Concurrent programming in SCOOP: a tutorial

18 April 2015

Ever more often, users want programs to be concurrent. A concurrent e-mail client, for example, can download new messages while you are reading earlier ones. The alternative is a *sequential* program, which does only one thing at a time: a sequential e-mail client would force the download to wait while you read, and, once downloading starts, would force you to wait before reading again. Not attractive.

SCOOP is the Eiffel mechanism that enables you to make your programs concurrent. The name means *Simple Concurrent Object-Oriented Programming*. Simplicity is indeed one of SCOOP's biggest draws. The "S" could also stand for Safe: concurrent programming with traditional approaches can be very tricky, but SCOOP removes many of the traditional pitfalls, such as "race conditions".

You can read in detail about the theory and rationale in the bibliographic references. This tutorial is a hands-on presentation of how to use SCOOP in practice. We will go through a simple example, an email client with its viewer and debugger.

You can download an Eiffel project with all the default SCOOP settings at http://..., load it into EiffelStudio, then fill in the initially empty class texts from the models below, compile them and run then as you go. At http://... you will find the final version of the example.

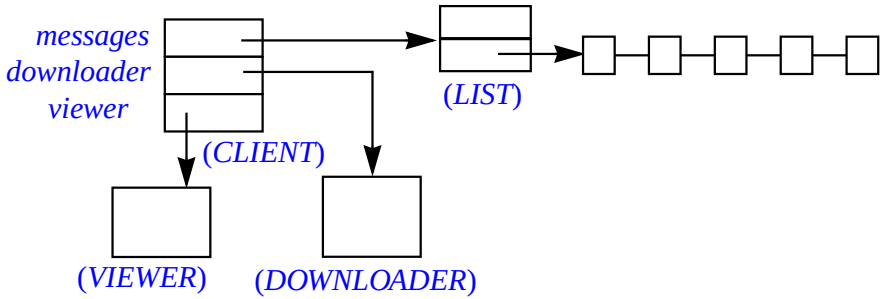
1 HERE AND THERE

The basic way to make a program a SCOOP program is to declare some of its entities as **separate**, so that operations on the associated objects run elsewhere.

Without **separate**, a sequential version of an email client could read:

```
class CLIENT feature
    messages: LIST [STRING]          -- Email messages received
    downloader: DOWNLOADER           -- Downloading engine
    viewer: VIEWER                   -- Message viewing engine
    … More features …
end
```

yielding at run time a simple object structure, with all objects in a single *region*:



Sequential*: *one region for all objects

The *CLIENT* object represents an email client and contains references corresponding to the attributes of the class:

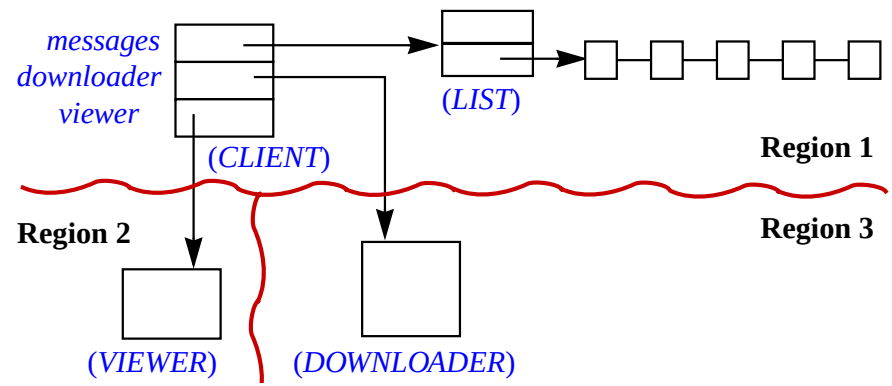
- *downloader*, a reference to an object of type *DOWNLOADER*.
- *viewer*, a reference to an object of type *VIEWER*.
- *messages*, a reference to a list header, which itself gives access to other objects, the actual list elements (messages).

Because these objects are all in one region, a single processor will take care of operations on all of them; if the class *DOWNLOADER* has a routine *download* to download messages and *VIEWER* has *view_one* to view messages, at most one of these routines will be running at any time.

With SCOOP you can make things concurrent by putting objects in different regions, handled by different processors. You declare something **separate** to specify that it will be in a different region:

```
class CLIENT feature
    messages: LIST [STRING]                -- Email messages received
    downloader: separate DOWNLOADER        -- Downloading engine
    viewer: separate VIEWER                -- Message viewing engine
end
```

The run-time picture changes to reflect the distribution of objects into regions:



Concurrent*: *three separate regions

The existence of three regions, delimited in the figure by curvy red lines, follows from the **separate** declarations. The list header and list element objects are not separate from the *CLIENT* object, so they belong with it in region 1. But since *viewer* is **separate** the viewer object is in a different region, number 2 in the figure. The *downloader* also has its own region, called region 3.

Here each of the classes *CLIENT*, *DOWNLOADER* and *VIEWER* needs only one instance (they are "singleton" classes), but in general a class may have many instances, spread over any number of regions.

You now know the basic rule of concurrency with SCOOP: declaring entities as **separate** to ensure that the corresponding objects belong to different regions.

2 PROCESSORS

To handle operations on objects, we need *processors*. A processor in SCOOP is simply a mechanism that can execute instructions in sequence. It can be a physical processing unit (one of the cores in a computer) but more commonly it will be a software mechanism such as a *thread*, as in the current SCOOP implementation.

Whatever the implementation, a processor has two fundamental properties:

- It is sequential. Each processor executes instructions one at a time. You get concurrency by using *several* processors.
- Every processor is associated with a region. The processor handles all operations on objects in the region; we say it is their **handler**. The example has three processors: one handling the email client, the list header and the list elements (region 1); another handling the viewer object (region 2); and a third, handling operations on the downloader (region 3).

To "handle" an object means to execute all the operations on that object. In object-oriented programming, every operation indeed works on an object, in the sense that *x*.*r* (…) works on the object attached to *x*. That object is in a region, and the region's handler will execute all such operations.

Class *DOWNLOADER* needs a routine for what downloaders do — download:

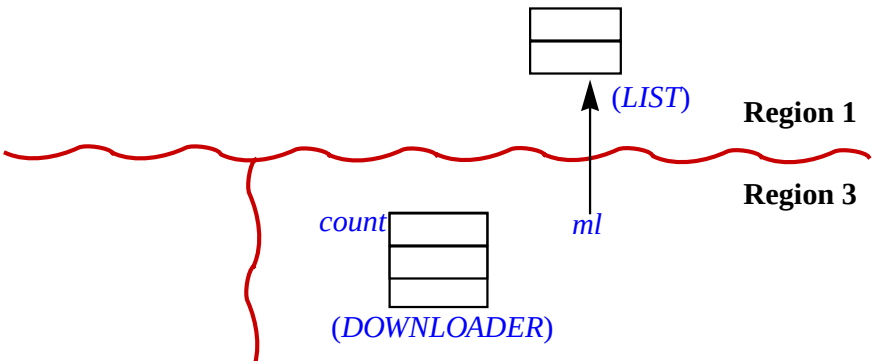
In class DOWNLOADER.

```
download_one (ml: separate LIST [STRING])
            -- Read one message and record it into ml.
    local
        latest: STRING
    do
        count := count + 1                          -- Increment message count
        latest := "message " + count.to_string      -- Make up new message
        print ("Adding message: " + latest)         -- Display that message
        ml.extend (latest)                          -- Add message to list
    end
```

In an actual email system, the downloader would get message from a file or socket. In the simulated downloader above, the *n*-th message is simply a string of the form "Message *n*". ("+" between strings is concatenation, and *to_string* gives the text representation of an integer.) The call to *print* is not needed for the downloader's operation but will help us follow the execution when we try the email system in a few moments.

To record the message, *download_one* uses the procedure *extend* from *LIST*, which adds an element at the end of a list.

The processor handling the *DOWNLOADER* object executes all the instructions of *download_one*: the first three, which do not involve separate variables, and the call to *record_one* (see the illustration below). But in the instruction *ml*.*extend* (*m*) of *record_one*, *ml*, the list of messages, is separate, so it has a different handler, and that processor — the handler for all region 1 objects — will execute the call to *extend*.



