

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 3

The D Framework

“20. Keep related words together.

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

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

3. The D Framework

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

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

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

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

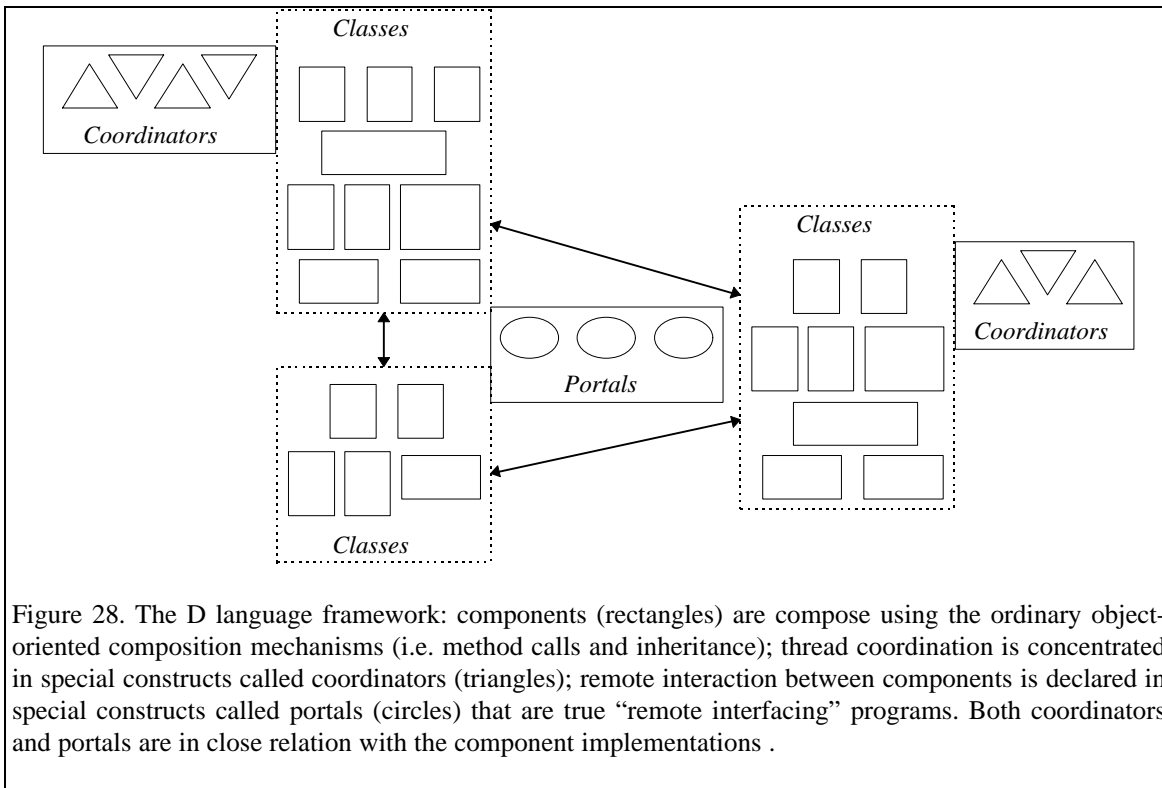


Figure 28. The D language framework: components (rectangles) are composed using the ordinary object-oriented composition mechanisms (i.e. method calls and inheritance); thread coordination is concentrated in special constructs called coordinators (triangles); remote interaction between components is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

3.1. Design Principles

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

3.1.1. Separation of Concerns: Identification of Aspects

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

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

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

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

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

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

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

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

3.1.2. Control over the Separation

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

This is a critical issue in the design of any programming language. First, it involves limiting what the languages can do, defining the “right” cuts on generality. As Hoare has put it in his now

famous quote, “the most difficult problem of language design is deciding what to leave out.” This is particularly important for aspect languages: they should be kept under a tight control, so that they can’t introduce chaos in the components, and their presence shouldn’t be overwhelming or intrusive. Secondly, control over the separation involves defining new abstraction and composition mechanisms for addressing those protocols appropriately.

