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 2

Code Tangling

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

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

2. Code Tangling

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

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

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

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

2.1. How Programs Become Tangled

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

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

2.1.1. Implementing the Functionality

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

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

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

    return locations[i];
  }
}
```

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

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

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

2.1.2. Adding Synchronization Constraints

The specification also states that book locators should be able to process several requests concurrently. Having written the classes in Java, concurrency exists in the form of threads that execute the objects without any default synchronization. Since the three methods of the BookLocator class

use and update the same instance variables, additional code is necessary in order to avoid inconsistencies.

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

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

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

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

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

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

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

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

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

2.1.3. Providing for Remote Access

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

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

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

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

  // the constructor
  public BookLocator (int dbsize) {
    books = new Book[dbsize];
    locations = new Location[dbsize];
  }
  public void register(Book b, Location l)
  throws LocatorFull {
    beforeWrite();
    if (nbooks >= books.length) {
      afterWrite();
      throw new LocatorFull();
    }
    else {
      // Just put it at the end
      books[nbooks] = b;
      locations[nbooks++] = l;
    }
    afterWrite();
  }
  public void unregister (Book b) {
    Book abook = books[0]; int i = 0;
    beforeWrite();
    while (i < nbooks &&
           abook.get_isbn()!=b.get_isbn())
      abook = books[++i];
    if (i == nbooks) {
      afterWrite();
      return;
    }
    // simply shift down the rest
    while (i < nbooks - 1) {
      books[i]= books[i+1];
      locations[i]= locations[++i];
    }
    --nbooks;
    afterWrite();
  }
// class BookLocator continued
  public Location locate (String str)
  throws BookNotFound {
    Book abook = books[0];
    int i = 0; boolean found = false;
    Location l;
    synchronized (this) {
      while (activeWriters > 0)
        try { wait(); }
        catch (InterruptedException e) {}
      ++activeReaders;
    }
    while (i < nbooks && found == false){
     if(abook.get_title().compareTo(str)==0 ||
        abook.get_author().compareTo(str)==0)
      found = true;
     else abook = books[++i];
    }
    if (found == false) {
      synchronized (this) {
        --activeReaders; notifyAll();
      }
      throw new BookNotFound (str);
    }
    l = locations[i];
    synchronized (this) {
      --activeReaders; notifyAll();
    }
    return l;
  }
  protected synchronized void beforeWrite() {
    // as before
  }
  protected synchronized void afterWrite() {
    // as before
  }
}
public class Book implements BookI {
  // possible implementation of Books
  String title, author;
  int isbn;  Project owner;
  OCRImage firstpage;

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

Figure 6. Providing remote access to book locators.

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

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

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

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

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

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

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

public class ProjectManager
       extends UnicastRemoteObject
       implements Pmanager {
  ProjectList projects;

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

  }
  // other methods ommited
}
```

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

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

Figure 7. Adding a new component to the application.

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

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

Marshaling propagates from the parameter object to its variables, recursively. When it is done in context, the context must also propagate in some way. In this case, a book is marshaled differently for two services because it involves a different marshaling of the `owner` field. Therefore, the

class Project must implement a marshaling routine for this particular situation. (Those methods, `pack1` and `unpack1` for class `Project`, are not shown).

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

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

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

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

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

Figure 8. Marshaling parameters depending on the service.

