Concurrent programming in SCOOP: a hands-on tutorial

Ever more often, users want programs to be concurrent. A concurrent email client, for example, lets you download new messages while you are reading earlier ones. The alternative is a *sequential* program, which does only one thing at a time: with a sequential email client, the download would wait while you read, then once downloading starts you would have 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. In particular, SCOOP entirely eliminate the worst of them, "**data races**" (read-write concurrency conflicts).

You can study the concepts in the SCOOP pages at eiffel., particularly eiffel.org/doc/solutions/Concurrency, and in the documents of the Concurrency Made Easy project at cme.ethz.ch. This tutorial is a hands-on presentation of the practice of SCOOP through a simple example, a bare-bones email client.

As you read the text you should edit and run the example. Download EiffelStudio for your platform at eiffel.org, install it, then download the code from eiffel.org/files/uploads/mail_client.zip. Unzip it to get three directories (folders): sequential, concurrent and scratchpad. The tutorial starts from the sequential version and makes it concurrent. At any time you can cheat by looking at the final version in concurrent. The idea is to leave the code in sequential and concurrent untouched, for reference; make your changes in scratchpad, which is originally identical to sequential. If at some point you want to restart from the original version, copy all the **.e** files (Eiffel class texts) back from sequential to scratchpad.

1 About the example

The sequential and concurrent versions differ only slightly; basically, we will obtain the concurrent version by adding a **separate** mark to the declarations of a few variables, to specify that certain objects are to be managed concurrently.

That's one of the fundamental properties of SCOOP: you can retain most of the classical (sequential) form of programming you know well, adding concurrency properties only where needed.

In both versions, sequential and concurrent, the system relies on six classes:

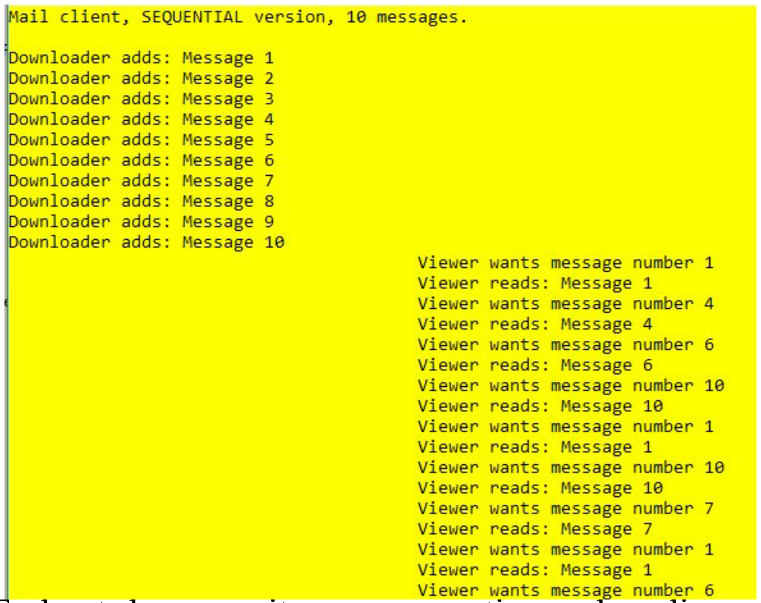
- *APPLICATION*: the small "root class" which starts execution by creating objects and calling the client.
- *CLIENT*: there will be one instance of this class, representing the email client which coordinates between the downloader and viewer.
- *DOWNLOADER*: also with just one instance, this class is in charge of downloading email messages. We don't really download them from a network, we just simulate the process by creating boring emails whose texts reads "Message 1", "Message 2" and so on.
- *VIEWER*: here too just one instance, a simulated email viewer which randomly selects a message number (like a human user who would decide to read one email or another from the list displayed by, say, Microsoft Outlook) and displays it. If the downloader has not produced the requested message yet, the viewer will have to wait.
- Auxiliary classes: *GENERATOR* for producing random messages and waiting times (between downloads or views) and *GLOBAL* for global information.

2 Running the code

We start by compiling and running both versions, to see how they behave. Then we will look at the sequential code (section 3), and in section 4 we will make it parallel.

Go to the sequential directory. Click the file *mail_client.ecf*, where *ecf* stands for Eiffel Control File. This starts EiffelStudio. (The screens in this tutorial were produced on Windows. On other systems the appearance may vary.) If this is the first time you are bringing up this project, you will be invited to compile it. Also, if this is the first time you are using EiffelStudio, it might do some bookkeeping, such as precompiling libraries, which take some time but won't have to be done again.

Before we look at the code, let's see its execution. Click the Run icon on the top icon bar, center-right. (Or just hit the F5 function key.) In the console window that comes up, on the left, the downloader will generate messages; on the right, the viewer will request and read messages:

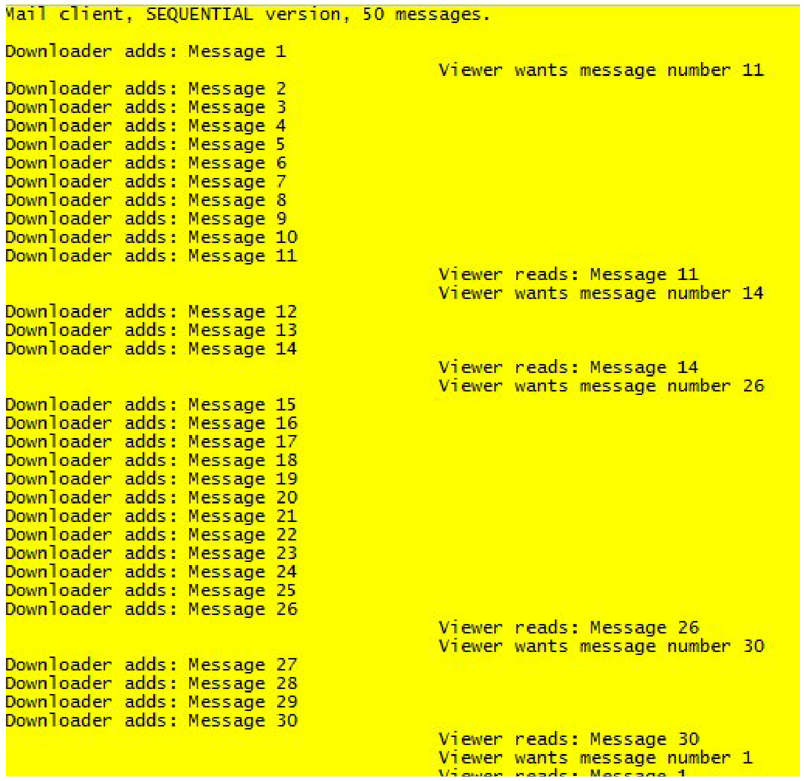


```
Mail client, SEQUENTIAL version, 10 messages.

Downloader adds: Message 1
Downloader adds: Message 2
Downloader adds: Message 3
Downloader adds: Message 4
Downloader adds: Message 5
Downloader adds: Message 6
Downloader adds: Message 7
Downloader adds: Message 8
Downloader adds: Message 9
Downloader adds: Message 10
                                        Viewer wants message number 1
                                        Viewer reads: Message 1
                                        Viewer wants message number 4
                                        Viewer reads: Message 4
                                        Viewer wants message number 6
                                        Viewer reads: Message 6
                                        Viewer wants message number 10
                                        Viewer reads: Message 10
                                        Viewer wants message number 1
                                        Viewer reads: Message 1
                                        Viewer wants message number 10
                                        Viewer reads: Message 10
                                        Viewer wants message number 7
                                        Viewer reads: Message 7
                                        Viewer wants message number 1
                                        Viewer reads: Message 1
                                        Viewer wants message number 6
```