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

3.1.3. Integration with Existing Languages

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

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

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

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

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

3.2. Specification of the Languages

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

3.2.1. Conventions and Notation

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

Cross-references

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

Notation

Terminal symbols are shown in `fixed width` font, and keywords are shown in **bold fixed width** font. Non terminal symbols are shown in *Italic* type. The definition of a non terminal is introduced by the name of the non terminal being defined followed by a colon. One or more alternative right-hand sides for the non terminal then follow; the alternatives are separated by the char-

acter “[]. New lines and indentation are meaningless. The subscripted suffix “*opt*”, which may appear after a terminal or non-terminal, indicates an optional part. Cross-references to a forward definition of a non terminal may appear further right of the occurrence of the symbol.

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

Foo_List:
Foo |
Foo_List Foo

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

Foo_CommaList:
Foo |
Foo_List , Foo

3.2.2. The Component Language

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

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

3.2.2.1 *Types, Values and Variables*

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

There are two categories of data values: primitive and reference values. A primitive data value is a value of a primitive type. A reference value holds the reference to an object. An object is a dy-

namically created instance of a class, and it has a unique identifier (i.e. its reference). In other words, objects are always handled indirectly through their references. (DD§3.3.2.2)

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

3.2.2.2 *Classes*

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

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

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

3.2.2.3 *Creation of Threads*

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

3.2.2.4 *The Meaning of Objects, Threads and Execution Spaces*

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

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

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

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

3.2.3. The Visible Elements of Components

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

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

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

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

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

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

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

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

3.2.4. The Coordination Aspect Language

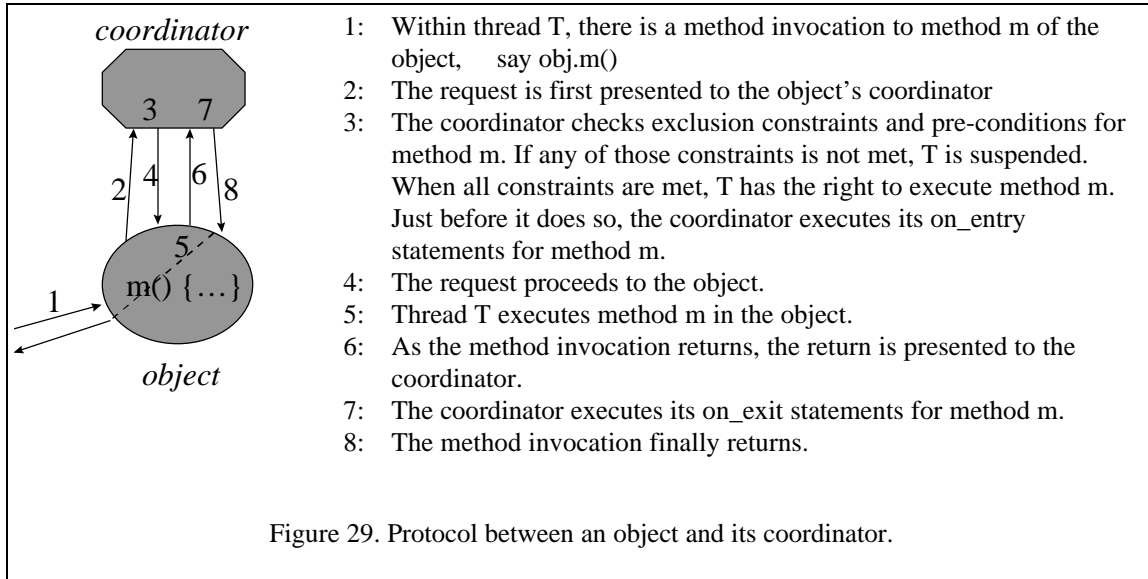
COOL provides means for dealing with mutual exclusion of threads, synchronization state, guarded suspension and notification, in relative separation from the classes. A COOL program consists of a set of coordinator modules:

COOLProgram:
CoordinatorDeclaration_List (§3.2.4.2)

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

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

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



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

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

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

The exclusion sets capture the strategies for mutual exclusion of threads over the execution of the methods; these strategies are expressed in a declarative form. The condition variables capture the synchronization state; synchronization actions, i.e., suspension and notification of threads, are performed based on the synchronization state. Ordinary variables keep track of the rest the coordinator's state that doesn't lead directly to synchronization actions, but that may affect the synchronization state (typically, it is used for keeping track of method invocation histories). The method managers operate on the coordinator's variables (both condition and ordinary variables), declaring pre-conditions and modifying the values of the coordinator's variables. Changing the value of a

condition variable results in issuing automatic notifications to threads that are waiting on pre-conditions including those variables. Coordinators are atomic entities, that is, all the computation made by a coordinator itself (pre-condition checks and state changes) is guaranteed to be thread-safe and free of race conditions.