The rule is simple:

- Any operation of the form *x*.*some_feature* (…), where *x* is separate, will be executed by the handler of *x*.
- The handler of the current object will, by default, execute all other operations: simple operations such as the assignment *count* := *count* + 1, unqualified feature calls such as *some_feature* (…), and qualified calls *x*.*some_feature* (…) where *x* is not separate.

3 STARTING A PROCESSOR

How do we get new regions and the associated processors? Simply by creating separate objects. If we start from a *CLIENT* object, it can create the viewer and the downloader as part of its own creation procedure (constructor):

```
class CLIENT create
    make
feature -- Initialization
    make
        do
            create {LINKED_LIST} messages.make    -- Create message list
            create downloader.make (messages)
            create viewer.make (messages)
        end
feature -- The attributes as before:
    messages: LIST [STRING]
    downloader: separate DOWNLOADER
    viewer: separate VIEWER
end
```

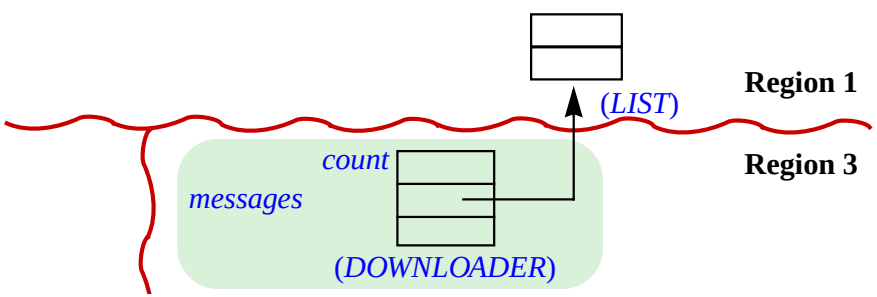
The creation procedure *make* of *CLIENT* starts by creating the list of messages. Since *messages* is not separate, this first instruction is a normal creation. The next two, however, have targets *downloader* and *viewer* that are declared **separate**; the effect is to create a new region for each of them, producing the situation shown in the last two figures, and also to start a processor for each of these regions.

What does it mean to "start a new processor"? In the default thread-based implementation of SCOOP, the simplest is to start a new thread. But the implementation can be smarter; for example it can reuse an existing thread from a "thread pool". And remember that nothing constrains SCOOP to use threads. What matters, independently of the implementation, is that when you create a processor it will be able to host objects in its associated region and sequentially execute operations on them.

In the example, the client passes *messages* as argument to the creation procedures of both separate objects, because both need access to the message list: the downloader to add messages, and the viewer to find messages. (All the classes involved have a creation procedure with the name *make*.)

4 KEEPING REFERENCES TO SEPARATE OBJECTS

Take a closer look at the downloader object.



Concurrent*: *three separate regions

To retain access to the message list in region 1, the object needs to keep a reference to it; you can declare that reference in the class as

```
messages: separate LIST [STRING]
```

In class DOWNLOADER.

Since this *messages* field will be a reference to an object in another region, it is declared as **separate**. (They both use same name, *messages*, which does not create any confusion since they are in different classes.)

It is the job of the *make* creation procedure of *DOWNLOADER* to record that reference. We saw that the client calls that procedure, as part of its own initialization, through the creation call

```
create downloader.make (Current)
```

In class CLIENT.

passing itself as argument, so that the downloader knows for which client it is working. So here is how *DOWNLOADER* looks so far:

```
class DOWNLOADER create
    make
feature -- Initialization
    make (c: separate CLIENT)
            -- Initialize downloader so that it will download messages for c.
        do
            messages := c.messages
        end
feature
    messages: separate LIST [STRING]
    count: INTEGER                  -- Number of downloaded messages
    download_one (ml: separate LIST [STRING])
        do … end        -- As seen above (page 3)
end
```

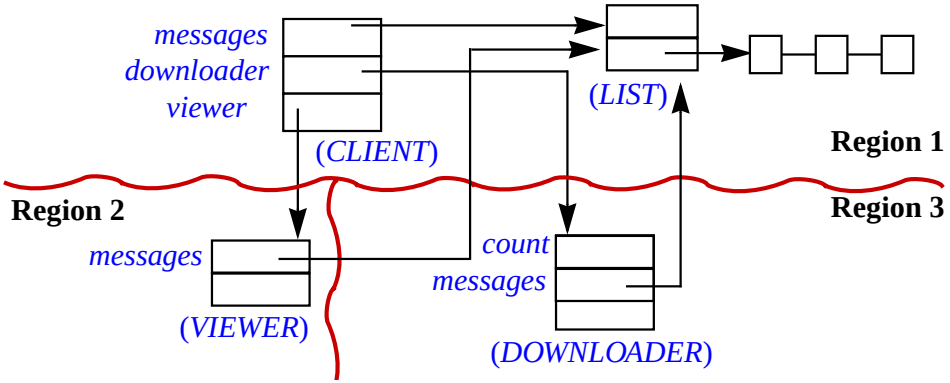
For simplicity, the attribute representing the message list has the same name, messages, in all relevant classes. In classes other than CLIENT, it is a separate reference to the messages list of the client region.

Both the argument *c* of *make* and the attribute *messages* are declared **separate**, because the client object and its message list belong to the client's region (region 1) and hence, seen from the downloader, are separate.

The viewer side uses exactly the same initialization

```
class VIEWER create
    make
feature
    make (c: separate CLIENT)
            -- Initialize viewer so that it will support viewing messages of c.
        do messages := c.messages end
    messages: separate LIST [STRING]
    … More features to come …
end
```

To refresh our minds here is the full run-time object picture:



Email system regions and objects

Both the downloader and viewer objects have a separate reference, called *messages* in both cases, to the message list in the client region. For information hiding, it may be preferable to gather all operations on the message list in class *CLIENT*, so that the viewer, downloader and other objects involved only keep a reference to the client object, and go through it rather than working directly on the list. The technique used here, however, illustrates the property that an object in any region can have separate references attached to any object in another region.

5 BUILDING A CONCURRENT PROGRAM

We have set up the stage and can now start building our little email system. The idea is to get right away to something that compiles and runs, so it is going to be *simulated* email (do not throw away Outlook and Gmail yet).

We add to *DOWNLOADER* a procedure that repeatedly downloads messages:

```
live
        -- Get messages and add them to the client message list.
    do
        from until is_over loop
            download_one (messages)
            wait (D_temporization)
        end
    end
is_over: BOOLEAN                -- Should we stop downloading? See section 13.
```

These four features go into class DOWNLOADER.

Until *is_over* is set, *live* repeatedly downloads a message and waits *D_temporization* seconds, using a system procedure *wait*. This is again simulated email; a real mail downloader, instead of instead of “pulling” messages at preset intervals, would be “pushed” to read messages as they become available.

On the simulated viewer's side, things are similar, with a *view_one* procedure and a procedure *live* describing the overall viewing process:

```
view_one (ml: separate LIST [STRING])
        -- Simulate viewing: if ml contains messages, display one, chosen randomly.
    do
            if not ml.is_empty then
            print ("Viewing message: " + ml.i_th (random (1, ml.count)))
                -- Assuming random yields a random integer in the given interval
        end
    end
live
        -- Simulate a user viewing a message o3fnce in a while.
    do
        from until is_over loop
            view_one (messages)
            wait (V_temporization)
        end
    end
is_over: BOOLEAN                -- Should we stop viewing? See section 13.
```

In VIEWER.

Do not confuse live and is_over with the features of the same name in DOWNLOADER, pages 6 and 7.

