

D: A Language Framework for Distributed Programming

Cristina Videira Lopes

PhD Thesis, College of Computer Science, Northeastern University. November 1997.

© Copyright 1997 Xerox Corporation. All rights reserved.

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]

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 loses 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	JAVA
<pre> 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; } } </pre>	<pre> 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; } } </pre>

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	JAVA
<pre> 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; } } </pre>	<pre> 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(); } } } </pre>

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	JAVA
<pre> 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}; } </pre>	<pre> 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(); } } </pre>

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	C++ AND PTHREADS
<pre> 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; } } </pre>	<pre> 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; } </pre>

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	JAVA
<pre> 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; } </pre>	<pre> 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; } } </pre>

DJ (CONT.)	JAVA (CONT.)
<pre> // 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; } } </pre>	<pre> // 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(); } } </pre>

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	JAVA
<pre> 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*/ } } </pre>	<pre> 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(); } } </pre>

DJ (CONT.)	JAVA (CONT.)
<pre> 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; } } </pre>	<pre> // 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; LabelMaker(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!"); } } </pre>

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	JAVA
<pre>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}; }</pre>	<pre>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 } }</pre>

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 data 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.

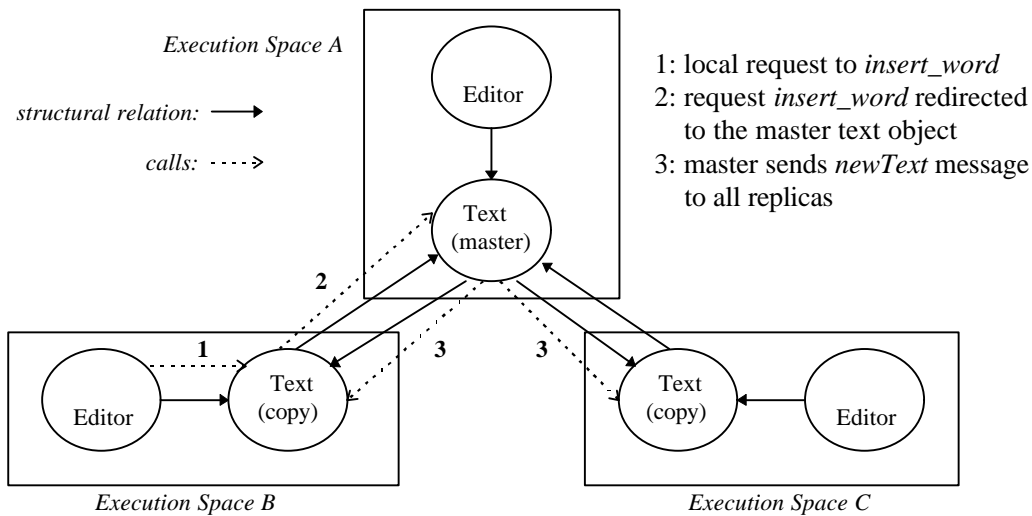


Figure 36. Structure of the distributed text editor and protocol for inserting a word remotely.

Interesting features: This application models a variety of distributed systems that use replication of data.

Comparison with: Java implementation.

DJ	JAVA
<pre> 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; } </pre>	<pre> 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 /*...*/; } </pre>

DJ (CONT.)	JAVA (CONT.)
<pre> 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+1]; 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}; } </pre>	<pre> 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+1]; 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]); } } } </pre>

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).

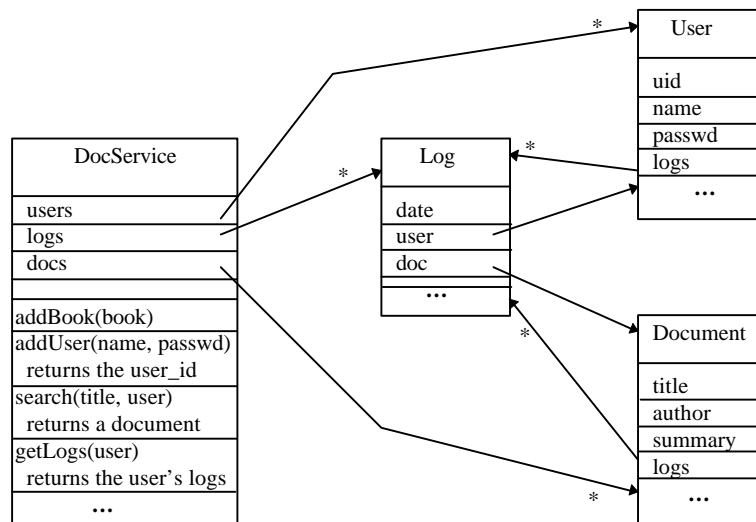


Figure 37. The document service.

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.