3.2.4.1 *Visible Elements of Classes*

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

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

QualifiedName:
ClassName . VisibleElementName

ClassName:
Identifier

VisibleElementName:
*Identifier | **

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

Related Design Decisions: DD§3.3.1.1.

3.2.4.2 *Coordinator Declaration*

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

CoordinatorDeclaration:
*Granularity*_{opt} **coordinator** *ClassName_CommaList*
CoordinatorBody

Granularity: **per_class**

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

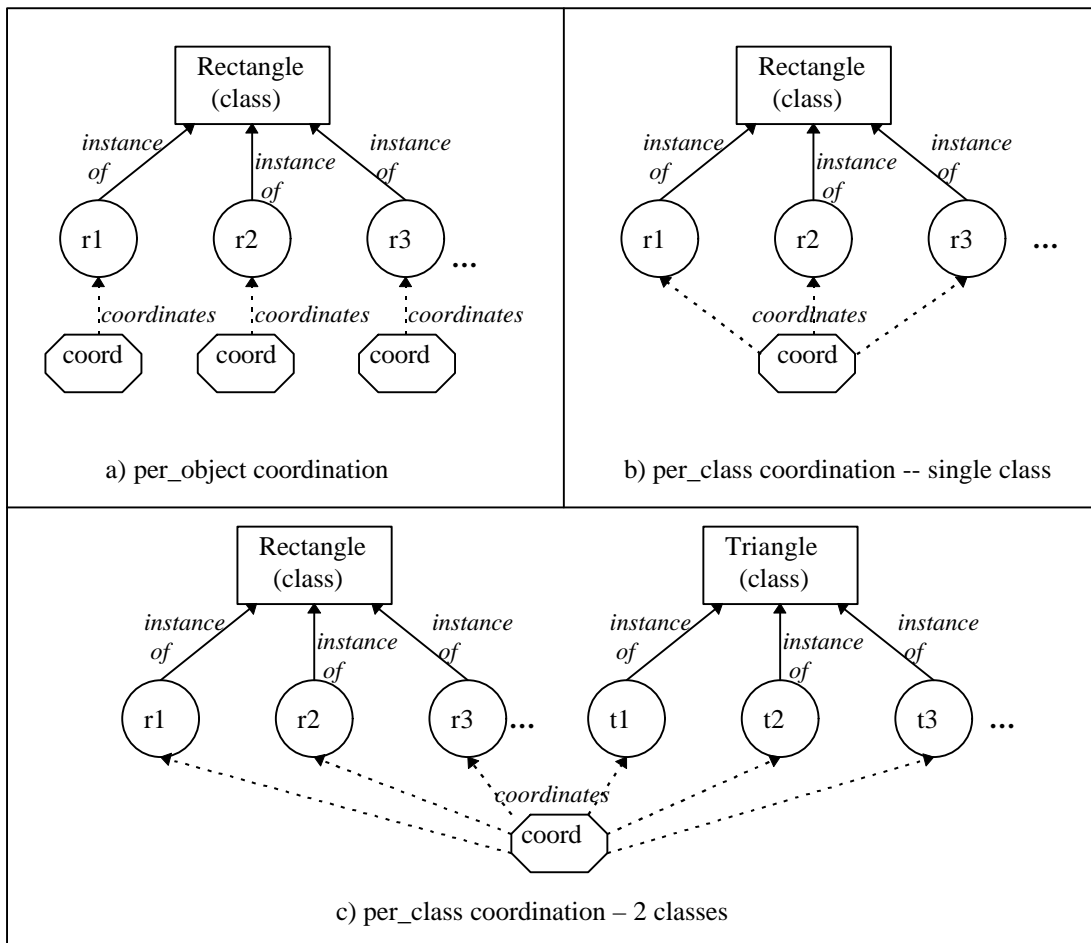


Figure 30. per_object and per_class coordination.