Execution: sequential version

(End not shown — it goes on wanting and reading a few more messages.) Well, we said this is a sequential version! There is only one thread of control, so even though the viewer requests messages in a random order it must wait patiently until the downloader has produced *all* messages. (That's why we limited the number of messages to 10 here — no need to wait for more messages to see the point.)

What we want, of course, is to permit the viewer to view a message as soon as it is available. The concurrent version does that. To see it in action, go now to the concurrent directory. It has the same files (in fact we will see that the sequential and concurrent versions of *CLIENT*, *DOWNLOADER* and *VIEWER* differ by only a handful of small details). As before, click *mail_client.ecf*. A separate EiffelStudio window comes up. Compile the system, and run it



```
Mail client, SEQUENTIAL version, 50 messages.

Downloader adds: Message 1
                                        Viewer wants message number 11
Downloader adds: Message 2
Downloader adds: Message 3
Downloader adds: Message 4
Downloader adds: Message 5
Downloader adds: Message 6
Downloader adds: Message 7
Downloader adds: Message 8
Downloader adds: Message 9
Downloader adds: Message 10
Downloader adds: Message 11
                                        Viewer reads: Message 11
                                        Viewer wants message number 14
Downloader adds: Message 12
Downloader adds: Message 13
Downloader adds: Message 14
                                        Viewer reads: Message 14
                                        Viewer wants message number 26
Downloader adds: Message 15
Downloader adds: Message 16
Downloader adds: Message 17
Downloader adds: Message 18
Downloader adds: Message 19
Downloader adds: Message 20
Downloader adds: Message 21
Downloader adds: Message 22
Downloader adds: Message 23
Downloader adds: Message 24
Downloader adds: Message 25
Downloader adds: Message 26
                                        Viewer reads: Message 26
                                        Viewer wants message number 30
Downloader adds: Message 27
Downloader adds: Message 28
Downloader adds: Message 29
Downloader adds: Message 30
                                        Viewer reads: Message 30
                                        Viewer wants message number 1
                                        Viewer reads: Message 1
```

Execution: concurrent version

(Again the end is not shown. This version generates 50 messages: see the declaration of *Max_messages* in class *PARAMETERS*.) Because this version is concurrent the exact order of operations depends on scheduling and might be different on your system. But in any case the improvement over the previous version is clear: the downloader and viewer execute in parallel. In the execution shown here, the viewer first requests message 11, and displays it as soon as the downloader produces it; and so on for subsequent requests.

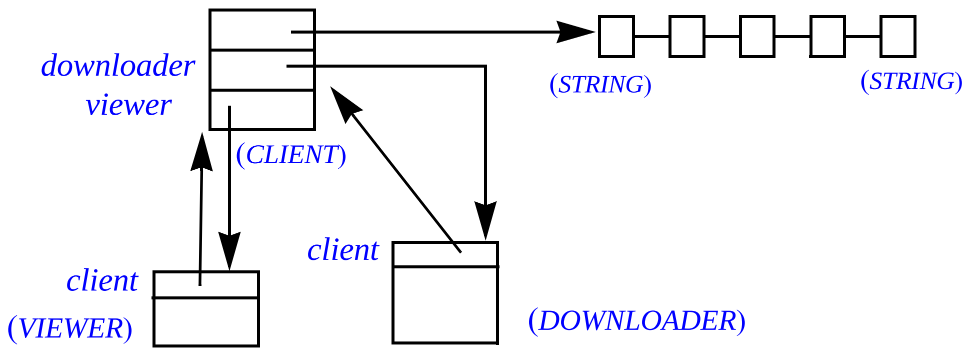
Now we are going to look at the code and see how SCOOP achieves this result.

3 LOOKING AT THE CODE

First, independently of concurrency properties, we must understand the structure and implementation of our little email client system. We look at the code in the sequential version (directory sequential); but the properties reviewed in this section will remain applicable to the concurrent version.

The data structure is simple, shown by the following illustration:

Basic data structure



The class names in parentheses indicate the type of each object. The *CLIENT* object maintains a list of strings, the email messages. It has references to a *DOWNLOADER* object and a *VIEWER* object, each of which knows about their client through the field client. That's it for the data structure.

To see the text of any class, select it in under *mail_client* in the tree view of the project on the right, or just type its name followed by Enter in the "Class" field. To view more than one class, you can use several tabs, as with a browser, with the same keyboard shortcuts (on Windows, Control-T for a new tab, and Control-Tab to alternate between tabs). Let's start with the root class, *APPLICATION*:

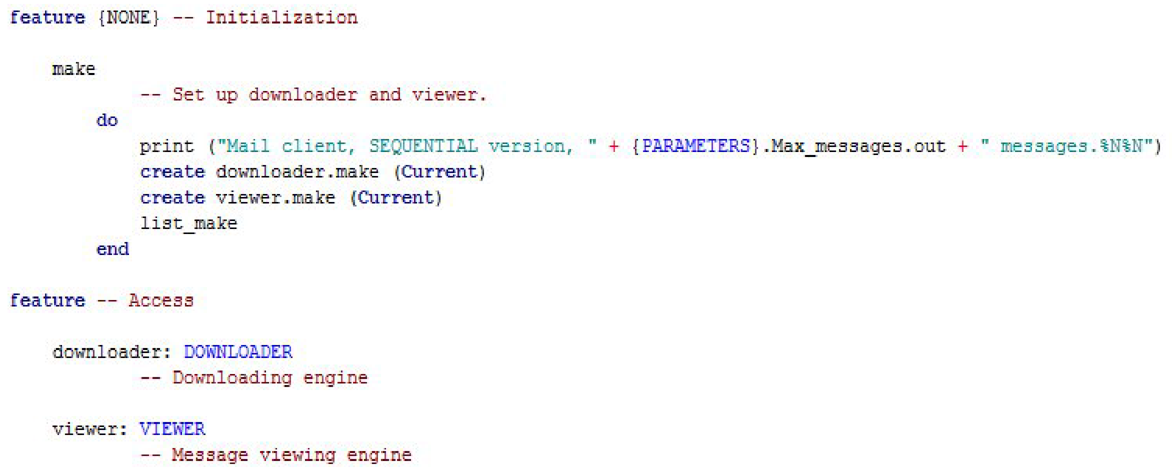
***In class** APPLICATION*

```
class
    APPLICATION

inherit
    ARGUMENTS

create
    make

feature {NONE} -- Initialization

    make
            -- Run application.
        local
            client: CLIENT
        do
            create client.make
            client.live
        end
```