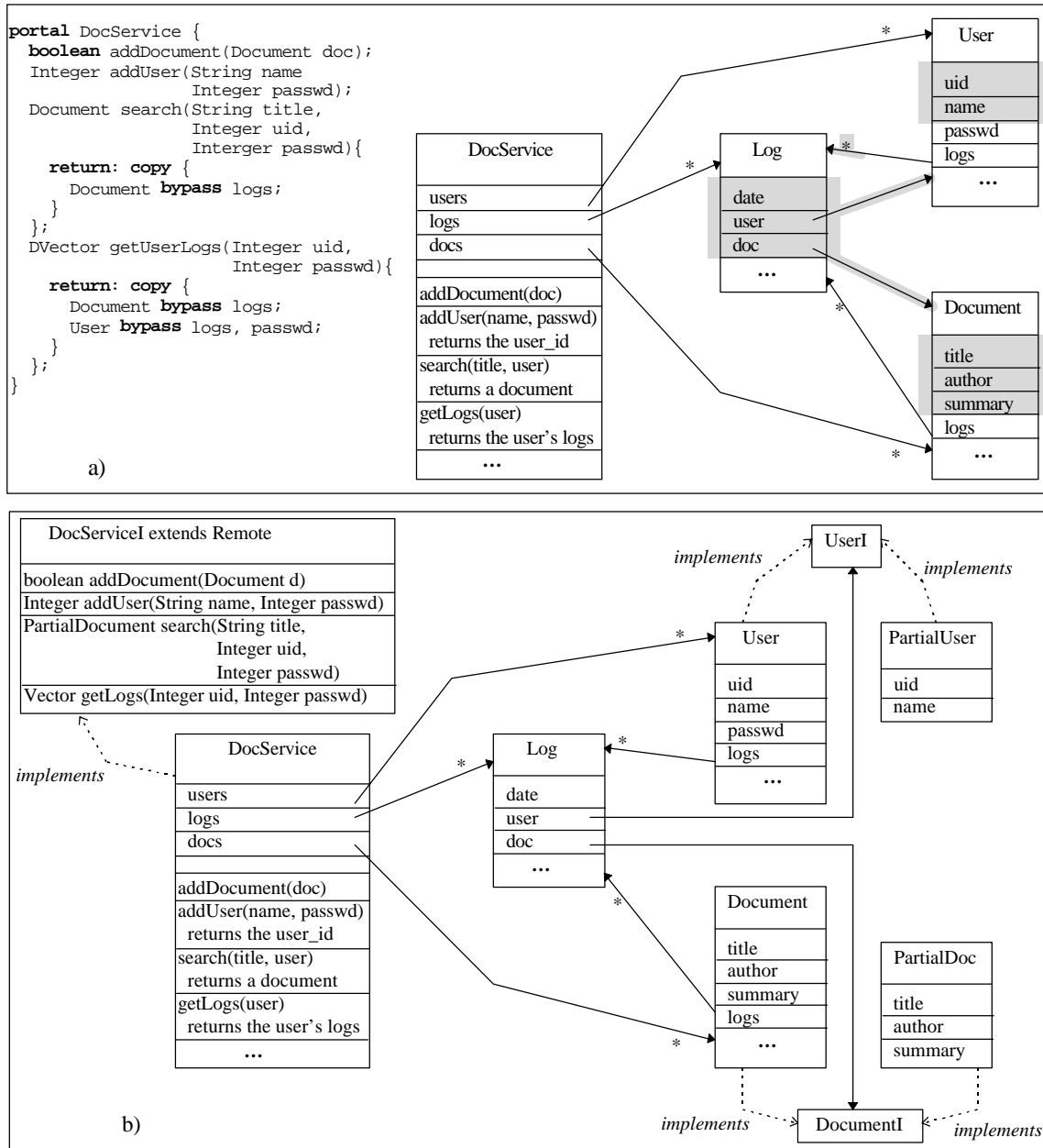


Figure 38. Class graphs for the implementation of the Document Service in a) DJ and b) Java.

The following pieces of code illustrate the DJ and Java implementations.

DJ	JAVA
<pre> 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; } } </pre>	<pre> 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; } } </pre>

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
<pre> coordinator DocService { selfex addDocument, addUser; mutex {addDocument, search}; mutex {addUser, search}; } </pre>	<pre> 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_* </pre>

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+RID L	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:

$$\text{aspectual bloat} = \frac{\text{LOC in Java} - \text{LOC in JCore}}{\text{LOC in Cool and Ridl}}$$

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:

$$\text{tangling} = \frac{\text{\# of transition points between aspect code and functionality code}}{LOC}$$

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 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	C	<i>Single thread calling a selfex method:</i> 28ms	<i>Single thread calling a synchronized method:</i> 7ms
2	O	<i>Two threads calling the same selfex method:</i> 90ms	<i>Two threads calling the same synchronized method:</i> 30ms
3	L	<i>Two threads calling 2 methods with requires, on_exit:</i> 13s	<i>Two threads, calling 2 methods with wait, notification:</i> 12s
		<i>Calling a remote operation</i>	<i>Calling a remote method</i>
4	R	<i>with no parameters:</i> 10s	<i>with no parameters:</i> 10s
5	I	<i>with one gref parameter:</i> 24s	<i>with one parameter of type Remote:</i> 24s
6	D	<i>with one copy parameter (object with 4 Integer fields):</i> 26s	<i>with one parameter of type Serializable (object with 4 Integer fields):</i> 30s
7	L	<i>with a copying directive that selects 3 out of 4 Integer fields of a parameter:</i> 35s	<i>with one parameter with 3 Integer fields that is partially copied from an object of another class (design in case-study 9):</i> 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.

