point U
    } finally {
      _BB3Coord.exit_BB3isEmpty(this);                 // See key point T
    }
  }
  protected boolean _d_isFull() {
    System.out.println ("This is isFull");
    return super._d_isFull();
  }
  public boolean isFull() {
    _BB3Coord.enter_BB3isFull(this);                   // See key point T
    try {
      return this._d_isFull();
    } finally {
      _BB3Coord.exit_BB3isFull(this);                  // See key point T
    }
  }
  protected int _d_get_size() { return size; }         // See key point V
  public int get_size() {                              // See key point V
    _BB3Coord.enter_BB3get_size(this);                 // See key point T
    try {
      return this._d_get_size();
    } finally {
      _BB3Coord.exit_BB3get_size(this);                // See key point T
    }
  }
}

// From COOL's coordinator
public class BB3Coord {                                // See key point W
  MethState BB3put = new MethState();
  MethState BB3take = new MethState();                 // See key point X
  MethState BB3isEmpty = new MethState();
  MethState BB3isFull = new MethState();
  MethState BB3get_size = new MethState();

  // condition variables
  boolean empty = true;
  boolean full = false;

  public static BB3Coord createCoord() {
    return new BB3Coord();
  }
  public synchronized void enter_BB3put(BB jcoreobj) {
    while (// conditions for waiting
           BB3put.isBusyByOtherThread()   /* from selfex */ ||
           BB3take.isBusyByOtherThread() /* from mutex */ ||
           BB3get_size.isBusyByOtherThread() /* from mutex  */ ||
           !(!full)      /* from requires */ ) {
      try { wait(); } catch (InterruptedException e) {};
    }
    BB3put.in();
  }
  public synchronized void exit_BB3put(BB3 jcoreobj) {
    // Exactly the same as exit_BBput. (except for the name: BB3)
  }
  public synchronized void enter_BB3take(BB3 jcoreobj) {
    while (// conditions for waiting
           BB3take.isBusyByOtherThread()  /* from selfex */ ||
           BB3put.isBusyByOtherThread()  /* from mutex */ ||
           BB3get_size.isBusyByOtherThread() /* from mutex  */ ||
           !(!empty)     /* from requires */ ) {
      try { wait(); } catch (InterruptedException e) {};
    }
    BB3take.in();
  }
  public synchronized void exit_BB3take(BB3 jcoreobj) {
    // Exactly he same as exit_BBtake. (except for the name: BB3)
  }
  public synchronized void enter_BB3get_size(BB3 jcoreobj) {
    while (// conditions for waiting
           BB3take.isBusyByOtherThread() /* from mutex */ ||
           BB3put.isBusyByOtherThread()  /* from mutex  */ ) {
      try { wait(); } catch (InterruptedException e) {};
    }
    BB3get_size.in();
  }
  public synchronized void exit_BB3get_size(BB3 jcoreobj) {
    BB3get_size.out();
    if (BB3get_size.depth == 0) notifyAll();
  }
  public synchronized void enter_BB3isEmpty(BB3 jcoreobj) { }
  public synchronized void exit_BB3isEmpty(BB3 jcoreobj) { }
  public synchronized void enter_BB3isFull(BB3 jcoreobj) { }
  public synchronized void exit_BB3isFull(BB3 jcoreobj) { }
}
```

Key points:

R. Since BB3 is now associated with its own COOL coordinator, this class is extended with a new variable that will hold the coordinator object. The type of this variable is the class type that is generated from the COOL coordinator. The initialization of this variable is done at the end of every constructor of BB3.

S. BB3 inherits the implementation of `put` and `take`, but redefines their coordination. In order to correctly handle these cases, the weaver must override `put` and `take` in BB3, so that the calls to the inherited implementation are wrapped around calls for the new coordination. (See §4.2.6)

T. The coordination for instances of BB3 is done by calls to their `_BB3Coord` variables. The inherited `_BBCoord` variable is completely ignored.

U. As mentioned before, the situation here is that BB3 inherits the implementation of `put`, `take`, and `isEmpty`, but redefines their coordination. The redefinition of the coordination is implemented by overriding the methods, and calling the new coordination object; the inheritance of method implementation is achieved by calling the "implementation" methods of the superclass.

V. BB3 implements the new method `get_size`. Therefore, this method also follows the usual renaming/duplication scheme.

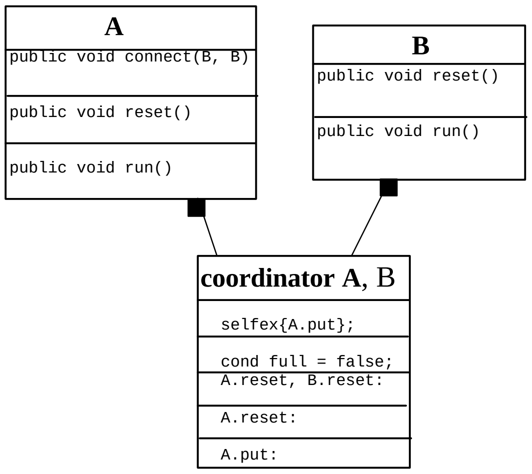
W. The new COOL coordinator for BB3 is translated following the same scheme as the coordinator for BB (see example 1). There is no inheritance relation between classes BB3Coord and BBCoord, because, in general, that would require multiple inheritance for handling the redefinition of multi-class coordination, and Java does not provide that.

X. The methods being monitored in BB3Coord are all the methods inherited, implemented or re-implemented in class BB3.

4.2.7.4 Multi-class Coordination (per_class)

Consider the following classes and their coordinator:

```
public class A implements Runnable {
  B firstB, secondB;
  int a[] = new int[10];
  int index = 0;
  public void connect(B first, B second) {
    firstB = first; secondB = second;
  }
  public void put(int n) { a[index++] = n; }
  private void reset() { index = 0; }
  public void run() {
    while (true) {
      firstB.reset(); secondB.reset();
      reset();
    }
  }
}
public class B implements Runnable {
  A theA;
  int myn, counter;
  public B(A a, int n) {
    theA = a; counter = myn = n;
  }
  public void reset() {
    counter = myn; }
  public void run() {
    while (true) {
     theA.put(counter++);
    }
  }
}

coordinator A, B {
  selfex A.put;
  cond full = false;
  A.reset, B.reset: requires (full);
  A.reset: on_exit { full = false; }
  A.put: requires (!full);
    on_exit { if (index == 10) full = true; }
}
```



The target output code is as follows:

```
public class A implements Runnable {
  ABCoord _ABCoord;
  B firstB, secondB;
  int a[] = new int[10];
  int index = 0;
  public A() {
    super();
    _ABCoord = ABCoord.createCoord();                    // See key point Z
  }
  // … the usual renaming/duplication scheme for all methods of A
  // and the usual accessor methods
}

public class B implements Runnable {
  ABCoord _ABCoord;
  A theA;
  int myn, counter;
  public B(A a, int n) {
    theA = a; counter = myn = n;
    _ABCoord = ABCoord.createCoord();                    // See key point Z
  }
  // … the usual renaming/duplication scheme for all methods of B
  // and the usual accessor methods
}

public class ABCoord {
  static boolean one = false;                            // See key point Z
  static ABCoord theABCoord;
  MethState Aconnect = new MethState();
  MethState Aput = new MethState();
  MethState Areset = new MethState();
  MethState Arun = new MethState();
  MethState Breset = new MethState();
  MethState Brun = new MethState();
  boolean full = false;

  public static synchronized ABCoord createCoord() {
    if (!one) {                                         // See key point Z
      theABCoord = new ABCoord();
      one = true;
    }
    return theABCoord;
  }
  public synchronized void enter_Aput(A jcoreobj) {
    while (// conditions for waiting
           Aput.isBusyByOtherThread()  /* from selfex */ ||
           !(!full)      /* from requires */ ) {
      try { wait(); } catch (InterruptedException e) {};
    }
    Aput.in();
  }
  public synchronized void exit_Aput(A jcoreobj) {
    Aput.out();
    if (jcoreobj._dget_index() == 10) full = true;
    if (Aput.depth == 0) notifyAll();
  }

  // … similar for all other methods of A
  public synchronized void enter_Breset(B jcoreobj) {
    while (// conditions for waiting
           !full      /* from requires */ ) {
      try { wait(); } catch (InterruptedException e) {};
    }
    Breset.in();
  }
  public synchronized void exit_Breset(B jcoreobj) {
    Breset.out();
    if (Breset.depth == 0) notifyAll();
  }
  // similar for all other methods of B
}
```

Key point:

Z. This is a multi-class coordination case. It is implemented by having all instances of the involved classes share the same coordinator object. For that reason, the `createCoord` method is slightly more sophisticated than the previous cases, working over a static variable that is set at most once and that holds the single coordinator object. This is the only difference between per_object and per_class coordination, and everything else of the translation and the weaving is the same.

4.3. Target Architectures for Implementing RIDL

The implementation of RIDL is considerably more complex than the implementation of COOL. Besides having a more complicated run-time, the details of RIDL's implementation, which are essential to make it work, can only be fully understood by those who have a relatively deep knowledge of Java RMI. This explanation focuses on the architectural issues of the implementation, while disclosing some of the important details without which the architecture does not work.

In using Java RMI, one might be tempted to directly translate RIDL's portals into Java's `Remote` interfaces, and make the corresponding JCore class implement that interface. That, however, doesn't implement the semantics of RIDL, for one fundamental reason: in RIDL, any object can be passed by global reference or by copy, and that is defined by the programmer on a per remote operation basis; in Java, the parameter passing mode is statically associated with the objects on a type basis. That is, in Java, when a class implements the `Remote` interface, its instances are always passed by global reference, and when a class implements the `Serializable` interface, its instances are always copied. A direct translation from RIDL's portals into Java's `Remote` interfaces wouldn't handle, for example, the following case:

```
portal ANode {
  ANode getLeft(ANode other) {
    other: gref;
  }
  int add(ANode other) {
    other: copy;
  }
}
```

The basic idea is, then, to define a protocol — the RIDL protocol — that uses Java RMI as a lower level protocol. The RIDL protocol is responsible for implementing the customized data transfers that are specified in the portals. The next subsections describe the most important points in the RIDL protocol: (1) an overview of the run-time; (2) interaction with the name server; and (3) data transfer protocols.

4.3.1. Run-time Architecture

Figure 35 shows the resulting run-time architecture for when an object in execution space 2, `aObj`, is referenced by some other execution space 1 (the client space). The implementation of the virtual reference spawns over three layers of protocol: the application, the RIDL layer and the RMI layer. Space 1 (the client), at the application level, contains a proxy for type-conformance, that, in turn, contains a reference to a RIDL proxy, `objPP`, that interacts with Java RMI. Space 2 (the

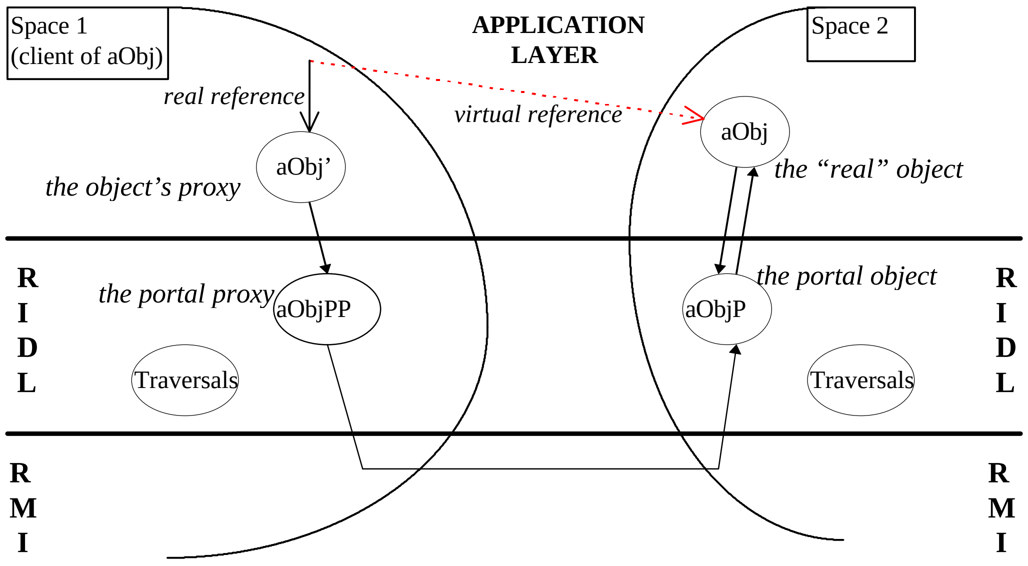


Figure 35. Run-time architecture for D's remote objects.

provider), at the application level, contains the “real” object, `aObj`, which, in turn contains a reference to an object at the RIDL layer, `aObjP`, that represents `aObj`’s portal. All incoming calls to `aObj` are, in fact, calls to `aObjP`, who directs them to `aObj`. The purpose of P and PP objects is to implement RIDL’s protocols.

The architecture shown in Figure 35 follows some well known design patterns that have been identified in the literature, namely the Proxy and the Adapter patterns [21]. The following subsections describe each of the pieces in detail.

4.3.1.1 Application-level Proxies for D’s Remote Objects

Proxies are local handles for remote objects. They respond to, at least, the same set of operations, and they are responsible for, at least, redirecting the calls to the remote object. One of the most important constraints that proxies must observe is that proxy classes must be of the same type as the application classes they represent, so that client programs are unaware of proxies but still type check correctly.[4] There are a number of ways of implementing proxies in accordance to this constraint. This implementation of RIDL takes the simplest approach: proxy objects are of the same class as the objects they represent. In the implementation space, the class of a D remote object provides two kinds of behaviors: one for the objects as defined by the JCore class, and one for proxies. An instance of such class is either a JCore object or a proxy, but not both at the same time; the behavior is defined at instantiation time and doesn’t change. Consider, for example, the following class and its associated portal:

`class A {`	`portal A {`
`  void f() { … }`	`  void  f();`
`  int g() {…}`	`  int g();`
`  double h(int i) { … }`	`}`
`}`	

Disregarding a number of details, the output woven class is as follows:

```
class A {
  APP _pp = null; // by default, these are real objects
  A(APP proxy) { _pp = proxy; } // constructor called by the run-time,
                                // if this is a proxy
  protected void _d_f() { original implementation of f }
  void f() {
    if (_pp != null) // this is a proxy object; redirect the call
      _pp.f();
    else             // this is a real object; execute the method here
      _d_f();
  }
  // similar for g and h
}
```

[4] If the proxy classes and the classes they represent are not type-conforming, the purpose of the proxy mechanism is defeated, because at *compile-time*, client programs must decide whether to reference proxies to remote objects or local objects.

The pair of methods was explained in §4.2.6. The dual behavior of classes associated with portals can be seen in this code: when _pp is null, A objects behave as defined by the programmer; when _pp holds a reference to a portal proxy, A objects behave as application-level proxies for type-conformance that simply redirect the invocation to the lower level of the protocol.

4.3.1.2 Portal Objects and their Proxies

Each instance of a class that is associated with a RIDL's portal (`aObj` in Figure 35, and hereafter called the "real" object) is associated, in the implementation space, with an object called the portal object (P). The purpose of a P object is two-fold: this is the Java's RMI `Remote` object that is passed around as a global reference instead of the "real" object, and it serves as the filter (i.e. the connector) to the real object, translating the parameters and return value into the types expected by DJ library. Therefore, the implementation of the portal object is a skeleton of the operations of the real object which simply redirects the incoming remote calls to the real object. Ps exist in the same execution space of their "real" objects.

Ps have remote counterparts called portal proxies (PPs). The purpose of these proxies is to detect illegal remote calls from clients (i.e. calls to methods that are not remote operations), as well as to translate the parameters and return value into the types expected by DJ library.

Considering again the example of the previous page, and disregarding some details, the pair of classes P and PP is as follows:

```
// the class of the portal proxy
class APP {
  APRI rself; // reference to the
              // remote portal object
  APP(APRI o) { rself = o; }

  void f() {
    rself.f(); // redirect
  }

  int g() {
    return rself.f(); // redirect
  }

  double h(int i) {
    // this is not a remote operation!
    throw new DInvalidException();
  }
}
```

```
// the portal class
class AP implements APRI {
  A myself; //reference to the real
            //object in the same space
  AP(A o) { myself = o; }

  void f() {
    myself.f();
  }

  int g() {
    return myself.g();
  }
}

// the portal interface
interface APRI extends Remote {
  void f() throws RemoteException;
  int g() throws RemoteException;
}
```

This is a simple example. The only sign of the RIDL protocol is in the portal proxy class APP, which throws an exception when a client space tries to make a remote invocation to method h. According to the portal, h is not a remote operation.

4.3.1.3 *Traversals and Traversal Classes*

Consider, for example, the following portal:

```
portal Library {
  BookCopy getBook(User u, String title) {
    return: copy {BookCopy bypass borrower; Book bypass copies;}
    u: copy {User bypass books;}
  }
  Book findBook(String title) {
    return: copy {Book bypass copies, ps;}
  }
}
```

In order to copy the parameter and return objects according to the copying directives, the DJ run-time relies on the existence of run-time representations of the traversal directives. The traversal directives declared in a portal are grouped in a class called "*ClassName*Traversals," where *ClassName* is the name of the class the portal is associated with. Traversal directives are represented at run-time by Traversal objects (see Appendix D). Traversal objects consist simply of a number of class names and associated "missing parts."

In the example above, the output class LibraryTraversals contains three traversal objects, each one representing a traversal directive of the portal. The first traversal object corresponds to the first copying directive in `getBook`; it associates the class name "BookCopy" with the field "borrower," and the class name "Book" with the field "copies," meaning that this directive excludes the `borrower` field of class `User`, and the `copies` fields of class `Book`. The second traversal object corresponds to the second copying directive in `getBook`; it associates the class name "User" with the field "books," meaning that this directive excludes the `books` field of class `User`. Etc. The code is as follows:

```
class LibraryTraversals {
  public static Traversal t1, t2, t3;
  static boolean once = false;
  public static synchronized void init() {
    IncompleteClass c;
    if (once) return; // initialization should be done only once

    t1 = new Traversal("t1", "LibraryTraversals");
    c = new IncompleteClass("BookCopy");
    c.bypass("borrower");
    t1.incompleteClass(c);
    c = new IncompleteClass("Book");
    c.bypass("copies");
```

```
        c.bypass("ps");
        t1.incompleteClass(c);

        t2 = new Traversal("t2", "LibraryTraversals");
        c = new IncompleteClass("User");
        c.bypass("books");
        t2.incompleteClass(c);

        t3 = new Traversal("t3", "LibraryTraversals");
        c = new IncompleteClass("Book");
        c.bypass("copies");
        c.bypass("ps");
        t3.incompleteClass(c);

        once = true;
      }
    }
```

4.3.2. The Name Service

The bootstrap for the proliferation of remote references is the Name Server. The Name Server is a remote object whose reference is globally known to all execution spaces. The name server in DJ is the one provided by Java RMI, but with a special DJ-specific portal to it, whose purpose is to bridge between Java's Remote objects and D's remote objects using the portal objects described before. The interface to DJ's name service is given below; the implementation of this interface is given in Appendix D.

```
  portal DJNaming {
    // Associate the given URL with the given DJ object
    void bind(String url, Object obj) {
      obj: gref;
    };

    // Lookup a DJ remote object that is associated with the given name
    Object lookup(String url) {
      return: gref;
    };
  }
```

4.3.3. RIDL's Data Transfer Protocols

4.3.3.1 Passing Primitive Data

The protocol for passing primitive data (integers, doubles, etc. – Strings are also considered primitive) is to use the RMI passing modes directly. The example in page 136 illustrates this part.

4.3.3.2 Passing Global References

In this implementation architecture, all global references that are passed between execution spaces, including with the Name Server, are, in fact Java's global references corresponding to portal objects. For example, when a JCore object exports its name to the Name Server:

```
public class ANode {
  public void exportName(String name) {
    DJNaming.bind(“rmi://globin.parc.xerox.com” + name, this);
  }
}
```

The DJ's interface to the Name Server exports, in fact, the portal object associated with `this` object. From the implementation of DJNaming.bind, in Appendix D:[5]

```
  public static void bind(String name, DObject obj) /* Exceptions omitted */ {
    Remote remoteObj = null;
    remoteObj = (Remote)(obj.getClass().getField("_p").get(obj));
    // exception catching omitted
    Naming.bind(name, remoteObj);
  }
```

On the client side, when a global reference is imported, there is the instantiation of proxies. For example, when a client class looks up a name in the Name Server:

```
// in some JCore client class
ANode n = DJNaming.looukup(“rmi://globin.parc.xerox.com”);
```

The DJ's interface to the Name Server imports the portal object's global reference, and instantiates the proxies. From the implementation of DJNaming.lookup, in Appendix D:

```
  public static Object lookup(String name) /* Exceptions omitted */ {
    Remote remoteObject = Naming.lookup(name);
    String className = getTheClassName(remoteObject);
    DObject theObject = null;
    Class theClass = (Class.forName(className));
    Class ppClass = (Class.forName(className + "PP"));
    theObject = (DObject)theClass.newInstance();
    Object ppObject = ppClass.newInstance();
    (theClass.getField("_pp")).set(theObject, ppObject);
    (ppClass.getField("rself")).set(ppObject, remoteObject);
    // exception catching omitted
  }
```

The imported global reference is stored in the portal proxy's variable called `rself` (remote self), and the reference to the portal proxy itself is stored in the D remote object proxy's variable called _pp. A slightly different version of this implementation pattern is used also whenever a JCore object is passed by reference in remote calls. The example in §4.3.3.2 illustrates this part of the protocol.

[5] Most “get” methods shown here are part of Java's reflection API.

4.3.3.3 Passing Copies

The solution devised to pass possibly incomplete copies of argument and return objects is to wrap those objects into special objects that know how to perform packing/unpacking traversals. All arguments of non-primitive types that are completely or partially copied in remote calls are represented at run-time by instances of the DJ library class DArgument. A DArgument associates a parameter or return object with a particular traversal object (that may be null, if the copy is to be deep copy). Traversal objects were presented in §4.3.1.3. For example, the following remote operation

```
portal Library {
  BookCopy getBook(User u, String title) {
    u: copy {User bypass books;}
    return: copy {BookCopy bypass borrower; Book bypass copies;}
  };
}
```

results, at run-time, in two DArgument objects, one for the User parameter and the other for the return object; those DArgument objects associate the respective D object with a Traversal object that represents the corresponding copying directive. The following pieces of code illustrate this part of the protocol:

```
// The portal proxy class (client side)
class LibraryPP {
  // everything else of this class omitted

  BookCopy getBook(User u,String title) {
    DArgument a;
    a = new DArgument(u, LibraryTraversals.t2);
    return (BookCopy)(rself.getBook(a, title).obj);
  }
}
```

```
// The portal class (provider side)
class LibraryP implements LibraryPRI {
 // everything else of this class
 // omitted

  DArgument getBook(DArgument u, String title) {
    BookCopy ret;
    ret = myself.getBook((User)(u.obj), title);
    return new DArgument (ret, LibraryTraversals.t1);
  }
```

The marshaling itself is done recursively by methods in each of the classes of the arguments and return values. These methods are generated by the weaver. The library class DArgument connects up to those marshaling methods (see class DArgument in Appendix D).

For example, given a class User:

```
class User {
  private String name;
  private BookCopy books[];
  private int index;
  // … methods …
}
```

Two special methods, one for packing and the other for unpacking, are needed in order to pass partial copies of user objects. The method for packing is, in pseudo-code,

```
void _d_write(ObjectOutput out, Traversal t) {
   for each of the variables of class User (name, books and index),
     if the variable name is a “missing part” in the given traversal,
         skip it;
     if the variable name is not a “missing part” in the given traversal,
         pack it into the ObjectOutput out;
         (Special attention must be given to arrays, null objects and
          non-D objects)
  }
```

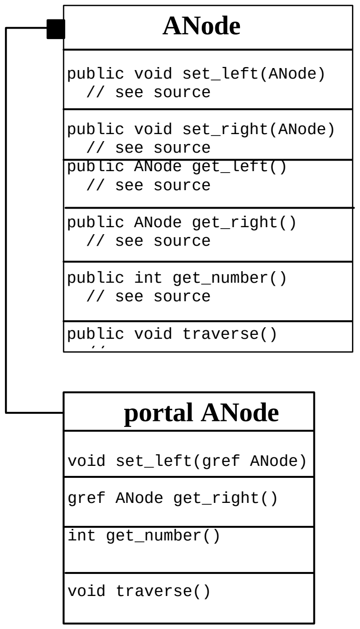
The method for unpacking is similar, but it reconstructs objects from the data in the `ObjectInput`. The example presented in §4.3.4.2 shows in much greater detail the protocol for passing incomplete copies.

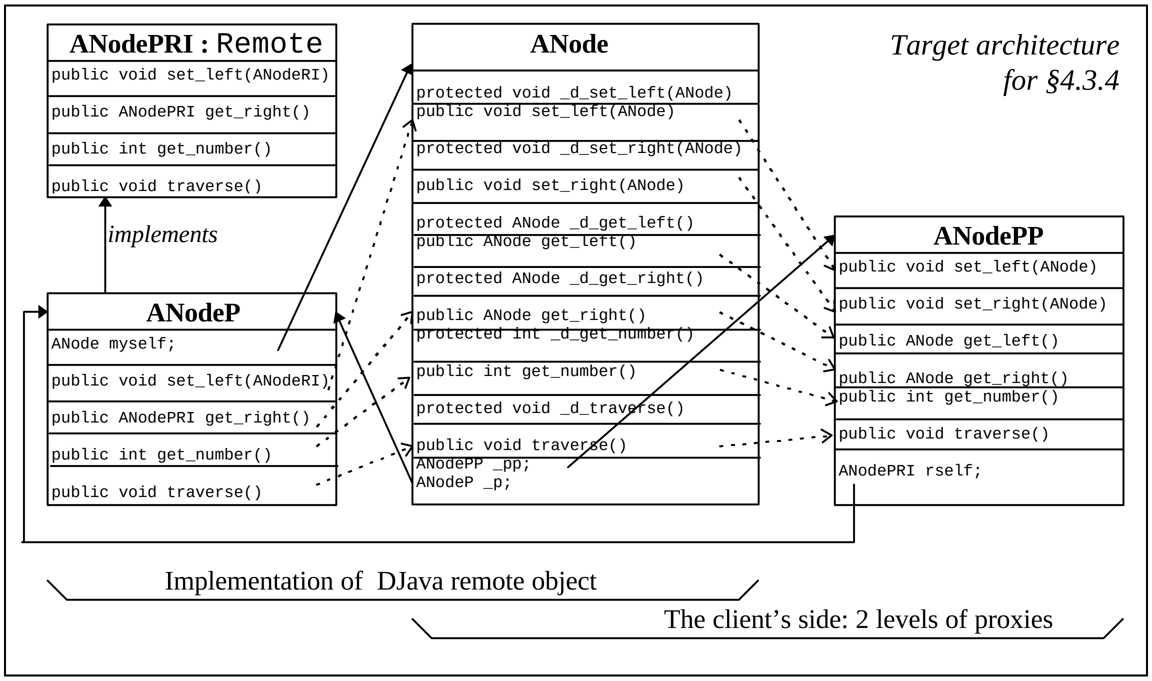
4.3.4. Examples

4.3.4.1 Arguments of Primitive Types and gref

Consider the following JCore class and its portal:

```
public class ANode {
  protected int mynumber;
  protected ANode left, right;
  public ANode(int n, ANode l, ANode r) {
    mynumber = n;
    left = l; right = r;
  }
  public void set_left(ANode l) { l = left; }
  public void set_right(Anode r) { r = right; }
  public ANode get_left() { return left; }
  public ANode get_right() { return right; }
  public int get_number() { return mynumber; }
  public traverse() {
    System.out.println(“Node ” + mynumber);
    left.traverse();
    right.traverse();
  }
}

portal ANode {
  void set_left(Anode l)  { l: gref; };
  ANode ANode get_right()  { return: gref; };
  int  get_number();
  void traverse();
}
```




ANodePRI : Remote
public void set_left(ANodeRI)
public ANodePRI get_right()
public int get_number()
public void traverse()
implements
ANodeP
ANode myself;
public void set_left(ANodeRI)
public ANodePRI get_right()
public int get_number()
public void traverse()
ANode
protected void _d_set_left(ANode)
public void set_left(ANode)
protected void _d_set_right(ANode)
public void set_right(ANode)
protected ANode _d_get_left()
public ANode get_left()
protected ANode _d_get_right()
public ANode get_right()
protected int _d_get_number()
public int get_number()
protected void _d_traverse()
public void traverse()
ANodePP _pp;
ANodeP _p;
Target architecture for §4.3.4
ANodePP
public void set_left(ANode)
public void set_right(ANode)
public ANode get_left()
public ANode get_right()
public int get_number()
public void traverse()
ANodePRI rself;
Implementation of DJava remote object
The client's side: 2 levels of proxies

From this source code, the following classes and interface are generated:

```
interface ANodePRI extends Remote {                    // See key point A
  int  get_number() throws RemoteException;            // See key point B
  void traverse() throws RemoteException;
  void set_left(ANodePRI l) throws RemoteException;    // See key point K
  void ANodePRI get_right(AnodePRI r) throws RemoteException;
}
public class ANodeP implements ANodePRI {              // See key point A
  ANode myself;                                        // See key point C
  RemoteStub mystub;
  public ANodeP(ANode s) {                             // See key point C
    myself = s;
    try {mystub = UnicastRemoteObject.exportObject(this);}
    catch (RemoteException e) {
      System.out.println (e.toString());
    }
  }
  public int get_number() throws RemoteException {     // See key point D
    return myself.get_number();
  }
  public void traverse() throws RemoteException {      // See key point D
    myself.traverse();
  }
  public void set_left (ANodePRI l) throws RemoteException {
    myself.set_left(new ANode(new ANodePP(l)));       // See key point L
  }
  public ANodePRI get_right (ANodePRI r) throws RemoteException {
    return myself.set_right(new ANode(new ANodePP(r)))._p;
                                                        //See key point L
  }
}
```

