JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 2, July-August 2002

Java Distributed Separate Objects

Miguel Katrib, Iskander Sierra, Mario del Valle and Thaizel Fuentes,
Computer Science Department, University of Havana, Cuba.

Abstract

Java supports distributed programming using threads and Remote Method Invocation (JRMI). However, a Java thread does not match well with the object concept, and JRMI cannot easily used to synchronize distributed objects. This work proposes a more object oriented model for concurrent and distributed programming using the notion of separate objects. Separate objects are asynchronous objects maybe running on different processors. These objects are also used for synchronization. The presented approach should encourage more seamless designs. The implementation is supported by JOODE (Java Object Oriented Development Environment) which supports the use of an adaptable platform of a net of processors.

1 IS CONCURRENCY IN JAVA TRULY OBJECT-ORIENTED?

To obtain asynchronous behavior of an action, a Java programmer must extend the class Thread, encapsulating the action in the `run` method. The classic production-consumption example can be illustrated with the following pattern.

A class Producer extends Thread and encapsulates the production algorithm in its overridden run method, and a class Consumer, also extending Thread, encapsulates the consumption algorithm. Both, the production and consumption threads, will synchronize using a buffer. To initiate the production-consumption behavior a method must create instances of Producer and Consumer and "activate" both instances by calling their corresponding `start` methods (Listing 1).

```
class Producer extends Thread {
  private Buffer buf;
  public void produce(){
  ..produces an object and puts it in the buffer..
  }
  ...
  public void run(){
    while (some condition) { produce(); }
  }
}
```

Miguel Katrib, Iskander Sierra, Mario del Valle and Thaizel Fuentes: *Java Distributed Separates Objects*, in Journal of Object Technology, Vol. 1, No. 2, July-August 2002, pages 119-142.
http://www.jot.fm/issues/issue_2002_07/article2

```
class Consumer extends Thread {
  private Buffer buf;
  public void consume(){
  ..consumes an object from the buffer..
  }
  ...
  public void run(){
    while (some condition) {consume();}
  }
}
class Main {
  public static void main(string[] args){
    Buffer b = new Buffer();
    Producer p = new Producer(b);
    Consumer c = new Consumer(b);
    /* do the production consumption in
       an asynchronous way */
    p.start();
    c.start();
    ..do other actions..
}
```

LISTING 1 Production - consumption using threads

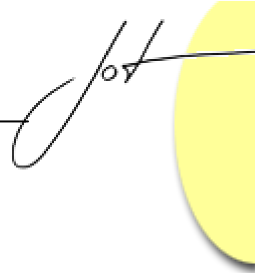
Supposing that the production process has two alternating production algorithms, the `Producer` class should be reprogrammed (Listing 2).

```
class Producer extends Thread {
  private Buffer buf;
  public void produce1(){..one algorithm..}
  public void produce2(){..other algorithm ..}
  ...
  public void run(){
    while (some condition) { produce1(); produce2();}
  }
}
```

LISTING 2 Two alternating production algorithms

But what if the production process changes again and we want two executions of the method `produce1` for each execution of `produce2`? The class `Producer` must be reprogrammed (or extended) changing (or overriding) the `run` method (Listing 3).

```
class Producer extends Thread {
  private Buffer buf;
  public void produce1(){
  ...one algorithm of production..}
  public void produce2(){
  ..other algorithm of production..}
  ...
```



```
    public void run(){
      while (some condition) { produce1(); produce1();
       produce2();}
    }
}
```

LISTING 3 Another production approach

The above approach is not particularly object oriented and does not enforce reusability because changes must be made in the `Producer` class and not in the class requiring the new production process. The only behavior that `Producer` should encapsulate is the two production algorithms, but how these algorithms should alternate in a production process should not be `Producer's` responsibility.

> *A cook (the producer) knows to prepare various recipes. He works concurrently with a waiter. But it is the waiter who knows the sequence of the client's requests. The waiter sequentially requests the cook for menu items while she/he should continues serving clients. The cook's routine must not change because the waiter's has changed.*

Another solution could be to have two classes `Producer1` and `Producer2`, one for each production method, and then create objects `p11` and `p12` of type `Producer1` and `p2` of type `Producer2`. The production strategy can now be placed in the client code (Listing 4).

> *But it is like having a cook for each recipe (a non too much efficient restaurant).*

```
class Producer1 extends Thread {
  private Buffer buf;
  public void produce(){..production algorithm 1..}
  ...
  public void run(){produce();}
}
class Producer2 extends Thread {
  private Buffer buf;
  public void produce(){..production algorithm 2..}
  ...
  public void run(){produce();}
}
class Main {
  public static void main(string[] args){
    Buffer b = new Buffer();
    Consumer c = new Consumer(Buffer b);
    c.start();
    Producer1 p11 = new Producer1(b);
    Producer1 p12 = new Producer1(b);
    Producer2 p2  = new Producer2(b);
    while (some condition){
      p11.start();
```

```
        p12.start();
        p2.start();
      ..do other actions..
  }
```

LISTING 4 Production objects with different threads

This solution looks more object-oriented than the previous but it hides some drawbacks. Observe that when running Listing 4, the behavior may not be as intended. The three different threads were started in the order `p11`, `p12` and `p2` but this does not mean that the result of their production will be in this order in the buffer. This depends on the duration of each production method and on the time slicing scheduling mechanism of the threads in the JVM.

Another drawback of this approach is the proliferation of classes. The original class `Producer` (Listing 2), having the two methods `produce1` and `produce2`, has been split into two classes (`Producer1` and `Producer2`), one for each production method. Nevertheless, in the original design, both production methods could share commonly defined information in their class. Such commonality could not be easily expressed in the new approach because there are two different classes with a single production method in each one. Where could this "common" information be located?

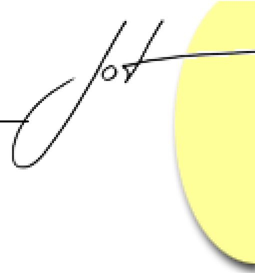
A Java fan could argue that the Java thread and the synchronized specifier are sufficient to model every concurrent behavior. But in opposition to object-oriented goals, the price to be paid could be derived in a clever design and low maintainability with little adaptability to a distributed environment. This paper attempts to offer a more object-oriented and expressive approach which serves as a foundation to the development of Java concurrent and distributed applications.