We can use the classes defined so far to build a mini-email system. To start the system, it suffices to create a *PLAYER* object and to call its routine *play1*, which concurrently runs the respective *live* routines on the downloader and the viewer:

```
note
    description: "Driver class for trying out email mechanisms"
class PLAYER create make feature
    client: separate CLIENT
    downloader: separate DOWNLOADER
    viewer: separate VIEWER
    messages: : separate LIST [STRING]
    make
        do
            create <NONE> client.make    -- Creates downloader and viewer
            downloader := client.downloader
            viewer := client.viewer
            messages := client.messages
        end

    play1
        do live_both (downloader, viewer) end
```

The <NONE> specification will be explained in section 14.

```
    live_both (d: separate DOWNLOADER; v: separate VIEWER)
            -- Run d and v concurrently.
        do
            d.live
            v. live
        end
end
```

You can compile a system with the above class and run *play1*. The execution output will look something like this:

```
Adding message: Message 1
Adding message: Message 2
Adding message: Message 3
Viewing message: Message 2
Adding message: Message 4
Viewing message: Message 4
Adding message: Message 5
Viewing message: Message 2
Viewing message: Message 5
…
```

"*Something like*" this, not necessarily exactly this, because the interleaving of messages of both kinds depends on several factors: the *random* function, the temporization values, and the speed of the physical processors and threads involved.

Such a behavior, where the program's results may depend on timing properties, is called *non-deterministic*.

Like cholesterol, non-determinism comes in both good and bad flavors. SCOOP supports "good" non-determinism and avoids the bad variety. Non-determinism is bad if it is an artefact of the implementation and hence undesirable. It is good if it follows from the problem specification.

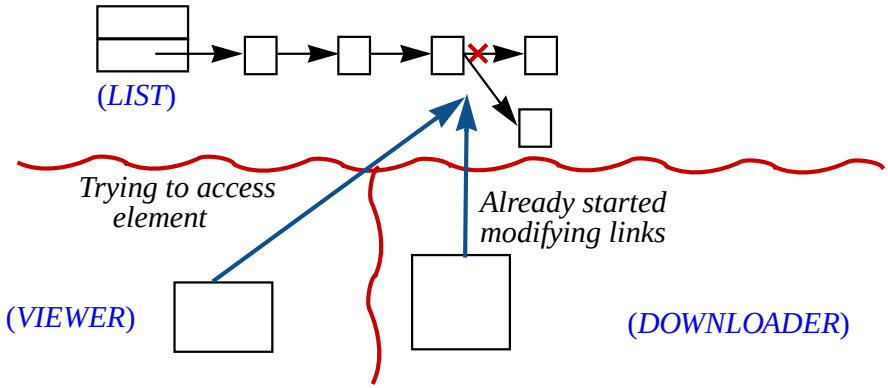
Here the non-determinism is good since it follows from the use of random values for the simulation. But non-deterministic does not mean arbitrary. The output shown here, and any variant that you get, will satisfy the following two properties:

- The download messages ("Adding message: …") appear in the full order of message numbers: 1, 2, 3, …
- No message is viewed before it has been downloaded.

Violating either of these properties would be an example of "bad" non-determinism.

6 CONTROLLING MULTIPLE ACCESS TO SHARED RESOURCES

As illustrated by the preceding example, concurrent applications typically start several threads of control but still need to access common resources, here the message list. In other approaches to concurrency, you need to add synchronization operations, to make sure that the various threads do not step on each other's toes. The technical term for such mix-ups is *race condition* (or "data race"). Here, for example, you might worry that the viewer will try to access the message list, through *i_th*, while the downloader is not finished adding an element to it, through *extend*. Indeed, look up the text of both of these routines in the library class *LINKED_LIST*: both need to traverse the list, but *extend* must modify the links between elements to add an element. If the execution were permitted to start *i_th* at some intermediate stage of that process, it would either produce an incorrect result or crash:



***Incorrect concurrent access* (*not in SCOOP*)**

With SCOOP such race conditions cannot happen because **a routine always has exclusive access to objects represented by its separate arguments**. Here the routines

```
download_one (ml: separate LIST [STRING])        -- In DOWNLOADER
view_one (ml: separate LIST [STRING])            -- In VIEWER
```

both guarantee exclusive access to the *ml* argument, the message list, in calls

```
download (messages)                              -- In DOWNLOADER
view_one (messages)                              -- In VIEWER
```

This guarantee — *no race conditions, period* — is a major benefit of SCOOP.

7 SIMULTANEOUS RESERVATION

Sometimes you will need exclusive access to more than one resource. In SCOOP you get it easily, as a generalization of the just seen mechanism of passing the corresponding separate objects as arguments.

Consider a "merger" object that takes two email clients and adds to the first client's message list all the messages from the second one. The routine may read

In class MERGER.

```
merge ( c1, c2: separate CLIENT )
    do
        merge_lists (c1.messages, c2.messages)
    end
merge_lists ( l1, l2: separate LIST [STRING] )
    do
        across l2 as e loop l1.extend (e.item) end
        l2.wipe_out
    end
```

(The implementation of *merge_lists* relies on list features from EiffelBase: *extends* add an element at the end of a list, and *wipe_out* removes all elements.)

We can assign the merger object its own region and processor (by now you know how to do this: just declare it **separate** and on creation it will spawn a new region and the associated processor). The *merge* operation will then run concurrently with all the other objects of the example, such as the downloader and the viewer. For the duration of its execution, it must have exclusive access to both client objects; otherwise you might get a race condition, inconsistent results and an botched data structure. The SCOOP rule meets this requirement: a call

```
merge (client_a, client_b)
```

gets exclusive access to **all** the separate arguments. The implementation of *merge* again relies on this rule, since it calls *merge_lists* with two separate arguments representing the clients' message lists, getting exclusive access to both of these lists.

To guarantee this exclusive access the execution may have to wait until their processors are available. With other concurrency mechanisms you might have to program this "wait on multiple resources" operation manually; it is hard to get it right. With SCOOP you get the multiple wait automatically. More generally, SCOOP transfers the responsibility for some of the most difficult issues in concurrent programming from the programmer to the compiler and runtime.

8 CONDITIONAL WAIT

Next we are going to learn how to wait for a condition in another region. More precisely we want to wait *smartly*. If we remove that requirement an easy solution may work: keep testing for the condition until it holds. This strategy is known as "busy wait" but in many cases it really is *silly* wait since it is wasteful of resources — and in fact it may not even be correct, as we will see shortly. Instead, SCOOP lets you specify an operation to be executed on a separate object *as soon as* a condition on that object becomes true, guaranteeing correctness and avoiding busy wait.

As an example, let us introduce a new separate object in our email system, of type *MOVER*. The mover object takes action whenever the size of the mailbox reaches a certain value *Max* (say 10,000 messages) and leaves only the last *Min* (say 1,000) in the mailbox. In practice it should archive the removed messages but we will ignore that part. The corresponding class can be as follows

```
class MOVER create make feature
    messages: separate LIST [STRING]-- Handled, as before, by the mail client
    make (c: separate CLIENT)
        … Set messages to c.messages (see DOWNLOADER, VIEWER, PLAYER) …
    live
            -- Keep watching for client's mailbox to reach Max messages,
            -- and when it does, remove all messages except last Min ones.
        do
            from until is_over loop move end
        end
    move
            -- When client's mailbox reaches size Max, trim it to Min.
        do
            "Wait until is_ready (messages)"
            trim (messages)
        end
    is_ready (ml: separate LIST [STRING])
            -- Has the size of ml reached Max messages?
        do Result := (ml.count >= Max) end
    trim (ml: separate LIST [STRING])
            -- Remove from ml all messages except last Min ones.
        do
            across 1 |..| Min as i loop
                ml [i.item] := ml [ml.count – Min + i.item]
            end
            ml.remove_tail (Min)        -- Cuts off list at position Min.
        end
    is_over: BOOLEAN            -- Should we stop moving? See section 13.
end
```

Warning: incorrect preliminary version. See the correct one below, page 14.

The actual moving is done by *trim*. Repeatedly, *move* waits for the size to reach *Max*, by testing for *is_ready*, and when that happens it executes *trim*.

