

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 4

Implementation

“HACK ATTACK *noun*.

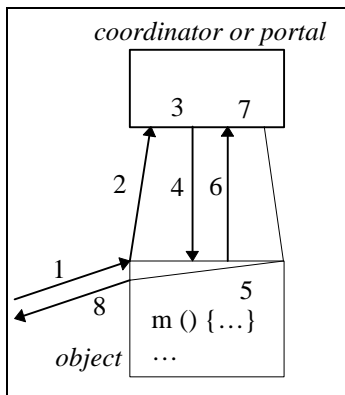
A period of greatly increased hacking activity.”

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:



act 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 OOP 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
<code>aClass.jcore</code> (JCore class)	<code>aClass.java</code> (woven class)
<code>aClass.cool</code> (coordinator)	<code>aClassCoord.java</code> (coordination class)
<code>aClass.ridl</code> (portal)	<code>aClassPRI.java</code> (RMI interface of portal) <code>aClassP.java</code> (portal class) <code>aClassPP.java</code> (proxy of the previous) <code>aClassTraversals.java</code> (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.

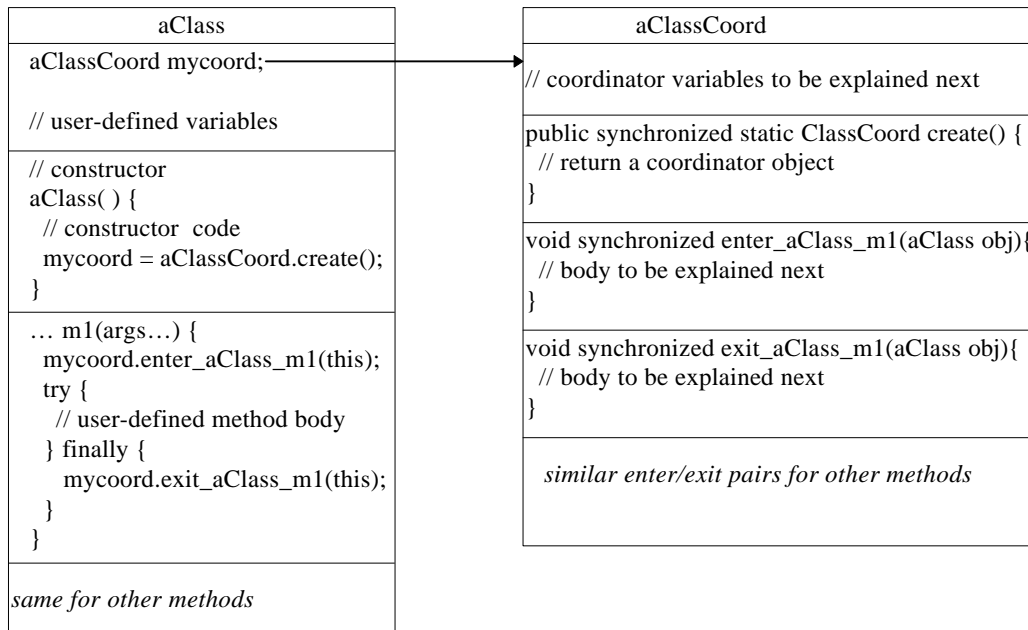


Figure 34. Output code architecture for weaving COOL.

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

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:

<pre>class A { void f() { ... } void g() { ... } }</pre>	<pre>coordinator A { selfex f; mutex {f, g}; }</pre>
<pre>class subA extends A{ void h() { ... } }</pre>	<pre>coordinator subA { selfex f; mutex {f, g, h}; }</pre>
<pre>class B { void k() { ... } }</pre>	<pre>coordinator B { selfex k; mutex {k, n}; }</pre>
<pre>class subB extends B{ void k() { ... } // overriding void n() { ... } }</pre>	