2 SEPARATE OBJECTS

The approach of Java separate objects is inspired by Meyer's proposal for the Eiffel language [Meyer 93, 97] but improves the synchronization model and at the same time attempt to fit better in a distributed and object-oriented environment.

A Java separate object is an instance of a separate class (a class implementing the interface `Separate`). In a separate class, asynchronous behavior is not only in one method (as in Java `run` method), but in every method. A separate object that will be the target of a call to a command method (`void` returning method) will execute the called method asynchronously with the caller, i.e. the execution flow of the caller (more exactly the execution of the method where the call is produced) will continue without waiting for a return. But if the called method is a query (non-void returning method) the call will be synchronous, i.e. the caller waits for the returned result.

```
class Producer implements Separate{
  private Buffer buf;
```



```
  public void produce1(){..production algorithm 1..}
  public void produce2(){..production algorithm 2..}
}
class Consumer implements Separate{  ...
  private Buffer buf;
  public void consume(){...}
}
class Main {
  public static void main(string[] args){
    Buffer b = new Buffer();
    Consumer c = new Consumer(b);
    Producer p = new Producer(b);
    while (some condition){
      p.produce1();...p.produce1();...p.produce2();
      ...c.consume();...c.consume();...c.consume();
    }
    ..do other actions while the
      production-consumption process is running
```

LISTING 5 The separate solution for the production-consumption problem

As will be explained later, the synchronization protocols are based on preconditions (boolean expressions acting as guards) preceding the methods.

The discussed production strategy in the previous example (Listing 4) is now solved in a straightforward manner (Listing 5). Because class `Producer` is separate, both methods `produce1` and `produce2` will run asynchronously with their callers. The production-consumption process is defined neither in the `Producer` class nor in the `Consumer`. It is defined in the different calling classes (`Main` in this example). Therefore, unlike the thread-based solution, a change in this process will result in a change in the caller classes and not in the existing classes `Producer` and `Consumer`. This results in a clearer and more maintainable approach.

Running the code in Listing 5 results in a scenario with three separate objects: the producer object `p`, the consumer object `c` and the main class (behaving as a "root" separate object). Due to the asynchronous behavior, the `main` method does not wait for the conclusion of the `p` production and the `c` consumption, but proceeds with other actions.

The Semantics of Separates Objects

Each separate object, target of a command method call, guarantees that the call will be "registered" while the caller can continue its execution without waiting for the execution of the call. Only when the call is a query does the caller wait until the target returns the calculated value. This "call and continue" policy is the basis of the asynchronous execution in the separate model.

Different callers can use a common target separate object. The target guarantees each caller that its calls will be executed in the order they were received. That is, calls coming from different callers could be interleaved, but keeping their relative order. For instance, a separate target `s` can receive from a caller `s1` the sequence of calls `s.h();...s.p();...s.r();` and from a caller `s2` the sequence `s.p();...s.g();...s.h()`. The actual order that `s` will execute the calls could be `h(from s1),p(from s2),g(from s2),p(from s1),h(from s2),r(from s1)` but could never be `h(from s1),p(from s2),g(from s2),r(from s1),h(from s2),p(from s1)` because `s1` executed the call `s.p()` before the call `s.r()` (see Figure 1).

The separate semantics will be implemented by attaching a "logical processor" to each separate object. Because logical processors may run on different machines, the separate model serves as basis for the development of distributed applications (some implementation issues will be discussed later in this paper).

Synchronously with the caller, a separate target of a call catches and "registers" the call for immediate or future execution, depending on whether the target is busy or not.

The execution of the call by the target looks like an atomic operation. Once the target begins to execute the call it will not initiate the execution of any other.

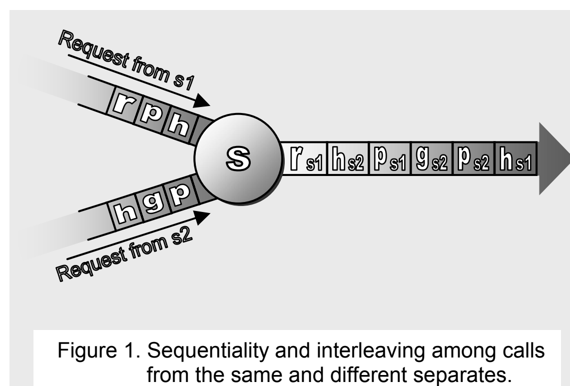
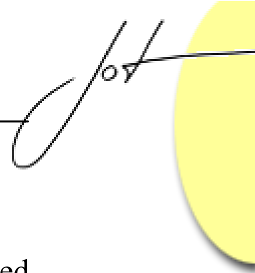


Figure 1. Sequentiality and interleaving among calls from the same and different separates.



This approach tries to keep a separate object (more exactly the logical processor attached to it) continuously busy. It can be either executing a call or suspended because it needs certain synchronization (synchronization will be explained later), but it is always ready to catch and register the calls.

If a separate object cannot initiate the execution of a call, or if it needs to suspend an executing call, then it will be able to execute other registered calls from other separate callers. That is, a separate object acts as if it has a different execution thread for each caller. Only one of the threads executes calls, while the others catch and register calls for future execution.

This approach is different from the one proposed by Meyer for Eiffel. In Eiffel, if a separate object is suspended while it is executing a call, then all other calls (even from other callers) must also wait.

There are cases where a caller would want to ensure, not only the atomicity of the execution of a call, but also atomicity of the execution of a sequence of calls to a same target. For example, it could be necessary to express that a sequence of calls, coming from a caller, must be executed by the target in a strictly consecutive order (without interleaving calls coming from other callers). This can be solved with the notion of transaction precondition which will be discussed in a later section.