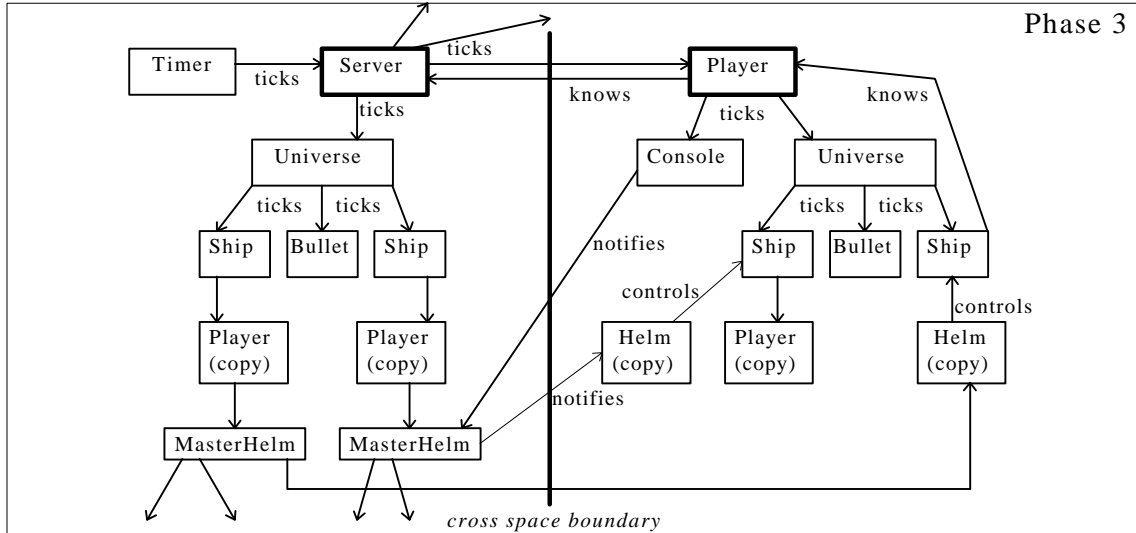


Figure 39. The diagram of the distributed space war application.

```

/* 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 getObject
always returns a consistent set of
objects currently registered with the
Universe
*/
coordinator Universe {
  mutex {register, unregister, getObject}
}

```

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
broadcast 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() {}
}

```

The four portals for the space war.

```

/* 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 broadcast) 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() {}
}

```

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.

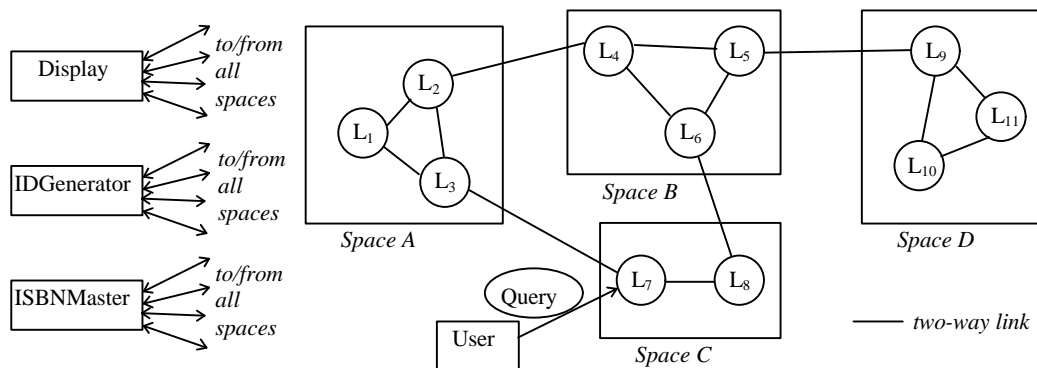


Figure 40. Interaction diagram for the distributed library system.

<pre> /* synchronizing the displayed network of libraries and searches */ coordinator Display { selfex addLink, removeLink; mutex {addLink, removeLink}; } </pre>	<pre> /* the newISBN must be synchronized, since it returns an ISBN number that is unique system-wide */ coordinator { selfex newISBN; } </pre>
<pre> // the getID method must be synchronized, // since it returns a query number // that is unique system-wide coordinator QueryNumGenerator { selfex getQueryNum; } </pre>	

<pre> 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() {} } </pre>	<pre> 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;} } } </pre>
	<pre> portal IDGenerator { int getID() {} int getIDBlock(int size) {} } </pre>
	<pre> portal ISBNMaster { ISBN newIsbn() { return: copy; } // for debugging.. long getNumBlocks() {} } </pre>

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 as-

pect 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.