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);
        }
    }
}

```

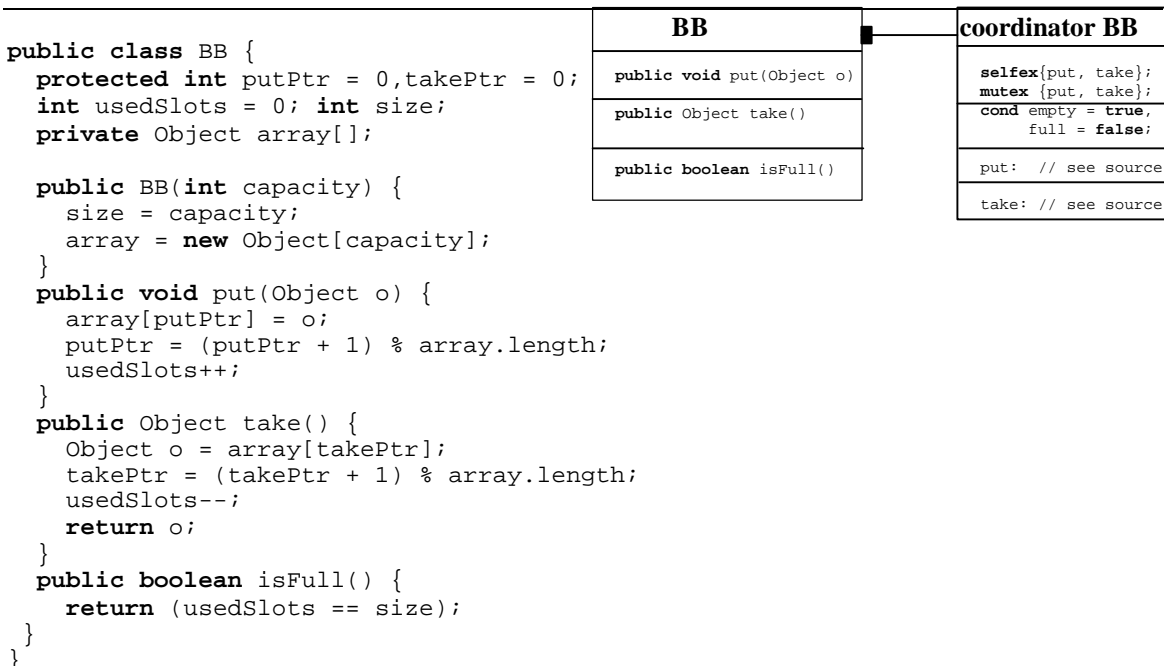
By doing this separation it is trivial to re-use the parts of the superclass that are necessary in the subclass. The implementation of method overriding consists in overriding the first method of the pair. For example, the woven class subB overrides the `_d_k` (the implementation) but not `k` (the coordination). 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.

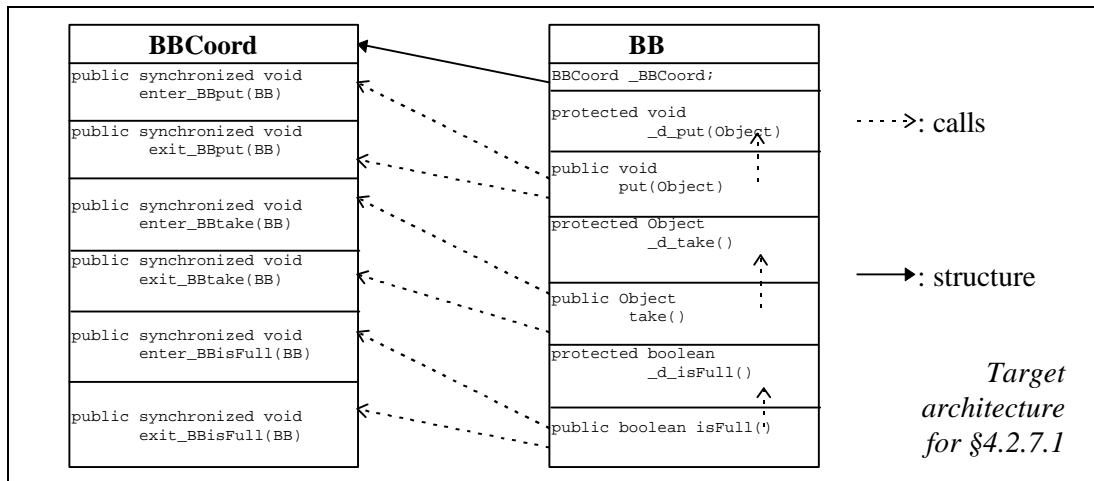
4.2.7. Examples

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:





```

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:

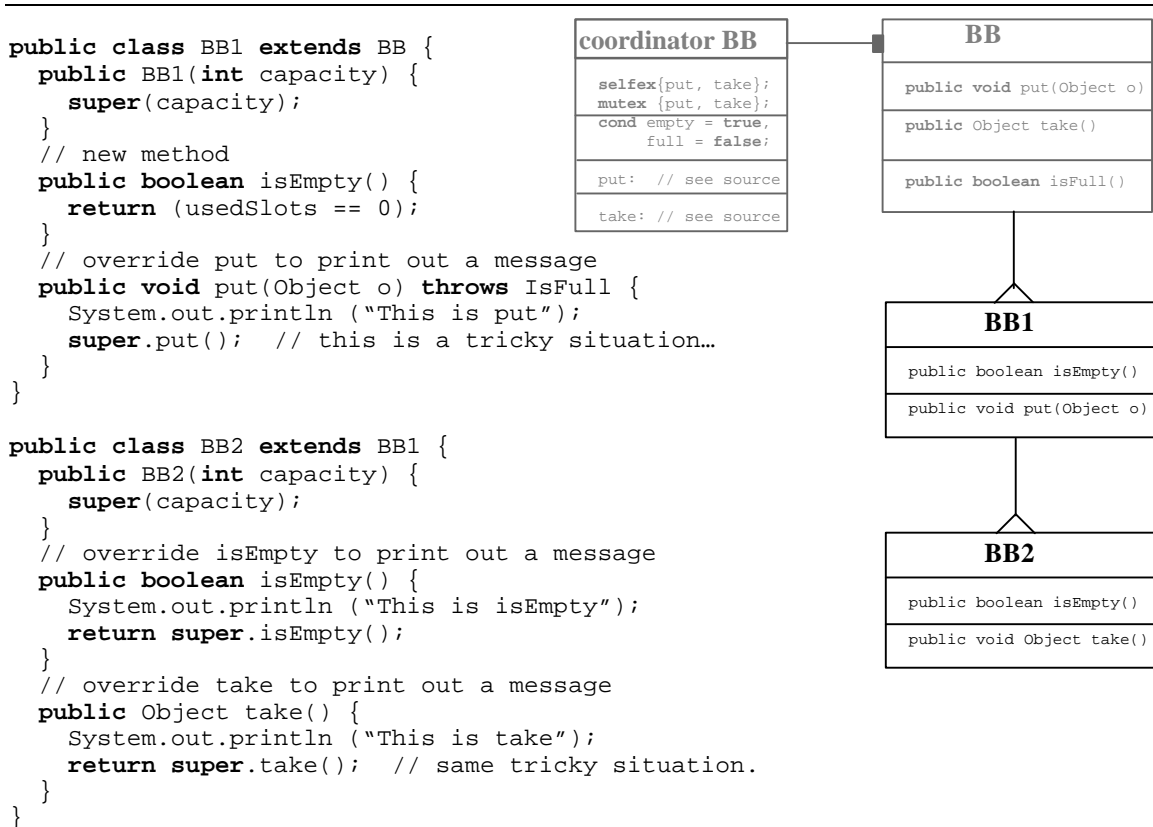
- A. Being associated with a coordinator, class BB gets an additional instance variable that will hold the reference to the coordinator object. The type of this variable is the class type that results from translating the coordinator of COOL into a Java class, in this case class BBCoord (see §4.2.1).