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

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

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

2.2. The Source of Tangling

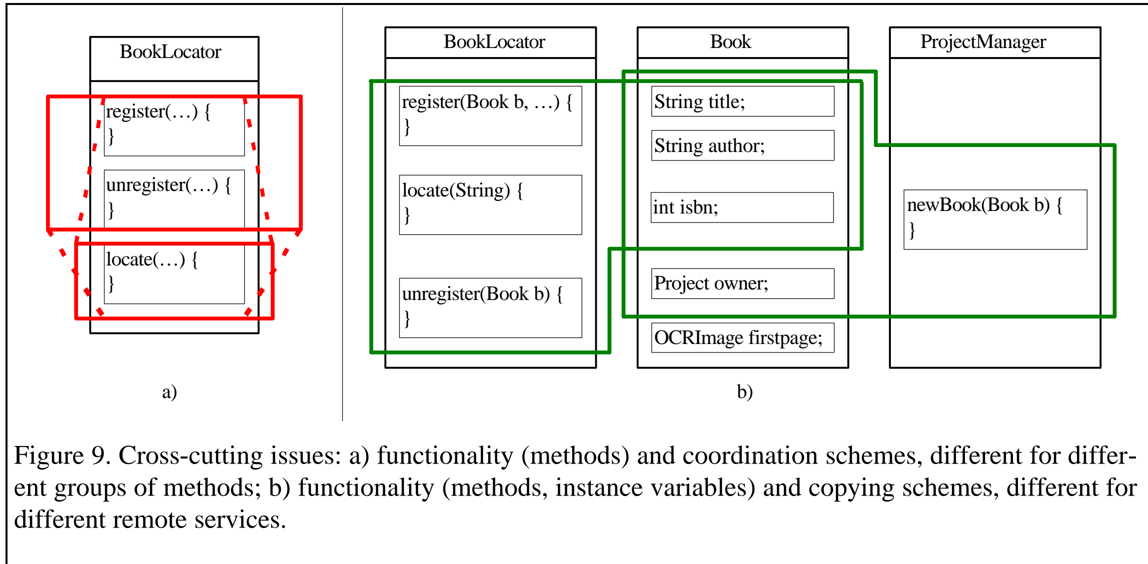

Figure 9. Cross-cutting issues: a) functionality (methods) and coordination schemes, different for different groups of methods; b) functionality (methods, instance variables) and copying schemes, different for different remote services.

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

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

This cross-cutting phenomena is directly responsible for the tangling in the code. The composition mechanisms the language provides us — method calling and inheritance — are well suited to

building up the functional units. But they are not so good for composing the functional units with cross-cutting issues, because they follow such different composition rules and yet must co-compose. This breakdown forces us to combine the properties entirely by hand — that's what happened in the tangled code presented in §2.1.

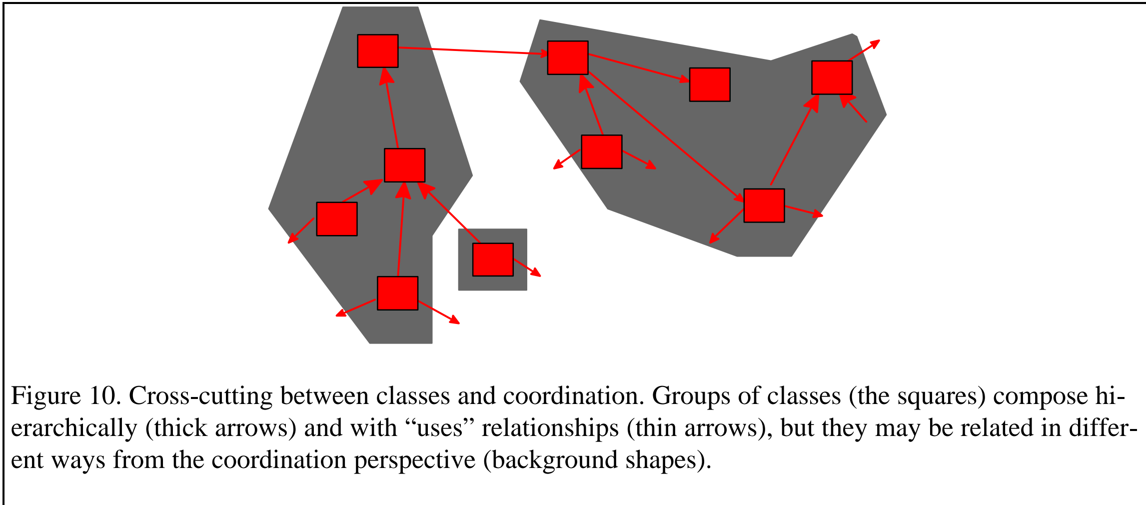

Figure 10. Cross-cutting between classes and coordination. Groups of classes (the squares) compose hierarchically (thick arrows) and with "uses" relationships (thin arrows), but they may be related in different ways from the coordination perspective (background shapes).