Recursion and Separate Execution

Consider the separate classes `S1` and `S2` (Listing 6) and let us consider a scenario with two objects `s1` and `s2` (of types `S1` and `S2` respectively) where `s1` is executing a call to `h1`. During the execution of `h1`, is done the call, `s2` catches and registers the call `s2.h2()` to `h2`. Later `h1` calls `s2.g2()` and waits for the result. `s2` catches the call to `g2`, registers it and continues executing `h2`. Only when `s2` finishes the execution of `h2` will it execute `g2`. Nothing is wrong, `s2` will register the call to `g2` from `s1` and it will execute this call after executing the previous call to `h2`.

```
class S1 implements Separate{
  ...
  public void h1(){s2.h2();...int m = s2.g2();...}
  public void r1(){...s2.f2();...}
  public  int f1(){...int k = s2.g2();...}
}
class S2 implements Separate{
  ...
  public void h2(){...}
  public void f2(){...int j = s1.f1();...}
  public  int g2(){...}
}
```

LISTING 6 Recursion and separate objects

Consider another scenario of the Listing 6 where `s1` is executing `r1`, `s1` calls `s2.f2()` and continues executing `r1`. `s2` catches the call to `f2` and begins to execute it. While executing `f2`, `s2` calls `s1.f1()` and waits for the result. `s1` catches and registers the call to `f1` to execute when it finishes with the previous `r1`. When `s1` executes `f1`, it calls `s2.g2()` and waits for the result to be returned. If `s2` catches and registers the call to `g2`, not executing it until the previous call to `f2` has been completed, then `s1` and `s2` will be in deadlock (Figure 2). `s2` is blocked because it requires `s1` to finish the execution of `f1`, and `s1` is blocked because it needs `s2` to finish the execution of `g2`!

> *Assume a restaurant's chef is the only one who approves all the flavors. The chef receives an order for a pie from a waiter. While he is preparing the pie's shell he asks a fellow cook to prepare the filling. To finish the filling the second cook needs the taste approved by the chef. If the chef defers tasting the filling until he finishes the pie then he will fall into deadlock because he needs the filling to finish the pie. Nevertheless, if this fellow cook calls the chef to taste a barbecue sauce that he is preparing (for a third cook), then in such a case there is no trouble if the chef decides to defer the sauce's approval.*

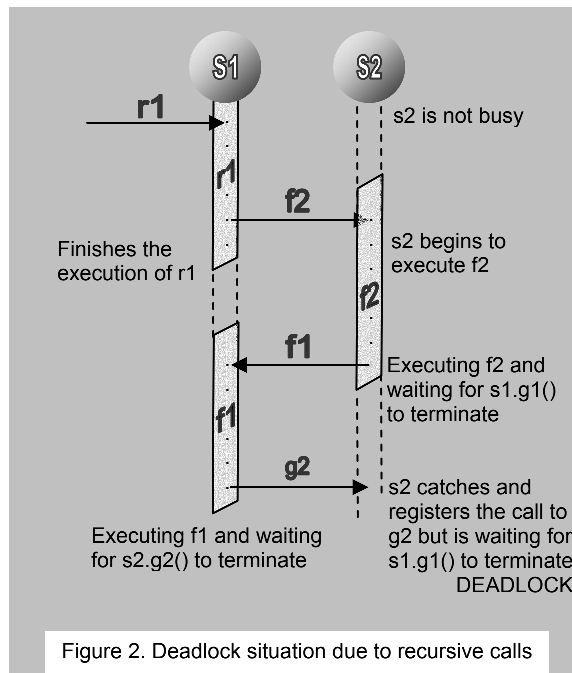
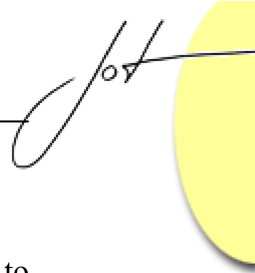


Figure 2. Deadlock situation due to recursive calls

To avoid this situation, apply the following rule to semantics of the separate objects:

"A separate object that does a first call to other separate is named the origin of the sequence of calls beginning in such call. When a separate is running a method and then catches a query call having the same "origin", then the separate will catch the call and will immediately execute it"



Note that the policy above avoids the deadlocks that could be created due to recursion loops.

The implementation section offers an idea of how this can be implemented by passing an implicit parameter with each query call. This parameter represents the "origin" of the sequence of query calls.

3 SYNCHRONIZING SEPARATE OBJECTS

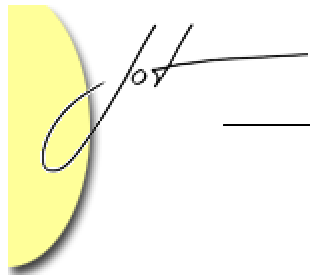
In practice, most applications using some asynchronous execution will require synchronization between the asynchronous components. Therefore the proposed separate model must include synchronization resources.

For instance, in the producer-consumer example (Listing 5) a producer `p` and a consumer `c` need certain synchronization when using the buffer. The consumer must wait for the buffer to be non empty. When inserting a product the buffer must wait for room if it is full (Note that due to the asynchronous behavior of a call to `produce`, it is not `p` that will wait but the buffer). The buffer must always work correctly if both `p` and `c` want to insert and extract at the same time. This classical synchronization example could be solved by integrating the Java synchronization resources with the separate model. However, this approach has the following drawbacks:

1. The semantics of the Java wait and notify methods must be extended by adapting them to the proposed separate behavior.
2. To prevent conflicting access to the buffer by `p` and `c`, the `Buffer` methods should be defined as `synchronized.` This can be done seamlessly if `p` and `c` are in the same address space as the buffer This is the case when each separate object is implemented using a different thread on the same processor [Avila 99], but this is not the case when we want to distribute the separate objects to be executed in different (and likely remote) processors. In this case, how would the distributed separate objects share a common buffer?
3. A separate caller could want two successive calls to a target not be interleaved with calls coming from a second caller. Such behavior is not easily expressed using the `synchronized` specifier.

Our approach proposes to use the separate objects themselves as the basis of the synchronization. This approach must suit a distributed model where separate objects do not share the same address space.

Any separate object `s` could be a "synchronization" object if it demands certain conditions to execute a method. We call such conditions synchronization preconditions.