```
public class ANodePP {                                  // See key point A
  ANodePRI rself;
  public ANodePP(ANodePRI s) { rself = s; }             // See key point E

  public int get_number() throws DInvalidRemoteOperation {
    try {return rself.get_number();}                    // See key point G
    catch (RemoteException e) {
      System.err.println("Remote exception in get_number");
      return 0;
    }
  }
  public void traverse() throws DInvalidRemoteOperation {
    try {rself.traverse();}                             // See key point G
    catch (RemoteException e) {
      System.err.println("Remote exception in traverse");
    }
  }
  public void set_left(ANode a1) throws DInvalidRemoteOperation {
    try {rself.set_left(a1._p);}                       // See key point M
    catch (RemoteException e) {
      System.err.println("Remote exception in set_left");
    }
  }
  public void set_right(ANode a1) throws DInvalidRemoteOperation {
    throw new DInvalidRemoteOperation();                // See key point F
  }
  public ANode get_left() throws DInvalidRemoteOperation {
    throw new DInvalidRemoteOperation();                // See key point F
  }
  public ANode get_right() throws DInvalidRemoteOperation {
    try {new ANode (ANodePP(rself.set_right()));}      // See key point M
    catch (RemoteException e) {
      System.err.println("Remote exception in set_right");
    }
  }
}

public class ANode implements DObject {                 // See key point H
  protected int mynumber;
  protected ANode left, right;
  public ANodeP _p = null;                              // See key point H
  public ANodePP  _pp = null;                           // See key point H
  // the null-ary constructor must exist, because of a
  // Java's serialization API undocumented requirement
  public ANode(){}
  public ANode(int n, ANode l, ANode r) {
    mynumber = n;
    left = l; right = r;
    _p = new ANodeP(this);                              // See key point H
  }
  public ANode(ANodePP r) { _pp = r; }                  // See key point H

  protected void _d_set_left(ANode l) { l = left; }     // See key point I
  public void set_left(ANode l) {                       // See key point I
    if (_pp != null) {                                  // See key point J
      try {_pp.set_left(l);}
      catch (DInvalidRemoteOperation e) {
        System.err.println("Invalid Remote operation set_left");
      }
    }
    else _d_set_left(l);
  }
  protected void _d_set_right(ANode r) { r = right; }   // See key point I
  public void set_right(ANode r) {                      // See key point I
    if (_pp != null) {                                  // See key point J
      try {_pp.set_right(r);}
      catch (DInvalidRemoteOperation e) {
        System.err.println("Invalid Remote operation set_right");
      }
    }
    else _d_set_right(r);
  }
  protected ANode _d_get_left() { return left; }        // See key point I
  public ANode get_left() {                             // See key point I
    if (_pp != null) {                                  // See key point J
      try {return _pp.get_left();}
      catch (DInvalidRemoteOperation e) {
        System.err.println("Invalid Remote operation get_left");
        return null;
      }
    }
    else return _d_get_left();
  }
  protected ANode _d_get_right() { return right; }      // See key point I
  public ANode get_right() {                            // See key point I
    if (_pp != null) {                                  // See key point J
      try {return _pp.get_right();}
      catch (DInvalidRemoteOperation e) {
        System.err.println("Invalid Remote operation get_right");
        return null;
      }
    }
    else return _d_get_right();
  }
  protected int _d_get_number() {  return mynumber; }   // See key point I
  public int get_number() {                             // See key point I
    if (_pp != null) {                                  // See key point J
      try {return _pp.get_number();}
      catch (DInvalidRemoteOperation e) {
        System.err.println("Invalid Remote operation get_number");
        return 0;
      }
    }
    else return _d_get_number();
  }
  protected void _d_traverse() {                        // See key point I
    System.out.println("Node " + mynumber);
    if (left != null) left.traverse();
    if (right != null) right.traverse();
  }
  public void traverse() {                              // See key point I
    if (_pp != null) {                                  // See key point J
      try {_pp.traverse();}
      catch (DInvalidRemoteOperation e) {
        System.err.println("Invalid Remote operation traverse");
      }
    }
    else _d_traverse();
  }
  public void writeExternal(ObjectOutput out) {
    // this method will be explained in the next example; in this example,
    // it is never called, because ANode objects are never passed by copy
  }
  public void _d_writeExternal(ObjectOutput out, Traversal t) {
    // this method will be explained in the next example; in this example,
    // it is never called, because ANode objects are never passed by copy
  }
  public void readExternal(ObjectInput in) {
    // this method will be explained in the next example; in this example,
    // it is never called, because ANode objects are never passed by copy
  }
  public void _d_readExternal(ObjectInput in, Traversal t) {
    // this method will be explained in example 4; in this example,
    // it is never called, because ANode objects are never passed by copy
  }
}
```

Key points:

A. From the RIDL's portal the translator generates two classes and one Java interface (See Figure 35). The class having the suffix "P" is used to instantiate interface objects, Ps, which are always associated with JCore's ANode objects; the class having the suffix "PP" is used to instantiate, on the callers side, the portal proxies that are associated with Ps; the interface having the suffix "PRI" is a Java requirement for exporting references outside execution spaces, and this interface is implemented by the P class (but not by the PP class, since PPs are proxies that are never passed to other execution spaces).

B. The Java interface declares only the remote operations declared in the RIDL portal.

C. P classes define the behavior of JCore objects when they are invoked from other execution spaces. Each P knows about its "real" JCore object, by holding its reference in the variable `myself`, which is initialized at instantiation time. Also at instantiation time, P references are exported to the RMI run-time (RMI requirement for passing global references).

D. The only operations implemented by P classes are the ones defined in the PRI interface (see B), and, in this case, these methods simply redirect the call to the "real" object. As it will be shown in later examples, these methods do something else when there are arguments of non-primitive types.

E. As for the PP class, it is used to instantiate P proxies on the callers side. It contains only one variable, `rself` (remote self), which holds a Java's remote reference to a PP. This variable is initialized when the PP is created.

F. Although ANode and ANodePP don't share any Java interface, ANodePP class implements exactly the same methods as ANode, with the same signatures; PPs are local representatives of remote objects, and at weave time it is not possible to determine if a reference to an ANode object holds a local or a remote object. Therefore, one of the responsibilities of PPs is to filter out remote method calls that are not declared remote operations in the RIDL interface, by throwing a `DInvalidRemoteOperation` exception. This implements the semantic feature of RIDL that states that, from other execution spaces, only the declared remote operations can be called.

G. For the valid remote operations, the PP redirects the call to the P. These methods do something else when there are arguments of non-primitive types.

H. The JCore class ANode is woven, and results in a Java class with the same name that implements the DObject interface (see Appendix D). There are two new variables: _p, which holds the reference to the corresponding P when the ANode object is a “real” object, and _pp, which holds the reference to a PP when the ANode object is a proxy that represents some remote ANode object. The implicit run-time invariant is that only one of these variables is not null (i.e. an instance of the resulting ANode class is either a “real” object or a proxy, but not both). Every constructor is extended with the initialization of the _p variable; that is, when the JCore application does `new ANode(…)` it always gets a “real” ANode. ANode proxy objects are instantiated by the DJ run-time through the special constructor that takes an ANodePP as argument.

I. As explained for COOL, each method of ANode is transformed in a pair of methods in the resulting ANode class: the “implementation” method with prefix “_d_”, and the wrapper method with the original name. The “implementation” method contains the original method body. This technique is used to implement the semantics of inheritance of aspect modules.

J. The wrapper method checks whether this ANode object is a “real” object or a proxy, and redirects the call accordingly. (When _pp is not null, the object is a proxy.)

K. Passing an ANode object by gref results in an operation that passes an ANodePRI object (i.e. a P, since Ps are the only classes that implement PRIs).

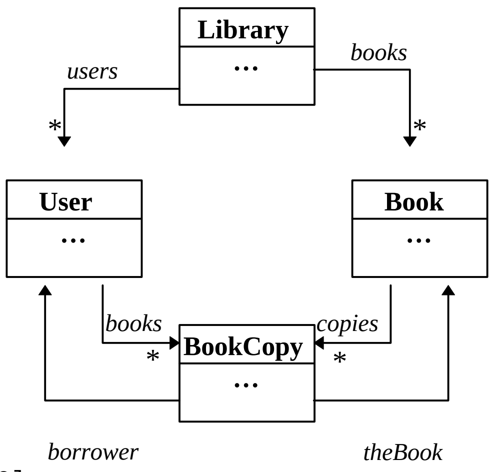
L. The implementation of the methods in the P class takes ANodePRIs as arguments and converts them to ANode objects before passing them to the “real” object. The conversion is done by instantiating an ANode proxy object, that is, an object of class ANode that behaves like a proxy (it’s _pp variable holds the PP that is instantiated here). Therefore, calls to that argument will eventually result in remote calls.

M. The implementation of the methods in the PP class takes ANode objects as arguments (exactly like the signatures in class ANode), and passes only the P object of those arguments. Java RMI takes care of passing the remote references of those objects.

4.3.4.2 Passing Copies

Consider the following classes and portal. This set of classes is part of a larger application. The focus of this piece of code is on the relations between the classes, which define how the marshaling traversals are to be done. Therefore, almost all methods were left out.

```
public class Book {
  private String          title, author, isbn;
  private PostScript      ps;
  private BookCopy[]      copies;
  private int             n_copies;
  public Book(String t, String a, String i, PostScript p) {
    title = t; author = a; isbn = i; ps = p;
    copies = new BookCopy[3];
    copies[0] = new BookCopy(0, this);
    n_copies = 1;
  }
  // all other methods omitted
}

public class BookCopy {
  private int mynumber;
  private Book theBook;
  private User borrower;
  public BookCopy(int n, Book b) {
    mynumber = n;
    theBook = b;
  }
  // all other methods omitted
}

public class User {
  private String name;
  private BookCopy books[] = new BookCopy[10];
  int index = 0;
  public User(String n) { name = n; }
  // all other methods omitted
}

public class Library {
  private Hashtable       books, users;
  public Library(int capacity) {
    books = new Hashtable(capacity);
    users = new Hashtable(100);
  }
  public BookCopy getBook(User u, String title) {
   // implementation omitted
  }
  public Book findBook(String title) {
   // implementation omitted
  }
}
```



```
portal Library {
  BookCopy getBook(User u, String title) {
    return: copy {BookCopy bypass borrower; Book bypass copies;}
    u: copy {User bypass books;}
  }
  Book findBook(String title) {
    return: copy {Book bypass copies, ps;}
  }
}
```

This code results in the classes shown below. Note that the P and PP classes, the PRI interface and the Traversals class resulting from the Library portal are shown, but the woven Library class itself is not shown (its weaving is similar to the weaving of class ANode in the previous example). The explanation of the marshaling routines is illustrated only with the class Book, because this class has the most diverse set of variables. However, all classes that pass through the RIDL Weaver (including Library, User and BookCopy) end up implementing the DObject interface, and therefore must implement the four marshaling methods, exactly in the same way as shown for the Book class.

```
interface LibraryPRI extends Remote {                    // See key point N
  DArgument getBook (DArgument u, String title) throws RemoteException;
  DArgument findBook (String title) throws RemoteException;
}

final public class LibraryTraversals {                   // See key point O
  public static Traversal t1, t2, t3;
  static boolean once = false;
  public static synchronized void init() {
    IncompleteClass c;
    if (once) return;

    t1 = new Traversal("t1", "LibraryTraversals");
    c = new IncompleteClass("BookCopy");
    c.bypass("borrower");
    t1.incompleteClass(c);
    c = new IncompleteClass("Book");
    c.bypass("copies");
    c.bypass("ps");
    t1.incompleteClass(c);

    t2 = new Traversal("t2", "LibraryTraversals");
    c = new IncompleteClass("User");
    c.bypass("books");
    t2.incompleteClass(c);

    t3 = new Traversal("t3", "LibraryTraversals");
    c = new IncompleteClass("Book");
    c.bypass("copies");
    c.bypass("ps");
    t3.incompleteClass(c);

    once = true;
  }
}
```

```
public class LibraryP implements LibraryPRI {
  Library myself;
  RemoteStub mystub;
  public LibraryP(Library s) {
    myself = s;
    try {mystub = UnicastRemoteObject.exportObject(this);}
    catch (RemoteException e) {
      System.out.println (e.toString());
    }
    LibraryTraversals.init();                          // See key point P
  }
  public DArgument getBook(DArgument u, String title)
  throws RemoteException {                             // See key point Q
    return new DArgument (myself.getBook((User)(u.obj), title),
                          LibraryTraversals.t1);
  }
  public DArgument findBook (String title) throws RemoteException {
    return new DArgument (myself.findBook(title),      // See key point Q
                          LibraryTraversals.t3);
  }
}
public class LibraryPP {
  public LibraryPRI rself;
  public LibraryPP() {LibraryTraversals.init();}
  public LibraryPP(LibraryPRI s) {
    rself = s;
    LibraryTraversals.init();                          // See key point P
  }
  public BookCopy getBook(User u, String title)
  throws DInvalidRemoteOperation {
    try {                                              // See key point R
      return (BookCopy)(rself.getBook(new DArgument(u, LibraryTraversals.t2),
                                      title).obj);
    }
    catch (RemoteException e) {
      System.out.println("Remote exception in getBook");
      return null;
    }
  }
  public Book findBook(String n) throws DInvalidRemoteOperation {
    try {
      return (Book)(rself.findBook(n)).obj;            // See key point R
    }
    catch (RemoteException e) {
      System.out.println("Remote exception in findBook");
      return null;
    }
  }
}
public class Book implements DObject {
  private String          title;
  private String          author;
  private String          isbn;
  private PostScript      ps;
  private BookCopy[]     copies;

  public Book() {}
  public Book(String t, String a, String i, PostScript p) {
    // exactly the same constructor code as in the source Book class
  }
  // all other methods omitted
  // Marshaling routines                              // See key point S
  public void writeExternal(ObjectOutput out) {       // See key point T
    try {
      out.writeObject(title);
      out.writeObject(author);
      out.writeObject(isbn);
      out.writeObject(ps);
      out.writeObject(copies);
    } catch (Exception e) {
      System.err.println("Error in packing Book.\n" + e.toString());
    }
  }
  public void readExternal(ObjectInput in) {          // See key point T
    try {
      title = (String)in.readObject();
      author = (String)in.readObject();
      isbn = (String)in.readObject();
      ps = (PostScript)in.readObject();
      copies = (BookCopy[])in.readObject();
    } catch (Exception e) {
      System.err.println("Error in packing Book.\n" + e.toString());
    }
  }
                                                      // See key point U
  public void _d_writeExternal(ObjectOutput out, Traversal t) {
    try {
      DPartCutter c = t.isIncompleteClass("Book");    // See key point V

      if (!c.bypassPart("title"))                  // See key points V, W
       out.writeObject(title);
      if (!c.bypassPart("author"))
       out.writeObject(author);
      if (!c.bypassPart("isbn"))
       out.writeObject(isbn);
      if (!c.bypassPart("ps")) {                   // See key points V, X
       if (ps == null || (ps != null && !(ps instanceof DObject))) {
         out.writeObject("Object");
         out.writeObject(ps);
       }
       else {
         out.writeObject("DObject");
         out.writeObject(ps.getClass().getName());
         ((DObject)ps)._d_writeExternal(out, t);
       }
      }
      if (!c.bypassPart("copies")) {             // See key points V, X, Y
       if (copies == null)
         out.writeObject(new Integer(0));
       else {
         out.writeObject(new Integer(copies.length));
         for (int _i = 0 ; _i < copies.length; _i++) {
           if (copies[_i] == null ||
              (copies[_i] != null && !(copies[_i] instanceof DObject))) {
             out.writeObject("Object");
             out.writeObject(copies[_i]);
           }
           else {
             out.writeObject("DObject");
             out.writeObject(copies[_i].getClass().getName());
             ((DObject)copies[_i])._d_writeExternal(out, t);
```

```
          }
        }
      }
      }
    } catch (Exception e) {
      System.err.println("Error in _d_packing Book.\n" + e.toString());
    }
  }
                                                      // See key point U
  public void _d_readExternal(ObjectInput in, Traversal t) {
    try {
      DPartCutter c = t.isIncompleteClass("Book");      // See key point V

      if (c.bypassPart("title")) title = null;     // See key points V, W
      else title = (String)in.readObject();
      if (c.bypassPart("author")) author = null;
      else author = (String)in.readObject();
      if (c.bypassPart("isbn")) isbn = null;
      else isbn = (String)in.readObject();

      if (c.bypassPart("ps")) ps = null;           // See key points V, X
      else {
       if (((String)in.readObject()).equals("DObject")) {
         String classname = (String)in.readObject();
         ps = (PostScript)Class.forName(classname).newInstance();
         ((DObject)ps)._d_readExternal(in, t);
       }
       else
         ps = (PostScript)in.readObject();
      }
      if (c.bypassPart("copies")) copies = null; //See key points V, X, Y
      else {
       int n = ((Integer)in.readObject()).intValue();
       if (n == 0) copies = null;
       else {
         copies = new BookCopy[n];
         for (int _i = 0; _i < copies.length; _i++) {
           if (((String)in.readObject()).equals("DObject")) {
             String classname = (String)in.readObject();
             copies[_i] = (BookCopy)Class.forName(classname).newInstance();
             ((DObject)copies[_i])._d_readExternal(in, t);
           }
           else
             copies[_i] = (BookCopy)in.readObject();
         }
       }
      }
    } catch (Exception e) {
      System.err.println("Error in _d_unpacking Book.\n" + e.toString());
    }
  }
}
```

Key points:

N. When the remote operations take arguments of non-primitive types that are to be passed by copy, as in this case, the PRI interface declares arguments of type DArgument in the corresponding positions. (Note: some types of the java.lang library, e.g. String, are considered primitive)

O. Because the RIDL portal for Library contains traversals, the class LibraryTraversals is also generated. The purpose of this class is to hold descriptions of the traversals specified in the portal. Those descriptions are stored in Traversal objects (see Appendix D). The naming of the traversal variables follows the order by which the traversal specifications appear in the RIDL portal. In this case, there are three traversals, therefore three Traversal variables: `t1` for the return BookCopy object in operation getBook, `t2` for the User argument in operation getBook, and `t3` for the return Book object in operation findBook. These variables are static, and are set only once through the `init` method (which is also static).

P. The `init` method of class LibraryTraversals is called whenever a P or a PP are instantiated. Therefore, when the DJ run-time needs the traversals for passing objects, the traversals are already initialized.

Q. The methods of the P class bridge between DArguments and whatever types are expected by the methods in the Library class. In the case of a parameter, the method takes the object (`obj`) that was constructed by the DJ runtime and sends it to the “real” object; in the case of the return object, the method constructs a DArgument from the return of the call to the “real” object, sending it the corresponding traversal object.

R. On the callers side (in the PP), the opposite is done: for parameters, the method constructs a DArgument associating the parameter with a traversal, so that the DJ run-time can marshal the object according to the traversal; for return objects, the method extracts the object (`obj`) that was constructed by the DJ run-time, and sends it to the upper proxy.

S. There are two pairs of marshaling methods: the `writeExternal`, `readExternal`, from Java’s Externalizable interface, and the `_d_writeExternal`, `_d_readExternal`, which are specific to DObjects. The write methods write the object into an output stream that is then sent across the wire, and the read methods construct objects from an input stream that has been received from elsewhere. Each pair of read/write methods is symmetric.

T. The `writeExternal`/`readExternal` pair is used to marshal DObjects in the absence of traversals. That is, if no traversal is specified in the RIDL interface, then a deep copy of the object is sent/received. This is achieved by writing/reading all variables of the Book class, recursively.

U. The `_d_writeExternal`/`_d_readExternal` pair is used to marshal DObjects in the presence of traversals. Therefore, they take a traversal object as the second parameter. Note that in these methods there isn't any reference to the class Library: the way these methods are engineered allows for Book instances to marshal all possible combinations of their parts, and by several different portals. This allows for classes to be woven in separate, independent of the traversals in which they are referenced.

V. Traversal objects contain a DPartCutter object that knows exactly which parts of which classes should not be passed. Therefore, the high-level view of the implementation of `_d_writeExternal`/`_d_readExternal` is to go through all the parts of the current object, check is they were cut or not, and pack/unpack them for the case they were not cut.

W. For primitive types, they are simply written/read into/from the stream, using Java marshaling.

X. For non-primitive types, more checks must be made, because non-primitive types may be DObjects or not, and may be null. In order for the reader method to call the appropriate unmarshaling method, there is the need for sending a tag saying if the part is a DObject or not. The reader reads the tag, and decides what to do. If the part is null, then the null value is sent as a non-DObject, so that it is safely unpacked. If the part is not null but also not a DObject, that means that the traversal cannot apply to it, and its default serialization is applied instead. If the part is a DObject, then its _d_ marshaling methods are called.

Y. Arrays must be handled with special care, by iterating through the elements and packing/unpacking each of them.

4.3.4.3 Inheritance of Portals

The implementation of inheritance of portals is exactly the same as inheritance of coordination. As already presented in the previous examples, the key engineering mechanism that handles inheritance and overriding of methods according to the given semantics is the isolation of the "implementation" code in one method and the insertion of another method which does the wrapping of the aspect at hand. For a detailed explanation of how to deal with inheritance of aspect modules, see §4.2.7.2 and §4.2.7.3.

4.4. Integrating COOL and RIDL

The last two subsections described the implementation architectures of COOL and RIDL in isolation from each other. COOL and RIDL are fairly independent, and the translation parts are completely independent. However, when a JCore class is associated both with a coordinator and a portal, the weaving of that class must be done carefully. The engineering problem that must be solved is how the two kinds of wrapper methods should be integrated. The two kinds of wrappers are, for example,

```
// From the output of weaving COOL:

protected void _d_put(Object o) {
  //code for inserting
}
public void put(Object o) {
  _BBCoord.enter_BBput(this);
  try { this._d_put(o); }
  finally {
    _BBCoord_exit_BBput(this);
  }
}
```

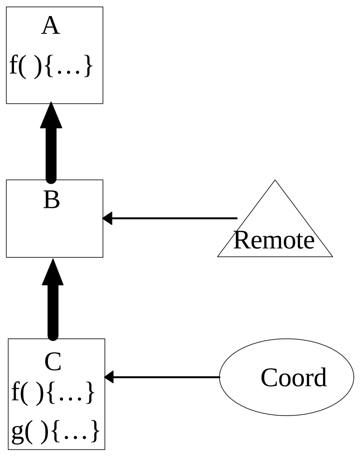
```
// From the output of weaving RIDL:

 protected void _d_put(Object o) {
  // code for inserting
 }
 public void put(Object o) {
  if (_pp != null) {
   try {_pp.put(o);}
   catch (DInvalidRemoteOperation e){
      System.err.println("Invalid…");
   }
  } else _d_put(o);
 }
```

This engineering problem is the manifestation of the issue of how proxies relate to the concurrent behavior of the remote objects they represent; and, at an even higher level of abstraction, how the aspects relate. For these two particular aspects, it is possible to establish a relation that is both intuitive and realizable.

Proxies should be unaware of the synchronization issues, since synchronization is done for the execution of methods of the "real" objects: according to the semantics described in Chapter 3, the coordination COOL targets is only the local coordination, not the distributed coordinated behavior. Therefore, the first thing that the wrapper methods must check is whether the object is a "real" object or a proxy; if it is a proxy, no synchronization should be done. The ordering of the wrappers must, then, be:

```
public void put(Object o) {
  if (_pp != null) {
    try{_pp.put(o);}
    catch (DInvalidRemoteOperation e) {
      System.err.println("Invalid…");
    }
  } else {
    _BBCoord.enter_BBput(this);
    try { this._d_put(o); }
    finally { _BBCoord.exit_BBput(this); }
  }
}
```

Although this idea is simple, its implementation is not so simple, because it must observe the semantics of inheritance and overriding of aspect modules when those modules are associated with different classes. Consider, for example, the following class hierarchy:



In this case, A-objects are neither remote nor coordinated, B-objects can be remote but are not coordinated, and C-objects can be remote and are coordinated. The difficult situation is that C- objects are coordinated by the coordinator for C and, at the same time, they are remote objects whose portal is the one inherited from B.

There are many ways of engineering the correct integration of the wrappers. One way is to always merge them, as proposed before — the actual implementation of DJ does this. The algorithm for merging wrappers is presented in Appendix C.

4.5. Summary

D was integrated with Java in a framework called DJ. This chapter described one implementation of DJ. Such implementation uses a pre-processor — the Aspect Weaver — that translates DJ programs into plain Java programs. The output Java programs contain specific patterns of code — the target architectures — that correctly implement the semantics of D. The architectures described here are simple and not optimized, but are they relatively easy to understand and reproduce.

This implementation preserves the modularities of D in the output code. Component and aspect modules are processed separately, and the dependencies in the output code preserve the dependencies given by the interfaces described in Chapter 3. Coordinators and portals are translated into Java classes. The instances of those classes — the “aspect objects” — execute the particular aspect run-time. The JCore classes are woven with hooks that transfer the control to the aspect objects at the beginning and at the end of the methods.

This chapter described the run-time structures for implementing COOL, RIDL and their integration with Java. The target architectures were explained in detail, and some input/output examples were given. The automation of the weaving/translation is given in Appendix C. The target architectures described here should not be seen as the final and best implementation strategy, but rather as a reasonable starting point for exploring the implementation space.

Chapter 5

Validation

“There is an implicit Whorfian hypothesis that the nature of languages shapes the way we think about problems. Thus, although we may not be able to measure it directly, most experts believe that the user of one of the more modern, structured languages is better equipped to think about complex problems than the user of the older languages (e.g., Fortran).”

William Wulf, in “*Trends in the Design and Implementation of Programming Languages*” [75]

5. Validation

D was designed to provide support for programming thread synchronization and remote interaction in separate from the implementation of the classes. The claim made about the framework is that the proposed re-modularization drastically decreases the tangling between functionality code and the code for programming those other two concerns, at a very low cost, and making programs easier to write and understand. This chapter validates this claim. It shows how effective the separation is, and what are the benefits and costs of doing it. The data presented here supports the following observations:

(1) The separation is extremely effective for the issues for which D was designed. By using D, the classes can, in fact, be freed from all the code for dealing with thread synchronization and remote data transfers.

(2) There are still important issues of distributed systems that are not captured by the aspect languages. Therefore, the programming of those issues is still tangled in the classes. The systematic approach to aspect language design, based on the analysis of code tangling, can overcome the limitations of the current version, so that, in the future, D can provide better support for those issues.

(3) On the human side, the re-modularization is intuitive and easily understood by programmers. Aspect modules were found to be very useful, in that they simplify the programming of some distribution issues.

(4) The locality of the aspect code and its relative separation from the classes is a major benefit of the framework. It allows programmers to reason, informally, about it, and to think more carefully about its consequences with respect to the rest of the application — something that they can hardly do when the aspect code is spread across the implementation of the classes.

(5) The costs of the framework are very low. The simple implementation described in Chapter 4 performs worse than plain Java, but still within acceptable bounds; a number of straightforward optimizations can make the framework unnoticeable at run-time. With respect to the size of programs, the aspect languages allow the development of programs that are at least as small as they would be if the framework was not being used, and, in many cases, smaller. With respect to human learning, programmers can assimilate the aspect languages fast.

The validation consists of three distinct sets of results: (1) a case by case comparison between implementations written in DJ and in other languages (§5.1); these are canonical examples that show the effectiveness of the separation in small scale, and that provide some hints about the strengths and weaknesses of the framework; (2) performance measurements; and (3) a usability experiment, in which four alpha-users were asked to write medium-sized applications using DJ (§5.2); this was a preliminary usability study, made to test the programmers' understanding of the re-modularization and of the new aspect interfaces introduced by D.

5.1. Case-Studies

This section contains a comparison between programs written in DJ and programs written in plain Java or C++. The goal is to make a theoretical, although not exhaustive, study of how D improves the quality of programs. Such study focuses on how D "behaves" in small, canonical examples, and it eliminates two important variables of practical software engineering: the programmers and the complexity of programming in the large. Nevertheless, it gives us some hints for which are the strengths and weaknesses of D.

Sub-sections §5.1.1 through §5.1.10 present ten small applications that can be seen as canonical examples of concurrent and distributed object systems. The presentation of these ten case-studies follows the following format: (1) a brief description of the functionality of the application, and additional requirements; (2) the reason why the case-study was selected; (3) the source for comparison; (4) two-column code comparison between pieces of code that illustrate the DJ and the alternative implementations, and eventual comments for clarifying specific points. For case-studies §5.1.8 and §5.1.9 some figures are also included, and only a portion of the code is shown. In order to be able to compare the DJ program with the alternative implementation, the intentions of the design are preserved in both implementations.

The analysis of the case-studies is concentrated at the end of this section. Sub-section §5.1.11 makes a quantitative study of the ten applications, introducing some ratios that help to measure the effectiveness of D with respect to improving the quality of programs.

In the presentation of the case-studies, some parts of the code are shadowed. The shadowing highlights the parts of the code that deal with synchronization and remote interaction. In the DJ implementations, portals and coordinators are shadowed. The identification of such blocks of code in the class implementations is less straightforward, but follows a simple rule: those are the pieces of code that would not be there if the execution of the objects was unsynchronized and non-distributed, that is, if the programmer would implement just the functional specifications. The shadowing follows a coarse-grain, optimistic approach. More precisely, in shadowing the classes, the following guidelines were used:

- the method qualifier `synchronized` is shadowed.
- synchronized statements are shadowed; the body of these statements may or may not be shadowed, depending on whether it deals only with synchronization issues or not.
- calls to `wait` and `notify` are shadowed.
- variable declarations used for holding synchronization state are also shadowed, as a block; the use of these variables is also shadowed.
- methods whose sole purpose is synchronization are shadowed; calls to those methods are also shadowed.
- the declaration “extends Remote” is shadowed, since its sole purpose is that the objects can be accessed remotely; however, those interfaces that extend the Remote interface are not shadowed, since they can be seen as types.
- the declaration of the exception RemoteException in the methods is shadowed, since it is there for reasons that have nothing to do with the implementation of the class (this exception is thrown by the RMI run-time).
- method signatures whose parameters are a number of parts of objects instead of the objects themselves are shadowed. As described in Chapter 2, the “splitting parts” re-design can be used to fix the problem of having to transfer only some parts of the objects, but it looses one important invariant, namely that the parts belong to the same object.
- classes whose sole purpose is to assist in the implementation of synchronization or remote data transfers are treated in a special way: the declaration is shadowed, and then each of its methods is shadowed as a single block. Method invocations to instances of those classes are also shadowed. It should be noticed that these classes can be a major source of confusion in the designs: they don’t exist in the functional specifications, but are included in the implementation in order to deal with the aspects.

The shadowing is a visual representation of the tangling phenomenon studied in Chapter 2, and it gives an immediate understanding of how effective D is in solving the code tangling problem. The shadowing itself is used as important data in two of the four metrics in §5.1.11.

5.1.1. The Bounded Buffer

Synopsis: A shared buffer object keeps an internal buffer. Producers insert objects in the buffer object, and consumers remove objects from the buffer object. When the buffer is full, producers wait until it's not full; when the buffer is empty, consumers wait until it's not empty. (Variations of this application were used throughout this thesis.)

Interesting features: This is one of the classical examples of synchronization. It needs mutual exclusion and guarded suspension. The amount of mutual exclusion depends on the implementation of the buffer class. Several implementations can be found in the literature.

Comparison with: Java implementation from [40] page 100.

DJ

```
public class BoundedBuffer {
  private Object array[];
  private int putPtr = 0, takePtr = 0;
  private int usedSlots=0;

  public BoundedBuffer(int capacity) {
    array = new Object[capacity];
  }

  public void put(Object o) {
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    usedSlots++;
  }

  public Object take() {
    Object old = array[takePtr];
    array[takePtr] = null;
    takePtr = (takePtr + 1) % array.length;
    usedSlots--;
    return old;
  }
}

coordinator BoundedBuffer {
 selfex put, take;
 mutex {put, take};
 cond full = false, empty = true;
 put: requires !full;
   on_exit {
     empty = false;
     if (usedSlots == array.length)
          full = true;
   }
 take:  requires !empty;
     on_exit {
       full = false;
       if (usedSlots == 0) empty = true;
     }
}
```

JAVA

```
public class BoundedBuffer {
  private Object[] array;
  private int putPtr = 0, takePtr = 0;
  private int usedSlots = 0;

  public BoundedBuffer (int capacity) {
    array = new Object[capacity];
  }

  public synchronized void put(Object o) {
    while (usedSlots == array.length) {
      try {
        wait();
      }
      catch (InterruptedException e) {};
    }
    array[putPtr] = o;
    putPtr = (putPtr + 1) % array.length;
    }
    if (usedSlots++ == 0)
      notifyAll();
  }

  public synchronized Object take() {
    while (usedSlots == 0) {
      try {
        wait();
      }
      catch (InterruptedException e) {};
    }
    Object old = array[takePtr];
    array[takePtr] = null;
    takePtr = (takePtr+1) % array.length;
    }
    if (usedSlots-- == array.length)
      notifyAll();
    return old;
  }
}
```

5.1.2. The Dinning Philosophers

Synopsis: Five philosophers are sitting at a table, eating and thinking alternately. In order to eat, they need to hold the two adjacent forks, which they share with their left and right neighbor respectively. Each philosopher can only eat if he hold both forks.

Interesting features: This is the other classical example of synchronization, that models the situations of threads having to grab several resources before they can proceed. Many implementations can be found in the literature. This particular implementation uses the monitor design.

Comparison with: Java implementation.

DJ

```
class Philosopher implements Runnable {
  // the global set up
  static final int max = 5;
  static Fork forks[]= new Fork[max];
  static int count = 0;
  // for each philosopher
  protected int    mynumber;
  protected Fork   left, right;
  protected Random time = new Random ();

  Philosopher() { /* initialization */ }
  public void run() {/*loop: think, eat */}
  private void think() { /* think */ }

  private void eat() {
    left.take();
    right.take();
    int x = time.nextInt();
    try{
      Thread.sleep(Math.abs(x % 500));
    } catch(InterruptedException e){};
    left.put();
    right.put();
  }
}

per_class coordinator Philosopher {
  condition eating[max]=new boolean[false];
  eat: requires !eating[(mynumber+1)%max]&&
             !eating[(mynumber+max-1)%max];
       on_entry {
         eating[mynumber] = true;
       }
       on_exit {
         eating[mynumber] = false;
       }
}
```

JAVA

```
class Philosopher implements Runnable {
  // the global set up
  static final int max = 5;
  static Fork forks[]= new Fork[max];
  static int count = 0;
  static Object PhiLock = new Object();
  static boolean Eating[] = {false, false,
                      false, false, false};
  // for each philosopher
  protected int    mynumber;
  protected Fork   left, right;
  protected Random time = new Random ();

  Philosopher() { /* initialization */ }
  public void run() {/*loop: think, eat */}
  private void think() { /* think */ }

  private void eat() {
    synchronized (PhiLock) {
      while (Eating[(mynumber+1)%max] ||
             Eating[(mynumber+max-1)%max]){
        try {PhiLock.wait();}
        catch (InterruptedException e) {}
      }
      Eating[mynumber] = true;
    }
    left.take();
    right.take();
    int x = time.nextInt();
    try{
      Thread.sleep(Math.abs(x % 500));
    } catch(InterruptedException e){};
    left.put();
    right.put();
    Eating[mynumber] = false;
    synchronized (PhiLock) {
      Phi.notifyAll();
    }
  }
}
```

5.1.3. The Shape

Synopsis: A shape object maintains both location and dimension information, along with time-consuming methods `adjustLocation` and `adjustDimensions` that independently alter the location and dimension of the shape.

Interesting features: The class has two sets of methods, each set having be synchronized, but independently of each other.

Comparison with: Java implementations from [40] pages 71-75.

DJ

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

  double x() { return x_(); }
  double y() { return y_(); }
  double width(){ return width_(); }
  double height(){ return height_(); }
  void adjustLocation() {
    x_ = longCalculation1();
    y_ = longCalculation2();
  }
  void adjustDimensions() {
    width_ = longCalculation3();
    height_ = longCalculation4();
  }
}

coordinator Shape {
  selfex adjustLocation, adjustDimensions;
  mutex {adjustLocation, x, y};
  mutex {adjustDimensions, width, height};
}
```

JAVA

```
public class Shape {
  protected AdjustableLocation loc;
  protected AdjustableDimension dim;

  public Shape() {
    loc = new AdjustableLocation(0, 0);
    dim = new AdjustableDimension(0, 0);
  }
  double x() { return loc.x(); }
  double y() { return loc.y(); }
  double width(){ return dim.width(); }
  double height(){ return dim.height(); }
  void adjustLocation() { loc.adjust(); }
  void adjustDimensions() { dim.adjust(); }
}

class AdjustableLocation {
  protected double x_,  y_;
  public AdjustableLocation(double x,
                            double y) {
    x_ = x; y_ = y;
  }
  synchronized double x() { return x_; }
  synchronized double y() { return 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 width() {
    return width_;
  }
  synchronized double height() {
    return height_;
  }
  synchronized void adjust() {
    width_ = longCalculation3();
    height_ = longCalculation4();
  }
}
```

5.1.4. Concurrent Matrix Multiplication

Synopsis: Matrix multiplication is performed by a number of concurrent threads. The output matrix is divided into *dimension/n_threads* parts, and each thread operates over each part.