In *move*, the waiting is — temporarily — represented by pseudocode, "Wait until *is_ready* (*messages*)". A naïve way to implement this pseudocode is

```
-- Warning: for discussion only, don't program like this!
from until is_ready (s) loop wait_a_little end
```

for *s* of some type *S*, where *wait_a_little* waits for some preset time. This solution uses busy wait, but the waste of CPU cycles is the least of its problems. The solution is in fact incorrect because of a typical concurrency error: between the time *is_ready* (*messages*) tests positive and the time we take advantage of it to call *trim* (*messages*), another client may have changed the *messages* list. This is precisely the kind of tricky errors that arise in pre-SCOOP approaches. Tricky because the second call generally comes so quickly after the first that in most cases things will work well; but every once in a while, depending on inputs and on relative execution speeds, a wrong result will ensue. Non-determinism makes the problem next to impossible to reproduce and identify.

As guaranteed by the SCOOP rules, both *is_ready* and *trim* need to reserve the *messages* list, but the two operations need to run within the *same* reservation of that shared object. Another attempt, again naïve, would be to transfer the wait (implemented by the code above) to the beginning of *trim*'s implementation, to retain the reservation of *messages*:

Warning: *still an incorrect solution.*

```
trim (ml: separate LIST [STRING])
    do
        "Wait until is_ready (ml)"
        … The rest as above (across loop and call to remove_tail) …
    end
```

This solution gets rid of the concurrency conflict noted above, but only by creating a worse problem: because *trim* now starts by reserving the *messages* list, no one else, such as the downloader, can access or modify that list; so nothing will ever happen! The execution gets stuck. (Your CPU will show a lot of activity, since it keeps executing the busy wait, but there is no *useful* activity, defined as activity that could change the list and hence allow the real execution to proceed.)

The solution is to associate the condition with *trim*. For this purpose, SCOOP relies on *preconditions*, part of Eiffel's Design by Contract mechanism. The line of pseudocode (the attempted waiting) disappears; and so does the need for routine *is_ready*. The correct version of *trim* is:

```
trim (ml: separate LIST [STRING])
        -- When ml reaches Max messages, remove all except last Min ones.
    require
        ml.count >= Max
    do
        … As above (across loop and call to remove_tail) …
    end
end
```

The condition *ml.count >= Max* in the precondition is known as a **separate precondition clause** because it uses a call with a target, here *ml*, that is a separate argument of the routine. The effect of a separate precondition clause in SCOOP is to make the call wait until the condition holds. At that point it will proceed as usual, reserving all the separate arguments, here *ml*. This policy is exactly what we need.

How to program the wait is the responsibility of the SCOOP implementation, not the application programmer (you!). The implementation does not have to use busy wait: if it finds the condition initially not satisfied, it can free the object for use by other processors, such as the downloader's, then check back at appropriate moments until the condition becomes true and it can proceed with the body of *trim*.

Since we started with an incorrect version of class *MOVER* then modified it a few times, here is the actual version, consolidating for ease of reference the various elements introduced above. The class is significantly simpler since we no longer need the procedures *move* and *is_ready*:

Replaces version of page 12.

```
class MOVER create make feature
    messages: separate LIST [STRING]

    make (c: separate CLIENT)
        … Set messages to c.messages (see DOWNLOADER, VIEWER, PLAYER) …
    live
            -- Keep watching for client's mailbox to reach Max messages,
            -- and when it does, remove all messages except last Min ones.
        do
            from until is_over loop trim (messages) end
        end

    trim (ml: separate LIST [STRING])
            -- When ml reaches Max messages, remove all except last Min ones.
        require
            ml.count >= Max
        do
            … As above (across loop and call to remove_tail) …
        end
    is_over: BOOLEAN                -- Should we stop moving? See section 13.
end
```

9 THE BUM WRAP?

The example classes discussed above, *DOWNLOADER*, *VIEWER*, *MOVER*, all perform operations on the client's message list, to which they have access through a separate attribute

```
messages: separate LIST [STRING]
```

You might have expected that they manipulate that list directly, as in

```
messages.some_list_feature          [1]     -- Invalid, see below.
```

(for example *messages.extend* (*latest*) in procedure *download_one* of class *DOWNLOADER*). But that is not what they do. Instead, the code achieves the required effect through calls of the form *r* (*messages*), for a routine *r* with an argument *ml*: **separate** *LIST* [*STRING*], where the body of *r* executes

```
ml.some_list_feature                [2]
```

Obviously this form does what [1] was intended to do, but at the cost (for the programmer) of extra coding, since you must write a wrapper routine *r* to which you pass the separate attribute as an argument.

This wrapping is not just a peculiarity of the examples seen so far. In a call *x.f* (…) where the target *x* is separate, *x* must by default be a formal argument of the enclosing routine. So you are not permitted to write the call [1] above, but must wrap it into a routine and pass *messages* as actual argument to that routine.

Why this rule? It guards against race conditions. It would be just too natural without the rule to write something like

```
messages.extend (v1)                -- Invalid.
messages.extend (v2)                -- Invalid.
```

believing that you are adding two values, *v1* and *v2*, next to each other at the end of the list. The trouble here is that *most of the time* you are. But in some execution out of a million someone else will capture messages between the two calls and you will end up with a transient bug (the jargon term is "Heisenbug") which is next to impossible to track and fix. The SCOOP convention forces you to get hold of separate objects before you can work on them, so if you do care about the elements ending up in consecutive positions you will use *add_two* (*v1*, *v2*), having declared

```
add_two (ml: LIST [STRING])
        -- Add v1 and v2, in this order, next to each other, at the end of ml.
    do
        ml.extend (v1)
        ml.extend (v2)
    end
```

While it is always necessary to reserve separate objects before you can work on them, there is a simplified notation to avoid introducing lots of small wrapper routines such as *add_two* in straightforward cases. It uses the keyword **separate** again and is called the **inline separate** instruction. You can get rid of the routine *add_two* if you replace the call *add_two* (*v1*, *v2*) by the separate inline instruction

```
separate messages as ml do
    ml.extend (v1)
    ml.extend (v2)
end
```

The syntax is self-explanatory; note the use of a local name, *ml*, to capture the separate object within the separate inline instruction. The separate inline instruction provides an implicit form of wrapping, not requiring the declaration of a routine, and has exactly the same effect as the explicit wrapping form presented above.

As with the routine call *merge_lists* (*c1.messages*, *c2.messages*), you can use an inline separate instruction to reserve two or more separate objects together. For example if this is the only call to *merge_lists*, you can get rid of this routine, replacing the call by

In class MOVER, page 11.

```
separate c1.messages as l1, c2.messages as l2 do )
        -- Rest of code identical to the body of merge_lists in the previous form:
        across l2 as e loop l1.extend (e.item) end
        l2.wipe_out
end
```

The separate inline instruction can also include a **require** clause to cause waiting on a separate precondition. For example in the *MOVER* class you can entirely avoid the *trim* routine by writing *move* as

In class MOVER. See the original trim on page 12.

```
move
        -- When client's mailbox reaches size Max, trim it to Min.
    do
        separate messages as ml require
            ml.count >= Max
        do      -- The rest as in the earlier trim:
            across 1 |..| Min as i loop ml [i.item] := ml [ml.count – Min + i.item] end
            ml.remove_tail (Min)
        end
    end
```

The fundamental rule remains: you may only call a routine on a separate target if you have reserved the corresponding object, by either passing it as argument to a wrapper routine or by using an inline separate instruction.

10 THE ORDER PRESERVATION RULE

With concurrency comes non-determinism, but non-deterministic does not mean random or arbitrary. SCOOP enforces time-ordering constraints that enable you to reason about the execution of your programs in a way that is not so different from reasoning about sequential programs.

We come back to our little playground for trying out our email mechanisms: class *PLAYER*. The "play" features that follow are in that class and let us experiment with properties of concurrent computation. We already had *play1*, which called *live_both*, but we start again with something much simpler:

PLAYER was on page 8.

play1 was on page 8.

```
play2
        -- Download two messages.
    do
        separate downloader as d do
            d.download_one (messages)
            d.download_one (messages)
        end
    end
```

Remember that in our simulated downloader *download_one* prints out a message indicating the message it is downloading. So we may expect an execution of *play2*, starting from scratch, to produce the following output:

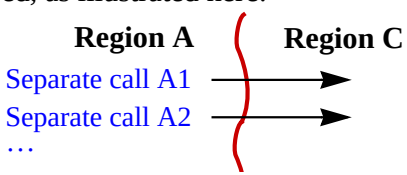
download_one was on page 3.

```
Adding message: Message 1
Adding message: Message 2
```

Run it; this is indeed what you will get. No visible non-determinism here: you can try any number of times, in any environment and will get the same result, with the two messages in order. Adding delays (*wait* (*t*) for some *t*) in *play2*, *download_one* or both does not affect this property.