Synchronization Preconditions

The idea of using "guarded" statements for synchronization was used in [Decouchant 89]. The separate approach proposed by Meyer for Eiffel uses the "double semantics" of a method's preconditions to synchronize. But when preconditions are used for synchronization, the Eiffel approach imposes a syntactic constraint requiring the use of only formal parameters declared as separate objects. This restriction disallows the use of a target separate objects own state for synchronization purposes. Note that, because its asynchronous character, the state of the target could change due to its interaction with other callers. Therefore, this constraint may result in unnatural designs from an object-oriented perspective, requiring the introduction of intermediate functions which query in their preconditions the separate objects "received" as parameters, and resulting in poor performance when separate objects are really remote objects.

The attempt to include assertions in Java has been largely discussed [Mannion 98, Meemken 98, RST Corp 1998]. In previous papers the authors also discuss the inclusion of assertions in Java and the role that assertions could play in concurrent programming [Fdez 98, Avila 99]. However, a full integration of the design by contract metaphor of assertions and the Java separate model should also do a proposition about the role of postconditions and invariants in the model. This theme goes beyond the current paper, so for the moment only preconditions will be examined.

A synchronization precondition has the form

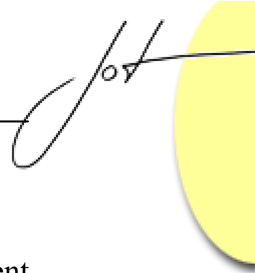
```
synchronization_precondition ::=
  require (list of boolean expressions)
```

A synchronization precondition goes between the method signature and the method body of a public method in a separate class.

```
method  ::=  method_signature  [synchronization_precondition]
method_body
```

To execute the called method, all the boolean expressions belonging to its precondition must evaluate to true. So the precondition acts as a guard for the method. As was explained above, when calling a method on a target separate object, that separate object captures the call and registers it, allowing the caller to proceed. Due to the asynchronous execution of the caller and target, if the method has a precondition, then that precondition will not be evaluated when the target catches the call but later when the target executes that call (i.e. after evaluating previous calls from the same caller).

It is important to note the foundation of the model. The caller method, whether the called method has a precondition or not, continues its execution (remember that we are talking about command methods because in a query call the caller always waits). It is only when the target separate object has executed previous calls from the same caller, that called method will be executed. If, at that moment, the precondition is not fulfilled, then the execution of the call is suspended and the separate object likely begins executing other hanging calls from other callers.



In traditional Java designs, the following two approaches can be applied to prevent insertion into a full buffer:

1. A check in the caller. The caller guarantees the synchronization:

```
synchronized(b){
  while (b.full()) b.wait();
  b.insert(x);
}
```

2. A check in the target. The called method guarantees the synchronization:

```
synchronized void insert(Object x){
  while (full()) wait();
  // insert the x
}
```

Both approaches can also be applied in JRMI for distributed applications.

The first approach puts the burden on each caller; the second one puts it on the target only.

The second approach is also more suitable in distributed (maybe remote) applications. Note that a remote call to the `insert` method will be made in both cases, but in the first approach a query remote call to `b.full()` is also made.

The role of precondition for synchronization used in the current separate model is inspired by the above second approach. It promotes designs where synchronization is encapsulated in the target, based as much as possible on the target state.

In the two above traditional Java approaches, both called and caller, will be really blocked. But in the separate model, based on the second approach, the caller will not be blocked. Due to the asynchronous semantics, the caller makes the call and continues. It is the target (the buffer) which will be blocked or not. Only if the same caller needs to later query this target, then the caller may have to wait.

So, this separate approach could result in improved performance when designing distributed and remote systems.

Each time a separate object finishes the execution of a call, and then all suspended calls (from other callers) will be "activated", because they may have a chance of proceeding because its precondition may now be fulfilled. This process is explained in more detail in the section "The Rationale of the Locking Mechanism".

The separate class `Buffer` of the production-consumption example can synchronize its usage by various producers and consumers by placing preconditions in its methods (Listing 7).

```
class Buffer implements Separate{
  ...
  public void insert(x) require (!full()){
].. inserts x in the buffer..}
  public Object get() require (!empty()){
  ..extracts an object from the buffer..}
  public boolean full(){...}
```

```
    public boolean empty(){...}
    public int total(){...}
    public int maxLength(){...}
  }
```

LISTING 7 A separate buffer with preconditions

Figure 3 shows a snapshot of pending calls to a buffer `b` from two producers `p1` and `p2` and from a consumer `c`. If `b` tries to execute the `insert` coming from `p1` when the buffer's memory is full, the execution of that call is suspended and `b` will try to execute a pending call from another caller (`p2`). But because `p2` call is also an `insert` it will also be suspended. Then `b` tries to execute the `get` coming from `c`, it will be executed because its precondition evaluates to true. When that execution of `get` finishes, all suspended calls will pass to the *ready* state and `b` will again try to reevaluate some of them. By default the precondition to be evaluated will be chosen in a non-deterministic way among the waiting preconditions (a different scheduling policy could be applied to fine tune this behavior).

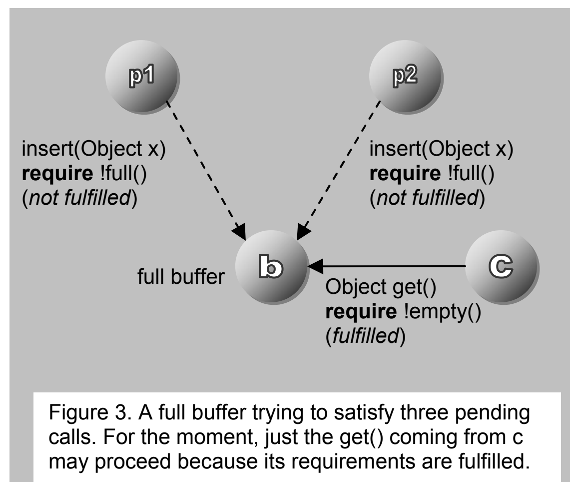
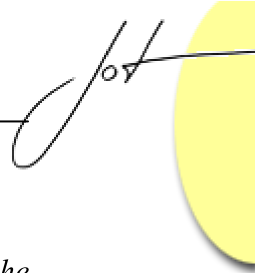


Figure 3. A full buffer trying to satisfy three pending calls. For the moment, just the get() coming from c may proceed because its requirements are fulfilled.