Interesting features: This models a variety of concurrent applications designed as master/slaves tasks, where each slave task does not share data. There is some synchronization only at the beginning and end of the computation.

Comparison with: C++ implementation from [41].

DJ

```
public class MatMul implements Runnable {
  static Matrix a, b, c;
  static int total_threads = 0;
  static int thrs_running = 0, queue = 0;
  static MatMul master = null;

  // The matrix multiplication function
  static void DoMatMul(Matrix aa,
                   Matrix bb, Matrix cc){
    a=aa; b=bb; c=cc;
    // execute the master object
    master = new MatMul();
    master.startThreads();
  }

  // Methods for master
  void startThreads() {
    for (int i = 0;i<total_threads; i++){
      // start a new slave thread
      new Thread(new MatMul()).start();
      thrs_running++;
    }
    printResults();
  }
  void printResults() {/* print them */ }
  int get_tid() { return queue++; }
  void work_done() { thrs_running--; }

  // Method for slaves
  public void run() {
   int start, stop, size = a.matsize;
   int tid=master.get_tid();
   start=tid*(size/total_threads);
   stop=start+(size/total_threads)-1;
   for (int row=start; row<=stop;row++)
    for (int col=0;col<size;col++)
     for (int j = 0; j < size; j++)
      c.data[row*size+col] +=
                 a.data[row*size+j] *
                 b.data[j*size+col];
    master.work_done();
  }
}

coordinator MatMul {
  selfex get_tid, work_done;
  cond allDone = false;
  printResults: requires allDone;
  work_done: on_exit {
     if (thrs_running == 0) allDone=true;
    }
}
```

C++ AND PTHREADS

```
struct thr_cntl_block {
  Matrix *a, *b, *c;
  int thrs_running, total_threads, queue;
  mutex_t start_mutex, stop_mutex;
  cond_t  start_cond, stop_cond;
} TCB;
// The matrix multiplication master function
DoMatMul(Matrix &a, Matrix &b, Matrix &c) {
  mutex_init(&TCB.start_mutex,USYNC_THREAD,0);
  mutex_init(&TCB.stop_mutex,USYNC_THREAD,0);
  cond_init(&TCB.start_cond,USYNC_THREAD,0);
  cond_init(&TCB.start_cond,USYNC_THREAD, 0);
  // more initialization of TCB omitted
  for (i = 0; i < TCB.total_threads; i++)
    thr_create(NULL,0,MultWorker,NULL,
               THR_BOUND|THR_DAEMON,NULL);
  mutex_lock(&TCB.start_mutex);
  TCB.thrs_running = TCB.total_threads;
  cond_broadcast(&TCB.start_cond);
  mutex_unlock(&TCB.start_mutex);
  thr_yield();
  mutex_lock(&TCB.stop_mutex);
  while (TCB.thrs_running)
    cond_wait(&TCB.stop_cond,&TCB.stop_mutex);
  mutex_unlock(&TCB.stop_mutex);
  printResults();
  return 0;
}
// Slave routine called from thrd_create
void *MultWorker(void *arg) {
  int row, col, j, start, stop, id, size;
  while (true) {
   mutex_lock(&TCB.start_mutex);
   cond_wait(&TCB.start_cond,
               &TCB.start_mutex);
   id = TCB.queue++;
   mutex_unlock(&TCB.start_mutex);
   size = TCB.a->getsize();
   start=id*(int)(size/TCB.total_threads-1);
   stop=start+(int)(size/TCB.total_threads)-1;
   for (row=start;row<=stop;row++)
    for(col=0;col<size;col++)
     for(j=0;j<size;j++)
      TCB.c->data()[row*size+col] +=
                 TCB.a->data[row*size+j] *
                 TCB.b->data[j*size+col];
   mutex_lock(&TCB.stop_mutex);
   TCB.thrs_running--;
   cond_signal(&TCB.stop_cond);
   mutex_unlock(&TCB.stop_mutex);
  }
  return 0;
}
```

5.1.5. Concurrent Graph Traversal

Synopsis: A graph is made of nodes which contain an integer value. The goal is to compute the sum of all the node values, by traversing the graph. A master object starts worker threads at some of the nodes of the graph; the result is the sum of the partial sums.

Interesting features: This application models a variety of concurrent applications designed as master/slaves tasks, where the slave tasks read and write shared data. In this example, the synchronization in the shared data is minimal: it consists of synchronizing the test and set of a flag indicating that the node has been visited. There is also some synchronization at the end, so that the master collects the results.

Comparison with: Java implementation.

DJ

```
class Node {
  Vector  neighbors = new Vector(6);
  int     id;
  transient boolean visited = false;
  // Initial set-up function
  static void connect(Node n1, Node n2) {
    // connect n1 to n2 and vice-versa
  }
  Node(int id) { this.id = id; }
  boolean was_visited() {
    if (visited) return true;
    visited = true;
    return false;
  }
  int sumup() {
    if (was_visited()) return(0);
    Enumeration elts=neighbors.elements();
    int      sum  = id;
    Node neighbor = null;
    while (elts.hasMoreElements()) {
      neighbor=(Node)elts.nextElement();
      sum += neighbor.sumup();
    }
    return sum;
  }
}

coordinator Node {
  selfex was_visited;
}
```

JAVA

```
class Node {
  Vector  neighbors = new Vector(6);
  int     id;
  transient boolean visited = false;
  // Initial set-up function
  static void connect(Node n1, Node n2) {
    // connect n1 to n2 and vice-versa
  }
  Node(int id) { this.id = id; }
  synchronized boolean was_visited() {
    if (visited) return true;
    visited = true;
    return false;
  }
  int sumup() {
    if (was_visited()) return(0);
    Enumeration elts=neighbors.elements();
    int      sum  = id;
    Node neighbor = null;
    while (elts.hasMoreElements()) {
      neighbor=(Node)elts.nextElement();
      sum += neighbor.sumup();
    }
    return sum;
  }
}
```

DJ (CONT.)

```
// The master and workers

class Traverser implements Runnable {
  static Graph graph;
  static transient int n_traversers = 0;
  static Traverser master;

  // Constructor for the master object
  Traverser (Graph g, int n_threads) {
    if (n_threads > 4) return;
    master = this; graph = g;
    Traverser workers[] = new Traverser[4];
    Integer   root[] = new Integer[4];
    // Initialization of 4 roots omitted

    for (int i = 0; i < n_threads; i++) {
      // start worker thread
      workers[i] = new Traverser(root[i]);
      workers[i].start();
    }
    printResult();
  }

  void printResult() {
    int sum = 0;
    for (int i = 0; i < n_threads; i++) {
     System.out.println("Sum[" + i + "]: "
                        + workers[i].sum);
     sum += workers[i].sum;
    }
    System.out.println("Total = " + sum);
  }
  void new_worker() { n_traversers++; }
  void work_done() { n_traversers--; }

  // Variables and methods for the workers
  Integer rootid; int sum = 0;
  Traverser(Integer r) {
    rootid = r;
    master.new_worker();
  }
  public void run() {
    sum = graph.sumup(rootid);
    master.work_done();
  }
}

coordinator Traverser {
  selfex new_worker, work_done;
  cond allDone = false;
  wait_for_workers: requires allDone;
  work_done: on_exit {
             if (n_traversers == 0)
               AllDone = true;
             }
}
```

JAVA (CONT.)

```
// The master and workers

class Traverser implements Runnable {
  static Graph graph;
  static transient int n_traversers = 0;
  static Traverser master;

  // Constructor for the master object
  Traverser (Graph g, int n_threads) {
    if (n_threads > 4) return;
    master = this; graph = g;
    Traverser workers[] = new Traverser[4];
    Integer   root[] = new Integer[4];
    // Initialization of 4 roots omitted

    for (int i = 0; i < n_threads; i++) {
      // start worker thread
      workers[i] = new Traverser(root[i]);
      workers[i].start();
    }
    printResult();
  }

  synchronized void printResult() {
    try {wait();}
    catch (InterruptedException e) {}
    int sum = 0;
    for (int i = 0; i < n_threads; i++) {
     System.out.println("Sum[" + i + "]: "
                        + workers[i].sum);
     sum += workers[i].sum;
    }
    System.out.println("Total = " + sum);
  }
  synchronized void new_worker() {
    n_traversers++;
  }
  synchronized void work_done() {
    n_traversers--;
    if (n_traversers == 0)
      notify();
  }

  // Variables and methods for the workers
  Integer rootid; int sum = 0;
  Traverser(Integer r) {
    rootid = r;
    master.new_worker();
  }
  public void run() {
    sum = graph.sumup(rootid);
    master.work_done();
  }
}
```

5.1.6. Assembly Line

Synopsis: This application consists of a number of concurrent agents. Several Candy Makers produce candies, which they feed, concurrently, to a Packer; the Packer fills a packet with a maximum number of candy and passes the packet to a Finalizer agent; the Finalizer takes one packet from the Packer and one label from a Label Maker, glues the latter on the former, and produces the final candy packet. (see Appendix B for an illustration of the agents)

Interesting features: This small application models a variety of systems designed as collaborating concurrent agents. The coordination involves several agents.

Comparison with: Java implementation.

DJ

```
class CandyMaker implements Runnable {
  protected Packer thePacker = null;
  CandyMaker(Packer p) {thePacker = p;}
  public void run() {
    while (true) {
      Candy aCandy = makeCandy();
      thePacker.newCandy(aCandy);
    }
  }
  protected Candy makeCandy() {/*make it*/}
}

class Packer implements Runnable {
  static int    nCandyPerPack = 50;
  protected Finalizer theFinalizer = null;
  protected Pack candyPack = null;
  protected int  nCandy = 0;
  Packer(Finalizer f) {theFinalizer = f;}
  public void run() {
    while (true) {
      candyPack = makePack();
      processPack(candyPack);
      theFinalizer.newPack(candyPack);
    }
  }
  void newCandy(Candy aCandy) {
    candyPack.put(aCandy); nCandy++;
  }
  protected Pack makePack(){/* make it */}
  protected void processPack(Pack aPack) {
    /*process it */
  }
}

class LabelMaker implements Runnable {
  protected Finalizer theFinalizer = null;
  LabelMacker(Finalizer f){theFinalizer=f;}
  public void run() {
    while (true) {
      Label aLabel = makeLabel();
      theFinalizer.newLabel(aLabel);
    }
  }
  protected Label makeLabel() {/*make it*/}
}
```

JAVA

```
class CandyMaker implements Runnable {
  protected Packer thePacker = null;
  CandyMaker(Packer p) {thePacker = p;}
  public void run() {
    while (true) {
      Candy aCandy = makeCandy();
      thePacker.newCandy(aCandy);
    }
  }
  private Candy makeCandy(){/* make it */}
}

class Packer implements Runnable {
  static int    nCandyPerPack = 50;
  protected Finalizer theFinalizer = null;
  protected Pack candyPack = null;
  protected int  nCandy = 0;
  protected packDone = false;
  Packer(Finalizer f) { theFinalizer = f; }
  public void run() {
    while (true) {
      candyPack = makePack();
      synchronized (this) {
        while (nCandy < nCandyPerPack) {
          try {wait();}
          catch(InterruptedException e) {}
        }
      }
      processPack(candyPack);
      theFinalizer.newPack(candyPack);
      synchronized (this) {
        nCandy = 0; packDone = false;
        notifyAll();
      }
    }
  }
  synchronized void newCandy(Candy aCandy){
    while (nCandy == nCandyPerPack ||
           !packDone) {
      try {wait();}
      catch(InterruptedException e) {}
    }
    candyPack.put(aCandy); nCandy++;
    if (nCandy==nCandyPerPack) notifyAll();
  }
```

DJ (CONT.)

```
class Finalizer implements Runnable {
 protected Pack        thePack = null;
 protected Label       theLabel = null;
 public void run() {
   while (true) {
     glueLabelToPack();
     newDJCandyPack();
   }
 }
 void newPack(Pack aPack) {
   thePack = aPack;
 }
 void newLabel(Label aLabel) {
   theLabel = aLabel;
 }
 protected void glueLabelToPack(){/*glue*/}
 protected void newDJCandyPack() {
    System.out.println(“New Candy Pack!”);
  }
}

coordinator Packer, Finalizer {
  selfex Packer.newCandy;
  cond packDone = false, packFull = false;
  cond gotPack = false, gotLabel = false;

  Packer.newPack: on_exit{packDone = true;}
  Packer.newCandy: requires !packFull &&
                             packDone;
      on_exit {
        if (nCandy == nCandyPerPack)
          packFull = true;
      }
  Packer.processPack: requires packFull;
  Finalizer.newPack: requires !gotPack;
      on_exit {
        gotPack = true;
        packFull = false; packDone = false;
      }
  Finalizer.newLabel: requires !gotLabel;
      on_exit { gotLabel = true; }
  Finalizer.glueLabelToPack:
              requires gotPack && gotLabel;
  Finalizer.newDJCandyPack:
      on_exit {
        gotPack = false; gotLabel = false;
      }
}
```

JAVA (CONT.)

```
  // Continuation of class Packer
  protected synchronized Pack makePack(){
    gotPack = true;
    notifyAll();
    /* make it */
  }
  protected void processPack(Pack aPack) {
    /* process it */
  }
}
class LabelMaker implements Runnable {
  protected Finalizer theFinalizer = null;
  LabelMacker(Finalizer f){theFinalizer=f;}
  public void run() {
    while (true) {
      Label aLabel = makeLabel();
      theFinalizer.newLabel(aLabel);
    }
  }
  protected Label makeLabel() {/*make it*/}
}

class Finalizer implements Runnable {
 protected Pack        thePack = null;
 protected Label       theLabel = null;
 protected boolean  gotLabel = false;
 protected boolean  gotPack = false;
 public void run() {
   while (true) {
     synchronized (this) {
       while (!(gotPack && gotLabel)) {
         try{wait();}
         catch(InterruptedException e) {}
       }
     }
     glueLabelToPack();
     newDJCandyPack();
     synchronized (this) {
       gotLabel = false;
       gotPack = false;
       notifyAll();
     }
   }
 }
 synchronized void newPack(Pack aPack) {
   while (gotPack) {
     try {wait();}
     catch (InterruptedException e) {}
   }
   thePack = aPack;
   gotPack = true;
   notifyAll();
 }
 synchronized void newLabel(Label aLabel) {
   while (gotLabel) {
     try {wait();}
     catch (InterruptedException e) {}
   }
   theLabel = aLabel;
   gotLabel = true;
   notifyAll();
 }
 protected void glueLabelToPack(){/*glue*/}
 protected void newDJCandyPack() {
    System.out.println(“New Candy Pack!”);
  }
}
```

5.1.7. Distributed BookLocator/PrintService

Synopsis: A book locator is a service that maintains an association between books and their physical locations. The print service prints books.

Interesting features: The book locator and the printer are network services; they both use book objects, but they need different parts of the book data.

Comparison with: Java implementation.

DJ

```
portal BookLocator {
 void register (Book book, Location l);
 Location locate (String title)
 default:
   Book: copy{Book only title,author,isbn;}
}
portal Printer {
  void print(Book book) {
    book: copy { Book only title,ps; }
  }
}

class Book {
  protected String title, author;
  protected int isbn;
  protected OCRImage firstpage;
  protected Postscript ps;
  // All methods omitted
}
class BookLocator {
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  // Other variables omitted
  public void register(Book b, Location l){
    // Verify and add book b to database
  }
  public Location locate (String title) {
    Location loc;
    // Locate book and get its location
    return loc;
  }
  // other methods omitted
}
class Printer {
  public void print(Book b) {
    // Print the book
  }
}

coordinator BookLocator {
  selfex register;
  mutex {register, locate};
}
```

JAVA

```
interface Locator extends Remote {
 void register(String title,
               String author, int isbn,
               Location l)
      throws RemoteException;
 Location locate(String title)
      throws RemoteException;
}
interface PrinterService extends Remote {
  void print(String title, Postscript ps)
       throws RemoteException;
}
class Book {
  protected String title, author;
  protected int isbn;
  protected OCRImage firstpage;
  protected Postscript ps;
  // All methods omitted
}
class BookLocator
        extends UnicastRemoteObject
        implements Locator {
  // books[i] is in locations[i]
  private Book     books[];
  private Location locations[];
  // Other variables omitted
  public void register (String title,
                        String author,
                        int isbn,
                        Location l)
          throws RemoteException {
    beforeWrite(); //for synchronization
    Book b=new Book (title, author, isbn);
    // Verify and add book b to database
    afterWrite(); //for synchronization
  }
  public Location locate (String title)
          throws RemoteException {
    Location loc;
    beforeRead(); //for synchronization
    // Locate book and get its location
    afterRead(); //for synchronization
    return loc;
  }
  // other methods omitted
}
class Printer extends UnicastRemoteObject
              implements PrinterService {
  public void print(String title,
                    Postscript ps)
          throws RemoteException {
    // Print the book
  }
}
```

5.1.8. Distributed Text Editor

Synopsis: A text object can be accessed by several editors in the network. All editors can read and modify the contents of the text, but each of them maintains its own editing state (e.g. `cursor`).

Optimization: In order to speed up the “read” accesses to the text, the text object is replicated in every editor. There is one master copy of the text. Modifications to the text are always done through the master copy. Every time there is a modification, the master sends updated versions of the date to all the replicas. Figure 36 shows the main steps of the text modification protocol when a remote editor wants to insert a word in the shared text.

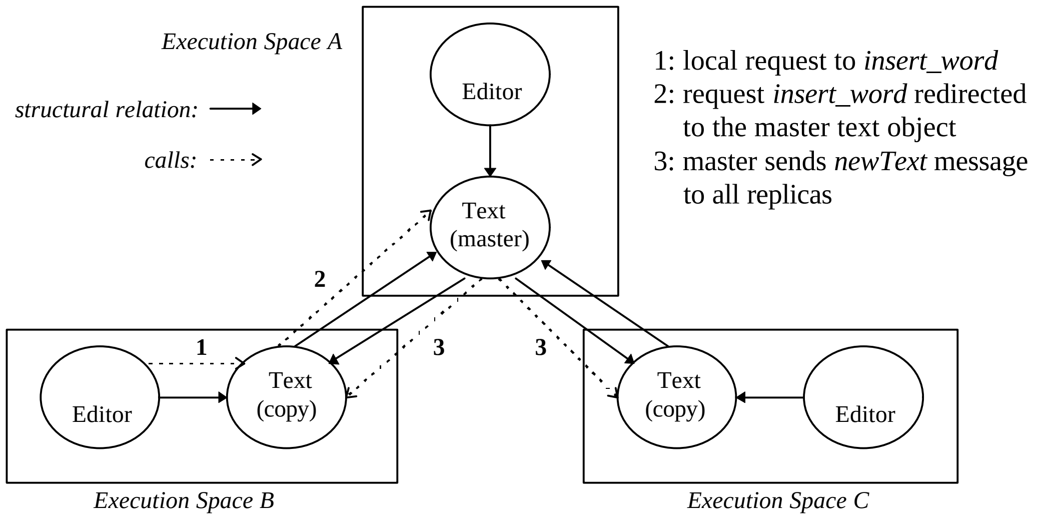

Execution Space A
structural relation:
calls:
Editor
Text (master)
1: local request to *insert_word*
2: request *insert_word* redirected to the master text object
3: master sends *newText* message to all replicas
2
1
3
3
Editor
Text (copy)
Text (copy)
Editor
Execution Space B
Execution Space C

Figure 36. Structure of the distributed text editor and protocol for inserting a word remotely.

Interesting features: This application models a variety of distributed systems that use replication of data.

Comparison with: Java implementation.

DJ

```
portal Text {
  void insert_word(char[] word, int size,
                   int pos);
  void remove(int c_position);

  // for the master copy
  void joinReplica(Text rep);
  void quitReplica(Text rep);
  // for the replicas
  Integer get_id();
  void newText(Text masterT,
               TextData tcopy);
  default: Text: gref;
}
```

JAVA

```
public interface TextI extends Remote {
  void insert_word (char[] word, int size,
                    int pos) throws /*…*/;
  void remove(int c_position) throws /*…*/;

  // for the master copy
  void joinReplica(TextI rep)
       throws RemoteException;
  void quitReplica(TextI rep) throws /*…*/;
  // for the replicas
  Integer get_id() throws /*…*/;
  void newText(TextI masterT,
               TextData tcopy)throws /*…*/;
}
```

DJ (CONT.)

```
class Text {
  TextData data = null;
  // Variables for the master copy:
  Hashtable replicas = new Hashtable(4);
  // Variable for the replicas:
  Integer id;
  TextI master; // if null, this is master

  // Only some methods are shown
  public void insert_word(char[] word,
                          int size,int pos){
    if (master == null) {
      for(int i=data.index-size;i>=pos;i--)
        data.t[i+size] = data.t[i];
      for (int i = 0; i < size; i++)
        data.t[i+pos] = word[i];
      data.index += size;
      updateLocalView();
      updateCopies();
    }
    else
      master.insert_word(word, size, pos);
  }
  public void remove(int pos) {
    if (master == null) {
      for (int i=pos; i<data.index-1; i++)
        data.t[i] = data.t[i+i];
      data.index--;
      updateLocalView();
      updateCopies();
    }
    else master.remove(pos);
  }
  public void newText(TextI masterT,
                      TextData tcopy) {
    master = masterT;
    data = tcopy;
    updateLocalView();
  }
  int size() {
    return data.index;
  }
  void display() {
    for (int i = 0; i < data.index; i++)
      System.out.print(data.t[i]);
  }
}

coordinator Text {
  selfex insert_word, remove;
  mutex {insert_word, remove};
  mutex {newText, display};
}
```

JAVA (CONT.)

```
class Text extends UnicastRemoteObject
           implements TextI {
  TextData data = null;
  // Variables for the master copy:
  Hashtable replicas = new Hashtable(4);
  // Variable for the replicas:
  Integer id;
  TextI master; // if null, this is master
  Object newCopyLock = new Object();

  // Only some methods are shown
  public synchronized void
         insert_word(char[] word, int size,
                     int pos) throws /*…*/{
    if (master == null) {
      for(int i=data.index-size;i>=pos;i--)
        data.t[i+size] = data.t[i];
      for (int i = 0; i < size; i++)
        data.t[i+pos] = word[i];
      data.index += size;
      updateLocalView();
      updateCopies();
    }
    else
      master.insert_word(word, size, pos);
  }
  public synchronized void remove(int pos)
         throws /* …*/ {
    if (master == null) {
      for (int i=pos; i<data.index-1; i++)
        data.t[i] = data.t[i+i];
      data.index--;
      updateLocalView();
      updateCopies();
    }
    else master.remove(pos);
  }
  public void newText(TextI masterT,
                      TextData tcopy)
         throws /* … */{
    synchronized (newCopyLock) {
      master = masterT;
      data = tcopy;
      updateLocalView();
    }
  }
  int size() {
    synchronized (newCopyLock) {
      return data.index;
    }
  }
  void display() {
    synchronized (newCopyLock) {
      for (int i = 0; i < data.index; i++)
        System.out.print(data.t[i]);
    }
  }
}
```

The shadows show the code related to synchronization and remote data transfers. The squares show the code related to replication. In this example, the DJ implementation also suffers from tangling with respect to replication, meaning that RIDL is not good at capturing this concern.

5.1.9. Distributed Document Service

Synopsis: This application consists of a document server that contains information about documents, and that provides a search engine that users can access. Users must first register, and provide a password, which they must then supply for future interactions with the document service, including searches. For each search request, the server logs the time, the document and the user that issued the request. Users can also request a list of all their logs. Figure 37 shows the class graph for implementing the basic functionality of the document service (the '*' represents a one-to-many relationship).

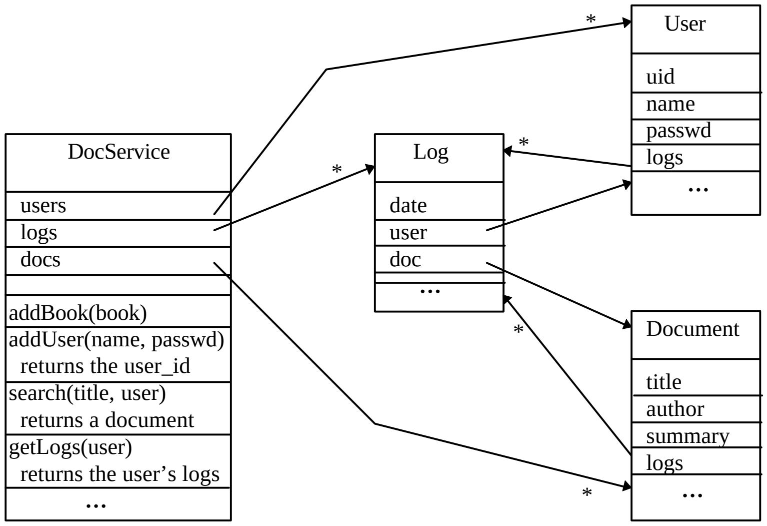

DocService
users
logs
docs
addBook(book)
addUser(name, passwd) returns the user_id
search(title, user) returns a document
getLogs(user) returns the user's logs
...
Log
date
user
doc
...
User
uid
name
passwd
logs
...
Document
title
author
summary
logs
...
*

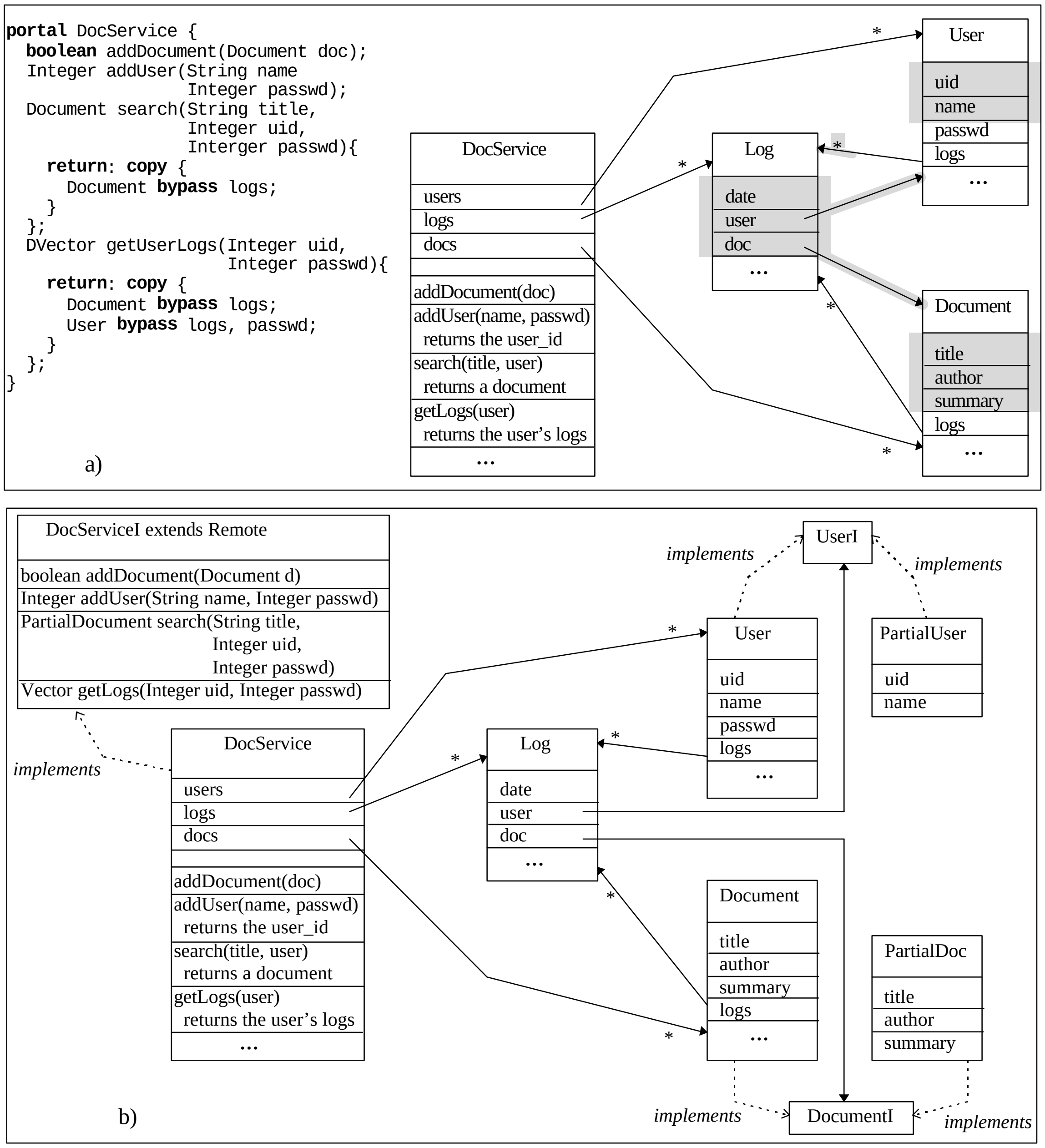
Figure 37. The document service.

Optimization: In order to speed up the accesses to the information, all the data is copied from the server to the clients and vice-versa. For example, when a user searches for a document, the document is copied to the user's machine, so that the browsing of the data is done locally.

Interesting features: This application models a variety of distributed systems (Web included) in which the data is selectively copied without any guarantees of consistency. This particular implementation of the document service contains cycles (Log, Document, Log and Log, User, Log) that need to be broken, or the whole data of the document service may be sent out to the clients.

Comparison with: Java implementation.

This application is too big for an exhaustive two-column code comparison. Instead, the class graphs are shown in Figure 38. The class graph for the DJ implementation (a) is exactly the same as the one shown in Figure 37. The selection of the data to send to clients is done in the portal. The gray area represents the portion of the data involved when passing the user's logs in the service `getUserLogs`. As for the Java implementation (b), the class graph must be extended by 2 more classes in order to be able to cope with the data selections.



```
portal DocService {
  boolean addDocument(Document doc);
  Integer addUser(String name
                  Integer passwd);
  Document search(String title,
                  Integer uid,
                  Interger passwd){
    return: copy {
      Document bypass logs;
    }
  };
  DVector getUserLogs(Integer uid,
                      Integer passwd){
    return: copy {
      Document bypass logs;
      User bypass logs, passwd;
    }
  };
}
```

a)
DocService
users
logs
docs
addDocument(doc)
addUser(name, passwd) returns the user_id
search(title, user) returns a document
getLogs(user) returns the user's logs
...
Log
date
user
doc
...
User
uid
name
passwd
logs
...
Document
title
author
summary
logs
...
*
b)
DocServiceI extends Remote
boolean addDocument(Document d)
Integer addUser(String name, Integer passwd)
PartialDocument search(String title, Integer uid, Integer passwd)
Vector getLogs(Integer uid, Integer passwd)
implements
DocService
users
logs
docs
addDocument(doc)
addUser(name, passwd) returns the user_id
search(title, user) returns a document
getLogs(user) returns the user's logs
...
Log
date
user
doc
...
UserI
User
uid
name
passwd
logs
...
PartialUser
uid
name
Document
title
author
summary
logs
...
PartialDoc
title
author
summary
DocumentI
implements
*

Figure 38. Class graphs for the implementation of the Document Service in a) DJ and b) Java.

The following pieces of code illustrate the DJ and Java implementations.

DJ	JAVA

public class DocService{
 Hashtable docs, users;
 Vector logs;
 	
 // All other methods omitted
 	
 public DVector getUserLogs(Integer uid,
                           Integer passwd){
   DVector ulogs = null;
   User user = (User)users.get(uid);
   if (user != null)
     ulogs = user.get_logs(passwd);
   return ulogs;
 }
}
	public class DocService
	       extends UnicastRemoteObject
	       implements DocServiceI {
	 Hashtable docs, users;
	 Vector logs;
	 int userReaders = 0, docReaders = 0;
	 int userWriters = 0, docWriters = 0;
	
	 public Vector getUserLogs(Integer uid,
	                          Integer passwd)
	        throws RemoteException {
	  User user = (User)users.get(uid);
	  if (user != null) {
	   Vector ulogs = user.get_logs(passwd);
	   if (ulogs != null) {
	    int size = ulogs.size();
	    Vector partulogs = new Vector(size);
	    Log log, partlog;
	    for (int i=0; i < ulogs.size(); i++){
	     log = (Log)ulogs.elementAt(i);
	     partlog = new Log(log.date,
	          new PartialDoc((Document)log.doc),
	          new PartialUser((User)log.user));
	     partulogs.insertElementAt(partlog,i);
	    }
	    return partulogs;
	   }
	  }
	  return null;
	 }
	}

In the Java implementation, the variables `userReaders`, `docReaders`, `userWriters`, `docWriters` are used for synchronization (not necessary for `getUserLogs`). The synchronization of the Java implementation uses the code pattern described in [40], page 133.