The creation procedure (constructor) *make* creates a client object, calls its own *make* creation procedure, which sets up the objects; then it starts *live* which performs the actual execution, a sequential on in this version.

Go to class *CLIENT* to see the details. Scrolling down a bit (we'll look at the top of the class in a second) we see the creation procedure and two declarations:

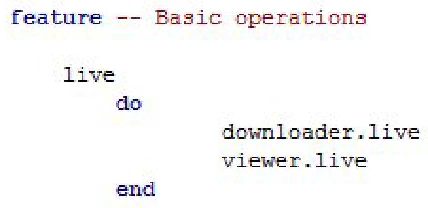
***In class** CLIENT*


```
feature {NONE} -- Initialization

    make
            -- Set up downloader and viewer.
        do
            print ("Mail client, SEQUENTIAL version, " + {PARAMETERS}.Max_messages.out + " messages.%N%N")
            create downloader.make (Current)
            create viewer.make (Current)
            list_make
        end

feature -- Access

    downloader: DOWNLOADER
            -- Downloading engine

    viewer: VIEWER
            -- Message viewing engine
```

The client declares the attributes representing the *DOWNLOADER* and *VIEWER* objects, and *make* creates the corresponding objects by calling their own creation procedures, also called *make*. (This is the standard convention, but you can call a creation procedure any way you like.) We pass **Current** to these procedures. **Current** is the current object ("this" in some other OO languages). The downloader and the viewer will need **Current** to know the client to which they are reporting.

The procedure *live* is simple: let the downloader and the viewer live, through their corresponding procedures:

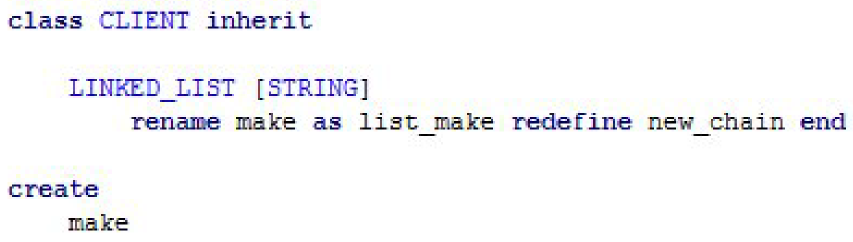
***In** CLIENT*


```
feature -- Basic operations

    live
        do
                downloader.live
                viewer.live
        end
```

Here we see the hitch in this sequential version: since we have only one thread of control, we can only let the downloader live its life first, then the viewer. When we go concurrent, we will start both in parallel, and let them produce and view messages (respectively) when they are ready to according to their own agendas.

Now scroll back to the beginning of the class:

***In** CLIENT*

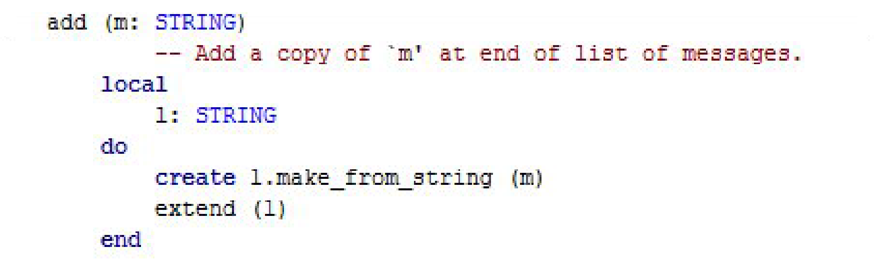

```
class CLIENT inherit

    LINKED_LIST [STRING]
        rename make as list_make redefine new_chain end

create
    make
```

You see that *CLIENT* inherits from *LINKED_LIST* [*STRING*]. That's because a client will maintain a list of messages. Using object-oriented techniques we simply declare that any *CLIENT* is a list. This is a perfectly legitimate use of inheritance, common in Eiffel style. Not everyone likes to use inheritance this way; if you prefer, you could also skip the inheritance and let *CLIENT* declare a distinct list of messages through an attribute:

messages: LINKED_LIST [*STRING*]

With this other approach the procedure *add*, which adds a message to the list, would be different. Here (scroll down a bit) it reads:

***In** CLIENT*



```
add (m: STRING)
        -- Add a copy of `m' at end of list of messages.
    local
        l: STRING
    do
        create l.make_from_string (m)
        extend (l)
    end
```

where the library procedure *extend* adds an element to the end of a list. If you did not use inheritance but a *messages* attributes, the last instruction would be

messages.*extend* (*l*)

but in the version as written above we just apply *extend* to the current *CLIENT* object which, through inheritance, is a list.

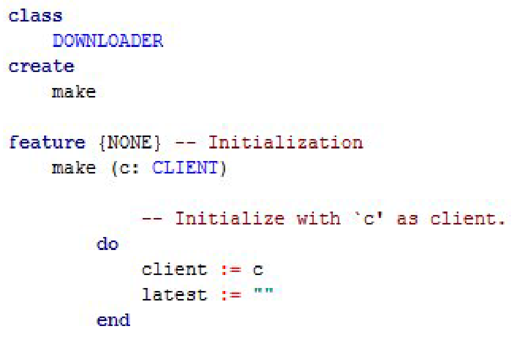
In this sequential version the procedure *extend* is more complicated than needed. In fact we could do without the local variable *l* and the creation. The whole procedure body (the **do** part) could just read

extend (*m*)

We made it a bit more wordy to prepare for the concurrent version. The creation procedure *make_from_string*, in the library class *STRING*, creates a string by copying another one.