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

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

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

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

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

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

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

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

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

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

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

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

3.2.4.3 Coordinator Body

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

```
CoordinatorBody:
{
  CondVarDeclaration_Listopt           (§3.2.4.4)
  VariableDeclaration_Listopt         (§3.2.4.5)
  SelfExclusiveMethodsopt           (§3.2.4.6)
  MutuallyExclusiveMethodSet_Listopt (§3.2.4.7)
  MethodManager_Listopt             (§3.2.4.8)
}
```

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

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

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

Related Design Decisions: DD§3.3.3.2

3.2.4.4 Condition Variables

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

CondVarDecl:

condition *VariableDeclarator* *CommaList* ;

VariableDeclarator:

Identifier = *CondVarInitializer* |
Identifier [] = *CondArrayInitializer*

CondVarInitializer:

true / **false**

CondArrayInitializer:

{ *CondVarInitializer* *CommaList* }

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

3.2.4.5 Ordinary Variables

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

VarDeclaration:

PrimitiveType VariableDeclarator_CommaList ;

VariableDeclarator:

Identifier = VarInitializer

Identifier [] = ArrayInitializer

VarInitializer:

Expression

ArrayInitializer:

{ VarInitializer_CommaList }

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

3.2.4.6 Self-Exclusive Methods

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

SelfExclusiveMethods:

selfex *QualifiedName_CommaList ;* (§3.2.4.1)

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

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

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

```

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

```

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

```

coordinator Fact {
  selfex f;
}

```

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

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

```

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

```

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

Related Design Decisions: DD§3.3.3.4

3.2.4.7 Mutual Exclusion Declarations

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

MutuallyExclusiveMethodSet:

```

mutex { QualifiedName_CommaList };

```

 (§3.2.4.1)

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

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

```

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

```

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

```

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

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

```

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

```

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

```

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

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

```

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

```

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

```

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

```

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

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

```

mutex {set_width, set_height, area, fill};

```

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

Related Design Decisions: DD§3.3.3.4

3.2.4.8 Method Managers

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

MethodManager:

<i>QualifiedName_CommaList</i> :	(§3.2.4.1)
<i>Requires_{opt}</i>	(§3.2.4.9)
<i>OnEntry_{opt}</i>	(§3.2.4.10)
<i>OnExit_{opt}</i>	(§3.2.4.10)

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

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

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

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

3.2.4.9 Guarded Suspension

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

Requires:

```
requires CondVarExpression ;
```

CondVarExpression:

```
VarRef /
Not CondVarExpression /
( CondVarExpression ) /
CondVarExpression ConditionalOp CondVarExpression
```

VarRef:

```
Identifier /
ArrayRef /
```

ArrayRef:
 Identifier[*ArrayIndex*]

ArrayIndex:
 Identifier /
 IntegerLiteral /

Not:
 !

ConditionalOp:
 && / ||

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

```

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

```

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

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

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

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

3.2.4.10 *On Entry and On Exit Statements*

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

OnEntry:
on_entry { *Statement_List* }

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

OnExit:
on_exit { *Statement_List* }

The statements consist of a sequence of conditionals and assignments:

Statement:
IfStatement |
AssignStatement

IfStatement:
if *Expression* { *Statement_List* } |
if *Expression* { *Statement_List* } **else** { *Statement_List* }

AssignStatement:
Identifier = *Expression* ; (See Appendix A for definition of *Expression*)

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

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

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

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

Related Design Decisions: DD§3.3.3.5

3.2.4.11 Some Examples of Coordinators

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

Coordinator for the classical bounded buffer:

```

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

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

```

Coordinator for the dining philosophers (monitor solution):

```

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

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

```

Coordinator for an assembly line:

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

```

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

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

  Packer.processPack: requires packFull;

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

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

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

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