Having a common target with all of its callers suspended does not mean that we are in a deadlock situation. In a distributed scenario, a new separate caller could appear on the scene to be attached to that target and then to call a method that changes the target's state (perhaps allowing some suspended calls to proceed).

In this example of the `Buffer` class (Listing 7) the preconditions include only information about the buffer, so that when evaluating a precondition, to decide about the execution of the called method, the buffer does not need to ask another separate object. It is a good approach, mainly in a distributed platform, because the buffer does not need to query another separate object (more costly operation). From the design perspective, this approach matches well with the following scenario:



> *Several separate objects are coordinating their tasks through one of them (the "coordinator"). Using preconditions in the methods, the coordinator should try to express as much coordination protocol as it can, attempting to not query other separate objects. Thanks to the asynchronous behavior, a separate object is not forced to wait for the coordinator (unless it requests some information from it). It only requests a service (makes a call) and continues. The coordinator will later execute the call (if the precondition is fulfilled).*

Nevertheless, not all of design situations can be handled using the above pattern. Likely, to execute some methods a target separate object cannot decide from its own state only but also needs to query other separate objects. This will be discussed in the next section.

Transaction Precondition

As it was explained previously, separate semantics guarantees that all the calls coming from a separate caller to a target will be evaluated in the order that the caller invoked them. But this does not mean that calls from other separate callers will not interleave them in the target queue. To enhance the design capability of the model, it is necessary to have a way to ensuring that a target must execute a sequence of calls, coming from a particular caller, without executing calls from other callers, i.e. a way for a separate object *own* other separate object for a certain period of time. During the *owned* state, the separate object can capture and register calls from other separate objects but can only execute those calls coming from its *owner*. Only when the owned object is *released* can it proceed to execute other pending calls.

In the `Buffer` example, separate semantics and the `Buffer` method preconditions guarantee that a consumer will consume two objects coming from a same producer in the same order that the producer produced them. But this does not mean that the consumer will not consume other interleaved objects from other producers. A solution to this situation is to include an `insert2` method in the class `Buffer`. This method should have the precondition

```
void insert2(Object x, Object y)
  require (total<=maxLength-2) {
  ...insert x and y contiguous in the buffer
}
```

So, a producer wanting two objects `x` and `y` to be consumed consecutively should call `b.insert2(x,y)` and not `b.insert(x); b.insert(y)`. This is a good solution because again the precondition requires information only from the target of the call (no remote access required). But what if another producer wants to guarantee that three consecutive objects will be consumed without any other object between them, or four, or five? Reprogramming (or extending) the class `Buffer` each time may not be a good approach. From the design point of view the main pitfall here is that the solution would not be a burden on the buffer but on the producer.

The proposed mechanism to solve this problem is the notion of *transaction preconditions*. If the precondition of a method `h` of a separate object `s` includes other separate objects (e.g. qualified calls to other separate objects `s1` and `s2`) then the precondition is a *transaction precondition*. The method `h` could be considered a "transaction" on the separate objects appearing in its precondition, i.e. during the execution of `h` no calls from a separate object other than `s` will be executed by `s1` and `s2` appearing in the `h` precondition.

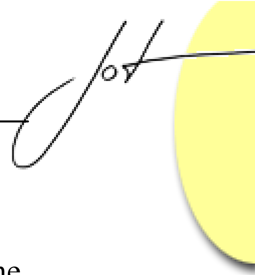
> *If the method precondition does not include queries to other separates, it could be considered as a transaction on the method itself. Note that while executing a method, a separate object can catch and register other calls, but the execution of the method is atomic, that is, the separate object will not execute any registered calls until it finished the current execution.*

For the example above, the claim that two consecutive productions result in two consecutive objects in the buffer can be now expressed in the producer and not in the buffer. A separate class `Producer`, using the separate buffer `b`, could include the method

```
public void doubleProduction()
  require (b.total()<=b.maxLength()-2){
..produce an object x..
b.insert(x);
..produce an object y..
b.insert(y);
  ...
}
```

Note the qualified queries `b.total()<=b.maxLength()` appearing in the precondition. To execute the `call p1.doubleProduction(x,y)` the separate object `p1` must first own the separate object `b` and then the precondition must evaluate true. During the execution of `doubleProduction`, `b` will not execute calls coming from separate objects other than `p1`, then the produced objects `x` and `y`, inserted into the buffer through the calls `b.insert(x) and b.insert(y)`, will be consecutive in the buffer without no other object `z` between them. Only when the execution of the method `doubleProduction` is finished, will release `b` and then other separate objects (maybe other producer or a consumer) waiting for `b` can proceed.

Compare the latter approach above with the former defining the `insert2` method in the class `Buffer`. The former may be better from the performance point of view because the precondition of `insert2` queries only target object (the buffer) and no remote queries (to other separate objects) are necessary. The latter may be better from the extendibility and maintainability points of view. A distributed application may be running where there's a need that two consecutive products will be consumed also consecutively. To include `insert2` in the `Buffer` class and to substitute dynamically (without stopping the application) the existing buffer `b` by a new one, will cause problems for existing producers and consumers. However, defining a new `Producer` class with the new



doubleProduction method, and introducing a new producer object on the scene, should not cause disharmony in the running application.

This tradeoff is a design decision. In the Eiffel approach, only the latter is possible because Eiffel requires a precondition to use only separate objects passed as parameters to the called method. This may result in a poor object-oriented design, and bad performance in simple cases such as the basic buffer methods. In the current proposition of the Java separate model, where both approaches are possible, the decision can be made by the developer.

```
class Fork implements Separate{...}

class Philosopher implements Separate{
  private Fork left, right;
  public Philosopher(Fork l, Fork r){
    left=l; right=r;}
  public void think(){...}
  public void eat()
   require (left.available() && right.available()){
   ...
  }

class Host {
  ...
  public void twoDinnersMeeting(int n){
    Fork[] forks = new Fork[n];
    for (int k=0; k<=forks.length-1; k++)
     forks[k]=new  Fork();
    Philosopher[] p = new Pilosopher[n];
    for (int j=0; j<=p.length-1; j++)
      p[j]=new Philosopher(forks[j], forks[(j+1)%n]);
    for (int j=0; j<=p.length-1; j++){
      p[j].think(); p[j].eat();
    }
    for (int j=0; j<=p.length-1; j++){
      p[j].think(); p[j].eat();
    }
  }
```

LISTING 8 The dining philosophers

How to avoid deadlock? The dining-philosophers metaphor

The proposed approach of synchronization based on preconditions also promotes the design of deadlock free applications [Katrib 99]. Applying a "multiple atomic owning policy" to separate objects appearing in a precondition facilitates avoiding situations where a separate object s owns a separate object s1 while it is waiting for a separate object s2, and in the meantime another separate object q owns s2 while waiting for s1.