DJ	JAVA

coordinator DocService {
  selfex addDocument, addUser;
  mutex {addDocument, search};
  mutex {addUser, search};
}
	public boolean addDocument(Document b){
	  before_write_docs();
	  try { // Implementation of addDocument
	  } finally { after_write_docs(); }
	}
	public Integer addUser(User u){
	  before_write_users();
	  try { // Implementation of addUser
	  } finally { after_write_users(); }
	}
	public PartialDoc search(String title, Integer uid) {
	  before_read_docs_users();
	  try { // Implementation of search
	  } finally { after_read_docs_users(); }
	}
	private synchronized void before_write_docs() {
	  while (docWriters>0 || docReaders>0) {
	    try {wait();}
	    catch (InterruptedException e) {}
	  }
	  ++docWriters;
	}
	private synchronized void after_write_docs() {
	  --docWriters; notifyAll();
	}
	// similar (but not the same) for
	// before_* , after_*

5.1.10. Message Queue

Synopsis: This class is a more sophisticated version of the bounded buffer. It manages a queue of messages, and it provides the services: open, enqueue, enqueueTail, enqueueHead, dequeueHead, dequeueTail, isFull and isEmpty.

Interesting features: It was found on the Web, as part of the ACE system [66]. From its documentation: “The MessageQueue class is a thread-safe message queueing facility, modeled after the queueing facilities in System V StreamS. It is the central queueing facility for messages in the ASX framework.”

Comparison with: A Java implementation extracted from the ACE system.

The class is too big to be shown here: 410 lines in the Java implementation and 323 lines in the DJ implementation. Instead, only its coordinator is shown.

```
coordinator MessageQueue {
  selfex open, enqueue, enqueueHead, enqueueTail,
         dequeueHead, dequeueTail,
         deactivate, activate, isFull, isEmpty;
  mutex {open, enqueue, enqueueHead, enqueueTail,
         dequeueHead, dequeueTail,
         deactivate, activate, isFull, isEmpty};

  cond full = false, empty = true;

  enqueueInternal, enqueueHeadInternal, enqueueTailInternal:
    requires: !full;
    on_exit {
      empty = false;
      if (currentBytes_ == highWaterMark_)
        full = true;
  }
  dequeueHeadInternal, dequeueTailInternal:
    requires: !empty;
    on_exit: {
      full = false;
      if (currentBytes_ == lowWaterMark_)
        empty = true;
    }
}
```

Note: the mutual exclusion constraints follow the original design found in the comparative Java implementation. `isFull` and `isEmpty` don’t need to be selfex. Using COOL’s **`selfex`** and **`mutex`**, the change is trivial, but using plain Java that would require a considerable change in the class implementation.

5.1.11. Analysis

The purpose of the case-studies is to isolate situations that occur frequently in concurrent and distributed systems, and to show that D addresses those situations better than languages that don't provide a separate mechanism for dealing with aspects. The word "better" is obviously ambiguous, and it can mean "faster," "slower," "smaller," or "bigger," depending on the particular issue that is being studied. In this context, the word "better" means "not bigger and more localized." This subsection analyses the ten case-studies under this perspective. For that, four metrics are used: (1) lines of code (LOC), (2) aspectual bloat; (3) number of methods affected by aspect code; and (4) tangling ratio. The analysis of the results is concentrated in §5.1.11.5.

5.1.11.1 LOC

APPLICATION		DJ			ALTERNATIVE IMPLEMENTATION	% SMALLER
#	NAME	JCORE	COOL+RIDL	TOTAL		
1	Bounded Buffer	48	16	64	64	0%
2	Philosophers	43	11	54	58	7%
3	Shape	24	5	29	48	40%
4	Matrix Multiplication	87	8	95	147	35%
5	Graph Traversal	92	12	104	104	0%
6	Assembly Line	108	25	133	152	13%
7	BookLocator/Printer	149	15	164	205	20%
8	Text Editor	215	15	230	232	1%
9	Document Service	246	12	258	369	30%
10	Message Queue	323	25	348	410	15%

Table 2. Lines of Code (LOC) in the implementations of the case-studies. The numbers shown here include the classes and small test clients.

5.1.11.2 *Aspectual Bloat*

This index is given by the following ratio:

$$aspectual\ bloat = \frac{LOC\ in\ Java - LOC\ in\ JCore}{LOC\ in\ Cool\ and\ Ridl}$$

APP # ⇒	1	2	3	4	5	6	7	8	9	10
BLOAT ⇒	1	1	5	N/A	1	2	4	1	10	4

Table 3. Aspectual bloat.

The aspectual bloat, as defined above, can only be used for comparing DJ with Java, or more generally, DX with X. It is a measure of how poorly the component language X, without D, captures the aspect programs in D's coordinators and portals. When the aspectual bloat is 1, it means that, using plain Java, the number of lines of aspect code within the components is the same as the number of lines in the corresponding portals and coordinators. An aspectual bloat much smaller than 1 would mean that D aspect programs were more lengthy than necessary. On the other hand, aspectual bloats much larger than 1 indicate that Java does not capture the aspect code as succinctly as the aspect languages of D.

5.1.11.3 *Methods Affected by Aspect Code*

Neither LOC nor the aspectual bloat capture the real issue for which D was designed, namely the tangling problem. One way of measuring the code tangling is by counting the number of methods affected by aspect code. This metrics does not capture the distribution of the aspect code within the methods themselves; it simply captures the tangling on a method basis and with respect to the number of methods of the application. The aspect code is identified according to the guidelines given in §5.1 (page 161). Table 4 summarizes the results.

APP #	TOTAL # OF METHODS		# OF METHODS AFFECTED		% OF METHODS AFFECTED	
	DJ	OTHER	DJ	OTHER	DJ	OTHER
1	6	6	0	2	0%	33%
2	5	5	0	1	0%	20%
3	6	15	0	6	0%	40%
4	12	7	0	2	0%	29%
5	13	13	0	4	0%	31%
6	18	18	0	6	0%	33%
7	15	21	0	11	0%	52%
8	17	17	7	10	41%	59%
9	25	37	0	17	0%	46%
10	29	29	0	18	0%	62%

Table 4. Percentage of methods affected by aspect code. Constructors also count as methods. Both aspects, that is, synchronization and remote data transfers, are considered. In case 8 (the Line Editor) the code for replication is considered aspect code.

5.1.11.4 Tangling Ratio

In order to capture the tangling of aspect code within the implementations, and its importance with respect to the size of the application (as opposed to the number of methods), the following metrics can be defined:

$$tangling = \frac{\begin{array}{c}\#\,of\ \ transition\ points\\ between\ aspect\ code\ and\ \ functionality\ code\end{array}}{LOC}$$

Transition points are the points in the source code where there is a transition from a non-shadowed area to a shadowed area and vice-versa. The intuition behind it is that they are the points in the program text where there is a “concern switch,” and this intuition applies not only to the aspects dealt with in D, but to virtually every possible concern that we can think of. For the study of D, the concerns are (a) the implementation of the functionality — base concern, (b) the implementation of thread synchronization and (c) the implementation of data transfers between execution spaces. Additionally, one other concern is also studied: code dealing with distributed replication.

For each of the case studies, the program texts were analyzed line by line in order to count the transition points. The identification of aspect code followed the guidelines given in §5.1 (page 161). In the DJ implementations, the transition between the classes, as a whole, and the aspect modules, as a whole, counts as one transition point. Table 5 summarizes the results.

APP #	# OF TRANSITION POINTS						TANGLING	
	SYNCHRONIZATION		DATA TRANSFERS		REPLICATION			
	DJ	OTHER	DJ	OTHER	DJ	OTHER	DJ	OTHER
1	1	12					2%	19%
2	1	6					2%	10%
3	1	32					3%	67%
4	1	14					1%	10%
5	1	12					1%	12%
6	1	34					1%	22%
7	1	14	1	30			1%	21%
8	2	12	1	22	24	24	12%	25%
9	1	26	1	56			1%	22%
10	1	60					0%	15%

Table 5. Tangling ratio. The empty cells mean that the application doesn't deal with the particular concern.

The tangling ratio is an indicator of intermingling. The higher this ratio, the more intermingled the aspect code is within the implementation of the components; the lower this ratio, the more localized the aspect code is. The numerical results are consistent with the visual effect presented in the case-studies. However, there is no absolute quantity being measured here. The percentages shown are only meaningful in relative terms. They are relative to a) the concerns that were being searched; b) the method for counting transition points. Any small variation of these two factors results in drastic changes of the numbers.

5.1.11.5 Analysis of the Results

Table 2 validates the claim that D makes the implementations not bigger than the alternatives. In fact, in six of those cases, the DJ implementations were considerably smaller. However, the actual values shown here should not be taken as a indicators of code reduction in general, since the case-studies are too small.

Table 3 shows that programming the aspects using D is not more lengthy than programming in plain Java, and in some cases it is much shorter. Aspectual bloats much larger than 1, such as in the case of the Shape, the BookLocator/PrintService, the Document Service and the Message Queue, indicate that Java does not capture the aspect code as succinctly as the aspect languages of D. For example, in the case of the Document Service, for each line of aspect code in the DJ implementation there are 10.3 lines of aspect code in the Java implementation.

The results show that aspectual bloats greater than one correspond to a significant code reduction in the DJ implementations. Therefore, the expected code reduction of an application depends on how strong the presence of the aspects is in that application.

Table 4 and Table 5 validate the claim about locality of the aspect code. Table 4 shows that, for the aspects for which D was designed, D completely removes the aspect code from the implementation of the classes. Using plain Java, the number of methods affected by aspect code, even in larger applications, can be very high. Table 5 shows a finer measure of the tangling and puts it in the perspective of the size of the application. Even in this perspective, the effectiveness of D in localizing the aspect code is apparent: the code for addressing the concerns is well localized in modules. The following application-specific observations are supported by the results in Table 4 and Table 5:

- The Shape (case 3), although too small to be seen as a reference, indicates that doing component refactoring purely for purposes of dealing with a particular implementation aspect, distributes the responsibilities of that implementation throughout the code.
- The DJ implementation of the Assembly Line (case 6) is not much smaller than the Java implementation, but the tangling in the former is null, whereas in the latter is considerably high.
- The matrix multiplication (case 4) and the graph traversal (case 5) have similar synchronization needs; their tangling is also similar (the ratio in the matrix multiplication is slightly smaller because the C++ code is very lengthy).
- Case 8 shows that DJ is weak in localizing the replication concern, but is still better than Java because the code for synchronization and remote data transfers is more localized.
- The Java implementation of the Document Service (case 9) can be seen as a refactoring for purposes of remote data transfers, and its effects are also pervasive.

- Case 10 shows how synchronization code can be spread in real situations, and it also shows that COOL is well prepared for coping with it.

The size and number of the case-studies is obviously insufficient for predicting what the results will be in general. For larger applications written in plain Java, we can expect smaller values of the tangling ratio, since larger applications usually have large functional components (GUIs, etc.). For example, the Remote Control Game that comes with the Java RMI distribution, was also studied using the metrics presented here. The application is 1960 LOC, and consists of 27 classes, of which only 3 have distribution intentions; the tangling ratio is 6%, affecting 17% of the methods.

5.2. Performance

This section presents a brief performance evaluation of the framework. The goal of this evaluation is to have an idea of how DJ programs perform with respect to their equivalents written in plain Java. The results shown here were obtained with JavaSoft’s JDK 1.1.3 in a 75MHz Pentium PC with 24Mbytes of RAM. The times were measured with no special hardware.

Chapter 4 described in great detail the relatively simple but relatively naïve implementation of DJ. As a reminder, DJ was implemented as a pre-processor that generates Java programs. The pre-processor introduces additional method invocations in the beginning and in the end of the methods of the original JCore classes. The purpose of those additional invocations is to transfer the control to “aspect objects” that implement the aspect programs, as defined in the coordinators and portals.

There is, therefore, an expected overhead introduced by the framework that comes from (1) the additional method invocations in the output woven classes (2) the non-optimized implementation of the aspect classes and (3) the non-optimized library classes (Appendix D). The results shown here confirm this overhead.

The most basic run-time characteristics, and a comparison with Java equivalents, are given in Table 6. This table shows the results obtained when the test applications are deprived of any functionality. All tests consisted of measuring the elapsed time of 1000 method invocations. Tests 1, 4, 5, 6 and 7 are single-threaded. Test 2 consists of two threads calling the same method, each one making 1000 method invocations. Test 3 consists of two concurrent threads calling two different methods 1000 times: thread t_1 calls a method that suspends it, and thread t_2 calls a method for notifying t_1.

#		DJ		JAVA	
1	COOL	*Single thread calling a selfex method*:	28ms	*Single thread calling a synchronized method*:	7ms
2		*Two threads calling the same selfex method:*	90ms	*Two threads calling the same synchronized method:*	30ms
3		*Two threads calling 2 methods with requires, on_exit:*	13s	*Two threads, calling 2 methods with wait, notification:*	12s
	RIDL	*Calling a remote operation*		*Calling a remote method*	
4		*with no parameters:*	10s	*with no parameters:*	10s
5		*with one gref parameter:*	24s	*with one parameter of type Remote:*	24s
6		*with one copy parameter (object with 4 Integer fields):*	26s	*with one parameter of type Serializable (object with 4 Integer fields):*	30s
7		*with a copying directive that selects 3 out of 4 Integer fields of a parameter:*	35s	*with one parameter with 3 Integer fields that is partially copied from an object of another class (design in case-study 9):*	28s

Table 6. Run-time characteristics of the implementation described in Chapter 4. The times are elapsed times of 1000 method invocations.

In case 1, DJ is 4 times slower than Java. The difference is due both to the additional method invocations to the coordination object and to the computational overhead of the implementation of exclusion constraints (see Chapter 4). The test method body is empty; therefore, any additional instruction, especially method invocations, adds a significant time penalty.

In case 2, DJ is only 3 times slower than Java, and the elapsed time in both cases is more than just double (remember, there are 2000 method invocation here). The reason is that there are occasional conflicts between the two threads, introducing temporary suspensions of the threads. As case 3 shows, suspension is a costly operation in Java. Therefore, the penalty of DJ's exclusion constraints is less noticeable here than in case 1.

In case 3, the results suffer a penalty of two orders of magnitude with respect to case 2. Java's wait/notification mechanism is very costly. Because of that, DJ's coordination overhead is almost unnoticeable.

In cases 4 and 5, there are no differences in performance. When added to the base overhead of Java RMI, RIDL's extra layer of proxies is irrelevant.

In case 6, DJ is faster than Java. This is because DJ uses the Externalizable interface. Objects that implement the Externalizable interface are marshaled faster than those that implement the Serializable interface. The reason why the Java test used the Serializable interface was that objects that implement this interface have default marshaling methods — the programmers don't need to write them by hand. Most RMI applications that pass objects by copy (and *all* the examples that come with the RMI distribution) use the Serializable interface. Therefore, the comparison is fair.

Finally, in case 7 DJ takes 25% more time than Java. In RIDL's copying directives, the calls to the PartCutter object, which occur both at the remote object and at the client, and the non-optimized way in which the IncompleteClass is implemented (Appendix D) are the dominant factor.

These basic performance penalties are less noticeable when the applications do something. For example, the Graph Traversal application (case-study §5.1.5), which executes a very simple recursive function and includes synchronization in every node, is only 2 times slower in DJ than in Java. The matrix multiplication (case-study §5.1.4) is as fast in one as in the other. In the Document Service (case-study §5.1.9), the getUserLogs service also takes the same time (1000 remote invocations take about 4 minutes in each implementation).

There is reason to believe that a number of optimizations will decrease the differences between DJ and Java. A simple optimization consists in compiling the objects away, eliminating the overhead of method invocations that exists in the current target architectures and in the library. For example, for single class coordination, if instead of generating one coordinator class the weaver inlines the coordination code in the woven class, test 1 for DJ takes only 25ms (this number was obtained by weaving the test application by hand using the suggested inline). As this number suggests, there is plenty of space for improving the implementation of DJ.

5.3. Preliminary User-Studies

The previous sections showed that DJ helps localizing two important implementation concerns away from the core functionality of the applications, at a very low cost. But does this help programmers in any way? This section describes what happened when four alpha-users tried to use this technology to write medium-sized distributed applications. None of the alpha-users had previous experience in programming distributed systems.

The overall conclusion is that the alpha-users understood the aspect languages and their interaction with Java very well, and they found the aspect modules very useful. The succinctness and locality of the aspect code, as well as the simplicity with which D addresses synchronization and distribution, played a major role in them being able to assimilate the issues of distributed object systems so quickly. The aspect languages themselves were the basis for the vocabulary with which they designed and discussed some of the distribution issues in their applications.

5.3.1. The Summer Experiment

During the summer of 97, the Aspect-Oriented Programming group at Xerox PARC, together with Professor Gail Murphy from the University of British Columbia, set up an experiment to test the feasibility of DJ as a language framework to be used by ordinary programmers. The four alpha-users were not exactly “ordinary programmers;” they were very talented students that were given the task of writing DJ code, criticizing the result and suggesting improvements. The DJ weaver that they used was not the proof-of-concept described in Chapter 4, but another implementation that was developed by a number of people in the AOP group. The author of this thesis did not participate in this new implementation, other than making sure that it worked according to the most important points of the specifications. Not everything of D was implemented in this version, though. Inheritance of aspect code was not implemented correctly. In particular, RIDL (called RIDL-- in this version) was relatively incomplete, and the copying directives were slightly different from the specification. The following summarizes the major differences between RIDL-- and RIDL. In RIDL--:

- copying directives apply only to the classes of the parameters, and not to classes further down the traversals.
- the selection of the fields is expressed only in positive terms (no bypassing); the programmer must explicitly name which fields should be copied.
- in addition, RIDL-- provides a feature that did not exist in RIDL: the programmer can specify if a given field of the class of the parameter that is being copied is passed by gref.

In spite of the mismatches between documentation and reality, the alpha-users were able to learn the aspect languages and to write two fairly complex distributed applications in a very short period of time.

5.3.2. The Applications

The section presents an overview of the applications written by the alpha-users. The purpose of this section is three-fold. First, it shows the degree of complexity involved in those applications. Secondly, it shows the planning in the development of the applications. Finally, it presents the users’ concrete reaction to the framework, in the form of code and comments.

With respect to this last point, the data shows that the aspect modules are not only easy to understand, but that the users felt compelled to document and justify the aspect code at least as well as the classes. Proper documentation is one of the most important issues for program maintenance and evolution.

5.3.2.1 The Space War

Synopsis: This application is a distributed, highly interactive game, to be played by players in different machines. Each player has a ship that can fire bullets towards other ships. There are also robot-ships that fire randomly, and packets of energy that the ships can pick up.

Interesting features: The application was chosen because it makes heavy use of communication, has (almost) real-time requirements, and the game is asynchronous (events take effect as soon as they are generated).

Limitations: The performance of Java RMI was a major obstacle, and it forced the programmers into a centralized server game, with replication of the shared data in the clients — something that D is known to handle poorly.

Major milestones and timeline: The application was developed in three phases: (1) game running on a single execution space, with a single player; (2) game running on a single execution space, with multiple players, each one having a different keyboard mapping; and (3) game running on several machines, with multiple players. At the end of each phase, the users wrote a report. The learning of DJ and development of the application started in mid-June; the final version was running by July 20. Figure 39 shows the interaction diagram of phase 3.

Final numbers: 19 JCore and Java classes (the components); 2 coordinators; 4 portals; 1500 LOC.

Aspect programs:

- In phase 1, there was no aspect programming. The application ran the basic functionality of the game. However, there were two threads, the user and the robot ships. Some important methods were unsynchronized, and that introduced some inconsistencies when running the application.
- In phase 2, the synchronization aspect was introduced. The Registry class (that became the Universe class in phase 3) was associated with the following coordinator:

```
coordinator Registry {
  mutex {register, unregister, getObjects};
}
```

- Phase 3 involved a fair amount of aspect programming. The coordinators and portals are shown next to Figure 39, as found in the user's source files.

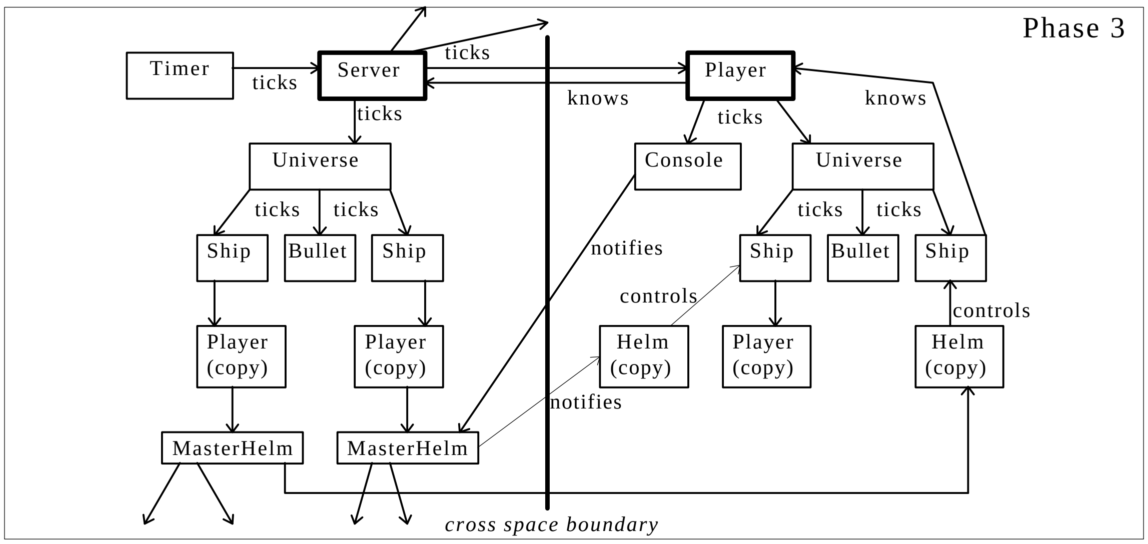


Figure 39. The diagram of the distributed space war application.

```
/* This coordinator synchronizes the
   activities of the game server and all of
   the MasterHelms that also reside on the
   server computer. */
per_class coordinator Server, MasterHelm {
  selfex Server.joinGame, Server.newShip,
         Server.clockTick,
         MasterHelmRotateCCW,
         MasterHelmRotateCW,
         MasterHelm.stopRotating,
         MasterHelm.thrustOn,
         MasterHelm.thrustOff,
         MasterHelm.fire;
  mutex { Server.joinGame, Server.newShip,
          Server.clockTick,
          MasterHelmRotateCCW,
          MasterHelmRotateCW,
          MasterHelm.stopRotating,
          MasterHelm.thrustOn,
          MasterHelm.thrustOff,
          MasterHelm.fire
        };
}
/* Broadcasting of the ship control events
   (MasterHelm methods) as well as
   Server.clockTick must be synchronized in
   order for all players to (1)see the same
   stream of events and (2) keep their
   universes in sync. We achieve this by
   requiring that only one event can be
   broadcast at a time, and it must be
   broadcast to all sites. (2) is met in
   MasterHelm.<event> and Server.clockTick
   method code. (1) is guaranteed by the
   following synchronization conditions.
   No event broadcast should occur while
   some player(s) has an inconsistent
   Universe which is the same case during
   Server.joinGame and Server.newShip
   calls. We achieve this by simply
   stopping clockTick and message delivery
   processing (mutex) for the duration of
   those calls. */
```

```
/* This coordinator ensures that getObjects
   always returns a consistent set of
   objects currently registered with the
   Universe
 */
coordinator Universe {
  mutex {register, unregister, getObjects}
}
```

The two coordinators for the space war.

```
/* Player is called remotely by Server
   only
 */
portal Player {
  /* Server calls this method to inform the
     player that a new ship is being added
     to the Universe, so that the player
     inserts the ship into its copy of the
     Universe, creates a Helm for the ship
     and returns it by gref, so that a
     MasterHelm on the server can control
     the copy of the ship remotely
   */
  Helm addShip(Ship ship) {
    ship: copy;
    return: gref;
  }
  /* Server calls this method to assign a
     new MasterHelm to a player joining the
     game. Since MasterHelms only reside on
     Server, -m- is passed by gref to
     create a remote link between the
     Player and the MasterHelm it uses to
     brodcast control events
   */
  void setMasterHelm(MasterHelm m) {
    m: gref;
  }

  /* Server calls this method to create and
     pass to the Player a copy of the
     current Universe
   */
  void setUniverse(Universe u) {
    u: copy;
  }

  /* This is a remote copy method. It
     copies only those fields of Player
     that the Server needs in a copy of
     Player that it will pass around with
     the Universe. That copy needs to
     reference the same MasterHelm as the
     original Player does, hence
     gref: Player.masterHelm
   */
  Player remoteClone() {
    return: { copy: Player.score,
                    Player.name,
                    Player.playerID;
              gref: Player.masterHelm;
            }
  }

  /* A message that the Server sends to
     players remotely every clock tick
   */
  void clockTick();

  /* These are workarounds for the bug that
     doesn't allow copying non-public
     fields remotely
   */
  MasterHelm getMasterHelm() {
    return: gref;
  }
  String getName() {}
}
```

```
/* Server is called remotely by Player
   only
*/
portal Server {
  /* When player joins the game, it passes
     itself to the server, so that the
     server can establish a remote link
     to the player and send it tick events
  */
  void joinGame(Player newPlayer) {
    newPlayer: gref;
  }

  /* When player requests a new ship
     is passes itself by global ref
     again, so that the server can
     update the player's universe */
  void newShip(Player player) {
    player: gref;
  }
}
```

```
/* MasterHelm is called remotely by Helms
   (to submit an event for brodcast) and
   Players (to register an event listener)
 */
portal MasterHelm {
  /* addDrone is called by Player to
     register a remote listener (Helm) to
     the events this MasterHelm broadcasts.
     The purpose of this call is to
     establish a remote link between
     this MasterHelm and the Helm at the
     remote site */
  addDrone(Helm helm) {
    return: gref;
  }

  /* These are remote declarations for
     Helm events */
  void rotateCW() {}
  void rotateCCW() {}
  void stopRotating() {}
  void thrustOn() {}
  void thrustOff() {}
  void fire() {}
}
```

```
/* Helms receive simple events in the form
   of remote method calls from their
   respective MasterHelms
 */
portal Helm {
  /* This method is called remotely when
     the user requests a new ship. The ship
     comes by copy to become a part of this
     user's copy universe
   */
  void setShip(Ship s) {
    s: copy;
  }
  /* Called by MasterHelm to notify this
     Helm about user actions
   */
  void rotateCW() {}
  void rotateCCW() {}
  void stopRotating() {}
  void thrustOn() {}
  void thrustOff() {}
  void fire() {}
}
```

The four portals for the space war.

5.3.2.2 The Space War - Java and Sockets

Synopsis: The space war was re-implemented using plain Java and sockets.

Interesting features: The users wanted to know how different the application would look like.

Limitations: There aren't any significant performance improvements by using sockets. The design of the application was roughly the same.

Milestone: The development of a couple of classes for handling messages and transform them in application objects (and vice-versa). The manual weaving of synchronization constraints.

Timeline: One additional week. This time was used solely for the design and implementation of the interaction with sockets; the design of the application was re-used.

Final numbers: 23 Java classes; 2300 LOC — 35% bigger than the DJ version.

5.3.2.3 Distributed Library System

Synopsis: Library objects store books. There may be several collaborating library objects in an execution space, and there are several of these distributed across the network. Reader objects make queries to a library, looking for books. If some of the books are not found there, the library redirects the query to its neighbors. The query goes from library to library, and it keeps track of its route. This goes on until either the books are found or all the libraries have been searched. Figure 40 shows the interaction diagram.

Interesting features: This application is a mixture of a search engine and a network agent.

Timeline: Two weeks: one week to design and implement the basic functionality with a minimal search engine in non-distributed mode; another week to design and implement the distribution issues and to add more search capabilities.

Final numbers: 13 classes, 3 coordinators, 4 portals.

Aspect programs: The aspect programs are shown next to the interaction diagram.

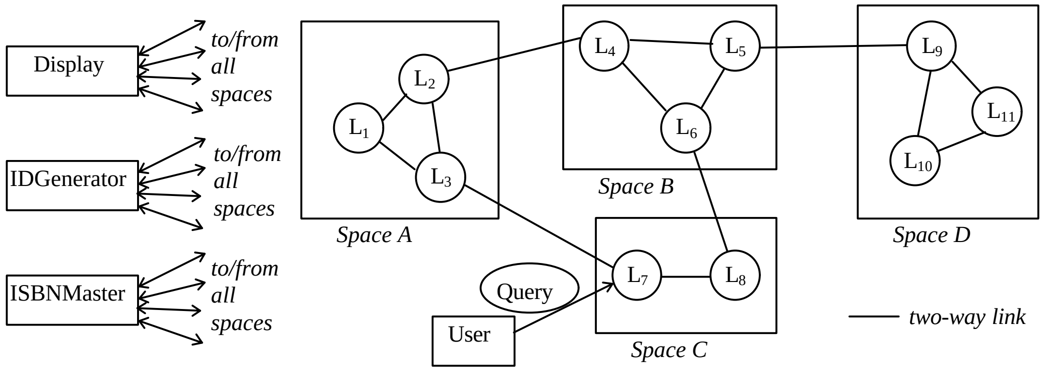


Figure 40. Interaction diagram for the distributed library system.

```
/* synchronizing the displayed network of
   libraries and searches
 */
coordinator Display {
  selfex addLink, removeLink;
  mutex {addLink, removeLink};
}
```

```
/* the newISBN must be synchronized, since
   it returns an ISBN number that is unique
   system-wide
 */
coordinator {
  selfex newISBN;
}
```

```
// the getID method must be synchronized,
// since it returns a query number
// that is unique system-wide
coordinator QueryNumGenerator {
        selfex getQueryNum;
}
```

```
portal Library {
  /* This method is called remotely by
     readers and libraries. Ideally, when
     the query is returned to the reader,
     only the book information should be
     copied.
   */
  Query search(Query query) {
    query: copy;
  }

  /* Called by the Display */
  long numBooks() {}
  long numLibs() {}

 /* Called by the drivers during setup
     (locally) and readers, sometimes
     remotely
   */
  void addBook(Book b) {
    b: copy;
  }

  /* Called by drivers during setup */
  void addLibrary(Library l) {
    l: copy;
  }
  Integer getID() {}
}
```

```
portal Display {
  /* Display needs only to know about the
     ID of the objects, so it may hash it.
   */
  void addLink(Object nodeA, Object nodeB){
    nodeA: {copy Library.id, Reader.id;}
    nodeB: {copy Library.id, Reader.id;}
  }

  void removeLink(Object nodeA,
                  Object nodeB) {
    nodeA: {copy Library.id, Reader.id;}
    nodeB: {copy Library.id, Reader.id;}
  }
}
```

```
portal IDGenerator {
  int getID() {}
  int getIDBlock(int size) {}
}
```

```
portal ISBNMaster {
  ISBN newIsbn() {
    return: copy;
  }

  // for debugging…
  long getNumBlocks() {}
}
```

5.3.3. Alpha-Users' Reports

By the end of the summer, the alpha-users were asked to individually report their experience in using DJ. Because the user's views are of such exceptional quality, their final reports and their answers to a survey are given in Appendix E. A brief summary of the reports and survey follows.

All users reported finding COOL and RIDL very easy to use. They all reported no difficulty in understanding the effect of the aspect code on the component code. They also reported that the aspect languages greatly eased the burden of programming the distribution issues for which those languages were designed: the languages' simplicity made the aspect modules easy to write, understand and modify. However, the users reported that there were still distribution issues in the applications they wrote that were not well captured by D, namely replication and distributed coordination. And they did indicate that we cannot expect aspect modules to capture intent.

Moreover, the users pointed out some conceptual problems and suggested new features that both show their deep understanding of the framework and how D can evolve. Most feature requests were related to RIDL. They include the naming of traversal directives, sender-side transfer specifications, object-sensitive traversal directives, automatic deduction of how much is needed to pass to support a given interface (inference), distributed coordination, support for replicated objects, and support for controlling the scheduling of threads.

5.4. Final Remarks

The locality of aspect code and its separation from the functional components is the most important design feature of D. The reason for wanting the locality of aspect code and its separation from the functional modules in the first place, is that it can simplify the development and evolution of the applications. When programming and evolving distributed systems using an object-oriented language, programmers can concentrate on different implementation issues at different times; whenever an aspect needs to be programmed, changed, or explained to other people, it is better to have it in one module than to have it spread all over the methods and even across classes, intermingled with the rest of the code. The results show that D achieves the desired locality and separation effectively and at a very low cost. The data exposes the benefits and costs in a number of promising results.

There are, however, some important goals of D whose fulfillment is still to be proven. How well does this separation scale? With respect to plain Java, how much more understandable and clear are DJ programs when they are 10K and 100K lines of code? Are non-trivial DJ programs always smaller than their Java equivalents? In what ways do the aspect modules help programmers in developing real world applications? Can they write them much faster? Does the division of labor in the source code help the division of labor in a working team of programmers? Does the vocabulary introduced by D capture important design issues in ordinary software development practices?