```

3.2.5. The Remote Interface Aspect Language

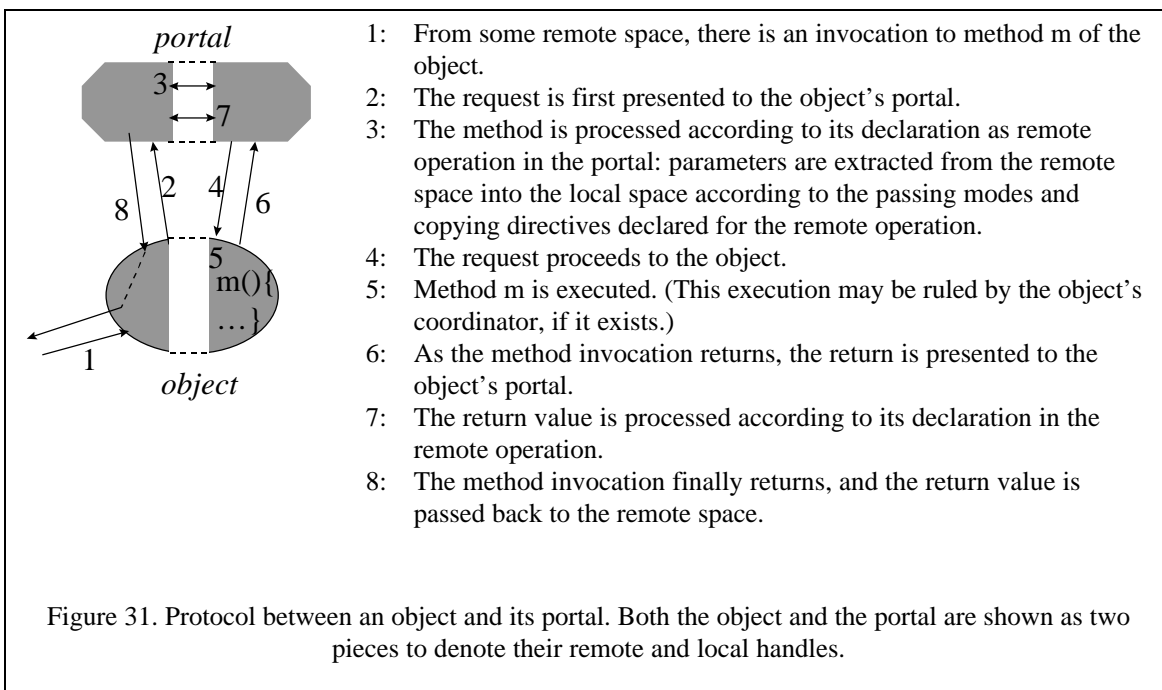
RIDL provides means for dealing with data transfers between different execution spaces in relative separation from the classes. A RIDL program consists of a set of portal modules:

RIDLProgram:
PortalDeclaration_List (§3.2.5.2)

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

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

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



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

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

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

3.2.5.1 *Visible Elements of Classes*

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

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

Related Design Decisions: DD§3.3.1.1.

3.2.5.2 *Portal Declaration*

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

PortalDeclaration:
portal *ClassName PortalBody*

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

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

3.2.5.3 Portal Body

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

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

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

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

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

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

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

Related Design Decisions: DD§3.3.4.2.

3.2.5.4 Remote Operations

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

```
RemoteOperation:
  ReturnType MethodName ( Parameter_CommaListopt )
  RemoteOperationBodyopt ;
```

```
ReturnType:
  Type /
  void
```

Parameter:

Type ParamName

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

ParamName:

Identifier |
ParamName []

RemoteOperationBody:

{ *ObjectTransferSpec_List* } (§3.2.5.5)

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

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

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

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

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

```

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

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

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

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

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

```

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

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

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

```

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

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

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

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

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

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

```

3.2.5.5 Object Transfer Specifications

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

ObjectTransferSpec:
ObjectName : *Mode* ;

ObjectName:
Identifier |
return

Mode:
gref /
copy *CopyDirective_{opt}* (§3.2.5.9)

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

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

Related Design Decisions: DD§3.3.4.1.

3.2.5.6 *Type Transfer Specifications*

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

TypeTransferSpec:
ReferenceType : *Mode* ;

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

Related Design Decisions: DD§3.3.4.1.

3.2.5.7 *Passing Global References*

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

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

Related Design Decisions: DD§3.3.4.1, §3.3.4.3.