The classical example of the dining philosopher's metaphor could be expressed without deadlock as shown in Listing 8. Note the precondition `require (left.available() && right.available())` of the method `eat`. A separate philosopher `p` either takes control of both left and right forks or does not take anyone, giving another philosopher a chance to eat.

Note the use of the boolean method `available`. The `available` method could be applied to any separate object. It always returns true. To use this `available` in a method's precondition means that to execute the method it is necessary to own the separate object target of the `available` call.

The `available` method has also an added benefit. Used as another query statement inside the body of a method as in `f(){...s.h(); ...s.p();...boolean b=s.available();...}` it means that the caller to `f` will wait for `s` to resume all its previous calls (`h` and `p`).

To conceive of a fork as a truly separate object introduces other advantages. For example, promoting a less promiscuous and more hygienic behavior, the class `Fork` could include methods like `isClean`, `clean` and `dirty`. In this case the `eat` method should change its precondition to require `(left.isClean() && right.isClean())`.

So, the separate object "hosting" the philosopher's meeting should create an object `waiter` of the separate class `Cleaner` (see Listing 9)

In order to eat, the hygienic philosopher waits for his neighbor forks to be cleaned, while a humble waiter is always watching for dirty forks to clean.

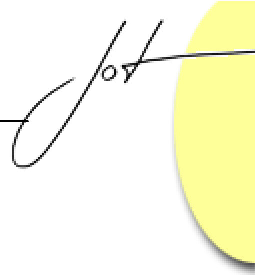
If, in place of the `clean` precondition, the "a priori" query style was applied before the call, as in

```
void doService(Fork[] forks){
  while (...meeting is open...)
    for (int i=0; i<=forks.length-1; i++)
      if (!forks[i].isClean())forks[i].clean();
  }
}
```

then the call `forks[i].clean()` won't be executed until the query call `forks[i].isClean()` has returned. But note that this query will not return while a philosopher is using the fork. The waiter is waiting for the philosopher to finish with the fork when he should be trying to clean another fork.

```
class Fork implements Separate{
  ...
  public boolean isClean(){...};
  public void clean()require (!isClean()){...};
}

class Philosopher implements Separate{
  //...as in Listing 8
```



```
}

class Cleaner{
  ...
  Cleaner( ){...}
  void doService(Fork[] forks){
    while (..meeting is on..)
      for (int i=0; i<=forks.length-1; i++)
        forks[i].clean();
  }
}

class Host {
  ...
  public void twoDinnersMeeting(int n){
    //create forks and philosophers as in Listing 8
    Cleaner waiter = new Cleaner( );
    waiter.doService(forks);
    //
```

LISTING 9 A healthy philosopher meeting

4 WHY NOT JAVA JRMI?

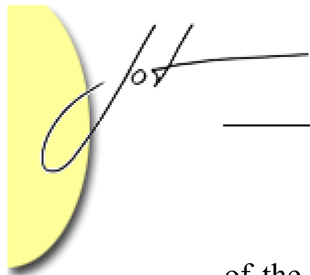
The Java distributed object model is known as JRMI (Java Remote Method Invocation). In this section the main drawbacks of the JRMI, compared to the proposed separate model, are analyzed. Another work to synchronize separate objects in JRMI or CORBA based environment based on the design by contract metaphor and using IDL languages is presented in [Pastrana 01].

JRMI is just a way of localizing remote objects and remotely executing calls on such objects. Implicitly, it does not assume asynchronous behavior between callers and those called methods. If we want to achieve such asynchrony, we must explicitly some patterns such as the following

1. Define a thread class `C` and encapsulate the remote call `r.h()` in the `run` method.
2. Substitute the intended original synchronous remote call `r.h()` by an instantiation `x` of the class `C` and execute the `start` method of `x` (likely passing the same parameters that the `run` method should pass in the remote call to `r`).

But it is an obligation on the developer and again, from the design and programming point of view, this has not an object-oriented flavor.

Moreover, a remote object does not have any kind of autonomous synchronization when it is called from several remote objects. Once more, the burden is on the developer



of the remote class. Accordingly, the developer must use the `synchronized` operator and the `wait` and `notify` methods, in the implementation of the remote methods. But, as previously discussed, such Java synchronization resources are not too expressive for the development of concurrent and deadlock free applications. Using these resources could easily result in a scenario in which a remote `r1` is blocked, by its work with another remote object `r2`, while `r1` could be serving another remote object `r3`. This problem was solved in the current separate model.

In JRMI a remote method declaration must include the `throws java.rmi.RemoteException` clause, forcing the caller to catch it. This requires more effort by the programmer when developing a highly distributed system with many remote calls. This policy could be justified because in JRMI objects are truly remote, and then there are more chances of connection failures.

The separate model is a more abstract model than that of JRMI. In the separate model, the separate objects are attached to logical processors. Two logical processors may be running on different remote processors or on the same machine. In both cases *JOODE* hides the details, encapsulating and protecting the connection operations (but the developers can change the maping between real and logical processors). This gives more flexibility and enhances expressiveness, hiding a lot of the dirty work. Therefore, many exceptions that could occur in JRMI, and that must be handled by the developer will now be captured and handled by the *JOODE* platform. The exceptions that are raised by the separate model are defined as Java runtime exceptions.

5 IMPLEMENTATION ISSUES

The JOODE platform

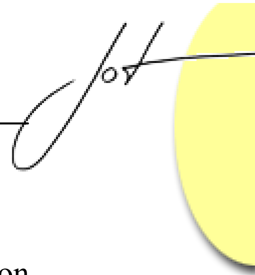
The Java Object Oriented Development Environment (*JOODE*) can join the "processing capability" of a number of physical processors located in a LAN. For an application's programming layer, *JOODE* may look like a single machine with multiple processors.