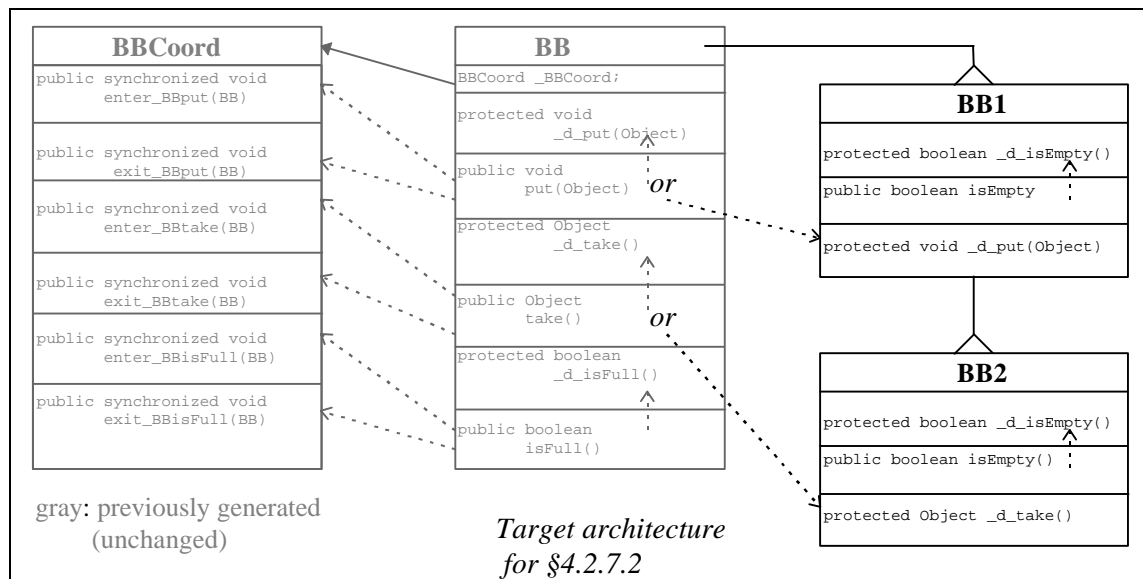
- B. The original methods of class BB are renamed. At the same time, new methods that take the original names are generated. These new methods make calls to the original, but renamed, methods. This renaming/duplication scheme ensures the separation between coordination (the public method of the pair) 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 through 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:





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);
    }
    protected boolean _d_isEmpty() {
        return (usedSlots == 0);
    }
    public boolean isEmpty() {
        return this._d_isEmpty();
    }
    protected void _d_put(Object o) {
        System.out.println ("This is put");
        super._d_put(o);
    }
    // no overriding of put
}
public class BB2 extends BB1 {
    public BB2(int capacity) { super(capacity); }
    protected boolean _d_isEmpty() {
        System.out.println ("This is isEmpty");
        return super.isEmpty();
    }
    public boolean isEmpty() {
        return this._d_isEmpty();
    }
    protected Object _d_take() {
        System.out.println ("This is take");
        return super._d_take();
    }
    // no overriding of take
}

```

// See key point N

// See key point O

// See key point P

// See key point Q

// See key point N

// See key point O

// See key point P

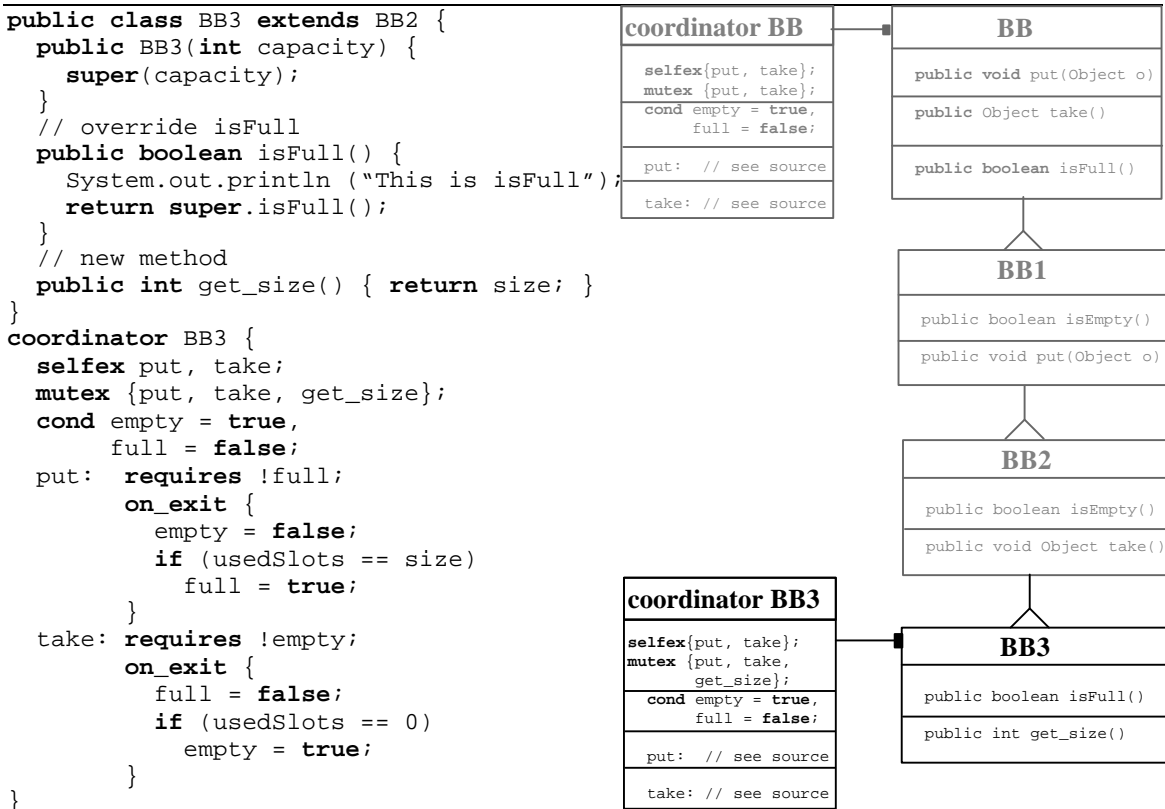
// See key point Q

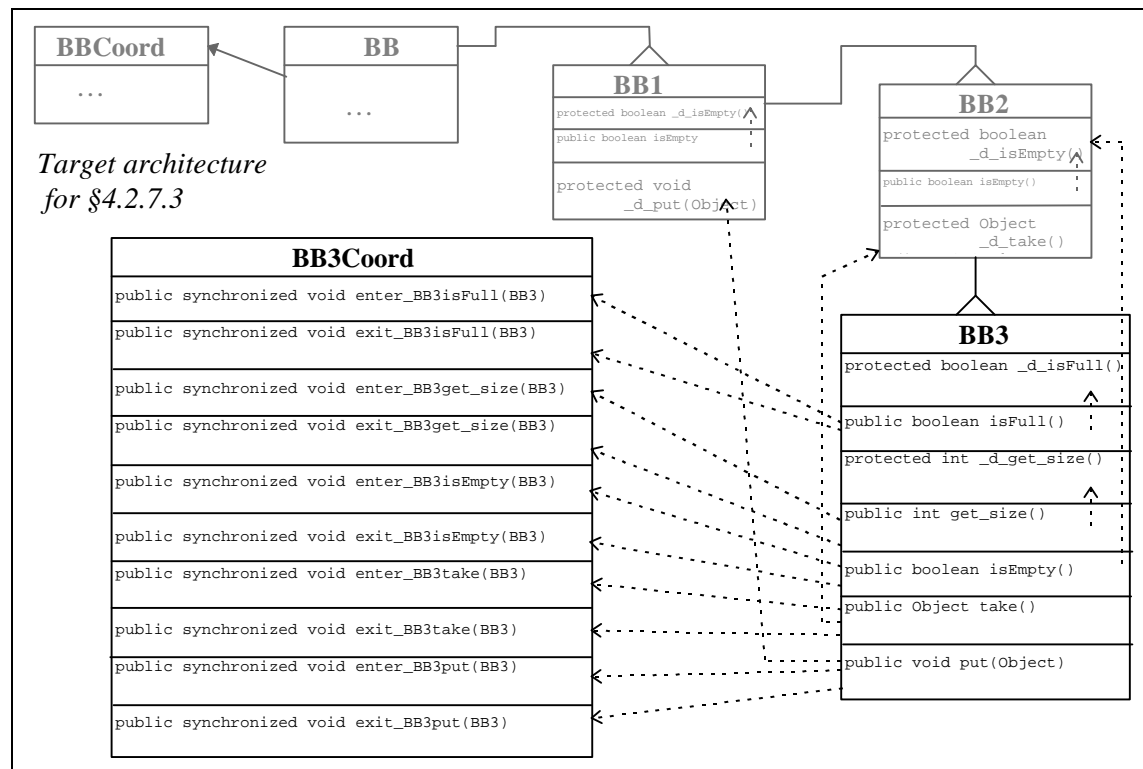
Key points:

- 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 coordination.
- 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 overridden.
- 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:





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:

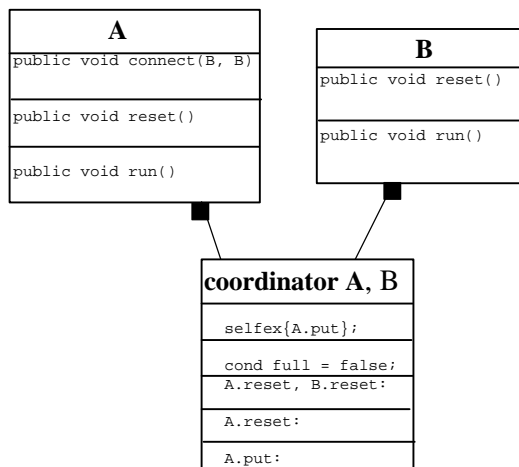
```