SCOOP provides an ordering guarantee: in this example, the order of execution of the two routines is the same as if we had a sequential (rather than concurrent) system; it is the order in which the algorithm of *play2* executes them.

This order preservation rule states that **if two separate calls have the same originating region and the same target region, the order of execution of the routines' bodies is the same as the order of the calls**. Here the originating region is the player's region, and the target region is the downloader's region. Their order will be preserved, as illustrated here:



Between given regions, call order preserved

Now let us see what happens when we change regions. Try this:

```
play3
        -- Download a message, view a message.
    do
        separate downloader as d, viewer as v do
            d.download_one (messages)
            v.view_one (messages)
        end
    end
```

Remember that *view_one* prints a message, chosen at random from the client's message list if not empty, but does nothing if the list is empty. If you run *play3* (from scratch) several times, you might get, in some executions

view_one was on page 8.

```
Adding message: Message 1                    "Two-line output"
Viewing message: Message 1
```

where the first line comes from *download_one* and the second from *view_one*. But in other executions you might also get

```
Adding message: Message 1                    "One-line output"
```

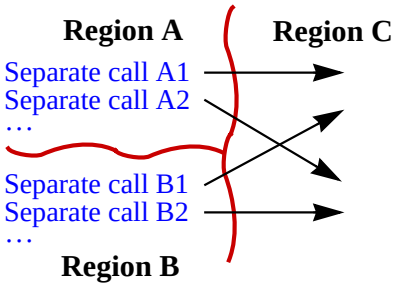
that is, just the first line! We call these two results the "two-line" and "one-line" outputs. You get one-line if the execution of *view_one* comes so fast on the heels of the call *v.view_one* (*messages*) that *download_one* has not had the time to start its execution yet. (If you do not see yet how this can happen, we will study the detailed timeline in the next section.)

Because computers are so fast these days it may be that on your system you never get the one-line output over repeated executions. To be almost sure to get it, add a line at the beginning of the text of *download_one*, in class *DOWNLOADER*:

```
wait (Delay)
```

where *Delay* is a time in milliseconds. For large *Delay*, say 1000, you should get the one-line output, because the call *v.view_one* will come much faster. For *Delay* = 0 you will probably get the two-line output. In-between, for small values, you may get different values in successive executions. In sequential computation this behavior would not make sense: whatever delays we introduce, we first execute *d.download_one*, and only then *v.view_one*; so the second operation will always find a non-empty list and print the two-line output.

What happened? We have three regions involved (player, downloader, viewer), and each has its own processor with its own timing. There is no guarantee of order preservation in such cases.



Between different regions, call order not necessarily

(In the example there is only one A call, to *download_one*, and one B call, to *view_one*.) The processor of region C will execute the routines of the A calls in order, and the routines of the B calls in order, but you cannot count on any guarantee between the As and the Bs.

This behavior is of the "good non-determinism" kind: ensuring any order constraint between more two or more calling processors, or two or more called processors, would require the execution of concurrent systems to rely on a global clock; that assumption would kill performance (since processors would spend their time resynchronizing), and is unrealistic anyway in distributed systems.

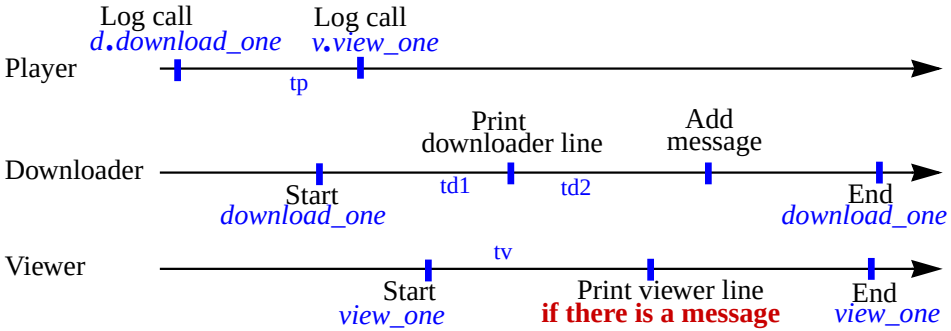
11 SEPARATE CALLS TO COMMANDS ARE BY DEFAULT ASYNCHRONOUS

Let us take a closer look at what happens in the execution of the two calls in *play3*

```
d.download_one (messages)
v.view_one (messages)
```

where *d* and *v* are both separate, and attached to objects in different regions ("**A**" and "**B**" in the last figure, both using the same client region "**C**" for the execution of the respective routines *download_one* and *view_one*).

Here is a possible timeline for the processors involved:



Processor timelines

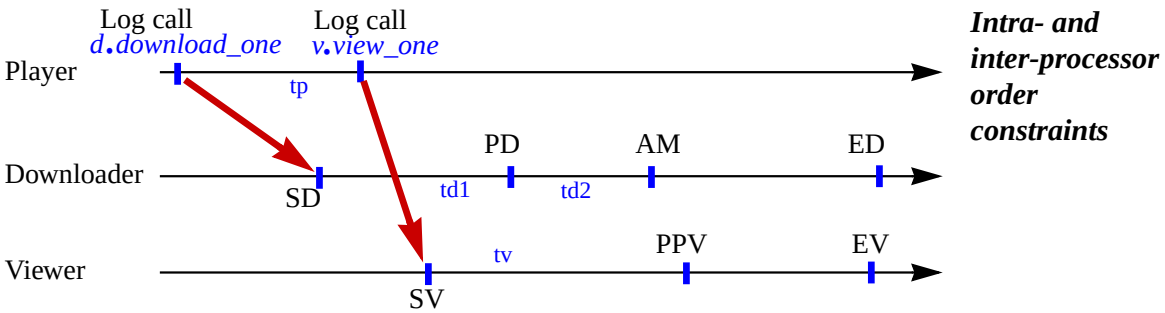
Following each of the timelines in turn:

- The player logs the call to the downloader, then the call to the viewer.

- The downloader, some time after the call to *download_one* has been logged, executes the body of that routine. The execution consists of printing a control line then adding a message to the list.
- The viewer, some time after the call to *view_one* has been logged, executes the body of that routine. The execution consists of finding out if there is a message in the list and *if so* to print it, otherwise to print nothing.

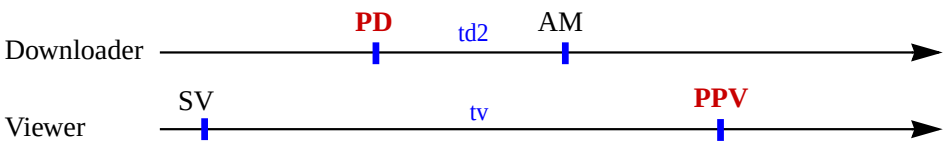
In what order will the two print instructions (one of them a potential print) occur? The preceding figure shows one possible order, but it is not the only possible one. The only timing guarantees are the following:

- Instructions in a single processor are executed in sequence (remember, a processor is sequential, and we get concurrency only by having several processors). This order guarantee corresponds to horizontal arrows in the figures above and below.
- In a separate call *x.r* (…), the logging of the call by the originating processor comes before the application of *r* by the target processor. This is a *causality* guarantee, represented in the figure below by the two red cross-processor lines.
- In addition, the order preservation rule ensures that *between two given regions* the order of routine body executions follows the order of the calls. We had that behavior in the first example, *play2*, but it does not occur in the current one.