The *JOODE's* dynamic behavior allows the addition or the leaving of a physical processor into (from) the platform. *JOODE* tries to preserve the calculated information and to keep the consistency of the system, avoiding application crashing.

JOODE supports the transparent distribution, communication and migration of Java objects throughout the physical processors existing in the platform.

The presence of several physical processors to run a Java application in the *JOODE* platform is not mandatory. Even if there is only one processor, *JOODE* can create each Java object locally, and assign a different execution thread to "run" each object.

Based on *JOODE*, the proposed separate model offers a greater level of abstraction to the developer. The developer does not have to detect and locate free processors in the



network, because this is done by *JOODE*. Therefore, a concurrent distributed application can be more flexible and more fault-tolerant.

JOODE has a default policy behavior that makes good use of this "computational distributed power". However, *JOODE* also includes a mapping tool which allows the developer to modify this policy or to decide on the actual location of the separate objects on the network.

Logical Processors and Calls-queue

For each separate class `C` the *separate* compiler renames it as `CCore,` and adds a class `C` which acts as a proxy decorator of `CCore.` This proxy class `C` has an interface which is analogous to the interface of its corresponding separate class. For each method `f(...)` in the original separate class `C` a corresponding method `f(...)`is included in the new `C`. This means that client classes of `C` do not need to change.

Each proxy object runs in the same address space (JVM) as the client creating (or using) the corresponding separate object.

Each separate object has an associated logical processor (*lp*) to execute the calls to it. This logical processor is mapped by *JOODE* to the physical resources. The proxy transforms each call from a client object to a "call" to the *JOODE* service layer. If the called method is a void method the proxy will return the control to the caller while the *JOODE* organizes the asynchronous execution of the call. But if the called method is a query (non-void method), then the proxy will not return to the client until the service layer returns the computed value to the proxy.

The *service layer* locates the *lp* associated with the separate object and passes the call to it. The lp puts the necessary information in a queue for its execution and immediately returns control to the service layer.

The queue corresponding to calls coming from a caller `s1` to a target `s2` is named the *calls-queue* `(s1,s2)`.

To make this easier to understand, imagine that calls in the calls-queues are executed based on a timestamp assigned when the *lp* captured the call. Different scheduling policies could be applied to enhance this behavior, for example to give a higher priority to calls close to their expiration time or to query calls because the caller is waiting. This could help avoiding starvation and similar unfair behaviors. *JOODE* includes resources so that the developer can fine tune the scheduling policy.

The basic unit of execution in an *lp* is a call inserted into one of its queues. As much as possible the scheduling policy will not request the *lp* to begin execution of another call if the *lp* is executing a call. But if the execution of a call in a particular calls-queue is not possible, due to synchronization precondition, then the *lp* can execute other calls from another calls-queue. Having one calls-queue for each caller makes easier to implement the avoidance of unnecessary delays in the lp behavior.

A calls-queue associated to an *lp* can be in one of the following three states:

1. *Running* .The lp is executing the call at the head of the queue.
2. *Ready*. The call at the head of the queue is waiting for a chance to be executed (it will be selected or not according to the scheduling policy)
3. *Suspended*. The call at the head of the queue is suspended because a previous evaluation of its precondition was unsuccessful. The call is waiting for an event occurrence (i.e the necessary separates used in the precondition were unlocked by their corresponding owners.) "signaling" that the call has a chance of fulfilling its precondition.

As was explained previously, a call always has a chance of being executed.

Figure 4 shows a snapshot of a likely scenario with two producers, one consumer and a buffer.

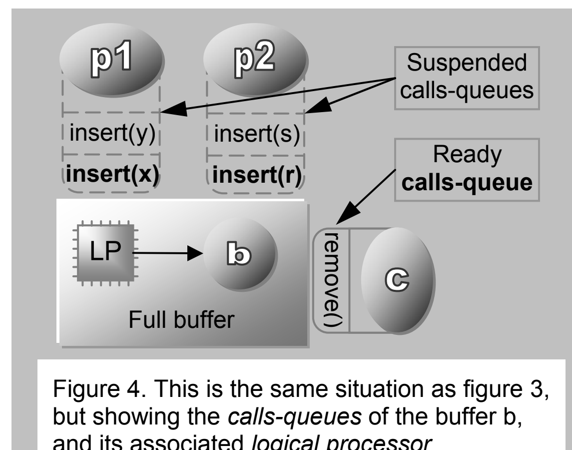


Figure 4. This is the same situation as figure 3, but showing the *calls-queues* of the buffer b, and its associated *logical processor*

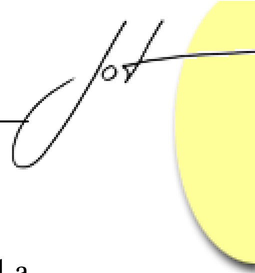
JOODE assigns a global identifier to each separate object. To enforce the recursion rule, a proxy passes this identifier with a call. The lp knows the global identifier of the call being executed. If the lp catches a new query call having the same global identifier, then it will not queue the call but will synchronously execute it just in time.

> *A well designed distributed application should not have a very long sequence of such calls. Furthermore, if we want to take advantage of having several distributed processors, a divide and conquer algorithm (typical of many recursive methods in the sequential approach) should be matched in different separates objects, so resulting in a non strictly recursive execution.*

The Rationale of the Locking Mechanism

An *lp* can be in one of two states locked or unlocked. An *lp* is locked when a caller owns the separate object running on the *lp*. The *lp* will execute only calls of the calls-queue of the separate object that owns it.

The locking mechanism is based on a mutexes' pool supported in *JOODE*.



In the mutexes' pool each *lp* has a reference to the separate object owning it, and a counter for the number of times it was locked.