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

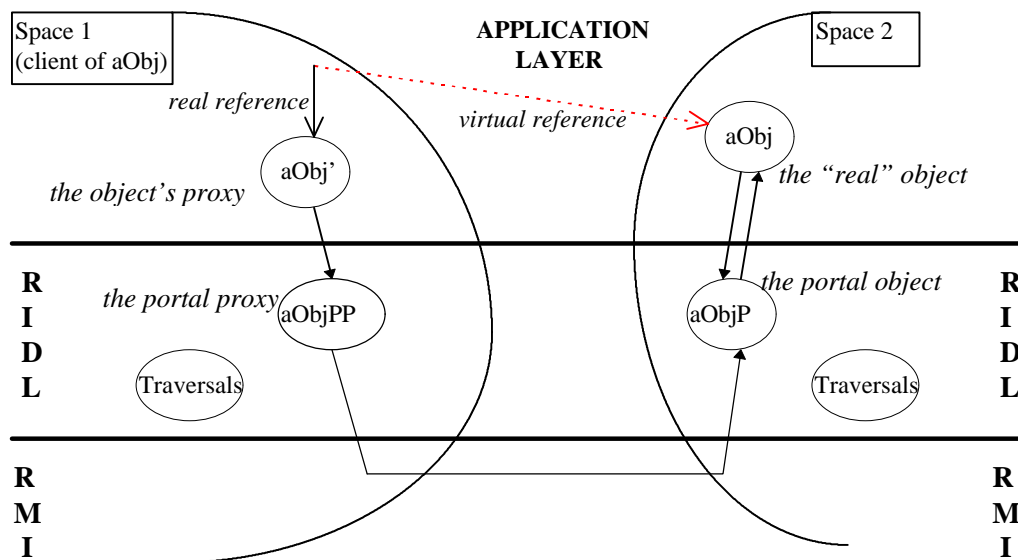


Figure 35. Run-time architecture for D's remote objects.

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:

<pre>class A { void f() { ... } int g() {...} double h(int i) { ... } }</pre>	<pre>portal A { void f(); int g(); }</pre>
---	--

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:

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

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 “*ClassNameTraversals*,” 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.lookup("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:

<pre>// 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); } }</pre>
<pre>// 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); } }</pre>

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 Object-Input. The example presented in §4.3.4.2 shows in much greater detail the protocol for passing incomplete copies.