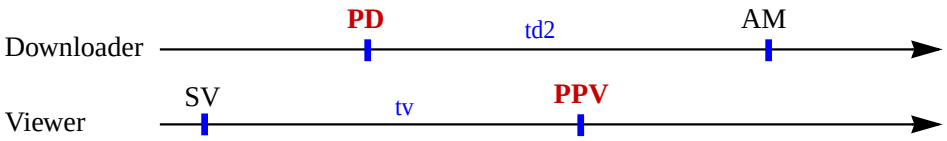
The figure is the same as the previous one, with all the order constraints represented. (Straightforward abbreviations: SD for Start Download, PD for Print Downloader line, AM for Add Message, ED for End Download, SV for Start Viewer, PPV for Potentially Print Viewer line, EV for End Viewer.) The *only* timing guarantees are the three horizontal arrows, one per processor, and the two cross-processor red arrows. The relative timing of the two critical operations, "Add message" in the downloader and "Print line if there is a message" in the viewer, depends on the speed parameters, shown in the figure as tp, td1, td2 and tv: instruction execution times in the player, downloader and viewer.

As a consequence, all of the following scenarios are possible. If PPV occurs after AM, we get two lines of output:



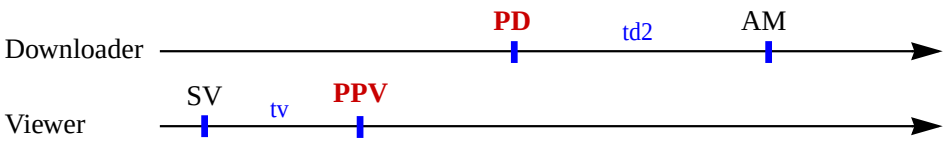
A two-line scenario

If that timing is reversed, however, we get just the downloader's line:



A one-line scenario

The relative order of PD and PPV does not matter, only the order of AM and PPV, so that the following ordering yields the same result as the previous one:



Another one-line scenario

You can play with the values of d2 and tv to force one of the two behaviors, by adding a *wait* (*Delay*) call in the body of *download_one* (in *DOWNLOADER*) or *view_one* (in *VIEWER*), where *Delay* is a time, for example1/10th of a second, orders of magnitude greater than computer-level execution times. Try it:

- If you add the call after the *print* instruction in *download*, you are making td2 very large, and are almost sure to get the one-line output.
- If you add it instead at the beginning of *view_one*, you are making tv very large, and are almost sure to get the two-line output.

In both cases the result is only "almost" sure: it is still in principle non-deterministic, but would only change under extreme differences between the speeds of the processors involved, more likely to occur on a network than on a single machine.

These examples of execution timeline highlight the asynchronous nature, by default, of procedure calls in SCOOP. In sequential programming, calls are always synchronous, in the sense that when you execute

```
some_object.some_routine (some_arguments)
other_instruction
```

you know that *other_instruction* will not start until the body of *some_routine* has run all the way to the end. This behavior is known as synchronous. No other is possible since in sequential programming there is only one processor, and it is busy executing *some_routine*. But in a concurrent setting if *some_object* resides in another region, which may have its own processor, the original processor — the

one executing the code above — can proceed on its own with *other_instructions*, independently of the other processor's handling of *some_routine's* body. The behavior in this case is said to be asynchronous.

Asynchrony is what makes a program concurrent. The rule is that **a procedure call on a separate target are by default executed asynchronously**. ("By default" because we will see a way to change that policy, but the behavior just described is the normal one and except for some advanced uses you should rely on it.)

A practical consequence, visible in the above timelines, is that for a separate call we need to distinguish between **feature call** and **feature application**:

- For the calling processor, executing an asynchronous call, such as *d*.*download_one* (*messages*) or *v*.*view_one* (*messages*) above, simply means "logging" the call: registering the information, so that the target processor will be notified. This operation is the feature *call*.
- The target processor must eventually execute the body of the routine, such as *download_one* for the downloader and *view_one* for the viewer. That execution can occur any time later, depending on processor speed and requests from other processors. It is the feature's *application*.

The distinction is irrelevant in sequential programming, and in SCOOP with non-separate calls, since feature application always follows feature call directly. But with concurrency the two are decoupled.

This concept also gives us a simpler and more precise version of the fundamental ordering rule of the previous section: **within one region or between two given regions**, **the order of feature applications is the order of feature calls**.

12 SEPARATE CALLS TO QUERIES ARE SYNCHRONOUS

For the next example remember that *count* in *DOWNLOADER* gives the number of downloaded messages; every application of *download_one* increases it by one. Add this routine to *PLAYER*:

```
play4
        -- Download two messages, find out how many were downloaded.
    local
        n: INTEGER
    do
        separate downloader as d do
            d.download_one (messages)
            d.download_one (messages)
            n := d.count
            print (n)
        end
    end
```

We can dispense with the local variable *n* if we replace the last two instructions by just one, *print* (*d*.*count*), but using the variable makes the details of the computation more visible. If you run this code you will get the output 2, as expected since every call to *download_one* increases *count* by one and the order preservation rule implies that the calls to *download_one* are applied before the evaluation of *count*. The rule is applicable here since the two processors involved are the same in all three cases.

With asynchrony, however, we might wonder whether we are guaranteed to get any result at all: if the call *d*.*count* were asynchronous, there would be no way to be sure that it will be finished when we assign its result to *n*. So we might be trying to print an undefined value.

You need not fear such a scenario: the call *d*.*count*, unlike the calls to *download_one*, is synchronous, meaning that the execution of the code in *play4*, in particular the assignment to *n*, will not continue until *download_one* has completed its application. The rule is that **separate calls to a query are executed synchronously**. Remember that a call *x*.*f* (…) is separate if its target *x* is separate. A feature *f* is a *query* if returns a result, meaning that it is either:

- An attribute, such as *count* here, describing a field present in every instance of the class.
- A function, computing the result through some algorithm.

As always in Eiffel, attributes and functions have the same effect when viewed from outside of their class. Try out this property by adding to *DOWNLOADER* the function

```
computed_count: INTEGER
            -- Number of messages in client's message list.
    do
        separate messages as m do
            Result := m.count
        end
    end
```

and to *PLAYER* a routine *play5* identical to *play4* except for the call *d*.*computed_count* replacing *d*.*count*. In other words, instead of relying on the downloader object's *count* field, incremented on each download, we compute the number of messages each time from scratch, as the length of the message list. The result does not change: the assignment to *n* cannot proceed until the execution of *computed_count* finishes. You can also experiment by adding delays to *play4* or *computed_count* and see that the result does not change.

The converse notion of "query" is "command", a synonym for "procedure": a feature that does not return a result. So the SCOOP synchrony policy is that, after logging of a call to a feature on a target handled by a different processor:

- If the feature is a command, its application, by default, proceeds asynchronously. This means that the calling processor can continue with the execution of instructions following the call; the target processor will execute the body of the command some time later.

- If the feature is a query, its application proceeds synchronously. This means that the calling processor will not continue with the execution of other instructions until the target processor has executed the body of the query, and returned the corresponding result.

13 THIRD-PARTY CONTROL

Instances of classes *DOWNLOADER*, *VIEWER* and *MOVER* are "active" objects in the sense that they have their own scenario, represented by the loop in their respective *live* procedures. The exit condition in all cases is called *is_over*.

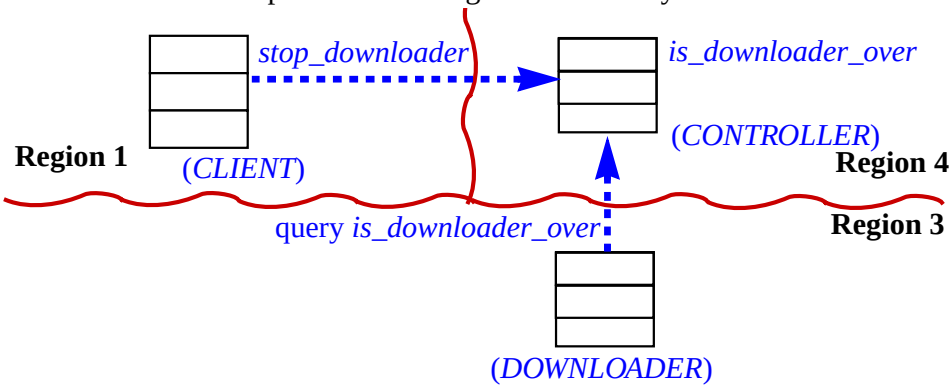
live procedures: pages 7, 8 and 14.