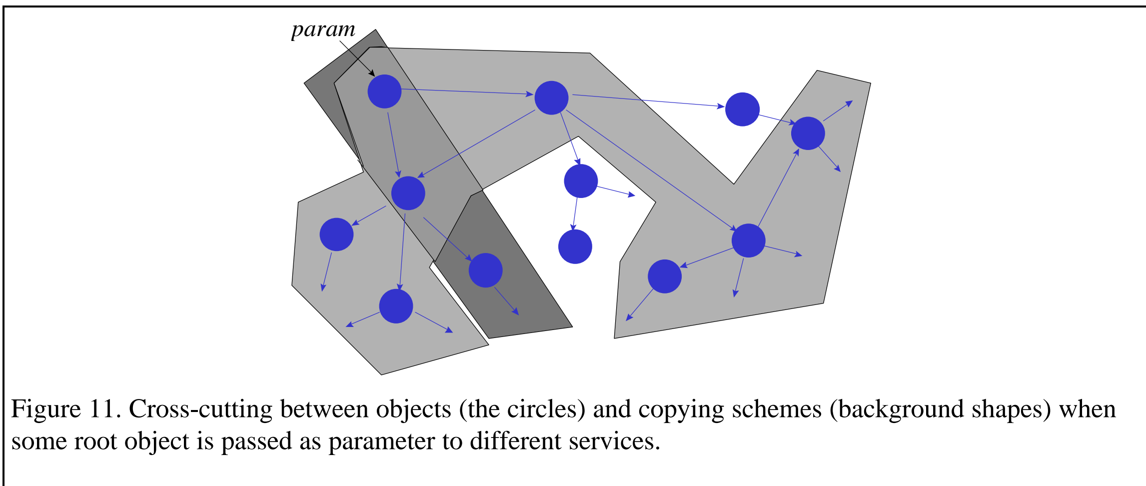

Figure 11. Cross-cutting between objects (the circles) and copying schemes (background shapes) when some root object is passed as parameter to different services.

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

2.3. Tangling and Programming Practices

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

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

2.3.1. Concurrency

2.3.1.1 Units of Synchronization

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

// same class Semaphore
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.3.1.2 Splitting Classes

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

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

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

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

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

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

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

2.3.1.3 Splitting Locks

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

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

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

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

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

2.3.1.4 Coordination State

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

This separation, however, is not enforced by languages like Java. As a consequence, programmers must make sure that suspensions and notifications happen at the right points in the code, and

for the right values of the instance variables. Defining and tracking the synchronization state is one of the most critical points of concurrent systems. Doing it in the objects' complete state space can be confusing and error-prone.

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

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

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

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

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

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

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

Figure 16.Tracking state variables.

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

2.3.1.5 Coordination by Subclassing

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

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

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

Coordination by subclassing is a good way of dealing with the *inheritance anomalies*. These anomalies have been studied before and they consist of the following: when coding a base class, if a number of precautions are not followed, the code related to concurrency control cannot be effectively inherited and/or redefined in a subclass without non-trivial redefinitions of the methods' implementations. This situation hampers the reuse of code by inheritance, and, therefore, weakens the usefulness of object-oriented languages in programming distributed applications.