3.2.5.8 *Passing Copies*

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

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

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

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

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

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

3.2.5.9 *Copying Directives*

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

```
CopyDirective:
  {
    SelectionDirective_Listopt
  }
```

SelectionDirective:

ClassSelector SelectionPrimitive VariableSelector_CommaList;

SelectionPrimitive:

only |
bypass

ClassSelector:

ClassName

VariableSelector:

VariableName |
all.*TypeName*

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

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

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

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

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

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

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

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

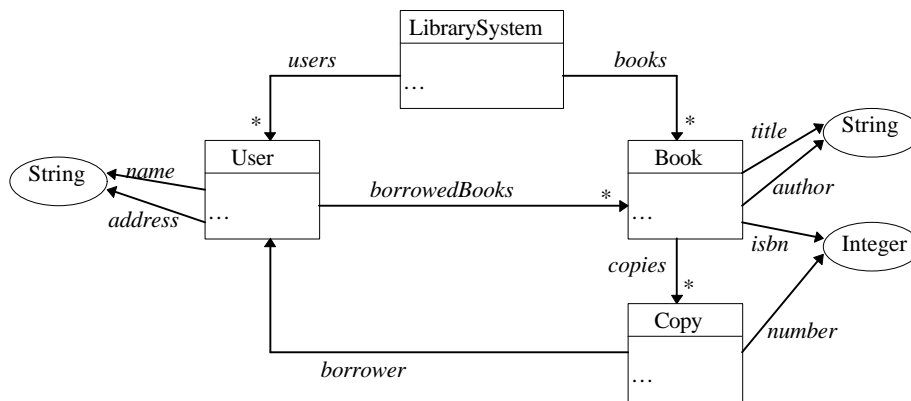


Figure 32. Class graph of a library application.

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

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

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

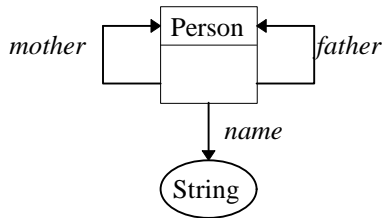
  BookList borrowedBooks(User user) {
    //for return object, exclude this edge; this breaks nasty cycle
    return: copy {Book bypass copies;}
    // for User, bring only the name
    user: copy {User only name;}
  };
}

```

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

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

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



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

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

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

Example 1: `{A bypass f1; A bypass f2;} ⇔ {A bypass f1, f2;}`

Example 2: `{A bypass f1; A bypass all.B;} ⇔ {A bypass f1, all.B;}`

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

Example 3: `{A only f1, f3; A only f2, f3;} ⇔ {A only f3;}`

Example 4: `{A only f1; A only all.B;} ⇔ {A only f1;}` if `f1` is of type `B`
`{A only ∅;}` otherwise

- when `bypass` and `only` constraints are given simultaneously, the resulting set of fields `F` is the intersection given by $F = \bigcap_{sets\ of\ fields\ of\ only\ constraints} \cup_{sets\ of\ fields\ of\ bypass\ constraints}$

Example 5: `{A only f1; A bypass f1, f2; A bypass f3;} ⇔ {A only f1;}`

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

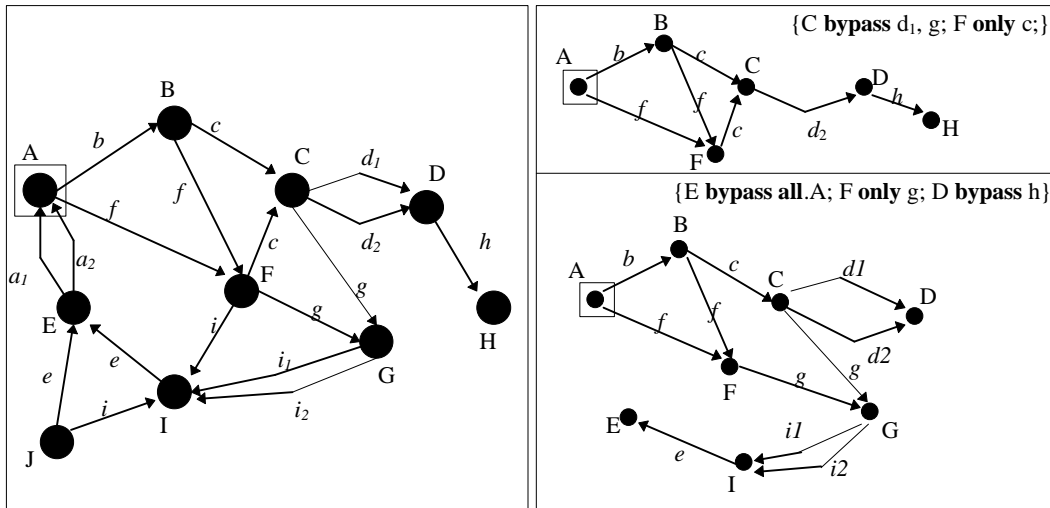


Figure 33. Two examples of copying directives. The circles are classes, and the edges are variables declared in them (e.g. class A contains a variable named *b* of type B and a variable named *f* of type F). The two graphs on the right are the result of applying the given copying directive to the graph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

Compatibility of copying directives:

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

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

Related Design Decisions: DD§3.3.4.4, §3.3.4.5.

3.2.5.10 *Classes that Must Have a Portal*

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

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

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

3.2.5.11 *Some Examples of Portals*

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

A portal for the bounded buffer:

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

BookLocator/ProjectManager:

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

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

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

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

3.2.6. Interaction between Aspects and Class Inheritance

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

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

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

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

