Chair of Software Engineering

# Implementing and evaluating an exception mechanism for SCOOP

Master Thesis

Florian Besser
ETH Zurich
fbesser@ethz.ch

March 16, 2013 - September 18, 2013

Supervised by:
Benjamin Morandi
Prof. Bertrand Meyer

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

**Abstract**

Exception handling is an important part of software development. Handling exceptions in concurrent software is even more complicated: There can be asynchronous exceptions, but because of design choices in each language, no exception mechanism can be used everywhere.

Several exception mechanisms have been proposed for SCOOP, a concurrent object-oriented programming model. We focus on two of these exception mechanisms, the EiffelSoftware exception mechanism as well as the accountability mechanism.

We implemented the accountability mechanism, an exception mechanism which handles exceptions in a more expressive and less restrictive way than the Eiffelsoftware exception mechanism. We also developed variations of that mechanism, making it more expressive. After that, we evaluated both the Eiffelsoftware exception mechanism as well as the accountability mechanism in terms of usability, expressiveness and performance. The evaluation has shown us two points on which we could improve: We refined the accountability mechanism by adding another variant, making it even more expressive and we preserved the types of asynchronous exceptions, thus providing developers with more useful information.

Additionally, we implemented the duel mechanism on top of the ordinary exception mechanism, allowing two or more SCOOP processors to duel for a resource, instead of having to wait until the resource is eventually released.

**Acknowledgments**

I would like to thank my supervisor, Benjamin Morandi, for always making time for me, his clarifications on how the accountability mechanism should be implemented, as well as for his help whenever problems arose that were out of my control.

Thanks to Ian King and the people working on SCOOP for their input and the effort with which they fixed problems.

Also thanks to Emmanuel Stapf, who pointed me in the right direction as to where I could implement my changes in this behemoth of a codebase.

Thanks also go to my friends and family, who loyally pretended to follow my train of thought as I raved on about processors and exceptions.

# Contents

# Chapter 1

# Introduction

In SCOOP multiple *processors* execute code in interleaved and potentially parallel fashion. SCOOP supports *asynchronous* (non-blocking) calls from objects on one processor to objects on another processor. This allows the *client* to continue while the *supplier* executes the received call. If the call terminates normally, the client does not need to be informed. But if the supplier *raises* an exception, the exception must be *propagated* to the client, who then has to change its control flow and *handle* the exception. The client may no longer be in a state where it can handle the exception though, since the client continued after placing the call on the supplier. An exception from such an asynchronous call is called an *asynchronous exception*. Such exceptions have a context, which consists of two parts: The *failed context* is the supplier's context, and the *responsible context* is the context of the client when it made the call. Exceptions are always the product of a $failure$ of the supplier, namely to hold its promise to the client.

Several exception mechanisms have been proposed for SCOOP. This paper is about implementing the accountability mechanism as described in [1]. We then evaluate the EiffelSoftware exception mechanism as well as the newly implemented one. The contributions are:

1. An implementation of the accountability mechanism as described in [1].
2. Several test programs to verify the correctness of the implementation.
3. An evaluation of usability, expressiveness, and performance of both the existing implementation as well as the accountability mechanism.
4. A duel mechanism, which builds upon the exception mechanism, and which allows processors to duel for resources.

Section 2 presents an overview of SCOOP and its features, section 3 explains the exception mechanisms. Section 4 explains the implementation. The exception mechanisms are evaluated in section 5. Section 6 covers improvements for the accountability exception mechanism. The duel mechanism is explained as well as evaluated in section 7. A user as well as a developer guide can be found

in section 8. Related work is discussed in section 9, references can be found in section 10. We present our conclusions in section 11.

# Chapter 2

# SCOOP

In SCOOP, every object is attached to a processor, and that processor is called the object's *handler*. A *processor* is a thread that can execute actions on objects. An object's class describes what $features$ can be applied to the object. A processor may be a CPU, but it can also be implemented as a thread; if a mechanism can execute calls sequentially, it can be a processor.

A variable $x$ that is attached to a processor can have pointers to other objects with the same handler (non-separate objects) as well as objects with a different handler (separate objects). In case of a non-separate object,the call $x.f$ is non-separate: It is synchronously executed by the handler. Here, $x$ is called the *target* of the feature call. In case of a separate object, the feature call is separate: The supplier executes the call asynchronously. Asynchronous feature calls support concurrent execution of programs.

A good example of these concepts can be found in a simple producer and consumer example, shown in listing 2.1. Assume that both the producer as well as the consumer have their own handlers. We generally use the same word for both the object as well as the handler, so in this case "consumer" refers both to the actual consumer object as well as its handler.

The keyword *separate* means that the objects may have a different handler than the current object. In this example, both the producer as well as the consumer have access to the same buffer, through which they will communicate.

To make sure that only one object accesses the buffer at the same time, the buffer must be locked by that object. Locks required to execute a feature are specified in the argument list. All possible targets of separate type must be listed in the argument list, and the feature will only be executed once all of them are locked by the current processor. Locks are held for as long as the client executes its feature. Targets that are locked are called controlled, and targets that are not locked are called uncontrolled.