Based on the results, there is reason to believe that D can be smoothly integrated with the existing software practices, and that such integration makes programs easier to write, understand and maintain. The benefits of clarity of the source code should be higher when programs are large and complex. Although D was designed with this goal in mind, this chapter does not provide enough data to validate such generalization. No claims of that magnitude can be made before D is disseminated and studied in industrial settings.

Chapter 6

Conclusions

“This is what I mean by "focusing one’s attention upon a certain aspect"; it does not mean completely ignoring the other ones, but temporarily forgetting them to the extent that they are irrelevant for the current topic. Such separation, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts that I know of.”

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

6. Conclusions

6.1. Summary

The design and implementation of distributed systems requires addressing a number of issues that do not arise in non-distributed systems. Two of the most important 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 effectively 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, 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.

6.2. Contributions

This thesis makes a number of contributions that can have an immediate impact on the design, implementation and documentation of distributed applications. It also makes contributions that, in the longer term, may affect the design of programming languages.

The concrete and most immediately useful contribution is DJ, the integration of D with Java™. This dissertation described one implementation of DJ that can be easily reproduced, either partially or in its entirety, and that performs within acceptable bounds.

But the most important contribution of this work is the design of an enforceable support for programming thread synchronization and application-level remote data transfers in separate from the implementation of the components, while using an ordinary object-oriented language for programming the components. The two new kinds of modules in D, coordinators and portals, are add-ons with respect to the class modules, and they control the behavior of the classes in concurrent and distributed environments. Coordinators and portals do not pursue the goal of being abstract descriptions that can be used by many different classes; instead, they simply aim for an effective separation of concerns. And, unlike reflective approaches, this separation is enforced by aspect-specific languages.

Systems like CORBA and Java RMI use the object-oriented composition mechanisms (inheritance and type implementation, respectively) to integrate remoteness with the OOPL. RIDL breaks away from the state-of-the art type-based remote interaction for distributed object systems by providing new abstraction and composition mechanisms that support a better division of labor and that capture a lot more of the issues involved in remote interaction than types can ever capture. These new mechanisms are equally well integrated with the OOPL. The declarative nature of the language makes it very easy to write relatively complex data transfer schemes.

Based on the study of the synchronization needs of concurrent object systems, COOL provides a small number of powerful language constructs that address those needs in a succinct way. The selfex and mutex declarations capture, to a large extent, the most common needs of multi-threaded object-oriented programs.

Coordinators and portals also improve the documentation of the applications. When using other languages, the tangling between functionality and aspect code makes it extremely difficult to understand *what* are the concurrency control schemes and data transfer protocols affecting those components. Coordinators and portals isolate and describe those issues. A positive side-effect of this has to do with user-written documentation (i.e. comments) that explains *why* those issues are programmed in a certain way. When using other languages, the documentation about the implementation decisions related to the distribution issues is frankly bad, and, in most cases, it simply does not exist. However, in the applications that were written in DJ, programmers were compelled to explain the decisions in the implementation of the aspect modules at least as well as the decisions in the implementation of the classes.

Another contribution of this thesis is the extensive study of code tangling with respect to the current programming practices and to the existing programming languages. This study provides the background for understanding some of the software engineering problems that programmers are faced with when developing distributed applications, and some of the solutions that they can apply. This study is a methodological research that sets up the motivation for “better” languages that, as Wulf puts it, “permit and even encourage the use of “good” program structures” [75]. Similar studies can be done for issues other than synchronization and remote data transfers.

In the longer term, D and this thesis suggest interesting directions for language design. The aspect languages were designed without having to modify or extend the component language. This is considerably different from all the previous approaches, where either completely new languages or extensions to the existing ones have been proposed. The approach taken here has one major benefit. The classes are programmed without explicit commitments to the aspect modules; therefore a class can be tested for what it does, independently of whether it will be used with D or not. The preliminary alpha-users study showed that, although the programmers had distribution in mind from the beginning, they first used and tested their classes in a non-distributed environment, and only then added coordinators and portals. The non-intrusion of the aspects into the component modules seems to be a useful design decision: programmers can simply “plug-in” or “plug-out” the aspect modules whenever they want.

In object-oriented languages, subclassing already established a relationship between modules that is not of the type client/provider. D went one step further, and showed that it is possible to establish many other kinds of interfaces between modules that are quite different from the normal client interfaces, but that are, nonetheless, extremely useful for structuring programs. This suggests one language design direction in which aspects are identified and aspect languages are added.

6.3. Future Work

A language framework very similar to DJ was implemented at the Xerox Palo Alto Research Center, and this implementation has been in alpha-usage since June of 1997. We plan to pursue the development and improvement of this framework, in many ways.

First, we intend to fix some design problems of D that were already detected. In RIDL, we need to include nested copying directives and clarify the relation between the components' subclassing relations and the types of the parameters that are passed.

Secondly, we intend to improve and extend the existing aspect languages. Some feature requests made by the alpha-users include support for replication, timeouts and new parameter passing semantics. One issue that will be further researched is the connection between types and RIDL. Another issue that will be investigated is the possibility of making the aspect languages more imperative than what they are now. Currently, they are mostly declarative, with the exception of the guarded suspension/notification in COOL. But it may be possible to add imperative features that give more flexibility to the languages, in particular RIDL. A third issue that needs urgent attention is error handling.

We intend to identify other aspects and design new aspect languages following the methodology in this thesis. That is, first we will look at many more Java programs in order to identify other kinds of code tangles; then we need to understand what are the good program structures that minimize those tangles — this gives us hints for what kinds of things an aspect language needs to be able to express; next, we design an aspect language and provide a weaving engine for it.

Finally, at the implementation level, the current weaver is strongly coupled with the two aspect languages. We plan to develop a generic weaver that can easily process the new aspect languages that will be designed.

6.4. Conclusion

This thesis has demonstrated that aspect-oriented programming is possible and useful. By separating the program modules into aspects and components, important issues that would otherwise be diluted in the program texts become visible and with well-localized effects. The particular aspect-oriented framework, D, has proven to be useful for small to medium size programs, and there is good reason to believe that its benefits will be even greater for larger, more complex programs.

Personally, I have learned that modularity means a lot more than dividing designs into components and implementations into classes. Better modularity can be achieved by including new kinds of modules that compose in new kinds of ways with each other and with the components, as long as those modules align well with issues in the design.

As important as the solution presented here, are the questions that this thesis raises. What other issues can be thought of as aspects? Are there other domains in which aspects can be useful? Is there a systematic way of defining aspect interfaces? How can we debug aspect-oriented programs and what kinds of visual programming interfaces would be appropriate? How do aspect modules scale? Would it be of any use to divide a software development team into component and aspect experts? The experience gained during the design and implementation of D and the many discussions it generated, gave me and my two groups, the AOP group at PARC and the Demeter group at Northeastern University, precious insights that will allow us, and the rest of the research community, to investigate this idea even further.

Appendix A. Syntax

I JCore

The only syntactic difference between JCore and Java is the keyword `synchronized`, which does not exist in JCore.

II COOL

Note: some of the productions are defined in §IV.

COOLTranslationUnit:
    CoordinatorDeclaration

CoordinatorDeclaration:
    Granularity$_{opt}$ **`coordinator`** *ClassName_CommaList* *CoordinatorBody*

Granularity: **`per_class`**

CoordinatorBody:
    `{`
      CondVarDeclaration_List$_{opt}$
      VariableDeclaration_List$_{opt}$
      SelfExclusiveMethods$_{opt}$
      MutuallyExclusiveMethodSet_List$_{opt}$
      MethodManager_List$_{opt}$
    `}`

CondVarDecl:
    `condition` *VariableDeclarator_CommaList* `;`

VariableDeclarator:
    Identifier = *CondVarInitializer* |
    Identifier `[]` = *CondArrayInitializer*

CondVarInitializer:
    `true` | **`false`**

CondArrayInitializer:
    `{` *CondVarInitializer_CommaList* `}`

VarDeclaration:

PrimitiveType VariableDeclarator_CommaList ;

VariableDeclarator:
Identifier = *VarInitializer* |
Identifier `[]` = *ArrayInitializer*

VarInitializer:
Expression

ArrayInitializer:
`{` *VarInitializer_CommaList* `}`

SelfExclusiveMethods:
`selfex` *QualifiedName_CommaList* `;`

MutuallyExclusiveMethodSet:
`mutex` `{` *QualifiedName_CommaList* `};`

MethodManager:
QualifiedName_CommaList `:`
$Requires_{opt}$ $OnEntry_{opt}$ $OnExit_{opt}$

Requires:
`requires` *CondVarExpression* `;`

CondVarExpression:
VarRef |
Not CondVarExpression |
`(` *CondVarExpression* `)` |
CondVarExpression ConditionalOp CondVarExpression

OnEntry:
`on_entry` `{` *Statement_List* `}`

OnExit:
`on_exit` `{` *Statement_List* `}`

Statement:
IfStatement |
AssignStatement

IfStatement:
`if` *Expression* `{`*Statement_List* `}` |
`if` *Expression* `{`*Statement_List* `}` **`else`** `{`*Statement_List* `}`

AssignStatement:
Identifier = *Expression* `;`

III RIDL

Note: some of the productions are defined in §IV.

RIDLTranslationUnit:
PortalDeclaration

PortalDeclaration:
`portal` *ClassName PortalBody*

PortalBody:
`{`
RemoteOperation_List
DefaultTransfers$_{opt}$
`}`

DefaultTransfers:
`default``:` *TransferableType_List*

RemoteOperation:
ReturnType MethodName `(` *Parameter_CommaList*$_{opt}$ `)`
RemoteOperationBody$_{opt}$ `;`

ReturnType:
Type |
`void`

Parameter:
Type ParamName

ParamName:
Identifier |
Identifier [ ]

RemoteOperationBody:
`{` *ObjectTransferSpec_List* `}`

ObjectTransferSpec:
ObjectName `:` *Mode* `;`

ObjectName:
Identifier |
`return`

Mode:

`gref` |
`copy` *CopyDirective$_{opt}$*

TypeTransferSpec:
ReferenceType `:` *Mode* `;`

CopyDirective:
`{`
SelectionDirective_List$_{opt}$
`}`

SelectionDirective:
ClassSelector SelectionPrimitive VariableSelector_CommaList;

SelectionPrimitive:
`only` |
`bypass`

ClassSelector:
ClassName

VariableSelector:
VariableName |
`all.`*TypeName*

IV General

Type:
PrimitiveType |
ReferenceType |
ArrayType

PrimitiveType:
`boolean` | **`byte`** | **`char`** | **`short`** | **`int`** | **`long`** | **`float`** | **`double`** |
`String`

ReferenceType:
Name

ArrayType:
PrimitiveType [ ] |
Name [ ] |
ArrayType [ ]

Name:
Identifier |
FullQualifiedName

ClassName:
    Identifier

VariableName:
    Identifier

QualifiedName:
    ClassName **.** *VisibleElementName*

VisibleElementName:
    Identifier | *

FullQualifiedName:
    Name . Identifier

Note: the following production accepts more than it should. It accepts, for example,

“3+(x == false)”

which is obviously unwanted. The production is given in such general form because this form is much simpler and shorter than the productions that would be necessary to disambiguate those situations. In any case, the ambiguities that this simplification accepts will be caught by the Java compiler after translation.

Expression:
    Literal |
    VarRef |
    UnaryExpression |
    (*Expression*) |
    Expression ArithmeticOp Expression |
    Expression ConditionalOp Expression |
    Expression RelationalOp Expression |
    Expression EqualityOp Expression

Literal:
    (as in Java’s grammar, [23] pages 19-27)

VarRef:
    Identifier |
    ArrayRef |

ArrayRef:
    Identifier[ArrayIndex]

ArrayIndex:

Identifier |
IntegerLiteral |

UnaryExpression:
AccessExpression ++ |
AccessExpression -- |
++ *AccessExpression* |
-- *AccessExpression*

ArithmeticOp:
`+` | `-` | `*` | `/` | `%`

ConditionalOp:
`||` | `&&`

RelationalOp:
`<` | `>` | `<=` | `<=`

EqualityOp:
`==` | `!=`

Not:
`!`

DJ: Primer

Version 0.3
June 1997

by Cristina Videira Lopes

XEROX
Palo Alto Research Center
3333 Coyote Hill Road
Palo Alto, CA 94304

I Introduction

DJ is an object-oriented language framework designed for facilitating the development and maintenance of concurrent and distributed applications. It uses the aspect oriented programming approach [37] to allow the code for the basic functionality of a distributed application to be written without having to explicitly deal with remote interactions and synchronization. Separate code deals with those issues.

This guide describes how to program in DJ. While using this guide, be sure to have two other documents at hand: “DJ: Framework Specification” and your favorite Java manual.

1 Overview of DJ

DJ consists of three relatively independent languages:

1) one full object-oriented language, JCore, for programming functional components;
2) one small language, COOL, for programming the aspect of thread synchronization over the execution of the components; and
3) one small language, RIDL, for programming the aspect of remote interactions between components.

JCore is Java™ 1.0 minus the following:

1) the keyword `synchronized`
2) the `Object` methods `wait`, `notify` and `notifyAll`
3) method overloading (method overriding is allowed!)

Everything else of Java, including the libraries, is available to DJ programs. However, the aspect modules of DJ (coordinators and portals) can only be applied to user programmed JCore classes, not to Java library classes.

2 Developing Programs in DJ

While concurrency and distribution should be taken into account right from the early stages of the *design* of a DJ application, the *implementation* of the application usually includes a number of iterations on the coding of the components and the aspects, alternately. First, the programmer should concentrate on getting the *functionality* right, independently of concurrency or distribution. That is, what the JCore classes do, the operations they provide, the composition between classes, etc. Much of the testing of the functionality can be done at this stage, without concurrency or distribution.

Then the programmer introduces the necessary concurrency (i.e. creation of threads) and distribution (i.e. many virtual machines), and with them the aspect programs in COOL and RIDL. In most cases, the issues related to concurrency should be introduced and tested before distributing the application over the network. After the aspect programs are introduced, it may be necessary or desirable to make small modifications in the implementation of the classes. (Beware of premature optimizations, though!).

As the functional and operational requirements of the application evolve, the process repeats: 1) think about the implementation of the classes, as independent as possible from concurrency and distribution issues; 2) introduce those issues and modify/extend the aspect programs; and 3) tune the classes, if necessary.

II Concurrency: Programming in COOL

COOL provides the means for dealing with mutual exclusion of threads, synchronization state, guarded suspension and notification, in relative separation from the method code. Coordination programs consist of a set of coordinator modules which are associated with the classes on a name basis. A coordinator may coordinate more than one class at the same time. The smallest blocks for synchronization are the methods.

1 Basic Example: The Bounded Buffer

This is the classical example of synchronization in concurrent systems. It models the situation in which a number of concurrent clients tries to access a limited shared resource. A bounded buffer maintains a fixed array of elements, and its clients concurrently put and take elements from it. So, a basic interface to these objects is:

```
public interface BoundedBuffer {
  public int capacity(); // invariant: capacity >= 0
  public int count();    // invariant: 0 <= count <= capacity
  public void       put(Object x) throws Full;
  public Object     take() throws Empty;
}
```

There are many ways of implementing bounded buffers. Let's start by the simplest one, which doesn't even consider issues of synchronization:

```
public class BoundedBufferV1 {
  private Object array[];
  private int takePtr = 0, putPtr = 0;
  protected int usedSlots = 0, size;

  BoundedBufferV1(int capacity) throws IllegalArgumentException {
    if (capacity <= 0) throw new IllegalArgumentException();
    array = new Object[capacity];
    size = capacity ;
  }

  public int count() { return usedSlots; }
  public int capacity()  { return size; }

  public void put(Object x) throws Full {
    if (usedSlots == size) throw new Full();
    array[putPtr] = x;
    putPtr = (putPtr + 1) % size;
    usedSlots++;
  }
  public Object take() throws Empty {
    if (usedSlots == 0) throw new Empty();
    Object old = array[takePtr];
    takePtr = (takePtr + 1) % size;
    usedSlots--;
    return old;
  }
}
```

A client of the bounded buffer may be something like:

```
public class Client implements Runnable {
  BoundedBufferV1  buf;
  Random           sleeptime=new Random ();
  boolean          done=false;
  boolean          shouldIput=false; // Indicates client is a producer

  public Client(BoundedBufferV1 b, boolean putp){
    buf=b;
    shouldIput=putp;
  }

  public void run(){
    while (!done){
      int x=sleeptime.nextInt();
      try {
        if (shouldIput) buf.put(new Integer(x)); // put if producer
        else buf.take();                         // else take
      } catch (BufferException e) {};
      try{
        Thread.sleep(Math.abs(sleeptime.nextInt() % 500));
      }catch(InterruptedException e){};
    }
  }
  public void finish(){
    done=true;
  }
}
```

If there is only client, then it sequentially executes the loop of random puts and takes, and everything works fine, except for occasional exceptions for when the buffer is full or empty.

When there are multiple concurrently clients inserting and removing objects from the same buffer, however, those requests need to be synchronized, because the internal variables of the bounded buffer are modified in the implementation of `put` and `take`; therefore there may be temporary inconsistencies within the buffer while these methods are running. To synchronize the access to the execution of the methods, we use COOL. For this particular implementation of the bounded buffer, we can define a coordinator as follows:

```
coordinator BoundedBufferV1 {
  selfex {put, take};
  mutex {put, take};
}
```

This declaration establishes an association between class `BoundedBufferV1` and this coordinator module. The body of this coordinator states that

- both methods `put` and `take` are self-exclusive, that is, if two threads try to execute `put` at the same time, one of them waits until the other is finished; the same for `take`.
- methods `put` and `take` are also mutually exclusive, that is, if one thread tries to execute `put` and another thread tries to execute `take` at the same time, one of them must wait until the other is finished.
- since count and capacity are not mentioned, their execution is not synchronized, and they always get executed, no matter what other executions there may be on the object.

The above coordinator, then, takes care of synchronizing the threads that try to execute the methods of the bounded buffer, guaranteeing the proper exclusion constraints.

But in the concurrent environment, if the buffer is full or empty we can make the threads wait until the buffer is not full or not empty, respectively — since other threads may eventually remove or insert, respectively, elements from the buffer. In COOL, this is done through the use of method managers that establish pre-conditions and modify special state that belongs to the coordinator. We can enhance the above coordinator as follows:

```
coordinator BoundedBufferV1 {
  selfex {put, take};
  mutex {put, take};
  condition empty = true, full = false; // coordination state with
                                        // initial values
  put:  requires !full;  // pre-condition; wait if false
        on_exit {
          // as soon as put finishes, change state accordingly
          if (empty) empty = false;
          if (usedSlots == size) full = true;
        }
  take: requires !empty; // pre-condition; wait if false
        on_exit {
          // as soon as take finishes, change state accordingly
          if (full) full = false;
          if (usedSlots == 0) empty = true;
        }
}
```

Note that coordinators have access to the variables defined in the classes they coordinate (in this case, `usedSlots` and `size`). But that's as much as they can do directly on the classes. They cannot assign values to those variables; and they cannot invoke methods.

Note also that this particular coordination strategy guarantees that no thread will ever remove objects from an empty buffer or insert objects on a full buffer. Therefore, assuming that the class will always be used along with its coordinator, on a later phase we can go back to the implementation of the class and remove the guard tests. In general, coordinators *add* constraints to the execution of the methods; hence, some failure states in a class implementation may be eliminated when a coordinator is associated with that class. The rule, however, is that you shouldn't do these optimizations unless you are 100% sure that the coordinator will always be there.

2 Subclassing and COOL

Coordinators are inherited by subclasses. Consider the following subclass of `BoundedBufferV1` defined in §1, which stores the objects in a linked list, instead of an array:

```
public class BoundedBuffer2 extends BoundedBufferV1 {
  private ObjectList olist;   // the elements, implemented as a list
  private Object putPtr, takePtr; // “pointers”
  public BoundedBuffer2 (int size) {
    olist = new ObjectList(); size = capacity;
    takePtr = putPtr = olist;
  }
  public void put(Object o) { // override the implementation
    putPtr.insert(o);
    putPtr = putPtr.next;
    ++usedSlots;
  }
  public Object take() {  // override the implementation
    Object old;
    old = takePtr.remove();
    takePtr = takePtr.next;
    --usedSlots;
    return old;
  }
}
public class ObjectList {
  Object o;
  public ObjectList next;
  public void insert (Object x) {
    this.o = x; this.next = new ObjectList();
  }
  public Object remove() {
    return this.o;
  }
}
```

The coordinator of the `BoundedBufferV1` applies to all of its subclasses, namely to `BoundedBufferV2`. For methods that are simply inherited but not redefined, it is relatively easy to understand the reason why: when instances of `BoundedBufferV2` are invoked for `capacity` or `count`, the actual method invoked is the one defined in the superclass. However, `put` and `take` are overridden here. Nevertheless, their execution is affected by the coordination strategy defined in the coordinator of the superclass, because they match the *name*. That is, coordinators manage method names, not the methods themselves.

But class inheritance does not necessarily imply that the coordination of a superclass is appropriate for the subclass. Coordination is relatively dependent on the implementation of the classes. Consider this other implementation:

```
public class BoundedBufferV3 extends BoundedBufferV1 {
  private Object array[];
  private int takePtr = 0, putPtr = 0;
  protected emptySlots;

  BoundedBufferV1(int capacity) throws IllegalArgumentException {
    if (capacity <= 0) throw new IllegalArgumentException();
    array = new Object[capacity];
    emptySlots = size = capacity ;
  }

  public int count() { return usedSlots; }
  public capacity()  { return size; }

  public void put(Object x) {
    do_put(x);
    increment_usedSlots();
  }
  public Object take() {
    Object x = do_take();
    increment_emptySlots();
    return x;
  }
  private void do_put(Object x) throws Full {
    if (emptySlots == 0) throw new Full();
    array[putPtr] = x;
    putPtr = (putPtr + 1) % size;
    emptySlots--;
  }
  private Object do_take() throws Empty {
    if (usedSlots == 0) throw new Empty();
    Object old = array[takePtr];
    takePtr = (takePtr + 1) % size;
    usedSLots--;
    return old;
  }
  private void increment_usedSlots()  {usedSlots++;}
  private void increment_emptySlots() {emptySlots++;}
}
```

The inherited coordination is still valid for this implementation (i.e. we can use it, and nothing bad will happen). However, a careful analysis of this code leads to the conclusion that the granularity of the synchronization can be finer for this class. There is no need to synchronize on the top `put` and `take`, but rather we can synchronize the private methods in a more efficient way:

```
coordinator BoundedBufferV3 {
  selfex {do_take, do_put, increment_emptySlots, increment_usedSlots};
  mutex  {do_take, increment_usedSlots};
  mutex  {do_put, increment_emptySlots};

  do_take: requires !empty;
        on_exit { if (full) full = false; }
  do_put:  requires !full;
        on_exit { if (empty) empty = false; }
  increment_emptySlots:
        on_exit { if (emptySlots == size) empty = true; }
  increment_usedSlots:
        on_exit { if (usedSlots == size) full = true; }
}
```

Associating this coordinator with class `BoundedBufferV3` overrides the inherited coordinator. Therefore in this class hierarchy, `BoundedBufferV1` and `BoundedBufferV2` are affected by `BoundedBufferV1`'s coordinator, but `BoundedBufferV3` is affected by its own coordinator.

3 The Dinning Philosophers: the Classical Monitor Solution

This is the other classical example of synchronization. It models the situation in which a number of concurrent clients tries to access a number of shared resources, and can only proceed if *all* the necessary resources are available. A number of philosophers are sitting around a table, eating spaghetti and thinking, alternately. Between each pair of neighbor philosophers there is exactly one fork. Before eating, each philosopher picks both left and right forks, and after eating he puts down the forks. The synchronization, then, is that each philosopher can only start eating if both left and right forks are free (that is, if their neighbors are not eating). So, the functionality can be implemented by the following class:

```
public class Philosopher implements Runnable {
  // the global set up
  static final int max = 5;
  static protected Fork forks[max];
  static protected int count = 0;
  // for each philosopher
  protected int      mynumber;
  protected Fork     left, right;
  protected Random   sleeptime = new Random ();
  protected boolean  done = false;
   Philosopher() throws MaxPhilosophers {
    if (count == max) throw new MaxPhilosophers();
    left = forks[count];
    right = forks[(count + 1) % max];
    mynumber = count++;
  }

  public void run() {
    while (!done) {
      think();
      eat();
    }
  }

  public void finish() {done = true;}

  private void think() {
    int x=sleeptime.nextInt();
    try{
      Thread.sleep(Math.abs(sleeptime.nextInt() % 500));
    } catch(InterruptedException e){};
  }

  private void eat() {
    // do something with the forks
    int x=sleeptime.nextInt();
    try{
      Thread.sleep(Math.abs(sleeptime.nextInt() % 500));
    } catch(InterruptedException e){};
    // do something else with the forks
  }
}
```

We need to synchronize the access to method eat. Any book on concurrent systems presents at least the monitor solution. In COOL, that solution looks like:

```
per_class coordinator Philosopher {
  condition OKToEat[] = {true,true,true,true,true};
  boolean eating[] = {false,false,false,false,false};

  eat: requires OKToEat[mynumber];
       on_entry {
         OKToEat[(mynumber+1) % max] = false;
         OKToEat[(mynumber-1) % max] = false;
         eating[mynumber] = true;
       }
       on_exit {
         if (eating[(mynumber+2) % max] == false)
           OKToEat[(mynumber+1) % max] = true;
         if (eating[(mynumber-2) % max] == false)
           OKToEat[(mynumber-1) % max] = true;
         eating[mynumber] = false;
       }
}
```

4 The Dinning Philosophers: another COOL Solution

In DJ we can do the synchronization in yet a different way, by taking advantage of inheritance. In the previous set up, there was only one class, which was instantiated exactly five times. In this set up, we define five subclasses of `Philosopher`, each one instantiated exactly once, and then we coordinate those five instances distinguishing them by their class names.

```
public class Philosopher0 extends Philosopher {
  static boolean exactlyOne = false;
  Philosopher0() throws MaxInstances {
    if (exactlyOne) throw new MaxInstances();
    exactlyOne = true;
    super();
  }
}
// …
public class Philosopher4 extends Philosopher {
  static boolean exactlyOne = false;
  Philosopher4() throws MaxInstances {
    if (exactlyOne) throw new MaxInstances();
    exactlyOne = true;
    super();
  }
}
```

In this set up, we use a multiple-class coordinator that coordinates the five philosopher classes (and, consequently, the five philosopher instances), and we can do the synchronization simply with mutual exclusion:

```
per_class coordinator Philosopher0, Philosopher1, Philosopher2,
                      Philosopher3, Philosopher4 {
  mutex {Philosopher0.eat, Philosopher1.eat};
  mutex {Philosopher1.eat, Philosopher2.eat};
  mutex {Philosopher2.eat, Philosopher3.eat};
  mutex {Philosopher3.eat, Philosopher4.eat};
  mutex {Philosopher4.eat, Philosopher0.eat}
}
```

5 *An Assembly Line*

Consider an assembly line for making candy packets, as depicted in Figure 41. Candy makers make candy, one piece at a time, and there can be many of them. They pass each candy they make to a packer (*newCandy*), which accumulates a number of candy for producing a packet. When the packet is ready, the packer passes it to the finalizer (*newPack*), which takes also a label (*newLabel*) from the label maker, glues the label on the packet and outputs the final packet. All this is done concurrently, i.e. each participant works on its own, and they only synchronize at certain points.

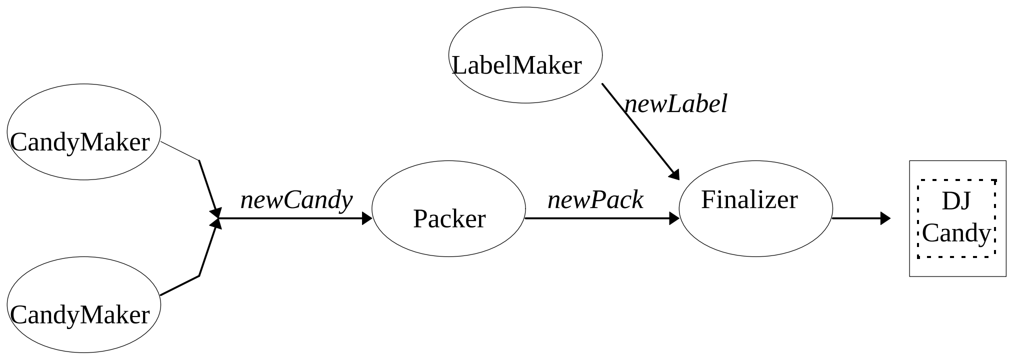


Figure 41. An assembly line for making candy packets.

The points of synchronization can be described as follows. When the packet in the packer is full, and while the packer is processing the packet, the candy makers must wait before passing any more new candy to the packer. As soon as the finalizer gets the packet from the packer, the packer restarts collecting candy from the candy makers into a new packet. The finalizer must wait for one packet and one label, and while it is gluing the label in the packet, both the packer and the label maker must wait before passing new items to the finalizer.

The remainder of this section shows the highlights of an implementation of this assembly line in DJ. First, the most important classes are presented. The coordinator is shown at the end.

Class CandyMaker:

```
public class CandyMaker implements Runnable {
  protected Random worktime = new Random ();
  protected Packer thePacker = null;
  CandyMaker(Packer p) {
    thePacker = p;
  }

  public void run() {
    Candy aCandy = null;
    while (!done) {
      aCandy = makeCandy();
      thePacker.newCandy(aCandy);
    }
  }

  public void finish() {done = true;}

  private Candy makeCandy() {
    Candy aCandy = new Candy();
    try {
      Thread.sleep(Math.abs(worktime.nextInt() % 500));
    } catch (InterruptedException e) {};
    return aCandy;
  }
}
```

Class Packer:

```
public class Packer implements Runnable {
  static final int maxPackers = 1;
  static int        count = 0;
  static int  nCandyPerPack = 50;

  protected int      myNumber = 0;
  protected Random worktime = new Random ();
  protected Finalizer theFinalizer = null;
  private Pack candyPack = null;
  private int  nCandy = 0;

  Packer(Finalizer f) throws MaxInstances {
    if (count == maxPackers) throw new MaxInstances();
    myNumber = count++;
    theFinalizer = f;
  }

  public void run() {
    Pack candyPack = null;
    while (!done) {
      candyPack = new Pack(nCandyPerPack);
      nCandy = 0;
      processPack(candyPack);
      theFinalizer.newPack(candyPack);
    }
  }

  public void finish() {done = true;}

  public void newCandy(Candy aCandy) throws Full {
    if (nCandy == nCandyPerPack) throw new Full();
    candyPack.put(aCandy);
    nCandy++;
  }

  private void processPack() {
    try {
      Thread.sleep(Math.abs(worktime.nextInt() % 500));
    } catch (InterruptedException e) {};
  }
}
```

Class LabelMaker:

```
public class LabelMaker implements Runnable {

  protected int      myNumber = 0;
  protected Random worktime = new Random ();
  protected Finalizer theFinalizer = null;

  LabelMacker(Finalizer f) {
    theFinalizer = f;
  }

  public void run() {
    Label aLabel = null;
    while (!done) {
      aLabel = makeLabel();
      theFinalizer.newLabel(aLabel);
    }
  }

  public void finish() {done = true;}

  private Label makeLabel() {
    Label aLabel = new Label;
    try {
      Thread.sleep(Math.abs(worktime.nextInt() % 500));
    } catch (InterruptedException e) {};
    return aLabel;
  }
}
```

Class Finalizer:

```
public class Finalizer implements Runnable {
  static final int maxFinalizers = 1;
  static int         count = 0;

  protected int      myNumber = 0;
  protected Random worktime = new Random ();
  private Pack       thePack = null;
  private Label      theLabel = null;

  Finalizer() throws MaxInstances {
    if (count == maxFinalizers) throw new MaxInstances();
    myNumber = count++;
  }

  public void run() {
    while (!done) {
      glueLabelToPack();
      newDJCandyPack();
    }
  }

  public void finish() {done = true;}

  public void newPack(Pack aPack) {
    thePack = aPack;
  }

  public void newLabel(Label aLabel) {
    theLabel = aLabel;
  }

  private void glueLabelToPack() {
    try {
      Thread.sleep(Math.abs(worktime.nextInt() % 500));
    } catch (InterruptedException e) {};
  }
  private void newDJCandyPack() {
    System.out.println (“New DJ Candy Pack!”);
  }
}
```

Finally, the coordinator:

```
coordinator Packer, Finalizer {

  selfex {Packer.newCandy};

  condition packFull = false, gotPack = false, gotLabel = false;

  Packer.newCandy: requires !packFull;
      on_exit { if (nCandy == nCandyPerPack) packFull = true; }

  Packer.processPack: requires packFull;

  Finalizer.newPack: requires !gotPack;
      on_entry { gotPack = true; }
      on_exit { packFull = false; }

  Finalizer.newLabel: requires !gotLabel;
      on_entry { gotLabel = true; }

  Finalizer.glueLabelToPack: requires (gotPack && gotLabel);

  Finalizer.newDJCandyPack:
      on_exit {gotPack = false; gotLabel = false;}
}
```

Note that this coordinator assumes that there is only one instance of the packer and only one instance of the finalizer. Otherwise, the coordination is incorrect — since the condition variables are simple variables, and different instances of packers and finalizers would conflict when modifying the coordination state. If more instances exist, the condition variables should be arrays of condition variables (see Philosophers).