That's it for *CLIENT* (you can ignore the *new_chain* implementation function at the end, a temporary need arising from a property of *LINKED_LIST* in EiffelStudio 18.11, not needed in future releases). Now let's look at *DOWNLOADER*. Its feature *client*: *CLIENT* is initialized in the creation procedure:

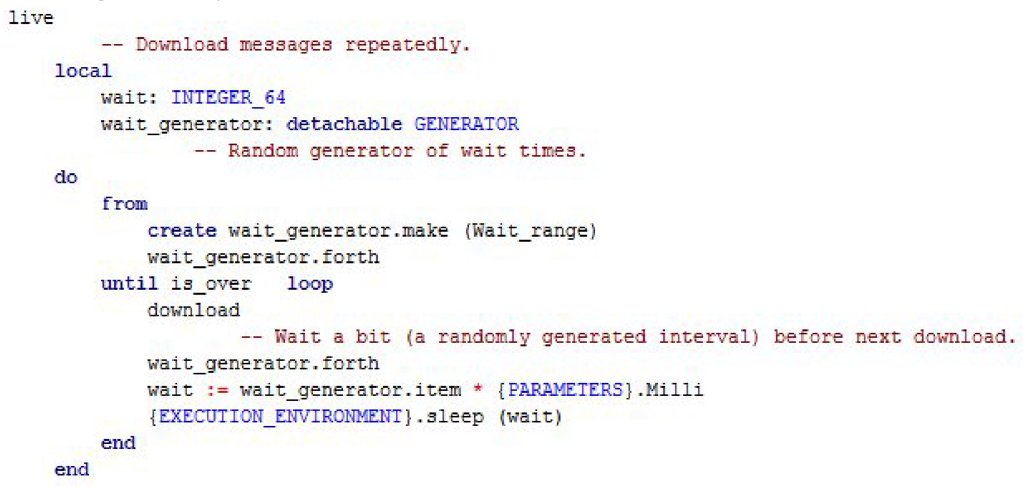
***In** DOWNLOADER*



```
class
    DOWNLOADER
create
    make

feature {NONE} -- Initialization
    make (c: CLIENT)

            -- Initialize with `c' as client.
        do
            client := c
            latest := ""
        end
```

By scrolling down you see how a downloader “lives”:

In *DOWNLOADER*



```
live
        -- Download messages repeatedly.
    local
        wait: INTEGER_64
        wait_generator: detachable GENERATOR
                -- Random generator of wait times.
    do
        from
            create wait_generator.make (Wait_range)
            wait_generator.forth
        until is_over   loop
            download
                    -- Wait a bit (a randomly generated interval) before next download.
            wait_generator.forth
            wait := wait_generator.item * {PARAMETERS}.Milli
            {EXECUTION_ENVIRONMENT}.sleep (wait)
        end
    end
```

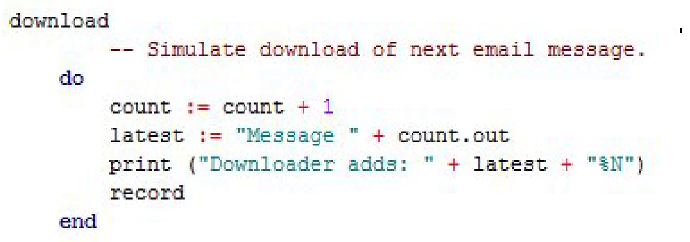
Looking at the body of the loop (after **loop**), we see that at each iteration the generator successively:

- Downloads a new message (we will next look at *download*).
- Computes a waiting time called *wait*.
- Waits for that duration, by calling *sleep (wait)*. The procedure *sleep* comes from the library class *EXECUTION_ENVIRONMENT*.

The waiting time, in the second step, is obtained through a random number generator called *wait_generator*. Such an object is like a little machine, with the operations forth which advances to the next step, producing a new number, and item, which gives that number. For the details you can look up the class *GENERATOR*, which relies on the library class *RANDOM*.

The download operation, used at every step of *live*, is simple:

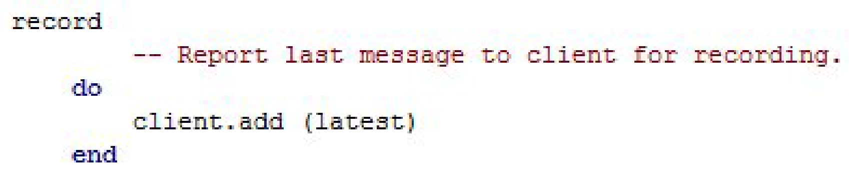
In *DOWNLOADER*



```
download
        -- Simulate download of next email message.
    do
        count := count + 1
        latest := "Message " + count.out
        print ("Downloader adds: " + latest + "%N")
        record
    end
```

It increments the message count by 1, produces a message *latest* including this count (+ on strings is concatenation, and *count*.*out* is the string representation of the number *count*), and tells the client to add this message at the end of the list (remember that the client *is* a list) through the procedure *add* seen above:

In *DOWNLOADER*



```
record
        -- Report last message to client for recording.
    do
        client.add (latest)
    end
```

The only difference with a real email system is that we make a message up (reading “*Message n*”) rather than downloading an actual email message from the network.

Finally, the viewer. It is based on similar conventions. Bring up class *VIEWER*. The structure is as with *DOWNLOADER*; scroll down to see the loop in *live*:

***In** VIEWER*

```
live
        -- Simulate user repeatedly choosing a message and reading it.
    local
        next: INTEGER
        wait: INTEGER_64
        wait_generator: GENERATOR
                -- Random generator of wait times.

        index_generator: GENERATOR
                -- Random generator of index of message to view.
    do
        from
            create index_generator.make ({PARAMETERS}.Max_messages)
            index_generator.forth
            create wait_generator.make (Wait_range)
            wait_generator.forth
        until is_over loop
                -- User randomly selects next message he wants to read.
            count := count + 1
            next := index_generator.item
            wants (next)
                -- User reads chosen message.
            view (i_th (client, next))
                -- Advance random message choice generator.
            index_generator.forth
            wait_generator.forth
            wait := wait_generator.item * {PARAMETERS}.Milli
            {EXECUTION_ENVIRONMENT}.sleep (wait)
        end
    end
```

At every iteration, the viewer picks a random number to select a message to view, then says it "wants" to view this message. Then it views it and waits a randomly generated time (using a different random generator). Here are "*wants*" and "*view*":