How can another object, such as the email client, tell one of these objects to stop its *live*? The declarations given above suggested that *is_over* is a boolean attribute. An initial idea for allowing clients to request termination is to include in the corresponding class, for example *DOWNLOADER*, a procedure to set that attribute:

```
stop
        -- Stop operation.
    do is_over := True end
```

This approach does not work, however: while the downloader is busy executing *live* its processor can do nothing else: whatever frantic attempts the client makes to call *downloader*.*stop* will have no effect.

A correct solution, using a SCOOP pattern known as *Third-Party Control*, relies on a third "controller" processor serving as intermediary:



Sending signals through a third-party controller

In both *CLIENT* and *DOWNLOADER*, we declare

```
controller: separate CONTROLLER
```

Then *is_over* in *DOWNLOADER* should no longer be an attribute but a function:

```
is_over: BOOLEAN
        -- Has operation termination been requested?
    do
        separate controller as c do
            Result := c.is_downloader_over
        end
    end
```

CONTROLLER, for its part, has the corresponding attribute and procedure:

```
is_downloader_over: BOOLEAN          -- Should downloader stop operation?
stop_downloader
        -- Record request to stop downloader operation
    do is_downloader_over := True end
```

The names explicitly refer to the downloader, so that a single controller can let the client stop other objects (with features *is_viewer_over*, *stop_viewer* and so on).

With this technique the client requests downloader termination through a call *controller.stop_downloader*; the next iteration of the downloader's *live* will query *is_downloader_over* and find out that it is now true. (Remember that all boolean attributes are initialized to **False** by default.)

The Third-Party Control pattern takes advantage of asynchrony to allow objects to pursue their own scenarios — to be "active" — while opening themselves up to interaction with other processors at defined times.

Another example where the pattern provides an elegant solution is a situation where a certain client user interface, for example in a browser, has started downloading a file and wants to display a progress bar, but not be stuck waiting for the download to proceed. Once the download has started, the client can engage in whatever other operations it wishes; every once in a while, it queries a third-party controller, and as a result updates the progress bar in the UI. For its part, the downloading operation notifies the controller whenever it has progressed by some preset amount such as 1%.

14 PASSIVE REGIONS

Every object is in a region. Every processor has its own region, where it handles all $x.f\,(\ldots)$ calls where x denotes an object in the region.

The reverse property does not have to hold: not every region has its own processor. Most do, but it is possible to define *passive* regions, in which calls will be handled by the calling processors.

You specify that a region is passive in the creation of the region's first object. Remember that regions get created through creation instructions of the form

```
create sep.p (...)
```

where the target *sep* is separate. (*p* is the optional creation procedure, which in all examples of this tutorial has the name *make*.) This instruction puts *sep* in the newly created region and starts the associated processor. The region in this case is "active": it has its own processor. You can, however, use

```
create <NONE> sep.p (...)
```

to get only a new region, and no new processor. In that case the new region is passive: any processor executing a separate call $x.f\,(\ldots)$ on an object in the region will carry out itself the execution of *f*.

Syntax note: if you specify an explicit type for the new object, it comes after the *<NONE>* mark, as in **create** *<NONE>* *{SOME_TYPE}* *sep.p* (...).

Using a passive region does not change the access rules: a separate feature application always has exclusive access to all separate arguments, and if there are any separate preconditions it will wait until they hold.

What does change is the possibility of asynchrony. When the target of a call is in a passive region, only one processor is in charge of both the call and the application, so the call is synchronous. The default policy of asynchronous calls for commands no longer applies (that is why it was only the "default" policy): all calls, whether to queries or commands, are now synchronous.

With passive regions we have the final picture of which calls are synchronous and asynchronous (empty entries indicate an irrelevant criterion):

Kind of call	Kind of feature	Target region	Behavior
Unqualified *f* (…)			Synchronous
Qualified *x*.*f* (…)	Query (attribute or function)		Synchronous
	Command (procedure)	Passive	Synchronous
		Active	**Asynchronous**

Calls are synchronous except in the bottom-right case, qualified command call to a target in an active region.

When should you use passive regions? The first part of the answer is: usually you should not! Most regions should be active, since the very aim of concurrent programming is to take advantage of the availability of several processors, each of which can proceed on its own. For that you need asynchrony.

Passive regions are an **optimization** mechanism. You should not bother with them until you have a running system and want to increase its performance.

Asynchrony is generally good for performance (as well as correctness). But there can be too much of it. An asynchronous call implies communication and other overhead: the target processor must register calls and manage the call queue. Making a region passive can help if all objects in it satisfy two properties:

- They represent *shared resources*, providing services to clients, but do not themselves have their own agenda (their own process). For example if in a class you find the need for a procedure such as *live* in *DOWNLOADER* or *VIEWER*, representing an object's independent lifecyle, instances of the class should be in active (non-passive) regions.
- A passive region makes sense if clients only need to reserve the corresponding resources for *short periods*. Otherwise it will hurt performance rather than help it, since it will prevent the client processors from proceeding asynchronously with other tasks.

live procedures: pages 7 and 8.

In the running example of this tutorial:

- It would be a mistake to use passive regions for the downloader, viewer and mover. These objects and the associated data structures need their own processors to give us the concurrency expected of an email system.
- The client, on the other hand, makes a data structure, the message list, accessible to other processors. It does not need a thread of control of its own and can be hosted in a passive region. We accordingly specified *<NONE>* in the corresponding object creation.

The passive creation was on page 8.

15 TYPE RULES AND ARGUMENT PASSING

The final step to mastering SCOOP is to understand a few rules that govern how you can combine separate and non-separate elements in the program text.

Add the following to class *PLAYER*:

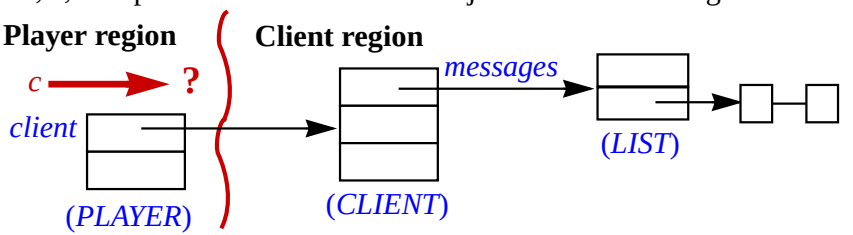
```
play6
    local
        c: CLIENT
    do
        c := client
        -- Operations on c to be added (see below)
    end
```

Invalid code, see explanation below.

Remember that *client*, a reference to the email client object, is declared of type **separate** *CLIENT*. Compile and run the resulting system. Rather, try to: you will not get to execute anything but will receive a compile-time error message:

```
VJAR: Source of assignment is not compatible with target.
```

The reason should be clear. You are trying to use a local variable declared as non-separate, *c*, to represent a reference to an object in a different region:



Non-separate references should not cross region boundaries

If such a reference must cross region boundaries, you must declare the corresponding variable as **separate**. Otherwise uses of that variable would violate concurrency rules and create havoc. For example, the body of the routine *play6* could include (where the comment above says "operations on *c*"):

```
c.messages.extend ("ABC")
```

which tries to add a string at the end of the message list, but without any of the exclusive access properties that SCOOP guarantees for shared resources. Data races would follow, in the form of conflicting modifications to the list structure. Fortunately, SCOOP forbids such games.

A correct declaration for *c*, instead of just *CLIENT*, is **separate** *CLIENT*.

Without this requirement, assignments such as *c* := *client* would, at run-time, create "**traitors**". A traitor is a reference — in this case, *c* — that denotes a separate object but is not declared accordingly.

The SCOOP type rules guarantee that no execution will produce traitors. There are three key rules. We have just seen the first one: **if the source of an assignment is of a separate type, the target must be of a separate type too**.