III Distribution: Programming in RIDL

RIDL provides the means for dealing with data transfers between different execution spaces in relative separation from the classes. RIDL programs consist of a set of portal definitions which are associated with the classes on a name basis. Portals are helpers with respect to the implementation of the classes: they take care of data transfers across space boundaries.

1 Remote Objects

In DJ, some objects may be promoted to being "remote objects." Remote objects are ordinary objects that can be invoked from other execution spaces. That is, a remote object can be invoked both locally and remotely.

For an object to be also a remote object, the programmer must define, along with the class, a *remote interface* to instances of that class. A portal identifies the subset of methods of the class that can be invoked remotely (the remote operations), and the parameters and return values that those remote operations take. For each of the parameters and return values, and optional mode may be supplied that describes how the data transfers are to be made between space where the remote object lives and the spaces where its clients live. A portal is, therefore, a contract to which communicating execution spaces agree.

From a client's perspective, invocation to remote objects has exactly the same form as invocation to local objects: *obj.method(args)*; that is invocation is location-transparent. *obj* may be bound either to a local or to a remote object. When *obj* is bound to a local object, then an ordinary local invocation occurs. When *obj* is bound to a remote object, then a remote method invocation occurs. In this case, the method invocation must comply with the object's portal. This implies two things:

1) The data transfer is done according to the remote object's portal.
2) The run-time exception `DJInvalidRemoteOp` may occur.

2 The Name Server

A noticeable difference between a non-distributed and a distributed environment, is that in the latter there is no automatic way of binding objects that live in different execution spaces. Therefore an additional bootstrap is necessary. The most common mechanism for doing this is through a Name Service that is implemented by one or more name server objects. A name server object is a remote object that maps names (i.e. strings) to remote object references. The name service in DJ is the one provided by Java RMI [27], but with a special DJ-specific portal to it:

```
portal DJNaming {
  // Associate the given URL with the given DJ object
  void bind(String url, Object obj) {
    obj: gref;
  };

  // Same as bind, but if the an association to the same url exists,
  // associate the url with the new object
  void rebind(String url, Object obj) {
    obj: gref;
  };

  // Lookup a DJ remote object that is associated with the given name
  Object lookup(String url) {
    return: gref;
  };

  String[] list();
}
```

DJNaming is the DJ-specific wrapper class that interacts with Java RMI’s Naming class. The functionality of the DJNaming class is the same as Java RMI’s Naming class. From the Java RMI reference manual: “The java.rmi.Naming class allows remote objects to be retrieved and defined using the familiar Uniform Resource Locator (URL) syntax. The URL consists of protocol, host, port, and name fields. […] The protocol should be specified as rmi, as in rmi://java.sun.com:2001/root.” Not all the fields need to be present.

Here's an example, of how to bind and look up remote objects:

```
BoundedBuffer1 bb = new BoundedBuffer(100);
String url = "rmi://parc.xerox.com/BoundedBuffer";
// bind url to remote object
DJNaming.bind(url, bb);
        ...
// lookup bounded buffer
bb = (BoundedBuffer)DJNaming.lookup(url);
```

3 Basic Example: The Distributed Bounded Buffer

The most simple distributed application is the one that consists of one “server” object and multiple clients that access it from other execution spaces. Let’s reuse the bounded buffer example of the previous section to demonstrate how this is done in DJ.

Functionality

The distributed bounded buffer has exactly the same functionality as the non-distributed bounded buffer, i.e. it maintains a fixed array of elements, and its clients, local or remote, concurrently put and take elements from it. So, we can reuse the exact same class. But to make certain points more clear, let's make the internal buffer to be an array of particular objects, say books. We need to change the signatures of the methods:

```
public class BoundedBufferV1 {
  private Book array[];
  private int takePtr = 0, putPtr = 0;
  protected int usedSlots = 0, size;

  BoundedBufferV1(int capacity) throws IllegalArgumentException {
    if (capacity <= 0) throw new IllegalArgumentException();
    array = new Object[capacity];
    size = capacity ;
  }

  public int count() { return usedSlots; }
  public int capacity()  { return size; }

  public void put(Book x) throws Full {
    if (usedSlots == array.length) throw new Full();
    array[putPtr] = x;
    putPtr = (putPtr + 1) % size;
    usedSlots++;
    System.out.println("BB got book:");
    b.print();
  }

  public Book take() throws Empty {
    if (usedSlots == 0) throw new Empty();
    Book old = array[takePtr];
    takePtr = (takePtr + 1) % size;
    usedSlots--;
    return old;
  }
}

public class Book {
  private int isbn = 0;
  private String title = null;
  Book(int n, String t) {isbn = n; title = t;}
  public void print() {
    System.out.println ("Book: " + isbn + title);
}
```

Portal

But for bounded buffers to be remote objects, we need to define their portal. Let's define it as follows:

```
portal BoundedBufferV1 {
  int capacity();
  void put(Book x);
  Book take();
}
```

This declaration establishes an association between class `BoundedBufferV1` and this portal module. The body of this portal states that

- methods `capacity`, `put` and `take` are remote operations.
- method `count` defined in the class is not a remote operation.
- the `Book` argument to put and the return `Book` object of take are to be passed according to the default transfer strategy, which is by deep copy. This means that when a client invokes `put` with a book object as an argument, the bounded buffer gets a replica of that book, and not the book object itself. And when a client invokes `take`, it gets a replica of the book that lives in the bounded buffer space.

The above coordinator, then, takes care of limiting the access that remote clients have to bounded buffers, and of establishing the data transfer strategies between the communicating execution spaces.

Exporting the object reference

The application that instantiates a bounded buffer should export its reference to the name server. For example:

```
public class StartBuffer {
  public static void main(String args[]) {
    BoundedBufferV1 bb = new BoundedBufferV1(100);
    try {
      DJNaming.bind(“rmi://goblin/BB”, bb);
    } catch (Exception e) {
      System.out.println("StartBuffer err: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

From this point on, clients all over the network may get the reference to the bounded buffer object that was instantiated in this applet.

Client

A client of the remote bounded buffer is exactly like the client of a local bounded buffer:

```
public class Client implements Runnable {
  BoundedBufferV1  buf;
  Random           sleeptime = new Random ();
  boolean          done = false;
  boolean          shouldIput = false; // Indicates client is a producer

  public Client(BoundedBufferV1 b, boolean putp){
    buf = b;
    shouldIput = putp;
  }

  public void run(){
    Book b;
    while (!done){
       int x=sleeptime.nextInt();
      try {
        if (shouldIput) {
          b = new Book(x, "aBook" + new Integer(x).toString());
          buf.put(b); // if producer, put
        }
        else {  // else take
          b = buf.take();
          System.out.println("Client got book:");
          b.print();
        }
      } catch (BufferException e) {};
       try{
         Thread.sleep(Math.abs(sleeptime.nextInt() % 500));
       }catch(InterruptedException e){};
    }
  }
  public void finish(){
    done=true;
  }
}
```

But whoever instantiates clients must first fetch the reference to the remote bounded buffer:

```
public class StartClient {
  public static void main (String args[]) {
    BoundedBufferV1 bb;
    bb = (BoundedBufferV1)DJNaming.lookup("rmi://goblin/BB");
    new Thread(new Client(bb)).start();
  }
}
```

Running the App

To run this distributed application, we need to first run the `StartBuffer` class and then, in a separate shell, we can run a `StartClient` classes. We should see the output messages displaying the books, both in the bounded buffer terminal and in the client terminals.

4 Multiple clients

We can run multiple clients in different execution spaces. In that case, we may end up with concurrency on the bounded buffer space. Hence, we need to synchronize the method invocations by using, for example, the coordinator shown in page 213.

5 Passing Remote References

In the previous example, the book objects are passed by copy. Let's change the portal of the BoundedBufferV1 class, so that the books are passed by global reference:

```
portal BoundedBufferV1 {
  int capacity();
  void put(Book x) {
    x: gref;
  }
  Book take() {
    return: gref;
  }
}
```

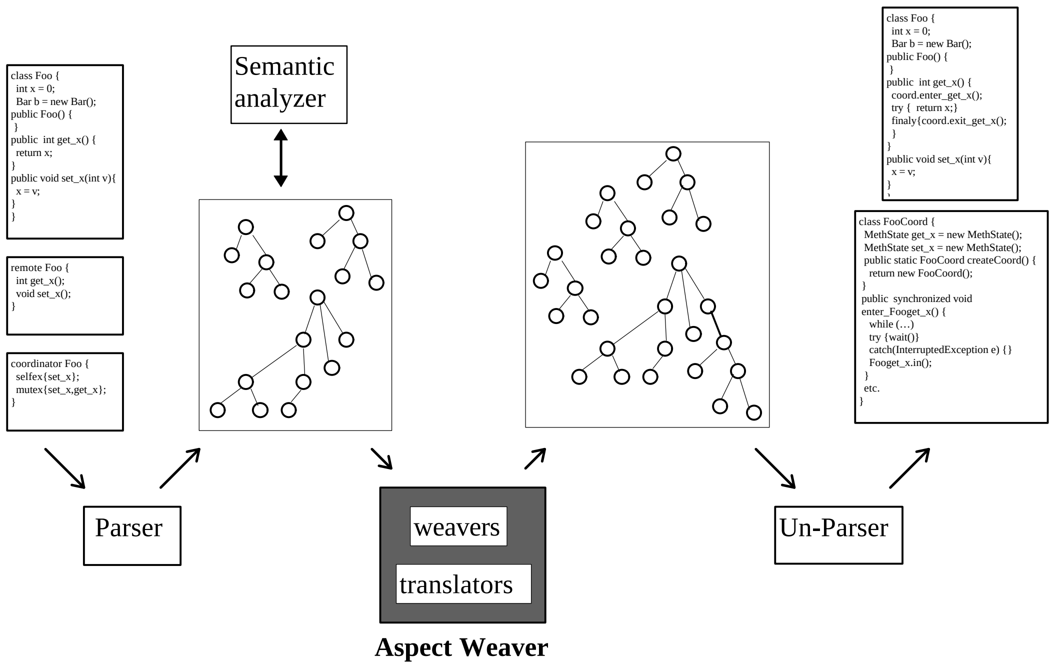
But now, books may also be remote objects. Therefore, they need a portal:

```
portal Book {
  void print();
}
```

After recompiling the bounded buffer and the clients of the bounded buffer, we can run the same set up. In this case, the book arguments and return objects are not replicated, and only their global references are passed. Therefore invocations to the print method will be displayed in within the space where the books where instantiated.

Appendix C. The Aspect Weaver

The translation of DJ programs into Java is done by a pre-processor, of which the Aspect Weaver is the most important module. The overall architecture is depicted in Figure 42. The Parser takes DJ programs and constructs representations of them (parse trees); the Semantic Analyzer checks the semantic validity of the tree, and, at the same stage, there may be some local transformations on the trees that facilitate the implementation of the Weaver (e.g. transforming COOL condition variable declarations into ordinary Java variable declarations, transforming RIDL traversal specifications into simpler structures, etc.); the Weaver takes those transformed parse trees and outputs new trees that represent woven Java programs; finally, the un-Parser takes those internal representations of Java programs and outputs Java.



```
class Foo {
  int x = 0;
  Bar b = new Bar();
public Foo() {
 }
public  int get_x() {
  return x;
}
public void set_x(int v){
  x = v;
}
}
```

```
remote Foo {
  int get_x();
  void set_x();
}
```

```
coordinator Foo {
  selfex{set_x};
  mutex{set_x,get_x};
}
```

```
class Foo {
  int x = 0;
  Bar b = new Bar();
public Foo() {
 }
public  int get_x() {
  coord.enter_get_x();
  try {  return x;}
  finaly{coord.exit_get_x();
  }
}
public void set_x(int v){
  x = v;
}
```

```
class FooCoord {
  MethState get_x = new MethState();
  MethState set_x = new MethState();
  public static FooCoord createCoord() {
    return new FooCoord();
 }
 public  synchronized void
 enter_Fooget_x() {
    while (…)
    try {wait()}
    catch(InterruptedException e) {}
    Fooget_x.in();
  }
  etc.
}
```

Figure 42. Architecture of the pre-processor that translates DJ programs into Java programs.

This appendix contains the algorithms implemented by the Aspect Weaver. They are presented as a mapping between the input and the output structures. The algorithms will be presented in pseudo-code, along with some comments.

I Notation

The pseudo-language that will be used is a mix of C and Lisp, and similar to any imperative language with type annotations. The only basic types are symbols, lists and integers. Symbols are shown in *italic*, and lists are denoted by `()`, with the elements separated by commas. New types are defined using the pseudo-instruction **`record`**. Records are denoted by `[<`*`type_of_record`*`>, …]`. For simplicity sake, the generic ***`null`*** symbol will be used to denote the null value of any type (i.e. symbols, lists, integers and records). The other pseudo-instructions are: variable assignment (`:=`); environments (**`global, let`**); loops (**`foreach, while`**); branching (**`if else`**); return from functions (**`return`**); de-structuring (**`given`**); function definitions (**`function`**) and function calls (e.g., `foo(arg1, arg2)`).

About the pseudo-records

Pseudo-records are used to represent the input and output structures of the Weaver (see Figure 42). A typed record is a finite list whose first element is a token that defines the record's type and each following element (called "field") of the list is a pair keyword/value. For example, the class

```
class Foo extends Bar {
  private int x = 0;
  public Foo() {super();}
  public Foo(int value) { x = value; }
  public synchronized int get_x() { return x; }
}
```

is represented by the record

```
[class, name: Foo, super: Bar,
        variables: ([vardecl, qualifiers: private, type: int, name: x,
                              init: [literal, 0]]),
        constructors: ([constructor, args: (), body: ([supercall, args:()])],
                       [constructor, args: (int),
                                  body:([assign, left: [var_ref, name: x],
                                            right:[var_ref,name: value]])]),
        methods: ([method, qualifiers: public synchronized, type: int,
                           name: get_x, params: (), body: [return, expr: x]]))
```

A record may be created empty, partially complete or complete. For example, `[class]` denotes an empty class record, with all fields defaulting to typed null values; `[class, name: `*`Foo`*`]` denotes a class named Foo, and with all other fields defaulting to typed null values.

For the sake of simplicity, the following notation will be used: `r.field` is the value of the element whose keyword is `field` in record `r` . For example, if `c` denotes the class record shown above, `c.super` denotes the value of the pair that has "`super`" as keyword, i.e. `Bar`.

II Definitions and Auxiliary Functions

1 *The Input and Output Records*

The input of the Weaver consists of representations of JCore classes, COOL coordinators and RIDL portals. The output consists of representations of Java classes and interfaces. All constructs of these languages are represented here by pseudo-records. The input records describe slightly transformed DJ parse trees, and the output records represent Java programs. Since JCore is Java without the synchronized statement, the input records are a super-set of the output records. The following list of record definitions describes the most important records used by the Weaver.

```
/*** JCore and Java ***/
// Only 3 records are shown.
record class          = [name: symbol, super: symbol,
                         interfaces: list of symbol,
                         variables: list of vardecl,
                         constructors: list of constructor,
                         methods: list of method]
record interface      = [name: symbol, supers: list of symbol,
                         methods: list of method] // these method bodies
                                                  // are always null.
record constructor    = [params: list of param, body: statement]
record method         = [qualifiers: combination of symbol,
                                     type: symbol, name: symbol,
                                     params: list of param,
                                     throws: list of symbol, body: statement]
/** COOL **/
record coordinator    = [granularity: one of {per_object, per_class},
                         classes: list of class,
                         vars: list of vardecl,
                         selfex: list of qualified_name,
                         mutexes: list of mutex,
                         mmanagers: list of mmanager]
record vardecl        = [type: symbol, name: symbol, init: expression]
record mutex          = [mux: list of qualified_name]
record mmanager       = [mnames: list of qualified_name,
                         requires: expression,
                         on_entry: statement,
                         on_exit: statement]
record qualified_name = [cname: symbol, mname: symbol]
/** RIDL **/
record portal         = [class: class, operations: list of operation]
record operation      = [type: ridl_type, name: symbol,
                               params: list of ridl_type]
record ridl_type      = [type: symbol, name: symbol,
                               mode: one of {gref, copy},
                               traversal: traversal]
record traversal      = [incompletes: list of incomplete_class]
record incomplete_class = [name: symbol, missing: list of symbol]
```

Note: From here on, there is, sometimes, a slight abuse of terminology by using, for example, “the class” instead of “the record representing a class”. Also, the handling of Java values is sometimes shortened: the literal records are dropped out, the type of the value is left unspecified, and only the value is shown.

2 Constants

```
// All of these constants are for weaving RIDL

ZERO            = [literal, value: 0]
ZEROOBJECT      = [new, class: Integer, args: (ZERO)]

WRITEEXTERNAL   = [method, qualifiers: public, type: void,
                           name: writeExternal,
                           params: ([param, type: OutputStream, name: out])]
READEXTERNAL    = [method, qualifiers: public, type: void, name: readExternal,
                           params: ([param, type: InputStream, name: in])]
D_WRITEEXTERNAL = [method, qualifiers: public, type: void,
                           name: _d_writeExternal,
                           params: ([param, type: OutputStream, name: out],
                                    [param, type: Traversal, name: t])]
D_READEXTERNAL  = [method, qualifiers: public, type: void,
                           name: _d_readExternal,
                           params: ([param, type: InputStream, name: in],
                                    [param, type: Traversal, name: t])]

BYPASSWRITETEST = [if,expr:[not,expr:[invocation,obj:c,meth:bypassPart]]]
BYPASSREADTEST  =  [if, expr: [invocation, obj:c, meth: bypassPart]]

READOBJECT      = [invocation, obj: in, meth: readObject]
READTOKEN       = [invocation, obj: [cast, type: String, expr: READOBJECT],
                               meth: equals, args: ([literal, DObject])]

NEWINSTANCE     = [invocation, obj: [invocation, obj: Class, meth: forName,
                                                 args: (classname)],
                                                 meth: newInstance]

CATCHINGWRAPPER = ([catch, exception: [param, type: DInvalidRemoteException,
                                             name: e],
                    body: [invocation,
                           obj: [field_ref, obj: System, field: err],
                           meth: println,
                           params: ([literal,
                                     value: "Invalid remote operation"])]])
CATCH_INMARSHALING = [catch, exception: [param, type: Exception, name: e],
                             body: [invocation,
                                    obj: [field_ref, obj: System, field: err],
                                    meth: println,
                                    params: ([invocation, obj: e,
                                                          meth: toString])]]
```

3 Auxiliary Functions

- `Append(<list>, <element`$_1$`>, …, <element`$_n$`>)`: appends the given elements to the given list and returns the list. The given list is extended with the new elements.
- `Clone(<record>)`: returns a record which is identical to the given record.

- `Concat(<lists of symbols> or <single symbols>)`: returns a symbol that is the concatenation of all the symbols given.
- `LookupClass(<symbol>):` the input is a class name. This function returns the class record representing the class with the given name. This record may be incomplete; the lookup of a class may result in filling only the record’s name, super, interfaces, and method signatures (i.e. no variables, no constructors and no method bodies). The return record itself may be the null record, if the class does not exist or if the given name is null.
- `LookupCoordinators()`: returns a list containing all coordinator records (e.g. from file extensions). These records are incomplete: they only contain the `classes` field, and each of those fields only contains the `name` of the class.
- `LookupClassWithAspect(<class record>, <symbol>)`: returns a record representing the closest class in the class hierarchy, starting at (and including) the given class, that is associated with an aspect module of the kind given by the symbol (i.e. *cool* or *ridl*). As in the function before, the resulting class record may be incomplete. If no class is associated with an aspect module of the given kind, the function returns ***`null`***.
- `Match(<symbol>, <list of symbols>):` returns *`true`* if the given list contains at least one occurrence of the given symbol; *`false`* otherwise.
- `Methods(<list of typed record>)`: returns a list of symbols corresponding to the “method” fields of all the given records. (similar to `Names`, below)
- `Names(<list of typed record>)`: returns a list of symbols corresponding to the “name” fields of all the given records. E.g. `Names(([class, name: `*`Foo`*`], [class, name: `*`Bar`*`]))` is `(`*`Foo`*`, `*`Bar`*`).`
- `Qname(<class record>, <method record>):` returns a symbol consisting of the concatenation of the given class name and the given method name. E.g. `Qname([class, name: `*`Foo`*`], [method, name: `*`bar`*`])` is *`Foobar`*.
- `Types(<list typed record>)`: returns a list of symbols corresponding to the “type” fields of all the given records. (similar to `Names`, above)

III The Weaving Engine

The weaving engines of COOL and RIDL are similar, and are captured by two generic weaving functions:

- `direct_weave`, which weaves a class when it is directly associated with an aspect module,
- `inherit_weave`, which weaves a class when it is not directly associated with an aspect module, but inherits an aspect module from a superclass.

The aspect-specificity of this weaving engine shows up only in the calls to `wrapper_body`. Wrapper bodies are considerably different for each aspect and for when aspects are combined together. But the core of the weaving engine, given by those two functions, is aspect-independent.

The fundamental reason why there are two generic engines, instead of one, is the following. According to the implementation architecture described in Chapter 4, when a class is directly associated with an aspect module, the weaver must process the class's own methods as well as *all* non-private methods of *all* superclasses, whereas when the class inherits an aspect module, the weaver simply needs to process the class's own methods.

```
//direct_weave is called when the class is directly associated with an aspect
// module (coordinator or portal).
// Input:  class: the class that is to be woven;
//         newvars: the variable declarations that must be woven;
//         init_code: initialization code to be appended to every constructor;
//         aspect: a token indicating which aspect is being woven;
// Output: the woven class.
function direct_weave (class oftype class,
                       newvars oftype list of vardecl,
                       init_code oftype statement,
                       aspect oftype symbol)
  let wclass = Clone(class) in
    // weave the extra variable declarations
    foreach var ∈ newvars,
      Append(wclass.variables, var)
    // weave the initialization in every constructor
    if wclass.constructors == null // no constructors. Weave one.
      wclass.constructors := ([constructor, body: init_code])
    else
      foreach const ∈ wclass.constructors,
        Append(const.body, init_code)

    // Process the methods implemented in the class. Make them into
    // implementation/wrapper pairs.
    foreach meth ∈ class.methods,
      let implmeth = Clone(meth), wrappermeth = Clone(meth)  in
        implmeth.name := Concat(_d_, implmeth.name)
        replace_calls_to_super(implmeth.body)
        wrappermeth.body := wrapper_body(wrappermeth,class.name,aspect)
        Append(wclass.methods, implmeth, wrappermeth)
    // Then, the methods that are inherited. Add only the wrapper method
    // in wclass for each method that is inherited.
    let super = LookupClass(class.super), methodNameList = () in
      while super ≠ null
        foreach meth ∈ super.methods,
          if Match(meth.name, methodNameList) == false // not seen before
             Match(meth.name, Names(class.methods)) == false //truly inherited
             and private ∉ meth.qualifiers
            Append(wclass.methods, [method, qualifiers: meth.qualifiers,
                                            type: meth.type,
                                            name: meth.name,
                                            params: meth.params,
                                            body:wrapper_body(meth,class.name,
                                                              aspect)])
            Append(methodNameList, meth.name)
        super := LookupClass(super.super)
    return wclass
end.
```

```
// inherit_weave is called when the class is not associated with an aspect,
// module but it has a superclass that is.
// New methods defined in this class are transformed into a pair of
// implementation/wrapper methods. Methods that are overridden from the
// super with aspect module, simply get transformed in implementation
// methods - the wrapper is inherited from the super.
// Input: class: the class record;
//        superc: the class record of the closest superclass that is directly
//                associated with a coordinator;
// Output: the woven class.
function inherit_weave (class oftype class,
                        supers_with_aspect oftype list of class)
  let wclass = Clone(class) in
    foreach meth ∈ class.methods,
      let implmeth = Clone(meth) in
        implmeth.name := Concat(_d_, implmeth.name)
        replace_calls_to_super(implmeth.body)
        Append(wclass.methods, implmeth)

      if Match(meth.name, Names(Methods(supers_with_aspect))) == false
        // the “wrapper” method. Here, it just calls the implementation
        // method. (Careful about the return type)
        let invocationToImplmeth = [invocation, meth: implmeth.name,
                                          args: Names(meth.params)] in
          if meth.type == void
            meth.body := invocationToImplmeth
          else
            meth.body := [return, expr: invocationToImplmeth]
      //else, the wrapper is the one in the super with aspect module.
    return wclass
end.


// wrapper_body dispatches the call according to the specific aspect
// that is being woven.
// Input:  meth: the original method for which the wrapper is being generated;
//         cname: the classname of the class where the method is implemented;
//         aspect: a token indicating the aspect that is being woven.
// Output: the body of the aspect-specific wrapper method.
function wrapper_body (wrappermeth oftype method,
                       cname oftype symbol, aspect oftype symbol)
  if aspect == cool
    return wrapper_body_cool(meth, cname)
  else
    if aspect == ridl
      return wrapper_body_ridl(meth)
    else
      return wrapper_body_coolridl(meth, cname, aspect)
end.
```

```
// replace_calls_to_super takes the body of the given method meth and replaces
// the direct calls to super.<meth.name> with calls to super._d_<meth.name>.
// (destructively). This is the most costly function to implement, since
// it implies that the parser must parse JCore method bodies.
// Implementers may simply chose to disallow direct calls to super(), to avoid
// parsing full Java.
// Input:  meth: a record representing an “implementation” method
// Output: the same record, but with the said substitutions.
function replace_calls_to_super (meth oftype method)
  foreach inv oftype invocation ∈ meth.body such that
                                  obj == super and meth == meth.name,
    inv.meth := Concat(_d_, meth.name)
end.
```

```
// weave is an auxiliary entry point to the weaving engine for weaving exactly
// one aspect in isolation of the other. It is called by weave_cool and
// weave_ridl.
// Subsection §3 presents a new version of this function for weaving both
// aspects at the same time.
// Input:  class: record representing the class that is to be woven
//         newvars: a list of new variable declarations to be woven
//         init_code: initialization code to be appended to every constructor
//         aspect: a token indicating which aspect is being woven
// Output: record representing the woven class.
function weave (class oftype class,
                newvars oftype list of vardecl,
                init_code oftype statement,
                aspect oftype symbol)
  let class_with_aspect = LookupClassWithAspect(class, aspect) in
    if class_with_aspect == null // no weaving.
      return Clone(class)
    else // test if it’s direct or inherited association with aspect module
      if class_with_aspect.name == class.name // direct
        return direct_weave(class, newvars, init_code, aspect)
      else // inherited
        return inherit_weave(class, (class_with_aspect))
end.
```

1 COOL-specific Functions

```
global coordvarname oftype symbol

// weave_cool is the top function for a stand-alone COOL weaver.
// Input:  class: the class record.
// Output: another class based on the input class but with woven code at
//         particular points.
function weave_cool (class oftype class)
  let allcoordinators = LookupCoordinators(),
    if ∃ coord ∈ allcoordinators such that  class.name ∈ Names(coord.classes)
      let coordclassname = Concat(Names(coord.classes), Coord) in
        coordvarname := Concat(_, coordclassname)
        let  newvars = ([vardecl, type: coordclassname, name: coordvarname]),
             init_code = init_coordinator_code(coordclassname) in
          return weave (class, newvars, init_code, cool)
    else return class // no weaving.
end.

// wrapper_body_cool generates the body of coordination wrapper methods.
// Input:  meth: the original method for which the wrapper is being generated;
//         cname: the classname of the class where the method is implemented;
// Output: the body of the wrapper method.
// Note:   coordvarname is a global variable that holds the name that the
//         coordinator variable has in the class that is being woven.
function wrapper_body_cool(meth oftype method, cname oftype symbol)
  let s = [sequence] in
    s.statements := ([invocation, obj: [var_ref, name: coordvarname],
                             meth: Concat(enter_, cname, meth.name),
                             args: this],
                     [try, body: try_body_cool(meth),
                           finally: [invocation,
                                     obj:[var_ref, name: coordvarname],
                                     meth: Concat(exit_, cname, meth.name),
                                     args: this]])
    return s
end.


// try_body_cool generates the body of the try statement inside the wrapper
// methods, which basically consists of a call to the implementation method.
// Input:  meth: the method record.
// Output: a Java statement that is either the direct invocation (if there
//         is no return value) or a return statement returning the result
//         of the invocation.
function try_body_cool(meth oftype method)
  let invocationToImplmeth = [invocation, meth: Concat(_d_, meth.name)
                                          args: Names(meth.params)]    in
    if meth.type == void
      return invocationToImplmeth
    else
      return [return, expr: invocationToImplmeth]
end.

// init_coordinator_code returns the assignment record that represents
// the initialization of the coordinator variable.
function init_coordinator_code(coordclassname)
  return [assignment, left: coordvarname, // coordvarname is global
                      right:[invocation, obj:coordclassname,
                                         meth:createCoord]]
end.
```

2 RIDL-specific Functions

```
global pvarname oftype symbol
global ppvarname oftype symbol

// weave_ridl is the top function of a stand-alone RIDL weaver.
// Input:  class: the class record.
// Output: another class based on the input class but with woven code at
//         particular points.
// Note:   the function marshaling_methods, called at the end, is part of
//         the RIDL weaver.
function weave_ridl (class oftype class)
  let pclassname = Concat(class.name, P),
      ppclassname = Concat(class.name, PP),
      traversalsclassname = Concat(class.name, Traversals)  in
    pvarname := Concat(_p, class.name),
    ppvarname := Concat(_rself, class.name),
    let newvars = ([vardecl, qualifiers: public,
                             type: pclassname,
                             name: pvarname],
                   [vardecl, qualifiers: public,
                             type: ppclassname,
                             name: ppvarname, init: null]),
        init_code = init_p_code(pclassname) in
    let wclass = weave(class, newvars, init_code, ridl),
        marshals = marshaling_methods(class) in
      Append(wclass.interfaces, DObject)
      foreach meth ∈ marshals,
        Append(wclass.methods, meth)
      return wclass
end.

// wrapper_body_ridl generates the body of portal wrapper methods.
// Input:  meth: the original method for which the wrapper is being generated;
// Output: the body of the wrapper method.
// Note:   ppvarname is a global variable that holds the name that the
//         PP variable has in the class that is being woven.
function wrapper_body_ridl(meth oftype method)
  return ([if, expr: [not_equal, left: ppvarname, right: null],
               then: [try, body: try_body_ridl,
                           catches: CATCHINGWRAPPER],
               else: [invocation, meth: Concat(_d_, meth.name),
                                  args: Names(meth.params)]])
end.


// try_body_ridl generates the body of the try statement inside the wrapper
// methods, which basically consists of a call to the PP object.
// Input:  meth: the method record.
// Output: a Java statement that is either the direct invocation (if there
//         is no return value) or a return statement returning the result
//         of the invocation.
function try_body_ridl(meth oftype method)
  let invocationToPP = [invocation, obj: ppvarname, meth: meth.name,
                                    args: Names(meth.params)] in
    if meth.type == void
      return invocationToPP
    else
      return [return, expr: invocationToPP]
end.
```

```
// init_p_code returns the assignment record that represents
// the initialization of the P variable.
function init_coordinator_code(pclassname)
  return [assignment, left: pvarname, // pvarname is global
                      right:[new, class:pclassname, args: this]]
end.



// The following functions generate the marshaling methods.
// In the following code, the field named “statements” of sequence records
// is omitted for making the code more readable; also omitted are the
// field names “left” and “right” of binary boolean expressions.

function marshaling_methods (class oftype class)
  let writeExt = Clone(WRITEEXTERNAL), readExt = Clone(READEXTERNAL),
      dwriteExt = Clone(D_WRITEEXTERNAL), dreadExt = Clone(D_READEXTERNAL) in
    writeExt.body  := simple_write_body(class.variables)
    readExt.body   := simple_read_body(class.variables)
    dwriteExt.body := traversal_write_body(class.name, class.variables)
    dreadExt.body  := traversal_read_body(class.name, class.variables)
    return (writeExt, readExt, dwriteExt, dreadExt)
end.

// Code for packing (write functions)
function simple_write_body(vars oftype list of vardecl)
  let s = [sequence] in
    foreach var ∈ vars,
      if static ∉ var.qualifiers
        Append(s.statements, [invocation, obj: s, meth: writeObject,
                                          args: (var.name)])
    return s
end.

function traversal_write_body(cname oftype symbol,vars oftype list of vardecl)
  let s = [sequence, (traversal_method_vardecl(cname))] in
    foreach var ∈ vars,
      if static ∉ var.qualifiers
        let bypasstest = Clone(BYPASSWRITETEST) in
          bypasstest.expr.expr.args := ([stringliteral, value: var.name])
          if is_primitive(var.type)
            bypasstest.then := [invocation, obj: s, meth: WriteObject,
                                            args: (var.name)]
          else
            if is_object(var)
              bypasstest.then := write_object(var.name))
            else // it’s array
              bypasstest.then := write_array(var.name))
          Append(s.statements, bypasstest)
    return s
end.

function write_object (v oftype expression)
 return
  [if, expr: [or,terms:([equal, v, null],
                        [and, terms:([notequal, v, null],
                                     [not, expr: [instanceof, v,DObject]])])],
       then: [sequence, (WRITEOBJECT,
                         [invocation, obj: out, meth: writeObject, args:(v)])]
       else: [sequence, (WRITEDOBJECT,
                         [invocation, obj: out, meth: writeObject,
                            args: ([invocation, obj: [invocation, obj: v,
                                                           meth: getClass],
                                                meth: getName])]
                         [invocation, obj: [cast, class: DObject, expr: v],
                                     meth: _d_writeExternal, args: (out,t)])]]
end.

function write_array (varname oftype symbol)
 return
  [if, expr: [equal, varname, null],
       then: [invocation, obj: out, meth: writeObject, args: (ZEROOBJECT)]
       else: [sequence, statements:
                ([invocation, obj: out, meth: writeObject,
                              args:([new, class: Integer,
                                          args:(array_length(varname))])]),
                 [for, base: [vardecl, type: int, name: _i, init: ZERO],
                       test: [lessthan, _i, array_length(varname)],
                       action: [incr, obj: _i],
                       body: write_object(element_ref(varname))])]]
end.