***In** VIEWER*

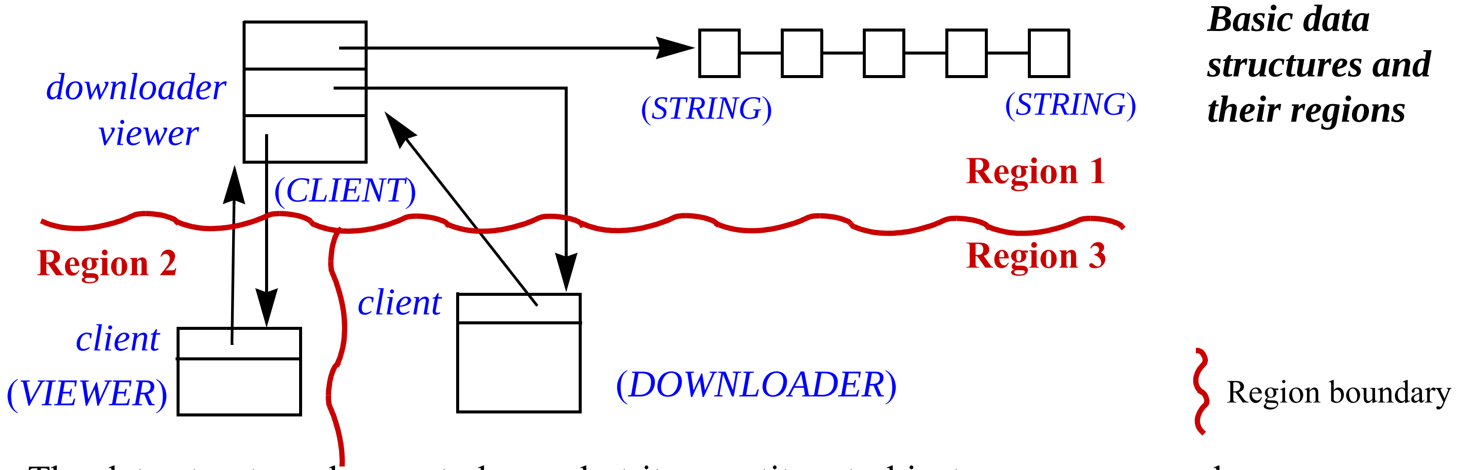
```
wants (i: INTEGER)
        -- Simulate user wanting to read message number `i'.
    do
        print ("%T%T%T%T%TViewer wants message number " + i.out + "%N")
    end

view (mess: STRING)
        -- Display message `mess' for user.
    do
        print ("%T%T%T%T%TViewer reads: " + mess + "%N")
    end
```

Each of these procedures simply outputs a line, saying respectively that the viewer wants a certain message, and that it is reading that message. %T is a tab character.

Now we are going to keep these structures and make the execution concurrent.

4 GOING SEPARATE

The key SCOOP concept is the notion of a separate object. This is also the only keyword you will ever have to learn to go concurrent: **separate**. To understand it, let us go back to the original data structure illustration and see what it becomes in the concurrent version:



Basic data structures and their regions

The data structure does not change but its constituent objects are now spread across three *regions*. A region is simply a group of objects. In SCOOP, every run-time object will belong to one of the regions. (In other words, the regions constitute a *partition* of the set of objects.) Every region has an associated **processor**; the role of the processor is to execute operations on objects of its regions. So if client denotes the *CLIENT* object above, any *client*.*add* (*message*) operation, adding message to the end of the list, will be executed by the processor of Region 1. Any operation on the viewer will be executed by the processor of Region 2, and any operation on the downloader by the processor of Region 3.

A "processor" not necessarily a physical CPU but simply a mechanism that can execute instructions in sequence. In the current SCOOP implementation it is usually a *thread*. Whatever the implementation, a processor is strictly sequential; concurrency comes from having *several* processors.

If two objects belong to different regions, they are said to be **separate** from each other. A reference to an object in another region is called a separate reference. Here the *downloader* and *viewer* references in the client, and *client* in both the viewer and the downloader, are all separate.

How do we get multiple regions and, as a result, separate references? Simple. If you want a reference to be potentially separate at run time, declare the type of the corresponding variable (or formal argument etc.) not just for example as *VIEWER* but as **separate** *VIEWER*.

That's what we are going to do now in the example: declare as **separate** everything that can, during an execution, denote a separate object.

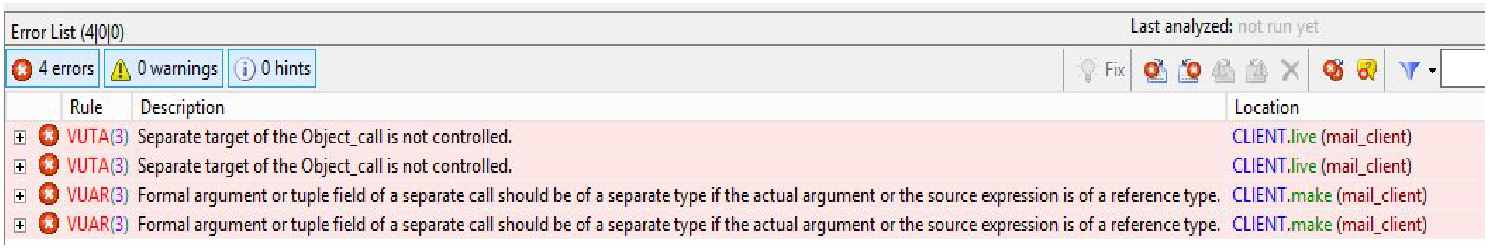
Leave the sequential and concurrent versions untouched for reference and go to the scratchpad directory where you will make the changes. The code is initially identical to the sequential version. Start a new instance of EiffelStudio on this project by clicking *mail_client.ecf*, and compile.

In class *CLIENT*, add the **separate** mark to the declarations of the downloader and viewer:

In CLIENT

```
    feature -- Access

        downloader: separate DOWNLOADER
                -- Downloading engine

        viewer: separate VIEWER
                -- Message viewing engine
```

If the viewer and downloader are separate from the client, the client is separate from each of them. Since you have not specified this property yet, if you recompile at this point you get an error message:



Error List (4|0|0)   Last analyzed: not run yet

4 errors | 0 warnings | 0 hints | Fix

	Rule	Description	Location
⊞	VUTA(3)	Separate target of the Object_call is not controlled.	CLIENT.live (mail_client)
⊞	VUTA(3)	Separate target of the Object_call is not controlled.	CLIENT.live (mail_client)
⊞	VUAR(3)	Formal argument or tuple field of a separate call should be of a separate type if the actual argument or the source expression is of a reference type.	CLIENT.make (mail_client)
⊞	VUAR(3)	Formal argument or tuple field of a separate call should be of a separate type if the actual argument or the source expression is of a reference type.	CLIENT.make (mail_client)

For the rest of this presentation, we are going to consider compiler error messages not as a nuisance but as precious help to get our concurrent software right. The big prize at the end is the guaranteed absence of data races. We are going to fix the last two error messages first. Declare the argument of *make* as **separate** in *VIEWER*:

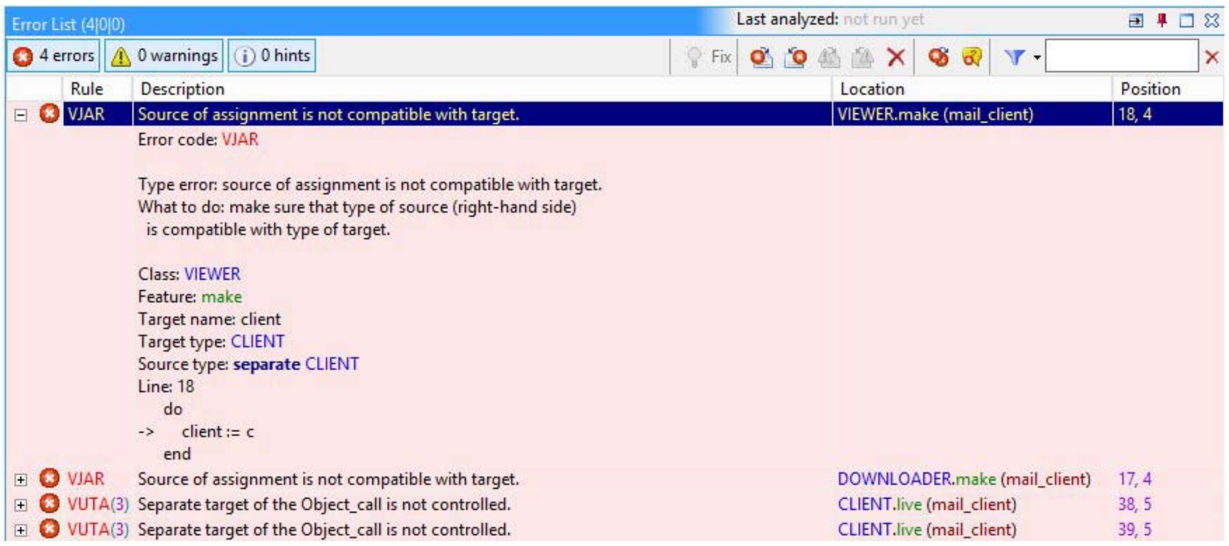
In VIEWER

```
    feature {NONE} -- Initialization

        make (c: separate CLIENT)
                -- Initialize with `c' as client.
            do
                client := c
            end
```

Do the same in *DOWNLOADER*: the type argument type of *make* in that class is *CLIENT*; change it to **separate** *CLIENT*.

When you recompile, you still get four errors, including two new ones. Click the "+" sign on the left to see the detailed message:

In VIEWER



Error List (4|0|0)   Last analyzed: not run yet

4 errors | 0 warnings | 0 hints | Fix

	Rule	Description	Location	Position
⊟	VJAR	Source of assignment is not compatible with target.	VIEWER.make (mail_client)	18, 4

```
Error code: VJAR

Type error: source of assignment is not compatible with target.
What to do: make sure that type of source (right-hand side)
  is compatible with type of target.

Class: VIEWER
Feature: make
Target name: client
Target type: CLIENT
Source type: separate CLIENT
Line: 18
    do
->     client := c
    end
```

	Rule	Description	Location	Position
⊞	VJAR	Source of assignment is not compatible with target.	DOWNLOADER.make (mail_client)	17, 4
⊞	VUTA(3)	Separate target of the Object_call is not controlled.	CLIENT.live (mail_client)	38, 5
⊞	VUTA(3)	Separate target of the Object_call is not controlled.	CLIENT.live (mail_client)	39, 5

The message says it all: "*Source of assignment is not compatible with target*". A basic type rule of SCOOP is that you cannot assign a separate expression, here *c*, to a non-separate target, here *client*. Why? Looking at the figure on page 9, *client* is a field of an object in region 1, but assigning it the reference *c* will cause it to denote an object in Region 1. This is the right behavior, but to reflect the run-time separateness of this reference we must declare *client* **separate** as well as c.

A variable not declared **separate**, but denoting an object in another region, would be called a **traitor**. As part of avoiding data races, the SCOOP type system ensures that no execution will every produce a traitor.

Note that, the other way around, *sep := nonsep* (with *sep* separate and *nonsep* non-separate) is OK: declaring a variable as "separate" means that the corresponding objects *might* be in another region, not that they have to. It could happen that in some executions *sep* stays in the same region.

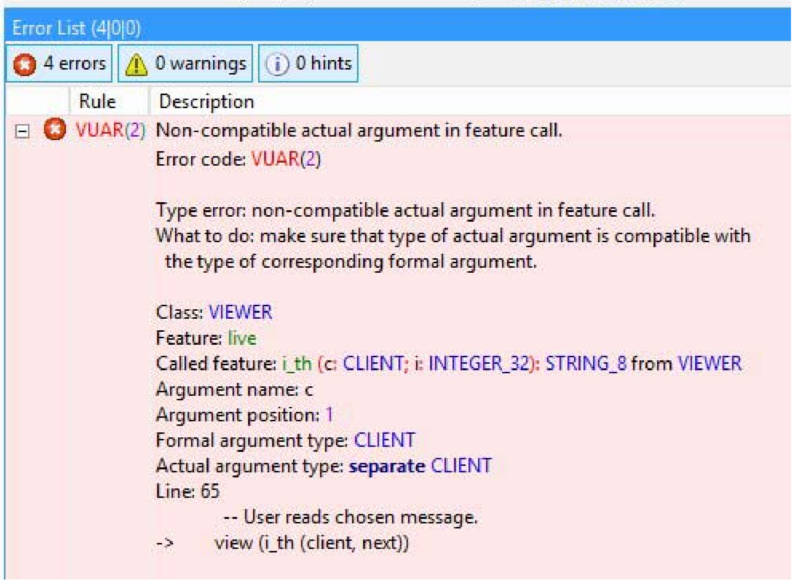
Of course *client* must be separate in *VIEWER* since (see the figure page 9) the viewer is in region 2 and the client in region 1. Fix the declaration of *client* (towards the end of class *VIEWER*)

***In** VIEWER*

```
feature {NONE} -- Implementation

    client: separate CLIENT
            -- (Shared)client from which we are getting messages.
```

Do the same for *client* in *DOWNLOADER* and recompile.