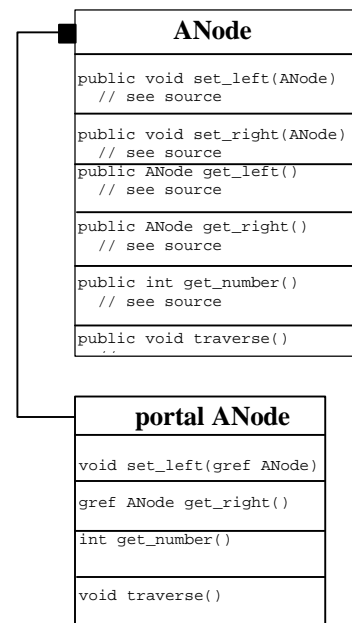
4.3.4. Examples

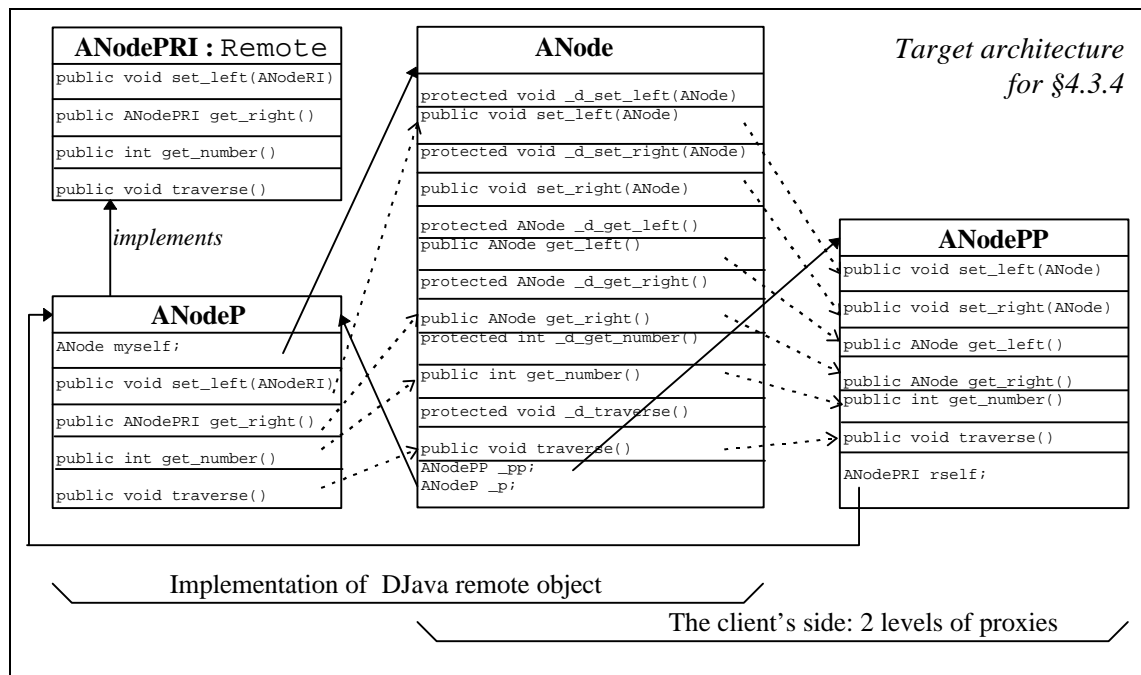
4.3.4.1 Arguments of Primitive Types and *gref*

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();
}
```





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

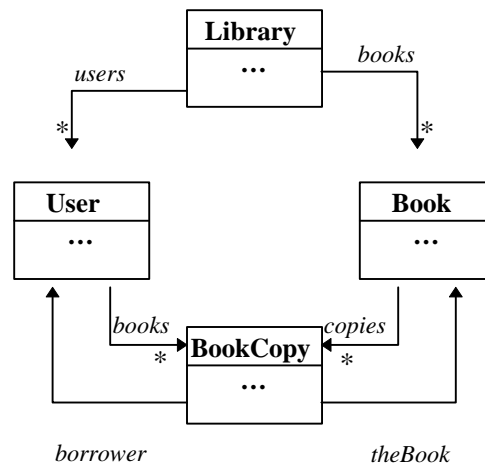
Consider the following classes and portal. This set of classes is part of a larger application. The focus of this piece of code is on the relations between the classes, which define how the marshaling traversals are to be done. Therefore, almost all methods were left out.

```
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);
                    }
                }
            }
        }
    }
}

```


sponding 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 `DObject`s 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 if 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 `DObject`s 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);
    }
}

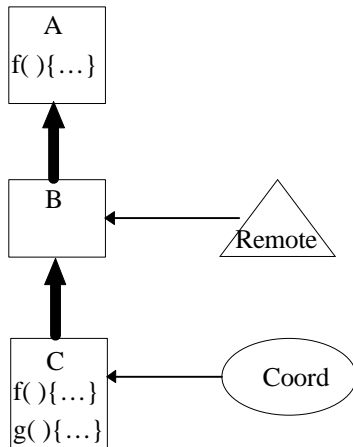
// From the output of weaving RIDL:
protected void _d_put(Object o) {
    // code for inserting
}
public void put(Object o) {
    if (_pp != null) {
        try {_pp.put(o);}
        catch (DInvalidRemoteOperation e){
            System.err.println("Invalid...");
        }
    } else _d_put(o);
}
```

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.