// Code for unpacking (read functions)
function simple_read_body(vars oftype list of vardecl)
  let s = [sequence] in
    foreach var ∈ vars,
      if static ∉ var.qualifiers
        Append(s.statements, [invocation, obj: s, meth: readObject,
                                          args: (var.name)])
    return s
end.


function traversal_read_body(cname oftype symbol, vars oftype list of vardecl)
  let s = [try, body: sequence, (traversal_method_vardecl(cname))
               catches: (CATCH_INMARSHALING)] in
    foreach var ∈ vars,
      if static ∉ var.qualifiers
        let bypasstest = Clone(BYPASSREADTEST) in
          bypasstest.expr.args := ([literal, value: var.name])
          bypasstest.then := ([assignment, left: var.name, right: null])
          if is_primitive(var.type)
            bypasstest.else := assign_after_read(var.name, var.type,
                                                 [invocation, obj: in,
                                                         meth: ReadObject])
          else
            if is_object(var)
              bypasstest.else := read_object(var.name, var.type)
            else // it's array
              bypasstest.else := read_array(var.name, var.type))
          Append(s.body.statements, bypasstest)
    return s
end.


function read_object(v oftype expression, type oftype symbol)
  return
    [if, expr: READTOKEN,
         then: [sequence, (READCLASS,
                           assign_after_read(v,type,[cast, type: type,
                                                            expr:NEWINSTANCE]),
                           [invocation, obj: [cast, type: DObject, expr: v],
                                        meth: _d_readExternal,
                                        args: (in, t)])]
         else: [assignment, left: v, right: [cast, type: type, READOBJECT]]]
end.

function read_array(varname oftype symbol, type oftype symbol)
  return
    [sequence, (READLENGTH,
                [if, expr: [equal, n, [literal, 0]],
                     then: [assignment, varname, null],
                     else: [sequence,
                            ([assignment, varname, [new, class:type, size:n]],
                             array_iteration(varname,
                                    read_object(element_ref(varname),type))])])]
end.

function assign_after_read(var oftype expression, type oftype symbol,
                           read_expr oftype expression)
  return [assignment, left: var, right: [cast, type: type, expr: read_expr]]
end.

function traversal_method_vardecl(cname oftype symbol)
  return [vardecl, type: DPartCutter, name: c,
                 init: [invocation, obj: t, meth: isIncomplete,
                                   args: ([literal, value: cname])]]
end.

function array_length (varname oftype symbol)
  return [field_ref, obj: varname, field: length]
end.

function element_ref (varname oftype symbol)
  return [array_ref, obj: varname, index: _i]
end.

function array_iteration (varname oftype symbol, body oftype expression)
  return [for, base: [vardecl, type: int, name: _i, init: ZERO],
               test: [lessthan, _i, array_length(varname)],
               action: [incr, obj: _i],
               body: body]
end.
```

3 Weaving COOL and RIDL Together

In order to weave COOL and RIDL together, a few extra functions are necessary, two at the top level and one at the bottom. However, the generic weaving engine consisting of `direct_weave` and `inherit_weave` is used and remains unchanged.

```
// all global variables from each of the separate weavers are set here
global coordvarname oftype symbol
global pvarname oftype symbol
global ppvarname oftype symbol
// a couple of extra ones that are used for generating the wrapper
global class_with_coord oftype class
global class_with_portal oftype class
```

```
// weave_both is the top function of for weaving COOL and RIDL together.
// It simply sets all global variables, and constructs the representations
// of the new cool and ridl variable declarations and initialization code.
// Input:  class: the class record.
// Output: another class based on the input class but with woven code at
//         particular points.
function weave_both (class oftype class)
  let pclassname = Concat(class.name, P),
      ppclassname = Concat(class.name, PP) in
    pvarname := Concat(_p, class.name),
    ppvarname := Concat(_rself, class.name),
  let allcoordinators = LookupCoordinators(),
    if ∃ coord ∈ allcoordinators such that  class.name ∈ Names(coord.classes)
      let coordclassname = Concat(Names(coord.classes), Coord) in
        coordvarname := Concat(_, coordclassname)
  let newcoolvars = ([vardecl, type:coordclassname, name: coordvarname]),
      init_cool_code = init_coordinator_code(coordclassname),
      newridlvars = ([vardecl, qualifiers: public,
                               type: pclassname,
                               name: pvarname],
                     [vardecl, qualifiers: public,
                               type: ppclassname,
                               name: ppvarname, init: null]),
      init_ridl_code = init_p_code(pclassname) in
    let wclass = weave(class, newvars, init_code, ridl),
        marshals = marshaling_methods(class) in
      Append(wclass.interfaces, DObject)
      foreach meth ∈ marshals,
        Append(wclass.methods, meth)
      return wclass
end.
```

```
// weave is the entry point to the weaving engine for weaving both aspects
// at the same time. It is called by weave_both.
// Input:  class: record representing the class that is to be woven;
//         newcoolvars: a list of new variable declarations coming from cool;
//         newridlvars: a list of new variable declarations coming from ridl;
//         init_cool_code: initialization code coming from cool;
//         init__ridl_code: initialization code coming from ridl;
// Output: record representing the woven class.
// When weaving both aspects at the same time, there are exactly 9 different
// combinations for the association of the class with the two aspects:
// 1. no direct or inherited coordinator (coord, for short) or
//    portal(ptal, for short);
// 2. direct coord and no ptal;
// 3. inherited coord and no ptal;
// 4. direct ptal and no coord;
// 5. inherited ptal and no coord;
// 6. direct both coord and ptal;
// 7. direct coord and inherited ptal;
// 8. direct ptal and inherited coord;
// 9. inherited both coord and ptal (not necessarily the same super for both)
// Each of these combinations results in different weavings.
//
function weave (class oftype class,
                newcoolvars oftype list of vardecl,
                newridlvars oftype list of vardecl,
                init_cool_code oftype statement,
                init_ridl_code oftype statement)
  class_with_coord := LookupClassWithAspect(class, cool)
  class_with_portal := LookupClassWithAspect(class, ridl)
  if class_with_coord ≠ null or class_with_portal ≠ null                 // 1
    /*** The first 4 cases are single aspect weaving ***/
    if class_with_coord ≠ null and class_with_portal == null
      if class_with_coord.name == class.name                             // 2
        return direct_weave(class, newcoolvars, init_cool_code, cool)
      else                                                               // 3
        return inherit_weave(class, class_with_coord)
    else
      if class_with_coord == null and class_with_portal ≠ null
        if class_with_portal.name == class.name                          // 4
          return direct_weave(class, newridlvars, init_ridl_code, ridl)
        else                                                             // 5
          return inherit_weave(class, class_with_portal)
      /*** The next 4 cases are the new ones ***/
      else
        if class_with_coord.name == class.name and
           class_with_portal.name == class.name                          // 6
          return direct_weave(class, Merge(newcoolvars, newridlvars),
                              [sequence: (init_cool_code, init_ridl_code),
                              coolridl)
        else
          if class_with_coord.name == class.name and
             class_with_portal.name ≠  class.name                        // 7
            return direct_weave(class, newcoolvars,init_cool_code,coolridl)
          else
            if class_with_coord.name ≠ class.name and
               class_with_portal.name ==  class.name                     // 8
              return direct_weave(class,newridlvars,init_ridl_code,coolridl)
            else                                                         // 9
              return inherit_weave(class,(class_with_coord,class_with_portal))
  else return Clone(class)
end.
```

```
// wrapper_body_coolridl generates the body of coordination wrapper methods.
// Input:  meth: the original method for which the wrapper is being generated;
//         cname: the classname of the class that is being woven;
// Output: the body of the wrapper method.
//
// This function needs to decide on the precise combination of wrappers, which
// depends on the superclasses with aspect modules. The logic is as follows:
// Case 1. class_with_coord.name = class_with_portal.name = cname
//         This means that the class that is being woven is directly
//         associated with both a coordinator and a portal.
//         Therefore, the wrapper must merge both aspect wrappers for this
//         class.
// Case 2. class_with_coord.name = cname and class_with_portal ≠ cname
//         This means that the class that is being woven is directly
//         associated with a coordinator and inherits a portal.
//         Therefore, the wrapper should contain only the COOL wrapper.
//         However, when the method overrides a method of class_with_portal,
//         the RIDL wrapper must be included too.
// Case 3. class_with_portal = cname and class_with_coord.name ≠ cname
//         This means that the class that is being woven is directly
//         associated with a portal and inherits a coordinator.
//         Therefore, the wrapper should contain only the RIDL wrapper.
//         However, when the method overrides a method of class_with_coord,
//         the COOL wrapper must be included too.
// Case 4. class_with_portal  ≠ cname and class_with_portal ≠ cname
//         does not occur here; such case is handled by inherit_weave. This
//         function is called only if both aspect modules are associated
//         with this class, and at least one of them is directly associated.
//
// Note:   global variables: coordvarname, ppvarname.
//
function wrapper_body_coolridl(meth oftype method, cname oftype symbol)
  let coolwrapper = [sequence,
                   statements: ([invocation, obj: coordvarname,
                                meth: Concat(enter_, cname, meth.name),
                                args: this],
                               [try, body: try_body_cool(meth),
                                finally: [invocation,
                                obj: coordvarname,
                                meth: Concat(exit_, cname, meth.name),
                                args: this])],
      nocoolwrapper = [invocation, meth: Concat(_d_, meth.name),
                                args: Names(meth.params)] in

    if (class_with_coord.name == cname and class_with_portal.name == cname) or
       (class_with_coord.name ≠ cname and           // the exception of Case 3
        Match(meth.name, Names(class_with_coord.methods))) or
       (class_with_portal.name ≠ cname and          // the exception of Case 2
        Match(meth.name, Names(class_with_portal.methods)))
      return ([if, expr: [not_equal, left: ppvarname, right: null],
                 then: [try, body: try_body_ridl,
                             catches: CATCHINGWRAPPER],
                 else: coolwrapper])
    else // not inherited from super with other aspect
      if class_with_coord.name == cname // only COOL wrapper
        return coolwrapper
      else // only RIDL wrapper
        return ([if, expr: [not_equal, left: ppvarname, right: null],
                   then: [try, body: try_body_ridl,
                               catches: CATCHINGWRAPPER],
                   else: nocoolwrapper])
end.
```

IV Translation Engine for COOL

```
// translate_coordinator is the top function of the translation for COOL.
// Input:  coord: a coordinator.
// Output: a class resulting from the given coordinator (coordinator class).
function translate_coordinator (coord oftype coordinator)
  let c = [class, name: Concat(Names(coord.classes), Coord)] in
    /*** The variables ***/
    if coord.granularity == per_class  // add the “theCoord” variable
      c.variables := ([vardecl, qualifiers: static, type: boolean,
                                name: one, init: false)],
                      [vardecl, qualifiers: static, type: c.name,
                                name: theCoord])

    // The method state variables. One per method of all the classes,
    // including methods inherited from superclasses
    foreach class ∈ coord.classes,
      let aclass = class, methodnames = ()  in
        while aclass != null
          foreach method ∈ aclass.methods,
            if Match(method.name, methodnames)==false // avoid repeating names
              Append(methodnames, method.name)        // of overridden methods
              Append(c.variables, [vardecl, type: MethState,
                                        name: Qname(class,method),
                                        init: [new, class: MethState]])
          aclass := LookupClass(aclass.super) // up the class hierarchy

    // The condition and ordinary variables
    foreach var ∈ coord.vars,
      Append(c.variables, [vardecl, type: var.type, name: var.name,
                                    init: var.init)])

    /*** The methods ***/
    // The factory method
    c.methods := ([method, qualifiers: public static synchronized,
                   type: c.name, name: createCoord,
                   body: factory_body(coord, c.name)])
    // The before and after methods
    foreach class ∈ coord.classes,
      let aclass = class, methodnames = () in
        while aclass != null
          foreach method ∈ aclass.methods,
            if Match(method.name, methodnames)==false // avoid repeating names
              Append(methodnames, method.name)        // of overridden methods
              Append(c.methods,[method,qualifiers: public synchronized,
                                       type: void,
                                       name:Concat(enter_,Qname(class,
                                                               method))
                                       body: before_body(coord, c.name)],
                               [method,qualifiers: public synchronized,
                                       type: void,
                                       name:Concat(exit_,Qname(class,
                                                              method))
                                       body: after_body(coord, c.name,
                                                         method)]
          aclass := LookupClass(aclass.super) // up the class hierarchy
    return wclass
end.
```

```
// The next three functions generate the bodies of the methods of the
// coordinator class that results from translating a COOL coordinator.
// factory_body returns a record representing the body of the factory method
//   Input:  granularity: the granularity of the coordinator;
//           cname: the name of the class that corresponds to the coordinator
//                  that is being translated.
//   Output: a statement record.
// before_body and after_body generate the body of a “before” method and a
// “after” method, respectively.
//   Input: coord:coordinator record; class: class record; meth: method record
//   Output: a statement record.

function factory_body (granularity oftype symbol, cname oftype symbol)
  let s oftype statement in
    if granularity == per_class
      s := [sequence, ([if, expr: [not, expr: [var_ref, name: one]],
                            then: [assignment, left: [var_ref,name: theCoord],
                                               right: [new, class: cname]]]),
                       [return, expr: [var_ref, name: theCoord]])]
    else s := [return, expr: [new, class: cname]]
    return s
end.

function before_body(coord oftype coordinator, class oftype class,
                     meth oftype method)
  let qname = [qualified_name, class: class.name, method: meth.name] in
    if qname ∉ coord.selfex and
       ∀ mutex ∈ coord.mutexes, qname ∉ mutex.mux and
       ∀ mm ∈ coord.mmanagers, qname ∉ mm.mnames
      return [] // Nothing to be coordinated. Return empty body.
    else
      let s oftype statement in
        s := [sequence,
              statements:([while, expr: waiting_condition(coord,class,meth),
                                  body: COOLWAITBODY],
                          [invocation, obj:[var_ref, name: qname], meth: in])]
        foreach mm ∈ coord.mmanagers,
          if mm contains on_entry statements for meth
            Append(s.statements, mm.on_entry)
        return s
end.
```

```
function after_body (coord oftype coordinator, class oftype class,
                     meth oftype method)
  let qname = [qualified_name, class: class.name, method: meth.name] in
    if qname ∉ coord.selfex and
       ∀ mutex ∈ coord.mutexes, qname ∉ mutex.mux and
       ∀ mm ∈ coord.mmanagers, qname ∉ mm.mnames
      return [] // Nothing to be coordinated. Return empty body.
    else
      let s oftype statement in
        // First, the call to “out” on the method state object
        s := [sequence,
              statements:([invocation, obj:[var_ref,name: qname], meth: out])]
        // Then, the on_exit statements
        foreach mm ∈ coord.mmanagers,
          if mm contains on_exit statements for meth
            Append(s.statements, mm.on_exit)
        // Finally, the call to notifyAll
        Append(s.statements, [if, expr: [equals,
                                         left: [field_ref, obj: qname,
                                                field: [var_ref, name: depth]]
                                         right: [literal, 0]]
                                  then: [invocation, meth: notifyAll]])
        return s
end.

// The next function generates the waiting condition for method m of the JCore
// class c, c.m. The waiting condition depends on the exclusion constraints in
// the self-exclusion set, the exclusion constraints in the mutual exclusion
// sets, and on the requires clause, if any, declared in a method manager. A
// thread wanting to execute method c.m must wait if:
//   - c.m is selfex and another thread is executing M; or
//   - for any other method c’.m’ , c.m is mutually exclusive with c’.m’
//     and another thread is executing c’.m’; or
//   - the pre-condition, as declared in a method manager, is false.
// Input:  coord: coordinator record; class: class record, meth, method record
// Output: a boolean expression
function waiting_condition (coord oftype coordinator, class oftype class,
                            meth oftype method)
  let qname = [qualified_name, class: class.name, method: meth.name],
      condSet = () in
    // Check if method is self-exclusive
    if qname ∈ coord.selfex
      Append(condSet, [invocation, obj: [var_ref, name: qname],
                                   meth: isBusyByOther])
    // Check mutual exclusion with other methods
    foreach c’ ∈ coord.classes
      foreach m’ ∈ c’.methods,
        let qname’ = [qualified_name, name: c’.name, method: m’.name] in
          if qname’ ≠ qname and ∃ mutex_k ∈ coord.mutexes such that
                                      qname ∈ mutex_k and qname’ ∈ mutex_k
            Append(condSet, [invocation, obj: [var_ref, name: qname’],
                                         meth: isBusyByOther])
    // Check if there exists a pre-condition
    if ∃ mm ∈ coord.mmanagers such that mm contains requires clause for qname
      Append(condSet, [not, expr: mm.requires])

    if CondSet == () return false
    else return [or, terms: condSet]
end.
```

V Translation Engine for RIDL

```
// translate_portal is the top translation function for RIDL.
// Input:  ptal: portal record.
// Output: a list of four class records:
//         pri: the record corresponding to the Java interface;
//         p: the record corresponding to the P class;
//         pp: the record corresponding to the PP class;
//         ct: the record corresponding to the traversals class.
function translate_portal (ptal oftype portal)
  return (generate_ri(ptal), generate_p(ptal),
          generate_pp(ptal), generate_traversals(ptal))
end.

// generate_pri generates the Java interface corresponding to the given RIDL
// portal
// Input:  ptal: a portal record.
// Output: the record corresponding to the Java interface.
function generate_pri (ptal oftype portal)
  let pri = [interface, name: Concat(ptal.class.name, PRI), supers: Remote] in
    foreach op ∈ ptal.operations, given op.params ≡ (rt1, …, rtn),
      Append(pri.methods, [method, type: ridl2java_type(type), name: op.name,
                           params: ([param, type: ridl2java_type(rt1),
                                            name: rt1.name],
                                    …
                                    [param, type: ridl2java_type(rtn),
                                            name: rtn.name]),
                           throws: RemoteException])
    return ptal
end.
// generate_p returns the record of the class used to instantiate P
// objects associated with D remote objects.
// Input:  ptal: a portal record.
// Output: the record corresponding to the class used to instantiate Ps
function generate_p (ptal oftype portal)
  let p = [class, name: Concat(ptal.class.name, P),
                  interfaces: Concat(ptal.class.name, PRI)] in
    p.variables := ([vardecl, type: ptal.class.name, name: myself],
                    [vardecl, type: RemoteStub, name: mystub])
    p.constructors := ([constructor, params: ([param, type: ptal.class.name,
                                                      name: s])
                                     body: PCONSTRUCTORBODY])
    // iterate over the remote operations, and generate one method for
    // each of them, attaching it to the methods of p
    foreach op ∈ ptal.operations, given op.params ≡ (rt1, …, rtn),
      Append(p.methods,
             [method,qualifiers: public,
                     type: ridl2java_type(op.type),
                     name: op.name,
                     params: ([param, type: ridl2java_type(rt1),
                                      name: rt1.name],
                              … ,
                              [param, type: ridl2java_type(rtn),
                                      name: rtn.name]),
                     throws: RemoteException],
                     body:p_method_body(op,
                                  traversal_names(ptal.class.name,
                                                  ptal.operations,
                                                  op.name))])
    return p
end.
```

```
// p_method_body returns the record representing the body of a method of
// the P class, which corresponds to the given remote operation.
// Input:  op: an operation record;
//         Tnames: the traversal names for the return and argument objects;
//                 traversal names may be null.
// Output: the record representing the body of a P method.
function p_method_body(op oftype operation, Tnames oftype list of field_ref)
  given op.params ≡ (rt₁, …, rtₙ), Tnames ≡ (tret, ta₁, …, taₙ),
  let calltoreal = [invocation, obj: myself, meth: op.name,
                               args: (java2ridl_obj(rt₁),  …,
                                      java2ridl_obj(rtₙ))]
    // Process the return type
    if op.type.name == void, // no return object.
                             // simply generate the call the real object.
      return calltoreal
    else if op.type.mode == gref, // return object is sent by gref.
                                  // return the p reference of the return of
                                  // the call to the real object
      return [return, expr: [field_ref, objname: calltoreal, field: _p]]
    else // return object is sent by copy.
         // return a new DArgument having as parameters the return of the
         // call to the p object and the traversal name
      return [return, expr: [new, class: DArgument, args: (calltoreal, tret)]]
end.
```

```
// generate_pp returns the record of the class used to instantiate PP
// objects associated with D remote objects.
// Input:  ptal: a portal record.
// Output: the record corresponding to the class used to instantiate PPs
function generate_pp (ptal oftype portal)
  let pp = [class, name: Concat(ptal.class.name, PP)] in
    pp.variables := ([vardecl, type: ptal.class.name, name: rself])
    pp.constructors := ([constructor], // the null-ary constructor
                        [constructor,params:([param,

type:Concat(ptal.class.name,PRI),
                                              name: s])
                                      body: PPCONSTRUCTORBODY])

  // iterate over the methods of the class, not over the remote operations
  let aclass = class, methodnames = () in
    while aclass != null
      foreach meth ∈ ptal.class.methods, given meth.params ≡ (p₁, …, pₙ),
        if Match(meth.name, methodnames)==false // avoid repeating names
          Append(methnames, meth.name)          // of overridden methods
          Append(pp.methods,
                 [method,qualifiers: public, type: meth.type, name: meth.name,
                         params: ([param, type: p₁.type, name: p₁.name], … ,
                                  [param, type: pₙ.type, name: pₙ.name])
                         body: pp_method_body(meth, ptal.operations,
                                     traversal_names(ptal.class.name,
                                                     ptal.operations,
                                                     op.name))])
      aclass := LookupClass(aclass.super)
    return pp
end.
```

```
// pp_method_body returns the record representing the body of a method of
// the PP class.
// Input:  meth: a method record representing a method from a JCore class;
//         remoteops: a list of remote operation records;
//         Tnames: the traversal names for the return and argument objects;
//                 traversal names may be null.
// Output: the record representing the body of a PP method.
function pp_method_body (meth oftype method,
                          remoteops oftype list of operation,
                          Tnames oftype list of field_ref)
  if Match(meth.name, Names(remoteops)) == false
    return INVALIDREMOTEOPERATION
  else
    let op = get_operation_from_name(meth.name, remoteops) in
      return [try, body: remote_call(op, Tnames),
                   catches: ([catch, exception: RemoteException,
                                     body: catch_body(op.type.name)])]
end.


// remote_call returns a record representing the call from the PP object to
// its counterpart P object.
// Input:  op: an operation record;
//         Tnames: the traversal names for the return and argument objects;
//                 traversal names may be null.
// Output: the call from the pp to the p.
function remote_call (op oftype operation, Tnames oftype list of field_ref)
  given op.params ≡ (rt_1, …, rt_n), Tnames ≡ (tret, ta_1, …, ta_n),
  let calltop = [invocation, obj: rself, meth: op.name,
                              args: (ridl2java_obj(rt_1, ta_1), … ,
                                     ridl2java_obj(rt_n, ta_n)))]
    // Process the return type.
    if op.type.name == void, // no return object.
                            // simply generate the call the P object
      return calltop
    else if op.type.mode == gref, // return object is passed by gref.
                                  // return the proxy to remote object of
                                  // the return of the call
      return [return, expr: [new, class: op.type.name,
                                  args:([new, class: Concat(op.type.name,PP),
                                              args: calltop])]]
    else // return object was passed by copy.
         // This method gets a DArgument back from the remote call to P.
         // Extract the object.
      return [return, expr: [field_ref, obj: calltop, field: obj]]
end.

function catch_body (typename oftype symbol)
  let s = [sequence, statements: (ERRORMESSAGE)] in
    // if the return type of the method is not void, we need to return
    // something when a RemoteException is thrown.
    if typename ≠ void
      Append(s.statements, [return, expr: NullValueForType(type)]
    return s
end.
```

```
// generate_traversals returns the record representing the class that
// contains the repository of traversals associated with a RIDL
// portal.
// Input:  ptal: a portal record representing a RIDL portal.
// Output: a record representing the traversals class, or null if there are
//         no traversal specifications in ptal.
function generate_traversals (ptal oftype portal)
  let vars = (), counter = 0 in
    // name the traversals sequentially as they appear in the portal
    foreach op ∈ ptal.operations,
      foreach type ∈ {op.type} ∪ Types(op.params),
        if is_primitive(type.name) == false and
           type.mode == copy and type.traversal ≠ null,

          Append(vars, [vardecl, qualifiers: static, type: Traversal,
                                 name: Concat(t, ToString(counter))])
          counter := counter+1
    if counter = 0, return [] // there are no traversals in r
    else
      let traversalclass = [class, name: Concat(ptal.class.name, Traversals),
                                   variables: vars] in
        Append(traversalclass.variables, THEONCEBOOLEAN)
        traversalclass.methods := ([method,
                                    qualifiers: public static synchronized,
                                    type: void, name: init,
                                    body: traversals_init_body(ptal)])
        return traversalclass
end.

// traversals_init_body returns a record representing the body of the
// method for instantiating the traversals associated with a portal.
// The statements in the return record are a translation of the traversals
// in RIDL.
// Input:  ptal: a portal record representing the RIDL portal
// Output: a record representing the body of the init method
function traversals_init_body (ptal oftype portal)
  let s = [sequence, statements:(THEINCOMPLETECLASSVARDECL, THEONCETEST)],
      counter = 0, ctname = Concat(ptal.class.name, Traversals) in
    // the traversals were named sequentially as they appear in the portal
    foreach op ∈ ptal.operations,
      foreach type ∈ {op.type} ∪ Types(op.params),
        if is_primitive(type.name) == false and
           type.mode == copy and type.traversal ≠ null,
          let tname = Concat(t, ToString(counter)) in
            Append(s.statements, [assignment, left: tname,
                                  right: [new, class: Traversal,
                                               args: (tname, ctname)]])
            foreach iclass ∈ type.traversals.incompletes,
              Append(s.statements, [assignment, left: c,
                                    right: [new, class: IncompleteClass,
                                                 args: iclass.name]])
              foreach part ∈ iclass.missing,
                Append(s.statements, [invocation, obj: c, meth: bypass,
                                                  args: part])
              Append(s.statements, [invocation, obj: tname,
                                                methname: incompleteClass,
                                                args: (iclass)])
          counter := counter+1
    Append(s.statements, THEONCEASSIGNMENT)
    return s
end.
```

```
// traversal_names returns a list containing the names of the traversals
// associated with the return and argument objects of a particular operation.
// Input:  classname: the name of the class associated with the portal
//                    that is being translated.
//         operations: a list of operation records representing the remote
//                     operations of a portal;
//         opname: the operation name.
// Output: a list of traversal names. The size of this list is exactly the
//         number of arguments of the operation plus one (return value). The
//         first traversal corresponds to the return value, the second to the
//         first argument, etc.
function traversal_names (classname oftype symbol,
                          operations oftype list of operation,
                          opname oftype symbol)
  if Match(opname, Names(operations)) == false // no such remote operation.
    return null
  else
    let the_op = get_operation_from_name(opname, remoteops) in
      let counter = objs_passed_by_copy_until_op(operations, the_op),
          traversalnames = () in
        foreach rtype ∈ {op.type} ∪ Types(op.params),
          if is_primitive(rtype.type) == false and
             rtype.mode == copy and type.traversal ≠ null
            Append(traversalnames,
                   [field_ref, obj: Concat(classname, Traversals),
                               field: Concat(t, ToString(counter))])
            counter := counter+1
          else
            Tnames ← (null)
end.

// The next 3 functions convert between Java types/objects and
// the types/objects used by the DJ library
function ridl2java_type (rtype oftype ridl_type)
  if is_primitive(rtype.type) == true return rtype.type
  else if rtype.mode == gref return Concat(rtype.type, RI)
  else return DArgument
end.
// Java2RIDL_arg converts arguments from the lower DJ library format
// to the source format
function java2ridl_obj (rtype oftype ridl_type)
  if is_primitive(rtype.type)  == true // simply return the argument name
    return rtype.name
  else if rtype.mode == gref // it’s a P. Instantiate a local proxy
    return [new, class: rtype.type,
                 args: ([new, class: Concat(rtype.type, PP)
                              args: rtype.name])]
  else // it’s a DArgument. Extract the internal object.
    return [field_ref, obj: rtype.name, field: obj]
end.
// RIDL2Java_arg converts arguments from the source format to the
// lower DJ library format
function ridl2java_obj (rtype oftype ridl_type, tname oftype symbol)
  if is_primitive(rtype.type) == true // simply return the argument name
    return rtype.name
  else if rtype.mode == gref// return the P object associated with argument
    return [field_ref, obj: rtype.name, field: _p]
  else                    // Build DArgument from argument and traversal.
    return [new, class: DArgument, args: (rtype.name, tname)]
end.
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// DArgument.java
//
// This class describes DArguments passed in remote operations. DArguments
// consist of an object (possibly null) and a traversal (possibly null).
// This class implements only the two marshaling methods that are called
// from the RMI run-time: writeExternal and readExternal.
//

import java.lang.reflect.*;
import java.io.*;

final public class DArgument implements Externalizable {
  public Object obj;
  Traversal trav;

  public DArgument() {super();}
  public DArgument(Object o, Traversal t) {
    obj = o; trav = t;
  }

  public void writeExternal(ObjectOutput s) {
    Trace.println("IN DArgument.writeExternal");
    try {
      s.writeObject(trav);
      if (trav == null) { // can be a DObject or not. Deep copy for DObjects.
       s.writeObject(obj);
      }
      else { // it's a DObject for sure
       if (obj == null)  { // send a null class
         s.writeObject(null);
       }
       else {
         // must send the class name, so that readExternal knows what
         // object to instantiate.
         String classname = obj.getClass().getName();
         s.writeObject(classname);
         ((DObject)obj)._d_writeExternal(s, trav);
       }
      }
    } catch (IOException e) {
      System.err.println("Error in DArgument.writeExternal\n" + e.toString());
    }
    Trace.println("OUT DArgument.writeExternal");
  }

  public void readExternal(ObjectInput s) {
    Trace.println("IN DArgument.readExternal");
    try {
      trav = (Traversal)s.readObject();
      if (trav == null) // can be a DObject or not
       obj = s.readObject();
      else { // it's a DObject
       // get the actual traversal object
       trav =(Traversal)Class.forName(trav.remoteinterface).//oops, must break
                              getDeclaredField(trav.name).get(null);
       String classname = (String)s.readObject();

       if (classname != null) {
         obj = Class.forName(classname).newInstance();
         ((DObject)obj)._d_readExternal(s, trav);
       }
       else {
         obj = null;
       }
      }
    } catch (Exception e) {
      System.err.println("Error in DArgument.readExternal\n" + e.toString());
    }
    Trace.println("OUT DArgument.writeExternal");
  }
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// DInvalidRemoteOperation.java
//

public class DInvalidRemoteOperation extends Exception {
  DInvalidRemoteOperation() {
    super();
  }
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// DJNaming.java
//
// This class is the DJ interface to Java's Naming class.
// On bind, it sends portal (P) references to the Name Server.
// On lookup, it gets the portal (P) references from the name server, and
// instantiates the two proxies (two levels) that are necessary.
//
// NOTE: this bind service is, in fact, the rebind service of Java.
//

import java.rmi.*;
import java.net.*;
import java.lang.reflect.*;

public class DJNaming{
  private final static String stubTailString="P_Stub";

  public static void bind(String name, DObject obj)
  throws InvalidRemoteObjectException, RemoteException, MalformedURLException{
    Remote remoteObj = null;
    try {
      remoteObj = (Remote)(obj.getClass().getField("_p").get(obj));
    }
    catch (IllegalAccessException e) {
      System.err.println("There has been a serious error."+
                   "Error Code: DJNRP");
      System.exit(9);
    } catch(NoSuchFieldException e){
      throw new InvalidRemoteObjectException
       (obj + " is not a valid remotable object\n");
    }
    Naming.rebind(name, remoteObj);
  }

  public static DObject lookup(String name)
  throws InvalidRemoteObjectException, RemoteException, NotBoundException,
    MalformedURLException, java.rmi.UnknownHostException{

      Remote remoteObject = Naming.lookup(name);
      String className = getTheClassName(remoteObject);
      DObject theObject = null;

      try{

       Class theClass = (Class.forName(className));
       Class ppClass = (Class.forName(className + "PP"));
       theObject = (DObject)theClass.newInstance();
       Object ppObject = ppClass.newInstance();
       (theClass.getField("_rself")).set(theObject, ppObject);
       (ppClass.getField("rself")).set(ppObject, remoteObject);