```

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

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

```

```

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

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

```

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

```

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

```

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

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

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

3.3. Design Decisions and Alternatives

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

3.3.1. General

3.3.1.1 *On Modules, Components, Aspects and Interfaces*

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

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

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

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

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

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

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

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

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

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

- (2) Aspect modules isolate the coding of those issues and free the components from code tangles. Besides the client and specialization interfaces, aspect modules introduce new kinds of interfaces for components, which focus *only* on particular subject matters. These interfaces are called “aspect interfaces.”
- (3) Aspect interfaces use particular sets of visible elements, which have been presented in §3.2.3, §3.2.4.1 and §3.2.5.1. Visible elements establish those dependencies between aspects and components that can be checked automatically. They have their counterparts in types, for example.

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

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

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

Documentation for clients:

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

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

Documentation for specialization:

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

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

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

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

Documentation for coordination:

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

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

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

Documentation for remote interaction:

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

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

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

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

3.3.1.2 The Need for Special Abstractions and Composition Mechanisms

[From §3.2.4.2, §3.2.5.2]

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

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

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

3.3.1.3 *Who Drives Who*

[From §3.2.4.2, §3.2.5.2]

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

3.3.2. The Component Language

3.3.2.1 *Class-based Language vs. Prototype-based Language*

[From §3.2.2.1]

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

3.3.2.2 *Uniform Reference Semantics*

[From§3.2.2.1]

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

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

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

3.3.2.3 *Threads*

[From§3.2.2.3]

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

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

3.3.2.4 *The Default Synchronization*

[From §3.2.2.4]

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

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

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

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

3.3.2.5 *The Default Communication*

[From §3.2.2.4]

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

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

3.3.3. COOL

3.3.3.1 *Coordination: Classes vs. Abstract Method Sets*

[From §3.2.4.2]

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

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

```

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

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

```

this uses only emptySlots

this uses only usedSlots

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

```

coordinator BoundedBuffer {
    selfex do_take, do_put;
    mutex {do_take, increment_emptySlots};
    mutex {do_put, increment_usedSlots};
}

```