The second rule is similar, but for argument passing rather than assignment. Add a routine

```
play7 (c: CLIENT)
    do end                          -- Empty body OK, the signature is enough
```

as well as

```
play8
    do play7 (client) end
```

The compiler will reject *play8* because it passes a separate expression, *client*, as actual argument to a routine which has a non-separate formal argument *c*. Were the call permitted, the body of *play7* (empty above) could include a non-protected call on a separate object, for example the same one as in the assignment example: *c.messages.extend* ("ABC"), creating the potential of race conditions.

The second type rule, then, is that **if a formal argument of a routine is of a separate type, the corresponding actual in any call should be of a separate type too**.

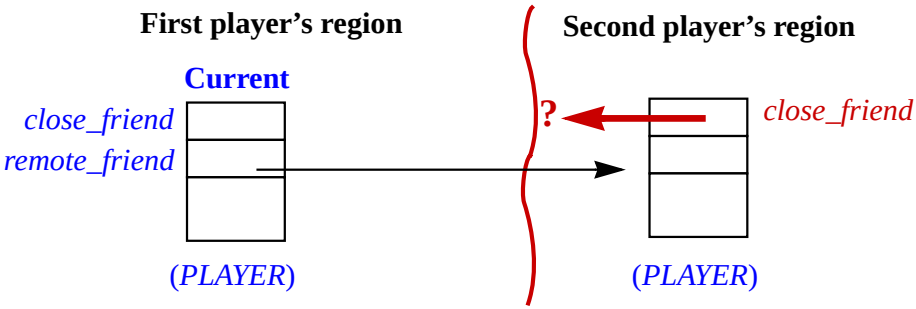
The third type rule reflects the third principal way one could try (sneakily) to create a traitor. To play, a player can have "friends", separate or not:

```
close_friend: PLAYER
remote_friend: separate PLAYER
set_close_friend (p: PLAYER)
        -- Make p this player's local (that is, non-separate) friend.
    do
        close_friend := p
    end
```

Now try this:

```
play9
    do
        create remote_friend.make
        remote_friend.set_close_friend (Current)
    end
```

The feature call, if we ever got to execute it, would create a traitor: we are passing ourselves (**Current** denotes the current object, like **this** in some other object-oriented languages) to an object in another region, which wants for *close_friend* an object in its own region. But, as illustrated below, **Current** is not in that region!

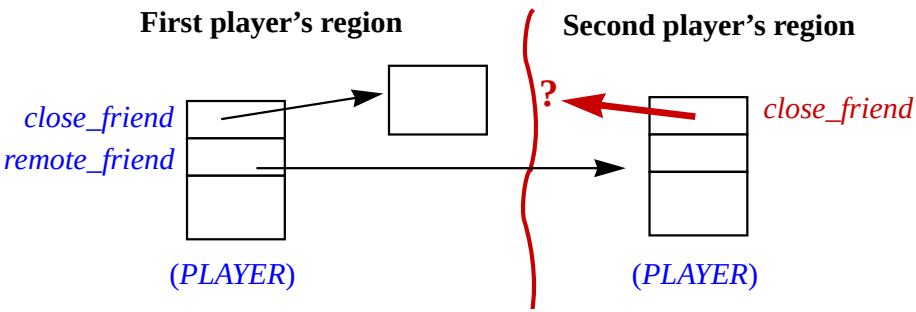


Should not pass oneself off as a separate object

Using **Current** is not critical to the example; any other object in the first player's region would cause the same problem, as if you replace the call in *play9* by:

```
remote_friend.set_close_friend (close_friend)
```

trying to make our remote friend use our own close friend as its close friend too. This is just as bad:



Should not pass off a local object as a separate object

These obvious cases of traitor creation are, however, not covered by the previous rules: the routine *set_close_friend* (*p*: *PLAYER*) has a non-separate formal argument, of type *PLAYER*, not **separate** *PLAYER*. That is how it should be since every player want a non-separate object as its "close" friend. The actual argument in the call to *set_close_friend*, whether **Current** or *local_player*, is also non-separate, so rule 2 lets it pass. The problem is that the call, *remote_friend*.*set_close_friend* (…), has a separate **target**, *remote_friend* (we say that it is a "separate call"), causing the actual argument to be separate *relatively to the target region* — "second player's region" in the figures — although the routine expects a non-separate reference.

The third rule addresses this case: **in a separate call, it is permitted to pass a reference as actual argument only if the corresponding formal argument is of a separate type.**

In the present example, the rule prohibits *any* separate call whatsoever to *set_close_friend*: calls must be either unqualified, as in *set_close_friend* (*a*), or qualified but with a non-separate target, as in *nonsep*.*set_close_friend* (*a*). In both cases, of course, *a* should be of a non-separate type. To support separate calls, *set_close_friend* should be changed to have a separate formal argument, although in this example the change does not make sense.

This third rule only applies to reference arguments. The overall behavior is the general one in Eiffel: the type of an expression is either a reference type or an expanded type. Expanded types (known in some other languages as value types) cover in particular basic types *BOOLEAN*, *CHARACTER*, *INTEGER* and *REAL*, and have copy semantics. So:

- If *a* is a reference to an object, what will be copied into the routine's formal argument in a call *f* (*a*) is that reference to that object; in SCOOP it may become a separate reference if the object belongs to a different region.
- If *a* is of an expanded type, its value is an object (possibly a very small object, such as a single word, for basic types such as *INTEGER*), and that object will be copied as part of the call. In SCOOP, the copy can take place across region boundaries.

As our last example, let us add to *CLIENT* a query returning the first message if any

```
first: STRING
        -- First message if any, otherwise empty.
    do
        Result := ""
        if not messages.is_empty then Result := messages [1] end
    end
```

and to *PLAYER* a procedure to print that first message from the client's list

```
play10
        -- Print first client message if any.
    do
        separate client as c do
            io.put_new_line (c.first )
        end
    end
```

Try compiling the system with this routine. Indeed it compiles (and runs). With the rules seen above, it should not! *STRING* is a reference type, since the value of a variable of type *STRING* is a reference to an object representing a string; the argument *c*.*first* is of type **separate** *STRING*, but the library routine *put_new_line* expects a plain *STRING*, so we are violating the second type rule.

In fact the routine *view_one* of *VIEWER* had the same problem, since it also prints a separate string.

view_one was on page 8.

One approach would be to make all routines that expect strings, such as *put_new_line*, work on potentially separate strings instead, but it would be heavy and inconvenient.

Sticking to *put_new_line* as it is, we can use an explicit solution by relying on the built-in string creation procedure *import* which, from a separate *STRING*, yields a non-separate string. Instead of *c*.*first*, the code of *play10* can pass to *put_new_line*, as actual argument, **create** {*STRING*}.*import* (*c*.*first*).

Because strings are so frequently used across region lines, class *STRING* declares *import* as a conversion procedure, taking advantage of Eiffel's conversion mechanism. As a consequence, you can use a **separate** *STRING* in any context that expects a plain *STRING*. A conversion will occur, importing a copy of the string to the desired region. This convention explains why *play10* as shown above, and *view_one* in *VIEWER*, are valid and run with the expected effect.

16 FURTHER READING

With this tutorial you have seen all of the key SCOOP concepts. There is considerable more information available, some of it comprehensive, some of it up to date. At the time of writing no document is both comprehensive and up to date, but we are working hard to fill this gap.

For the exact reference to the texts cited below see the documentation page of the SCOOP-based Concurrency Made Easy project at ETH Zurich: cme.ethz.ch/publications/.

The basic concepts and motivation behind SCOOP appear in Meyer's book *Object-Oriented Software Construction* (2nd edition).

For an informal but fairly detailed survey, see the SCOOP slides of the ETH "Concepts of Concurrent Computation" course.

Three ETH PhD theses have made major contributions to the SCOOP technology. Nienatltowski's 2007 thesis remains a prime reference on the concepts and many details, although some aspects of the model have changed since then. It contains in particular a more complete and systematic presentation of the type rules. Morandi's more recent thesis introduced the notion of passive regions (called "passive processors") and contains a detailed formal semantics. West's thesis, also from 2014, provides many insights and critical performance improvements.

SCOOP is part of the current implementation of EiffelStudio, which can be downloaded at eiffel.com. The SCOOP mechanisms are embedded in those of the Eiffel language, on which extensive information is available at eiffel.org.