The procedure by which an *lp* executes a call to a method `h` (including separate objects `a1`, `a2`, ...`an` in the precondition of `h`) at the head of the calls-queue `(s1,s2)`, involves the following steps (to simplify this, the *lp* associated with a separate object `ai` will also be referred to as `ai`):

1. If some of the `a1`, `a2`, ...`an` are locked by a separate object other than `s2`, then the calls-queue is updated to the suspended state.
2. If all `a1`, `a2`, ...`an` are unlocked (or locked by `s2` itself) then they will be locked (the lock counter is incremented). Using the mutexes' pool, *JOODE* guarantees that this locking procedure will be atomic.
3. After locking `a1`, `a2`, ...`an` the precondition expression is evaluated. If it evaluates to true, the call to `h` in the calls-queue is executed. If it evaluates to false, then the lock counters are decremented and the calls-queue passes to the *suspended* state.
4. When the execution of `h` finishes then the lock counters `a1`, `a2`, ...`an` are decremented. Note that this "unlocking" procedure does not need to be atomic, i.e. as soon as one of the `ai` is unlocked; other suspended calls (from the same or other separate objects) can take advantage of this.

A *JOODE* service layer maintains different *waiting* lists for the activation *suspended* calls-queues when the required *lp* is unlocked.

Note that a calls-queue `(s1,s2)` in a *ready* state will not necessarily pass to the running state. The ready state only means that the *lp* associated with `s2` intends to execute the call at the head of `(s1,s2)` (it could pass back to the suspended state if steps 1 and 3 once again do not succeed).

To avoid latency problems, all of the above locking-unlocking procedure is done by a *JOODE* service layer concentrated in one JVM address space. This *JOODE* service layer will only ask the *lp* to evaluate a suspended call when all the separate objects that the called method requires are available, and then locked by the service layer. So, useless requests to, likely a remote, *lp* are avoided.

6 CONCLUSIONS

The presented work explain a model and a tool to design an implement concurrent an distributed Java applications based on the notion of separate object. These separate objects serve also for synchronization purposes, when used in the method’s preconditions.

This approach intent to avoid some limitations and impedances, between the Java model of threads and the JRMI, to facilitate the design and implementation of object oriented and distributed Java applications.

The implicit synchronous-asynchronous policy, applied by default to the calls to void or non void methods of a separate target, is insufficient to develop more customized distributed applications. For example, contrary to the default policy, we could like that a call to a void method will done synchronously or a call to a non void will be done asynchronously. Furthermore, we could need that a call begins to be executed before an expiration time. In other work [Del Valle 02] authors will present the notion of *behavior object* acting as a *pattern* to customize the interaction between separate objects and the calls between them.

REFERENCES

[Decouchant 89] Decouchant D, et al. A synchronization mechanism for typed objects in a distributed system. ACM SIGPLAN Workshop on Object-based Concurrent Programming, vol. 24, pp 105-107. SIGPLAN Noticies, ACM Press. 1989.

[Del Valle 02] Del Valle M, Sierra I, Katrib M, Fuentes T Patrones de coordinación entre objectos separate Java, to be presented in V Workshop IDEAS 2002, La Habana, April 2002.

[Fdez 98] Fernández D, Katrib M, “JavaA: Including assertions in Java”, Computación y Sistemas, Revista Iberoamericana de Computación, México, September 1998.

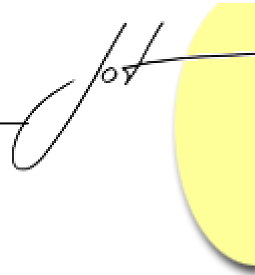
[Fuentes 02] Fuentes T, Katrib M, Sierra I, Del Valle M, JOODE Una plataforma multicapas para la distribución de objetos Java en una red local, to be presented in V Workshop IDEAS 2002, La Habana, April 2002.

[Avila 99] Avila F, Katrib M, Pimentel E, Objetos Concurrentes en Java, Proceedings of the 2nd Workshop Iberoamericano en Ingeniería de Requisitos y Ambientes de Desarrollo de Software, IDEAS'99, Costa Rica, March 1999, ISBN 9968-32-000-5

[Katrib 99] Katrib M, Pimentel E, Synchronizing Java threads using assertions, Proceedings of TOOLS 31st, Asia 99, Editors Jian Chen, Jian Lu, Bertrand Meyer, IEEE Computer Society, September 1999, ISBN: 0-7695-0393-4

[Mannion 98] Mannion M, Phillips R., "Prevention is Better than Cure. Design by Contract and Java", Java Report, September 1998.

[Meemken 98] Meemken D, JaWA: Java with Assertions, Universita’t Oldenburg, http://theoretica.informatik.uni-oldenburg.de/~jawa/



[Meyer 93] Meyer B, Systematic Concurrent Object-Oriented Programming, Communications of the ACM, vol. 36, no 9, September 1993.

[Meyer 97] Meyer B, OBJECT ORIENTED SOFTWARE CONSTRUCTION: 2nd Edition, chapter 30, Prentice Hall 1997.

[Pastrana 01] Patrana J.L, Katrib M, Pimentel E, Coordinación de Objetos Separate en Java, IV Workshop IDEAS 2001, Santo Domingo de Heredia, Costa Rica, Abril 2001.

[RST Corp 1998] RSTCorp. “AssertMate”, http://www.rstcorp.com/

About the authors

Miguel Katrib is Dr. in Computer Sciences and full professor of the Computer Sciences at the University of Havana. He authored several educational books and research papers and is member of the Program Committee of different congress and workshops. His educational and current research areas include programming methodologies, programming languages, compiling, object technology and web programming technologies. He is an “object from the very beginning enthusiast” and also a cine fan. He can be reached at mkm@matcom.uh.cu.

Thaizel Fuentes is postgraduate student of a Master Program in Computer Science at the University of Havana. She also works as assistant professor in the Computer Science Department at the University of Havana. She heads a research team on Remote Programming. Her research areas includes: object oriented technologies, concurrency, distribution, parallelism and web programming. She likes dance. She can be reached at thai@matcom.uh.cu.

Iskander Sierra is postgraduate student of a Master Program in Computer Science at the University of Havana. He is a member of the research group of Object Oriented Programming and Web Technology. He is also interested in reflection programming and oo compiling. He plays tennis. He can be reached at isk@matcom.uh.cu.

Mario del Valle is postgraduate student of a Master Program in Computer Science at the University of Havana. He is a member of the research group of Object Oriented Programming and Web Technology. He is also interesting in net administration and server side programming. He loves drawing and scuba diving. He can be reached at mariovm@matcom.uh.cu