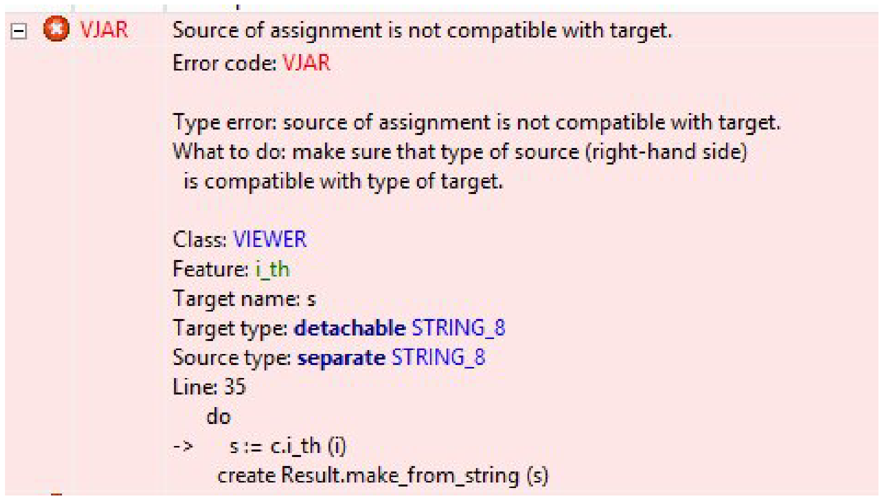
This actually yields more compilation errors but (believe it or not) we are progressing! The first error message points to the routine *live* of *VIEWER*:


Error List (4|0|0)
4 errors | 0 warnings | 0 hints

Rule	Description
VUAR(2)	Non-compatible actual argument in feature call.

```
Error code: VUAR(2)

Type error: non-compatible actual argument in feature call.
What to do: make sure that type of actual argument is compatible with
  the type of corresponding formal argument.

Class: VIEWER
Feature: live
Called feature: i_th (c: CLIENT; i: INTEGER_32): STRING_8 from VIEWER
Argument name: c
Argument position: 1
Formal argument type: CLIENT
Actual argument type: separate CLIENT
Line: 65
          -- User reads chosen message.
->      view (i_th (client, next))
```

At issue is the function *i_th* , called by *live*, which takes a *CLIENT* as its first argument. Obviously this should be **separate** *CLIENT*. Fix this and recompile; there is still something wrong with *i_th*:



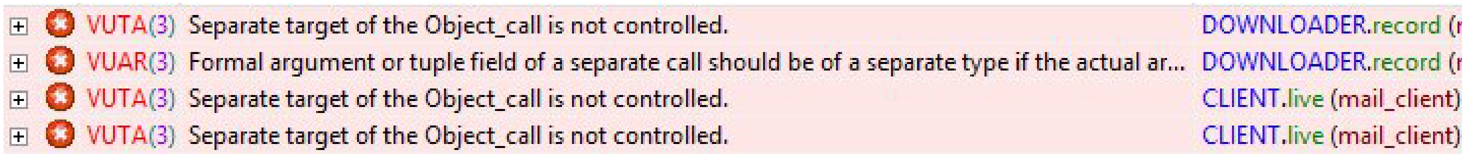
```
⊟ VJAR   Source of assignment is not compatible with target.
         Error code: VJAR

         Type error: source of assignment is not compatible with target.
         What to do: make sure that type of source (right-hand side)
           is compatible with type of target.

         Class: VIEWER
         Feature: i_th
         Target name: s
         Target type: detachable STRING_8
         Source type: separate STRING_8
         Line: 35
             do
         ->      s := c.i_th (i)
                 create Result.make_from_string (s)
```

Since *c*.*i_th* (*i*) returns the last element of the list of messages from the (separate) client, that list and the result of *i_th* are separate. So *s* should be separate too. Add **separate** to the declaration of *s*. This is not enough (as you will see if you compile now): for the Result of *i_th*, we want a string *in the same region*, not a separate one. So instead of *make_from_string* we must use *make_from_separate*, another creation procedure from the library class *STRING*, designed expressly for such cases: *make_from_separate* initializes a string by copying from a string in a possibly different region. Correct *i_th* to use this creation procedure:

***In** VIEWER*

```
i_th (c: separate CLIENT; i: INTEGER): STRING
        -- Message of index `i' in `c'.
    require
        i <= c.count
    local
        s: separate STRING
    do
        s := c.i_th (i)
        create Result.make_from_separate (s)
    end
```

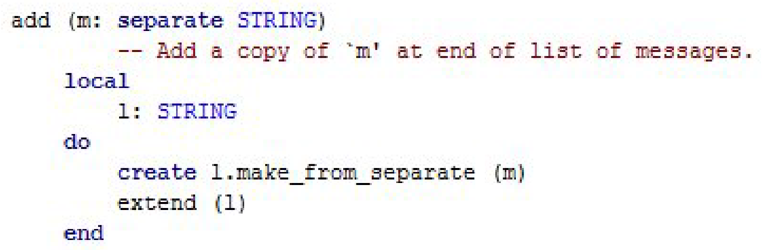
Recompile. Four errors remain :



```
⊞ VUTA(3) Separate target of the Object_call is not controlled.                                       DOWNLOADER.record (r
⊞ VUAR(3) Formal argument or tuple field of a separate call should be of a separate type if the actual ar... DOWNLOADER.record (r
⊞ VUTA(3) Separate target of the Object_call is not controlled.                                       CLIENT.live (mail_client)
⊞ VUTA(3) Separate target of the Object_call is not controlled.                                       CLIENT.live (mail_client)
```

The second one, having to do with *DOWNLOADER*.*record*, is similar to what was wrong with *live* and *i_th* in *VIEWER*. The offending routine is *add* of *CLIENT*. Correct it to make sure that its argument is separate and to use *make_from_separate* instead of *make_from_string*:

***In** CLIENT*

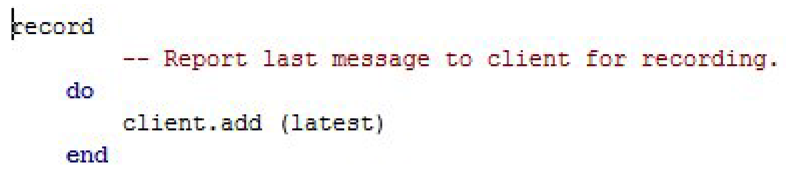


```
add (m: separate STRING)
        -- Add a copy of `m' at end of list of messages.
    local
        l: STRING
    do
        create l.make_from_separate (m)
        extend (l)
    end
```

Recompile; the other three errors are still there. They all say that the "*separate target*" of a call is not "*controlled*". We are seeing a key property of SCOOP. If you have a call *x*.*r* (...) where the target *x* is separate, then *x* must be "controlled", meaning that the caller has obtained exclusive access to its processor, and hence to all the objects in its region. Then no one else can access it and you have removed any possibility of data race.