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

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

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

    return locations[i];
  }
}
```

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

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

Figure 17. Coordinating the book locator class by subclassing.

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

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

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

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

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

2.3.2. Communication

2.3.2.1 Splitting Parts

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

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

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

public class ProjectManager
       extends UnicastRemoteObject
       implements Pmanager {
  ProjectList projects;

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

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

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

Figure 18. Splitting the objects into their smaller parts.

2.3.2.2 *Class Transformations*

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

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

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

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

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

2.3.2.3 The Serializer Design Pattern

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

2.3.3. Summary

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

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

2.4. Tangling and Programming Languages

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

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

2.4.1. Basic Linguistic Support for Distributed Programming

2.4.1.1 Synchronization

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

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

cessing that data, but they simply request the service to the monitor. Figure 15 showed the bounded buffer implemented as a monitor.

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

2.4.1.2 Communication

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

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

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

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

2.4.2. Concurrency in Object-Oriented Languages

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

2.4.2.1 Orthogonal Approach

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

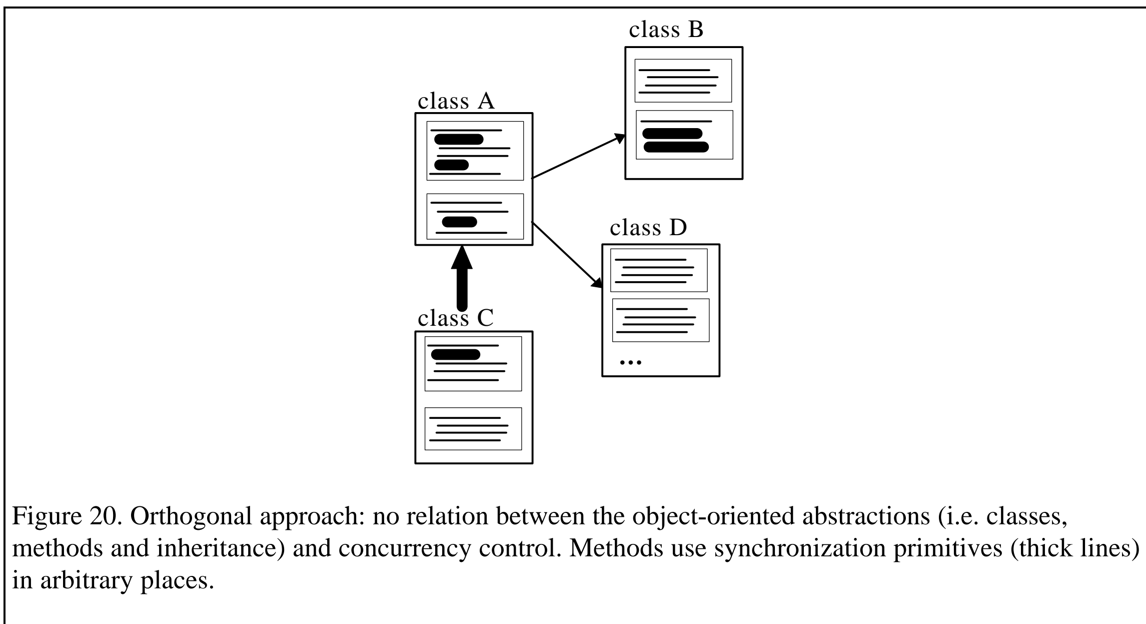

Figure 20. Orthogonal approach: no relation between the object-oriented abstractions (i.e. classes, methods and inheritance) and concurrency control. Methods use synchronization primitives (thick lines) in arbitrary places.

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

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

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

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

Figure 21. The bounded buffer written in Obliq.

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

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

The code tangling problem also occurs in the Obliq implementation, since the code for coordination is embedded in the implementation of the methods. However, Obliq provides more built-in

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

2.4.2.2 Full Integration

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

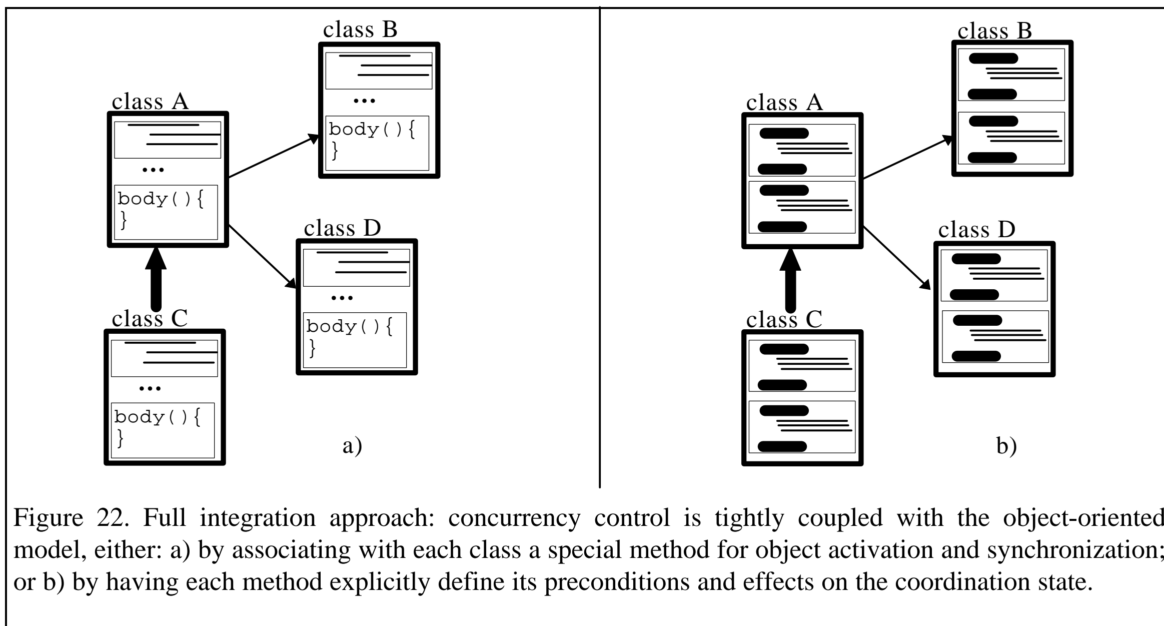

Figure 22. Full integration approach: concurrency control is tightly coupled with the object-oriented model, either: a) by associating with each class a special method for object activation and synchronization; or b) by having each method explicitly define its preconditions and effects on the coordination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

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

as-monitor is dispersed among the methods themselves, in the form of guards and post-conditions. Some Actor languages [1, 29] and Hybrid [56] took this approach.

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

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

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

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

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