```

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

```

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

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

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

3.3.3.2 Granularity of Synchronization

[From §3.2.4.2, §3.2.4.3, §3.2.6]

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

- (1) the provider (i.e. the class) is the one that defines the synchronization (monitor approach);
- (2) the smallest unit of synchronization is the method;
- (3) there is no middle ground between one instance and all instances of the same class;

- (4) the coordination is contained within one coordinator;
- (5) the association between an object and its coordinator is static.

The alternatives to these decisions are:

- (a) the synchronization would be driven not only by the class, but also by its clients;
 - (b) the units of synchronization would be not only methods, but also arbitrary blocks of code;
 - (c) synchronization strategies could affect arbitrary instances of arbitrary classes;
 - (d) the coordination of one object could be split among several coordinators;
 - (e) the association could change dynamically.
- (1) vs. (a): it may happen that the synchronization scheme affecting an object depends on the context in the caller. For example, the caller may need to ensure mutual exclusion of two objects simultaneously in order to execute safely. The only way of doing this is by using some kind of semaphore approach (e.g. locks). But, as discussed in Chapter 2, semaphores do terrible things to programs, because they establish hidden implementation dependencies between modules. COOL discourages those practices, and favors the monitor approach. If the synchronization is context-dependent, then that context should be abstracted in a class, and that class should then be coordinated. However, COOL does not prevent from programming with semaphores.
- (2) vs. (b): while (b) could, eventually, be supported — by, for example, including some kind of pattern matching primitives in COOL — its non-alignment with the object-oriented modularities raises some doubts about its clarity. That kind of unrulid synchronization can always be transformed into more solid designs, by encapsulating those blocks in methods. But this is certainly a point to have in mind while evolving COOL.
- (3) vs. (c): the decision comes as a natural consequence of using object-oriented languages based on classes. A class defines the same behavior for all of its instances, and it is not possible to define particular behaviors for particular instances. The only way to do that is with tests in the code. This is what happens in COOL, too (see coordinator for dining philosophers in §3.2.4.11).
- (4) vs. (d): splitting the responsibility of coordination would overcome some limitations of COOL. For example, we could have one coordinator for taking care of issues that affect the whole class (methods that affect class variables), and several per object coordinators for taking care of issues that affect each of its instances, individually. This will require

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

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

3.3.3.3 Synchronization: Local vs. Distributed

[From §3.2.4.2]

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

3.3.3.4 Exclusion Constraints

[From §3.2.4.6, §3.2.4.7]

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

3.3.3.5 Assignments

[From §3.2.4.10]

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

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

3.3.3.6 *Current Limitations*

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

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

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

3.3.4. RIDL

3.3.4.1 *Remote Interaction: Classes vs. Abstract Types*

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

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

However, many languages and systems before D have chosen to use abstract types as the basis for remote interaction. Some examples are Emerald [10], CORBA [57] and Java RMI [27]. The

benefits of connecting remote components using abstract types, as opposed to using low level point-to-point connections, can be summarized as follows: (1) a significant part of the interface, namely the valid operations, their parameters and return values, can be checked at compile time; (2) it allows for the possibility of changing the implementation of the services without changing the client code.

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

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

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

3.3.4.2 Granularity of Remote Interaction

[From §3.2.5.2, §3.2.5.3, §3.2.6]

The granularity of remote interaction in RIDL is very similar to the granularity of synchronization in COOL, which was discussed in §3.3.3.2. The alternatives are also similar. In RIDL the decisions were based on prudence. The current version of RIDL is highly influenced by the existing interface definition languages. IDLs have proven to be a good idea, but the existing ones are too

much grounded on, and therefore limited by, the concept of abstract type (see discussion in §3.3.4.1). The first step towards a powerful language that expresses many kinds of remote interaction protocols at the application level is to make the shift from the type-based approach to the aspect-based approach. Once that shift is done, the possibilities are immense, as the following list suggests.

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

- (1) the provider (i.e. the class) is the one that defines the remote interaction;
- (2) the smallest unit for remote interaction is the method;
- (3) the remote interaction is contained within one portal;
- (4) the association between an object and its portal is static;
- (5) there are no multi-class portals.

The alternatives to these decisions are:

- (a) the remote interaction would be driven not only by the class, but also by its clients;
- (b) the units would be not only methods, but also arbitrary blocks of code;
- (c) the remote interaction with one object could be split among several portals;
- (d) the association could change dynamically;
- (e) there could be specialized portals which would specify remote interaction strategies between particular pairs/sets of components.

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

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

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

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

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

3.3.4.3 *On the Semantics of `gref`*

[From §3.2.5.7]

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

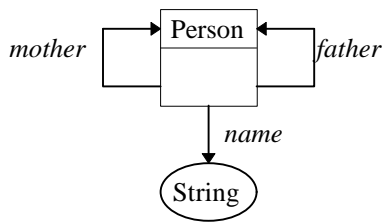
There are operational differences between the implementation with replication and the other implementation without replication, namely differences in performance (potentially, but not necessarily, better in the first case), availability and protection. In a distributed system it is important to be able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

3.3.4.4 *The Copying Directives*

[From §3.2.5.9]

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

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



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

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

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

3.3.4.5 The Missing Parts

[From §3.2.5.9]

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

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

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

3.3.4.6 *Current Limitations*

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

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

3.4. Final Remarks

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

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

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

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