There are two ways of controlling an object. Let's try the first one, starting with the first error message. It has to do with *record* in *DOWNLOADER*, which reads

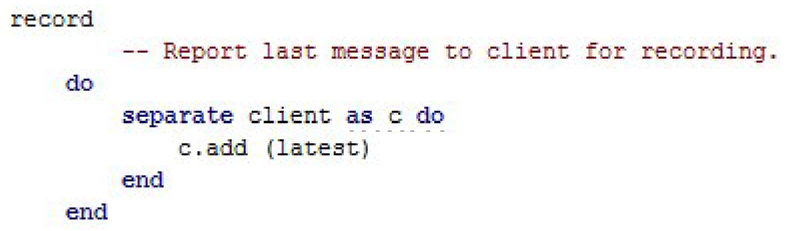
***In** DOWNLOADER*



```
record
        -- Report last message to client for recording.
    do
        client.add (latest)
    end
```

The call *client*.*add* (*latest*) is invalid in SCOOP because *client* is now separate but not controlled. We make it controlled through the **separate** instruction (not to be confused with the **separate** declaration), changing the declaration into:

***In** DOWNLOADER*



```
record
        -- Report last message to client for recording.
    do
        separate client as c do
            c.add (latest)
        end
    end
```

(If you compile now you see that the first error is gone.) What does the **separate** instruction mean? It directs the caller to gain exclusive access to *client* under the name *c*, meaning exclusive access to its processor (and all other objects in its region). Of course if that processor was already controlled by someone else we will have to wait. But when we finally get the *client* object, we have the guarantee that it is available to us, and to us only, under the local name *c*.

This resource reservation mechanism is very general; it lets you, in one shot, get exclusive control not just of one processor as here but of **any number** of objects and their processors. Just write **separate** *a* **as** *x*, *b* **as** *y*, ... **do** ... **end**. No need to use complex algorithms to lock one object, then the next, and to back up at that stage if you fail (to avoid deadlock). SCOOP guarantees that when you get the objects you get all of them.

We can take advantage of this property to remove the two remaining "target not controlled" errors, which are for *live* in *CLIENT*. Just replace

In *CLIENT*

```
live
    do
            downloader.live
            viewer.live
    end
```

by

In *CLIENT*

```
live
    do
            separate downloader as d, viewer as v do
                d.live
                v.live
            end
    end
```

In one shot, the **separate** instruction gives us simultaneous exclusive access to the processors for *both* the downloader and viewer, meaning to regions 2 and 3 in the figure of page 9.

You might fear that this instruction locks these regions for the entire execution, but that is not what it means. *live* needs exclusive access to the downloader and viewer for a very short period only: just long enough to start *d.live* and *v.live*. Such calls, having a separate target, are **asynchronous**, meaning they get executed on their own processors; the caller, here *live*, just logs the calls and moves on with its own business without waiting. Think of starting off a print job and proceeding with the rest of your work without waiting for the print's completion.

Recompile; this time the program compiles without error. Not only that, but it now has the desired concurrent semantics! (As last touches, in the message to be output at the beginning of execution by *CLIENT.make*, replace "*SEQUENTIAL*" by "*CONCURRENT*", and for more interesting results change the value of *Max_messages* in class *PARAMETERS* from 10 to 50.) Click Run and you will see the concurrent version at work, as in the figure of page 3. Victory!

5 Probing further

Let's consider three more properties of SCOOP visible on this example.

We noted above that the **separate** instruction is one of two ways to gain exclusive access. The other one is even simpler. If you call *r (x)* where the corresponding formal argument in *r* is separate, then the execution of *r* gains exclusive access to *x*. So when *DOWNLOADER.record* calls *CLIENT.add*, declared (see page 13) as

In *CLIENT*

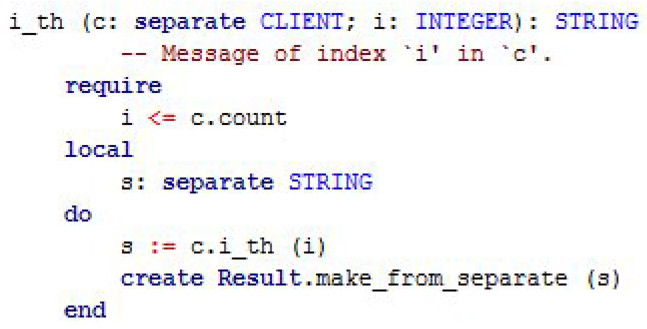
```
add (m: separate STRING)
        -- Add a copy of `m' at end of list of messages.
    local
        l: STRING
    do
        create l.make_from_separate (m)
        extend (l)
    end
```

this routine has exclusive access to *m* (and temporarily transfers it to *make_-from_separate*). As with the **separate** instruction, this mechanism is applicable to any number of separate arguments, not just one.

Next, with the concept of asynchronous call (as with *d*.*live* where *d* is the downloader) the question arises of how to *resynchronize* with a call that you have started asynchronously (like going to the printer to get your printout). The SCOOP answer is simple: as soon as you *query* the target object, for example with a function call such as *d*.*count*, you will have to wait for all previous calls to *d* to have completed.

Finally, preconditions as wait conditions. Consider the function *i_th* in *VIEWER*. It reads:

***In** VIEWER*



```
i_th (c: separate CLIENT; i: INTEGER): STRING
        -- Message of index `i' in `c'.
    require
        i <= c.count
    local
        s: separate STRING
    do
        s := c.i_th (i)
        create Result.make_from_separate (s)
    end
```

The precondition (**require** clause) states that we can only obtain a message that has already been made available to the client, meaning its message number *i* is no greater than the client's *count* of messages downloaded thus far. That precondition involves *c*.*count* where *c* is separate. Such a precondition is not (as in sequential code) a *correctness* condition, which would be useless here. It is a **wait** condition. The execution of *i_th* will wait until the client not only is available (since the argument *c* is separate) but also, per the precondition, has at least *i* messages.

This is the behavior we want: when the user requests a message, the viewer should wait until that message becomes available. Of course if there are several separate preconditions, execution will wait until *all* are satisfied. Think of how difficult it would be to program such behaviors manually; in SCOOP, you just write the wait conditions as precondition clauses.

This powerful mechanism is the SCOOP technique for waiting on single or multiple conditions.

6 On with SCOOP

With this tutorial you have seen all of the key SCOOP concepts (remember, the S is for "simple"). Of course there have many interesting consequences. To explore SCOOP in depth you can start from the references given in the introduction. Note in particular the collection of classical concurrency examples (dining philosophers, barber shop and many others), which complement the email client example.

SCOOP has been designed to make concurrent programming safe and pleasurable. We hope you enjoy SCOOP and use it to build many great concurrent systems.