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

Figure 24. Active objects with an explicit body.

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

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

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

Figure 23. Behavioral abstractions.

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

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

2.4.2.3 Separation of Coordination and Functionality

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

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

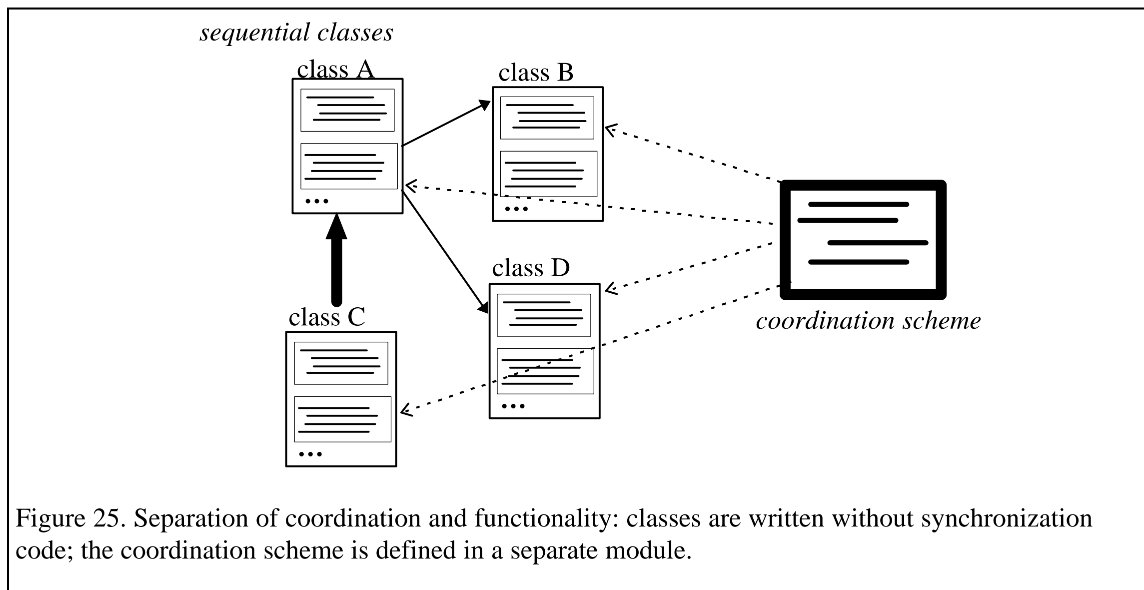

Figure 25. Separation of coordination and functionality: classes are written without synchronization code; the coordination scheme is defined in a separate module.

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

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

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

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

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

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

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

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

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

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

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

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

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

2.4.3. Communication in Object-Oriented Languages

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

2.4.3.1 Orthogonal Approach

One way of integrating remote communication in the object model is by using the low-level communication primitives that the operating system provides. This has been called by Atkinson [6] the "operating system" approach. One way of doing it is by wrapping the communication primitives in

classes, giving them object-oriented interfaces, say a Socket class, an IPAddress class, etc. A good example is Java's network API [26]. The resulting programs look very much like Figure 20.

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

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

2.4.3.2 Full Integration

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

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

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

cal purposes, because the number of cross-space method invocations increases drastically as the application executes, with uncontrollable, negative effects on the application's performance.

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

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

Obliq provides a fully transparent object reference space across the network, where objects are always bound to the location where they were created. Objects, other than of primitive types, are never copied. The language does not provide any primitives for location control. Of all the distributed object-oriented languages, Obliq is the one that most closely achieved the full integration between objects and communication (in one simple way: restricting the data sent across spaces to

being object references and primitive data values). Therefore, the code tangling due to issues of remote communication is basically null. But because of the full integration, the language suffers from the two drawbacks that were identified before (i.e. performance penalties and lack of protection). If programmers need to tune these issues, they need to decompose the objects in their basic data types and send the pieces, instead of the objects, in method calls (as discussed in §2.3.2.1). Obliq has been integrated with a graphical user interface development environment, which was used to develop simple distributed applications [8].

2.4.3.3 Hybrid Approaches

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

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

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

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

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

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

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

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

2.5. Final Remarks

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

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

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

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