      } catch(ClassNotFoundException e){

       System.err.println("There has been a serious error."+
                   "Error Code: DNLUPCNF");
       System.exit(4);
      }catch(InstantiationException e){
       throw new InvalidRemoteObjectException("Could not lookup object " +
                                 name + "\n"+
               "Probably because I could not find a 0-arity constructor\n"+
                                 " for "+className);
      }catch(IllegalAccessException e){
       System.err.println("There has been a serious error." +
                        "Error Code: DNLUPIA");
       System.exit(4);
      }catch(NoSuchFieldException e){
       System.err.println("There has been a serious error."+
                        "Error Code: DNLUNSF");
       System.exit(4);
      }
      return theObject;
  }


  private static String getTheClassName(Object obj)
  throws InvalidRemoteObjectException{
    String className = ((obj.getClass()).getName());
    if (className.endsWith(stubTailString)) {
      return
       className.substring(0,className.length()-stubTailString.length());
    } else {
      throw new InvalidRemoteObjectException (obj +" not valid remote object"
                                        + "Error Code: DNGRC" +
                                        className);
    }
  }

}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// DObject.java
//
// The interface that is common to ALL classes that pass through the
// RIDL weaver.
//

import java.io.*;

public interface DObject extends Externalizable {
  void _d_readExternal(ObjectInput in, Traversal t);
  void _d_writeExternal(ObjectOutput out, Traversal t);
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// DPartCutter.java
//
// Implemented by IncompleteClass and by UnknownIncompleteClass.
// DPartCutters are used in the marshaling methods of DObjects.
//

public interface DPartCutter {
  boolean bypassPart(String name);
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// IncompleteClass.java
//
// IncompleteClass stores which parts of the class are bypassed during
// marshaling. The class is identified by "name", and the parts are
// stored in the Vector of strings "bypass".
//

import java.util.*;

final public class IncompleteClass implements DPartCutter {
  public String name;
  public Vector bypass = new Vector (4, 4);

  public IncompleteClass(String n) { name = n; }

  // called during the set up of Traversals.
  public void bypass (String part) {
    bypass.addElement(part);
  }

  // called from the marshling methods during marshaling.
  public boolean bypassPart(String pname) {
    String p = null;
    for (Enumeration e = bypass.elements() ; e.hasMoreElements() ; ) {
      p = (String)e.nextElement();
      if (p.equals(pname)) {
       return true;
      }
    }
    return false;
  }
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// InvalidRemoteObjectException.java
//

import java.rmi.*;

public class InvalidRemoteObjectException extends RemoteException{
  public InvalidRemoteObjectException(){
    super();
  }

  InvalidRemoteObjectException(String str){
    super(str);
  }
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// MethState.java
//
// This is the only class in the DJlib that is used to support COOL.
// It is used to capture the execution state of methods of JCore objects.
//

import java.util.Vector;

final public class MethState {
  public int depth = 0; // to handle re-entrance
  private Vector tl = new Vector(5);

  final public boolean isBusyByOther() {
    if (depth > 0 && !tl.contains(Thread.currentThread()))
      return true;
    else return false;
  }
  final public void in() {
    depth++;
    tl.addElement(Thread.currentThread());
  }
  final public void out() {
    depth--;
    tl.removeElement(Thread.currentThread());
  }
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// Trace.java
//

public class Trace {
  static boolean    TRACE = true;

  public static void traceON() { TRACE = true; }

  public static void traceOFF() { TRACE = false; }

  public static void println(String str) {
    if (TRACE)
      print(str + "\n");
  }

  public static void print(String str) {
    if (TRACE) {
      System.out.print(str);
      System.out.flush();
      try { Thread.sleep(100); }
      catch (InterruptedException e) {}
    }
  }
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// Traversal.java
//
// This class represents traversal directives. Traversal directives
// are attached to remote interfaces (stored in remoteinterface), and
// during the translation process they are given a name (stored in name).
// They store the classes that are incomplete, in a Vector of
// IncompleteClasses.
//

import java.io.*;
import java.util.*;

final public class Traversal implements Serializable {
  public String remoteinterface;
  public String name;
  public Vector classes = new Vector(4,4);

  private static final DPartCutter Unknown = new UnknownIncompleteClass();

  public Traversal(String n, String ri) {
    remoteinterface = ri;
    name = n;
  }

  public void incompleteClass (IncompleteClass c) {
    classes.addElement(c);
  }

  public DPartCutter isIncompleteClass(String cname) {
    for (Enumeration e = classes.elements() ; e.hasMoreElements() ; ) {
      IncompleteClass c = (IncompleteClass)e.nextElement();
      if (c.name.equals(cname)) {
       return c;
      }
    }
    return Unknown;
  }

  private void writeObject(ObjectOutputStream s) {
    try {
      Trace.println("In Traversal.writeObject");
      s.writeObject(remoteinterface);
      s.writeObject(name);
    } catch (IOException e) {
      System.err.println("Error in packing Traversal.\n" + e.toString());
    }
    Trace.println("Out Traversal.writeObject");
  }

  private void readObject(ObjectInputStream s) {
    try {
      Trace.println("In Traversal.readObject");
      remoteinterface = (String)s.readObject();
      name = (String)s.readObject();
    } catch (Exception e) {
      System.err.println("Error in unpacking Traversal.\n" + e.toString());
    }
  }
}
```

```
//
// Copyright Xerox Corporation, 1997.  All rights reserved.
//
// UnknownIncompleteClass.java
//
// When a Traversal is asked whether a certain class is incomplete or not,
// and it finds that that class is not mentioned, then it returns an
// instance of this class. This class basically says that no bypass
// should be done (all parts should be copied).
//

final public class UnknownIncompleteClass implements DPartCutter {
  public boolean bypassPart(String name) { return false; }
}
```

Appendix E. User Reports

This appendix contains (I) the part of the users' final report of the space war application related to Cool and Ridl; (II) the email messages that the alpha-users have sent reporting their experience of using DJava; (III) their answers to a survey written by Prof. Gail Murphy. All documents are shown here with the permission of the people involved.

I Aspect Programming in the Space War Application

Distribution

The server and each player run on their separate machines. Remote calls occur when a player joins the game (and gets a copy of the universe from the server), a player transmits an action to the masterhelm object on the server's machine, and the player's local helm gets action an from the masterhelm. The player's clockTick method is called remotely by the server. Aside from joining the game (when information is swapped back and forth between player and server), remote communication is fairly simple.

Synchronization

Broadcasting of the ship control events (MasterHelm methods as well as Server.clockTick must be synchronized in order for all the players to see the same stream of events and keep their Universes in sync.

We achieve this by requiring that only one event can be being broadcast at a time and it must be broadcast to all the sites. The second part of this requirement is met in MasterHelm.<event> and Server.clockTick method code. The first part is guaranteed by the following synchronization conditions.

No event broadcast should occur while some player(s) has an inconsistent Universe which is the case during Server.joinGame and Server.newShip calls. We achieve this by simply stopping clock tick and message delivery processing (mutex requirement) for the duration of those calls. This is a simple but inefficient solution that is likely to change in the future.

In order to guarantee that all players are notified when a new ship has to be added to the Universe, we require selfexclusion and mutexclusion of Server.joinGame and Server.newShip.

I (Mark) find it pretty amazing (and somewhat alarming) that so many requirements are implemented by only 10-line piece of Cool spec.

Also, there is a synchronization issue that we had to solve in component code, rather than in Cool. In order to get the desired response-time we had to lower the priority of the Timer thread that calls Server.clockTick() method. Without that, the Java thread scheduler often kept choosing clockTick() thread over event threads when both were waiting on a mutex semaphore. This resulted in unacceptably long delays in event delivery.

II Users' Evaluation

Evaluation by Beth Seamans :

Anything I could say in direct response to this would repeat a bug report, a feature request, or a response to one of Gail's surveys (and usually both). John Lamping has compiled a fairly complete list of DJava shortcomings that reflect the bug report/feature request side pretty well. So rather than do that, I would like to expound briefly (if that's not an oxymoron) on a point that I think is important, and that I brought up in some recent meetings but has not really been expressed at any other time. Also, I'll be going back to the library code and re-working it for public consumption, so hopefully there will be (more) explicitly DJava-significant comments and explanations embedded. I'll let you know when that's done. If the above is insufficient, let me know and I'll give it another try.

Ridl and Cool make it much easier (faster, cleaner, more efficient) to express a concept, as long as Ridl or Cool was designed to express that concept. This has the side-effect of making it easier to think about both those concepts and the components where the concept _used_ to be expressed. I think that our experiences coding in DJava have shown this quite well. By providing a pre-defined set of concepts or tools to the user you coerce the user into using those tools even when inappropriate (when all you have is a hammer, everything looks like a nail), but this is an inherent limitation, and all you can do is define the 'best' set of tools you can (and part of our job this summer was to help discover what the best set looks like).

So far, so good. But I would say that a large part of what makes DJava so usable and so attractive is how it simplifies the design process by exposing the concerns, and little has been done to exploit this benefit in the language(s). I have heard a lot of comments from the Guinea Pig Team to the effect of, 'if you worked in OO but had a _really clean design_, the benefit of AOP is not that great'. Even if that's true, the key is it's not that easy to create that 'really clean design'. We demonstrated that, too, pretty effectively. A key point (although certainly not the only point) seemed to be the need to clearly distinguish _what_ the concerns are, and _how well_ they are expressed by the languages provided. An optimal mapping is crucial to gaining the promised benefits. I keep bringing up the Ridl/replication muddle that Spacewar went through. More generally, design-building tools (of the kind that would be successful in OO too) would help enormously when deciding what to express in those simple, friendly languages. This could be regarded as out of the DJava scope and in the user-interface domain, but I think if DJava needs it to thrive, it should be regarded as part of the language.

I also was interested to hear Gail's comment that reading the Ridl code was very hard. The 'what' was exposed beautifully, but the motivation remained completely obscure (sometimes to the programmers, too). Without the mechanism to express that motivation, a large part of the DJava advantage is lost. Good comments can fill that need, but good comments are as hard to find as good designs, especially since even the most diligent commenter can underestimate how much explanation is required. Building hooks into the language itself to express the 'why' as well as the 'what' could help fulfil the DJava promise.

(Beth Seamans is graduate student at Stanford University.)

Evaluation by Jared Smith-Michelson :

From my experience, the best thing about Cool and Ridl is that they greatly ease the burden of programming. Knowing nothing about the inner-workings of RMI, but having Ridl under my belt, I was able to write distributed programs. All it took was a separate Ridl file where I defined certain methods as being remote. I can only imagine the difficulties I would have had in working directly with RMI. Cool also simplified coding. Although Java has a synchronized method, it does not have explicit support for mutual exclusion. Without Cool, writing mutually exclusive methods would require the implementation of locks. The other nice feature of Cool and Ridl is that they succeed in separating concerns. Making a small change in how a field is passed or slighting reordering a mutual exclusion group is trivial in Ridl and Cool.

No distributed server-less design of the space war game was ever implemented. The main reason is that it would have been far to difficult. The synchronization issues would have been dizzying. Even with Cool and Ridl, many new abstractions would have had to have been developed to handle the communication between machines. It seems that although Ridl works very well for systems which make occasional remote calls, it does not solve the problem of making a evenly distributed, server-less, synchronizationally intense network. However, this may very well be asking too much. Perhaps it's the job of a new aspect language?

One feature I'd really like to see in DJava is support for per method copy directives. In Ridl, one should be able to specify not only which fields are passed and how, but whether or not specific methods are to be called locally or remotely. Let me offer two examples which I have come across.

The first is in the spacewar game. When a new Player joins the a game, a copy of the Universe is passed over. In the Universe, are copies of other Player objects. However, only a small portion of the Player object is needed, namely the id number, the name, score, etc. A large portion of the Player object need not be copied, including Console, Server, and others. Handling this problem is fairly straightforward using the current implementation of Ridl. But, it seems more natural to describe which methods will be needed, rather than which fields. For example, we wish the method Player.getName() to be called locally since it's called every time the Ship is painted to the screen. The getName() method could be declared as "passed by copy" and Ridl could then figure out which fields the getName methods needs. The second example is regarding the distributed library. Library objects contain hashed lists of other remote Libraries to which they can forward Book Queries. Obviously, the lists need to be of grefs to Libraries (you can't copy a Library!) But in order to hash an Object, it needs to support the methods hashCode and equals. If the Library is regularly rehashing its indices, these methods should be remote calls. Currently, it is not possible to make certain calls remote if all you have is a gref. It would therefore be nice if Ridl supported per method copy directives.

Another feature which might be nice to have in DJava is gref to local copy conversion. Say object Foo creates an object Bar and passes it out as a gref. Then, way Foo receives the very same object Bar back again (as a gref). Could there then be some way of converting that reference to the actual Bar object, since it's in the same computation space already? Could DJava recognize that Foo already has a copy of the Bar object and that there's no need for a gref?

It would also be nice to have some more aspect languages for DJava. I was thinking along the lines of error handling, and tract debugging.

(Jared Smith-Michelson is a junior student at the Massachusetts Institute of Technology.)

Evaluation by Mark Marchukov:

I liked the way Cool handled synchronization. A short coordinator was sufficient to guarantee that all the clients in our game see the same stream of game events. On the other hand, I think that someone unfamiliar with the design of Spacewar would have a hard time understanding what purpose that coordinator really served in the program. A long comment was necessary to fully describe its purpose.

We found that we didn't need to use all the features of Cool in Spacewar, at least not in the version we finally came up with. Simple self-exclusion and mutual-exclusion declarations were enough. So I think that Spacewar was not really coordination-intensive.

Ridl with its convenient support for global references was useful at the initialization stages of the game when the references between objects have to be set up. It also pushed us towards a design that looked like there was no distribution: we used global references to deliver messages to individual ships across the network as opposed to delivering them to *clients* that would in turn deliver them to local ships. But since RMI is grossly inefficient, our program was slow too and no part of Ridl could help us make it faster by sending less data over the network. That's a pity.

And lastly, the design of Spacewar that we used in the distributed version was heavily based on replication of objects and keeping replicas on different sites in sync. Ridl offered us no direct control over replication and distributed synchronization aspects. After all, it wasn't its job, it wasn't designed for that. This is I think why the DJava version of Spacewar has approximately the same complexity as the Java-only version that we wrote later.

(Mark Marchokov is a graduate student at the University of Virginia)

III Prof. Murphy's survey

The part of the survey that is shown here is the following:

A Survey for the AOP Trailblazers
July 23, 1997

This survey consists of eleven main questions (some with sub-questions). Your answers need not be long, but try to refer to concrete examples (through pointers to code, etc.) whenever possible.

Using Aspect-Oriented Programming.

1. Please estimate the amount of time you spend thinking about aspects when:
   i. designing a component
   ii. implementing a component
   iii. Are there any particular components/aspects for which the time has been much different than the time stated above? If so, which ones?
2. Are the components you have written independent from their aspects (e.g., could you remove the aspects and still have functioning components, is there partial dependence, etc.). If applicable, describe an instance in which there was coupling between the component and the aspect. (i.e., the component was written differently because of the aspect)
3. In your first report, you described four different "processes" for AOP design. Which one most reflects the way you are doing Aspect-Oriented design? *(NOTE: not sufficient context for understanding this question; the answers were omitted)*
4. Do you find it (easy/moderately difficult/difficult/impossible) to read someone else's aspect code and understand it? How do you resolve any problems you experience in understanding an aspect?
5. How do you decide if your aspect code is working? (e.g., do you run test cases, reason through the code, etc.?)
6. What procedures do you use to debug your DJava code?

Answers by Beth Seamans:
1.
i. designing a component
some
ii. implementing a component
very little

'Some' is hard to define. Most of the time, the aspects did not require an inordinate amount of attention. When Ridl reality did not match our understanding, or when a Ridl operation supporting a clean design did not exist, we were forced to spend much more time. Regarding the implementation, we usually did a detailed design of the class interface structure, where most of the aspect issues crop up, so the implementation was fairly routine.

We have been thinking that there may be more sophisticated and efficient ways to use Cool, and that will take more consideration.

iii. As noted above, the unsolved Ridl bugs and unimplemented Ridl feature requests require more time on the design.

2. I think the components are independent, but would be quite inefficient since there is currently a lot of duplication of information across machines.
   The Masterhelm/Helm relationship is explicit in the component code, and arises from the distributed aspect.

4. The aspect code, since it is so minimal, is usually easy to understand. It can be more difficult to see the 'why', but the 'what' is clear. [Resolving problems of understanding] Talking and drawing diagrams. Repeat as necessary. Having four people together, with ready access to expert help, makes the confusion/clarification cycle very small.

5. We reason through the code and play the game to test. Since it is non-deteRMInistic, it's difficult to make sure all bases are covered, but quite a bit of hands-on playing gets done.

6. We try to run it without the aspects first, if at all possible. The debugging process is mostly a matter of isolating through printed debug statements.

Answers by Jared Smith-Mickelson :

1

i. For the most part, I don't think too much about aspects while designing a component. Although, that's not to say I ignore them altogether. Sometimes, knowing that I have Cool and Ridl as tools effects the design of the component. For example, in the spacewar game, there is a Console object which paints graphics to the screen and a Universe object which maintains the positions and velocities of the SpaceObjects. The Server periodically sends clock ticks to the Universe so that the game stays synchronized. The problem we were facing was how to notify the Console when the Universe was ticked. We were thinking along the lines of having the Universe tick the Console or having the Server tick both, when it occurred to us that the best solution was to write some Cool code. We realized that Console doesn't even need a clockTick method. All it needs to do is continuously redraw the screen. A Cool file could describe a guard which would suspend the Console until the Universe was updated. This would keep synchronization concerns where they belong, in a Cool file.

ii. During implementation, aspects play a stronger role. This is mainly due to the fact that most of the gritty details are worked out here. When implementing components, I find myself thinking about remote argument passing quite often. I'm concerned about which calls should be remote, and whether or not object foo needs a reference to bar or a copy of bar. For the most part, I find myself thinking about the distribution aspect of the program. I usually save synchronization for the end, for I have found that most of the time method exclusion can be added without having to change underlying design or implementation.

2. I wouldn't call them independent. They'd still compile, but their functionality would likely be corrupted. However, I see nothing wrong with this. After all, components are dependent on other components, aren't they? Therefore, why can't they depend on aspects? To me, aspects and components are means by which to modularize the code, not decouple it. For example, in the Console/Universe example above, the program wouldn't work properly without the Cool

code. The Console, without sleeping or suspending its own thread, would hog the processor and virtually freeze everything else.
In version 0.3 of the spacewar program, there is a very complex protocol for the joining of Players to the game. This protocol lends itself almost entirely to what Ridl could and could not due at the time of implementation. For example, in a couple places, it was necessary for the Server to create a global reference to a local object. Since Ridl could not do this, objects had to be passed back and forth between remote spaces using methods not present in the original design.

4. I find aspect code to be extremely clear and easy to read. I can only imagine the nightmare of reading through woven Java output looking for deeper meaning behind the slew of Locks and TraversalPatterns.
5. When it comes to deciding whether my code is working properly, I treat aspects the same as I do components. First I run the program, looking for correct behavior. If something is not working the way I anticipated, I reason through the code. If I can't determine what's going wrong, I write smaller test cases, in an attempt to get to the root of the problem.
6. Usually, the compiler or runtime error messages will suffice. But if I have to, I put in print-outs to trace the execution path. It would be nice to have a debugger to step through the code, but alas...

Answers by Mark Marchukov:
1.
i. It depends on what you mean by "aspect". If distribution is an "aspect", then 60%. This includes discussing both .Cool and .Ridl declarations and component structure and actions. If only .Cool and .Ridl declarations are "aspects", then 20%.
ii. 20%. We are usually doing a very detailed design. I didn't include the time we spend to find work-arounds for weaver bugs.
2. Looks like your definition of "aspect" is close to the second one above... Let's see... Server and MasterHelm have to be synchronized. They will function without their synchronization aspects, but not in this program. Their Ridl parts can be removed. Not so for Helm that relies on the fact that setShip(s) gets a copy of s, not a reference. Universe could function without its coordinator, in a single-threaded program. Player would be unusable without its Ridl part. Player is a good example. Its remoteClone() method simply returns -this-. And it is all up to its Ridl aspect to determine what and how is returned. It is meaningless to have a method that returns -this- in a class, unless it has a distribution aspect.

4. Aspect code that we write is very small. I wouldn't even call it "code" because it doesn't have a flow of control in it. These are declarations. They are easy to understand. However, to understand what the program will do when the component code corresponding to this aspect specification executes, one must understand the *component* code. And this is not always easy. This would require looking at both component code and aspect specs and going "a-ha! when I make this call the result comes by copy, these fields are returned by global reference and these are skipped. That's why I'm getting a null pointer exception!"
5. I don't think I have had to look at an unfamiliar piece of aspect code so far. But if I had, I would have done what I described above: looked at both the aspect and its component(s).
6. System.out.println()

Bibliography:

1. Agha G. and Hewitt C. *Concurrent programming using Actors.* In *Object-Oriented Concurrent Programming* pp. 37-53, *Series in Computer Science*, Yonezawa A. and Tokoro M., Eds. MIT Press. 1987.
2. Aksit M., Wakita K., Bosch J. et al. *Abstracting object interactions using composition filters.* In proc. *ECOOP'93 Workshop on Object-Based Distributed Programming*, pp. 152-184, 1993.
3. America P. *POOL-T: A parallel object-oriented language.* In *Object-Oriented Concurrent Programming* pp. 55-86, Yonezawa A. and Tokoro M., Eds. MIT Press. 1987.
4. Armstrong J., Williams M., Wikström C. et al. *Concurrent Programming in Erlang.* Prentice Hall, 1996.
5. Assenmacher H., Breitbach T., Buhler P. et al. *PANDA -- Supporting distributed programming in C++.* In proc. *ECOOP*, pp. 361-383, Kaiserlautern, Germany, 1993.
6. Atkinson C. *Object-oriented Reuse, Concurrency and Distribution: an Ada-based approach.* Addison-Wesley, 1991.
7. Bergmans L. *Composing concurrent objects.* PhD thesis, Department of Computer Science, University of Twente, Twente, 1994.
8. Bharat K. and Brown M. *Building Distributed, Multi-User Applications by Direct Manipulation.* In proc. *ACM Symposium on User Interface Software and Technology*, pp. 71-81, 1994.
9. Black A. and Artsy Y. *Implementing Location Independent Invocations.* In proc. *9th International Conference on Distributed Computing Systems*, pp. 550-559, 1989.
10. Black A., Hutchinson N., Jul E. et al. *Distribution and abstract data types in Emerald.* In *IEEE Transactions on Software Engineering* vol. SE-13(1) pp. 65-76, 1987.
11. Briot J.-P. and Yonezawa A. *Inheritance and synchronization in concurrent OOP.* In proc. *ECOOP'87*, pp. 32-40, 1987.
12. Cardelli L. *Obliq: a language with distributed scope.* Digital Equipment Corporation, Palo Alto. Technical Report 122 1994.
13. Caromel D. *Programming abstractions for concurrent programming - a solution to the explicit/implicit control dilemma.* In proc. *TOOLS 3*, pp. 245-253, Sydney, Australia, 1990.
14. Champeaux D. d., Lea D. and Faure P. *Object-Oriented System Development.* Addison-Wesley, 1993.
15. Chiba S. *A Metaobject Protocol for Enabling Better C++ Libraries.* PhD dissertation, Department of Information Science, University of Tokyo, Tokyo, 1996.
16. Dahl O.-J., Dijkstra E. and Hoar C. A. R. *Structured Programming.* Academic Press, London, 1972.
17. Dijkstra E. *Cooperating Sequential Processes.* In *Programming Languages,* Genyus, Ed. Academic Press. 1968.
18. Dijkstra E. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

19. Dijkstra E. *Go to statments considered harmful.* In *Communications of the ACM* vol. 11(3) pp. 147-148, 1968.
20. Frølund S. *Inheritance of synchronization constraints in concurrent object-oriented programming.* In proc. *ECOOP'92*, pp. 185-196, 1992.
21. Gamma E., Helm R., Johnson R. et al. *Design patterns -- elements of reusable object-oriented software.* Addison-Wesley, 1994.
22. Gehani N. H. *Capsules: A Shared Memory Access for Concurrent C/C++*. In *IEEE Trans. on Parallel and Distr. Systems* vol. 4(7), 1993.
23. Gosling J., Joy B. and Steele G. *The Java^{TM} Language Specification.* Addison-Wesley, 1996.
24. Hoare C. A. R. *Monitors: An Operating System Structuring Concept.* In *Communications of the ACM* vol. 17(10) pp. 549-557, 1974.
25. Hürsch W. L. and Lopes C. V. *Separation of Concerns.* Northeastern University, Boston, USA. Technical report NU-CCS-95-03, February 1995.
26. Java *Java Network API.* JavSoft, Web Pages, http://www.javasoft.com:80/products/jdk/1.1/docs/api/Package-java.net.html
27. Java *Java Remote Method Invocation Specification, Revision 1.0.* JavaSoft, October 1996.
28. Jul E., Levy H., Hutchinson N. et al. *Fine-Grained Mobility in the Emerald System.* In *ACM Transactions on Computer Systems* vol. 6(1) pp. 109-133, 1988.
29. Kafura D. *Inheritence in Actor-based concurrent object-oriented languages.* In proc. *ECOOP'89*, pp. 131-145, 1989.
30. Kafura D. G. *Concurrent object-oriented real-time systems.* Department of Computer Science, Virginia Tech. Technical Report TR 88-47 1989.
31. Karaorman M. and Bruno J. *Introducing concurrency to a sequential language.* In *Communications of the ACM* vol. 36(9) pp. 103-116, 1993.
32. Kiczales G. *Foil for the Workshop on Open Implementation.* Xerox PARC, Web pages, http://www.parc.xerox.com/oi/workshop-94/foil/main.html
33. Kiczales G. *Why are Black Boxes so Hard to Reuse?* Invited Talk, OOPSLA'94, Video tape, Web pages, http://www.parc.xerox.com/oi/gregor-invite.html
34. Kiczales G., des Rivères J. and Bobrow D. G. *The Art of the Metaobject Protocol.* MIT Press, 1991.
35. Kiczales G. and Lamping J. *Issues in the design and specification of class libraries.* In proc. *OOPSLA'92*, pp. 435-451, Vancouver, Canada, 1992.
36. Kiczales G., Lamping J., Lopes C. et al. *Open Implementation Design Guidelines.* In proc. *International Conference on Software Engineering*, Boston, 1997.
37. Kiczales G., Lamping J., Mendhekar A. et al. *Aspect-Oriented Programming.* In proc. *European Conference on Object-Oriented Programming*, Finland, 1997.
38. Knuth D. *Structured programming with go to statements.* In *Computing Surveys* vol. 6, 1974.
39. Lamping J. *Typing the specialization interface.* In proc. *OOPSLA'93*, pp. 201-214, Washington, DC, 1993.
40. Lea D. *Concurrent Programming in Java^{TM}: design principles and patterns.* Addison-Wesley, 1996.
41. Lewis B. and Berg D. J. *Threads Primer: A Guide to Multithreaded Programming.* Sun Microsystems Press Books., 1995.

42. Lieberherr K. *Adaptive Object-Oriented Software: the Demeter Method with Propagation Patterns.* PWS Publishing Company, Boston, Massachusetts, 1996.

43. Lieberherr K. and Patt-Shamir B. *Traversals of Object Structures: Specification and Efficient Implementation.* Northeastern University, Boston. Technical Report NEU-CCS-97-15, September 1997.

44. Lieberherr K. J., Silva-Lepe I. and Xiao C. *Adaptive Object-Oriented Programming Using Graph-Based Customization.* In *Communications of the ACM* vol. 37(5) pp. 94-101, 1994.

45. Liskov B. *Distributed programming in Argus.* In *Communications of the ACM* vol. 31(3) pp. 300-312, 1988.

46. Löhr K.-P. *Concurrency annotations for reusable software.* In *Communications of the ACM* vol. 36(9) pp. 90-101, 1993.

47. Lopes C. V. *Adaptive parameter passing.* In proc. *2nd International Symposium on Object Technologies for Advanced Software,* pp. 118-136, Kanazawa, Japan, 1996.

48. Lopes C. V. *D: A Language Framework for Distributed Programming.* Thesis Proposal, Northeastern University, College of Computer Science, 1996.

49. Lopes C. V. and Lieberherr K. *Abstracting Process-to-Function Relations in Concurrent Object-Oriented Applications.* In proc. *European Conference on Object-Oriented Programming,* pp. 81-99, Bologna, Italy, 1994.

50. Lopes C. V. and Lieberherr K. *AP/S++: Case-Study of a MOP for Purposes of Software Evolution.* In proc. *Reflection'96,* pp. 167-184, S. Francisco, CA, 1996.

51. Matsuoka S., Wakita K. and Yonezawa A. *Synchronization constraints with inheritance: what is not possible - so what is?* University of Tokyo, Tokyo. Technical Report 1990.

52. Matsuoka S., Watanabe T. and Yonezawa A. *Hybrid group reflective architecture for object-oriented concurrent reflective programming.* In *ECOOP'91* pp. 127-144. Springer-Verlag. 1991.

53. Matsuoka S., Watanabe T. and Yonezawa A. *Hybrid Group Reflective Architecture for Object-Oriented Concurrent Reflective Programming.* In *European Conference on Object Oriented Programming* pp. 231-250. 1991.

54. Matsuoka S. and Yonezawa A. *Analysis of Inheritance Anomaly in Object-Oriented Concurrent Programming Languages.* In *Research Directions in Concurrent Object-Oriented Programming* pp. 107-150, Agha G., Wegner P., et al., Eds. MIT press. 1993.

55. McAffer J. *A Meta-level Architecture for Prototyping Distributed Object Systems.* PhD dissertation, Department of Information Science, University of Tokyo, Tokyo, 1995.

56. Nierstrasz O. *Active Objects in Hybrid.* In proc. *OOPSLA'87,* pp. 243-253, 1987.

57. OMG *The Common Object Request Broker: architecture and specification.* OMG. Reference Manual, December 1991.

58. OMG *CORBAServices: Common Object Services Specification.* Object Management Group. Specification Document, March 31, 1995. Updates March 28 1996.

59. Palsberg J., Xiao C. and Lieberherr K. *Efficient Implementation of Adaptive Software.* In *ACM Transactions on Programming Languages and Systems* vol. 17(2) pp. 264-292, 1994.

60. Papathomas M. *Concurrency issues in object-oriented programming languages.* In *Object-oriented development* pp. 207-245, Tsichritzis D., Ed. Universite de Geneve. 1989.

61. Parnas D. *Information Distribution Aspects of Design Methodology.* In proc. *IFIP'71,* pp. 339-344, Ljubljana, Yugoslavia, 1971.

62. Parnas D. *A Technique for Module Specification with Examples.* In *Communications of the ACM* vol. 15(5) pp. 330-336, 1972.

63. Parnas D. L. *On the Criteria to be Used in decomposing Systems into Modules.* In *Communications of the ACM* vol. 15(2), 1972.

64. Riehle D., Siberski W., Baeumer D. et al. *Serializer.* In *Pattern Languages of Program Design 3,* Martin R., Ed. Addison-Wesley, Reading, MA. 1997.

65. Schaffert C., Cooper T. and Bullis B. K., M. *An introduction to Trellis/Owl.* In proc. *OOPSLA'86,* pp. 9-16, 1986.

66. Schmidt D. *The ACE pages,* Web pages, http://www.cs.wustl.edu/~schmidt/ACE.html

67. Seiter L. M., Palsberg J. and Lieberherr K. J. *Evolution of Object Behavior Using Context Relations.* In proc. *Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 46--57, San Francisco, 1996.

68. Sousa P., Sequeira M., Zúquete A. et al. *Distribution and Persistence in the IK platform: overview and evaluation.* In *Usenix Computing Systems* vol. 6(4) pp. 391-424, 1993.

69. Stata R. *Modularity in the Presence of Subclassing.* PhD Dissertation, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1996.

70. Strunk W., Jr. and White E. B. *The Elements of Style, 3rd edition.* Allyn & Bacon, Needham Heights, Massachusetts, 1979.

71. Takada T. and Yonezawa A. *An implementation of an object-oriented concurrent programming language in distributed environments.* In *ABCL: an object-oriented concurrent system* pp. 133-155, *Computer Systems Series*, Yonezawa A., Ed. MIT Press. 1990.

72. Tomlinson C. and Singh V. *Inheritance and synchronization with enabled-sets.* In proc. *OOPSLA'89,* pp. 103-112, 1989.

73. Watanabe T. and Yonezawa A. *Reflection in an object-oriented concurrent language.* In proc. *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 88),* pp. 306--315, San Diego, CA, 1988.

74. Wirth N. *Programming in Modula-2.* Springer-Verlag, 1982.

75. Wulf W. *Trends in the Design and Implementation of Programming Languages.* In *IEEE Computer* vol. 13(1) pp. 14-24, 1980.

76. Yokote Y. and Tokoro M. *Concurrent programming in Concurrent Smalltalk.* In *Object-oriented Concurrent Programming* pp. 129-158, *MIT Press Series in Computer Systems,* Yonezawa A. and Tokoro M., Eds. MIT Press. 1987.

77. Yonezawa A., Shibayama E., Takada T. et al. *Modelling and programming in an object-oriented concurrent language ABCL/1.* In *Object-Oriented Concurrent Programming* pp. 55-86, Tokoro A. Y. a. M., Ed. MIT Press. 1987.