SCOOP supports *condition synchronization*: Preconditions are not simply evaluated to true or false, but they can be wait conditions. In the example 2.1, the precondition of *consume* states that the buffer must not be empty. This means that the execution of *consume* is delayed until the precondition evaluates

Listing 2.1: *Eiffel*: Producer-consumer example

```
1  producer: separate PRODUCER
2  consumer: separate CONSUMER
3  buffer: separate BUFFER[INTEGER]
4
5  consume ( buffer: separate BUFFER[INTEGER] ): INTEGER
6  require
7    not buffer.is_empty
8  do
9    Result := buffer.item
10 end
11
12 produce ( buffer: separate BUFFER [INTEGER]; a_element:
       INTEGER )
13 require
14   not buffer.is_full
15 do
16   buffer.put (a_element)
17 end
```

to true.

Listing 2.2 shows *lock passing*, another technique supported by SCOOP. In this scenario, the supervisor will act as the client, while the worker will be the supplier. The client has the ability to pass locks it currently holds to the supplier, including the lock to itself. This allows the supplier to place calls to the resources guarded by these locks. The client has to wait for the supplier to return the locks, so any call involving lock passing becomes synchronous.

SCOOP uses *wait by necessity* to determine when synchronization between supplier and client is necessary: The client will wait for a result from a call if that result is needed. In the example 2.1, $Result := buffer.item$ is a synchronous call, since the result from $buffer.item$ is required and therefore waited upon.

Listing 2.2: *Eiffel*: Supervisor-worker example

```
1  note
2    description: "The supervisor will place a workload on a
         WORKER, but will remain available should the WORKER
         have any problems/questions."
3
4  class SUPERVISOR
5
6    place_workload (a_worker: separate WORKER)
7      do
8        a_worker.do_work ( Current )  --Allow the worker to
             place callbacks to the supervisor
9      end
10
11 end
```

# Chapter 3

# The Exception Mechanisms

This chapter gives an overview of the two exception mechanisms. It explains the basic behaviour of processors, with a high-level event based algorithm.

## 3.1 EiffelSoftware Exception Mechanism

This exception mechanism was developed by EiffelSoftware. In this exception mechanism suppliers remember exceptions as well as the client responsible for them. The basic idea is that an exception caused by a client must eventually be returned to that client.

Algorithm 1 shows the interactions between a client p and suppliers $\{q_1$ ... $q_n\}$ for a feature request f. The event based notation follows Cachin et al. [4].

Whenever a client locks a supplier (see Feature_application_start), the supplier remembers the client and that it is locked (see $\langle$Get | p, [LOCK]$\rangle$). In case the supplier fails, it executes the associated rescue clause to reestablish its consistency. Without a retry instruction (see Rescue_clause_end), the supplier remembers the exception as well as the client associated with it. It will never again execute any calls from that client (see Feature_call, after Rescue_clause_end has been executed), and the client will always get the same exception if it tries to interact with the supplier. The failed supplier will release all locks it holds, and wait until the client releases the lock on the supplier. The client will receive an exception whenever it asks for a report from the supplier, and the supplier reports a failure (see $\langle$Get | p, [GET_REPORT]$\rangle$ and $\langle$Get | p, [GET_REPORT_WAIT]$\rangle$). The client queries for such a report before placing a call on a supplier and after the supplier executed a synchronous call (see Feature_call). When the client unlocks the supplier (see $\langle$Get | p, [UNLOCK]$\rangle$), the supplier will no longer be locked, but still try to execute any remaining calls placed on it.

This approach allows exceptions to be passed back to whomever caused them, allowing processors to delegate work to other processors, and then later query for the result or the exception. It also makes sure that the client will get informed of possible exceptions in a timely manner, while not requiring any

manual querying specifically for exceptions or other effort from the developer's side.

## 3.2 Accountability Mechanism

The core of this exception mechanism is *accountability*. Whenever clients query for any exceptions from suppliers, they are said to hold the suppliers *accountable*. A supplier is called *accountable* if it might in the future have to report exceptions to its client.

Algorithm 2 shows the interactions between a client p and suppliers $\{q_1 \ ... \ q_n\}$ for a feature request f.

Whenever a client locks a supplier (see Feature_application_start), the supplier sets itself accountable and locked (see ⟨Get | p, [LOCK]⟩). In case the supplier fails, it executes the associated rescue clause to reestablish its invariant. Without a retry command (see Rescue_clause_end), the supplier remembers the failure and cleans up: It deletes any remaining feature requests from the client and waits until the client decides to release the lock (see ⟨Get | p, [UNLOCK]⟩) or until the supplier is held accountable (see ⟨Get | p, [HOLD_ACCOUNTABLE]⟩). Should the client hold the supplier accountable, it will propagate the exception to the client (see Feature_call). If a supplier is held accountable, the supplier becomes unfailed, meaning it is allowed to receive more calls. The client automatically holds the supplier accountable if it does a synchronous feature call. As soon as the client releases the lock of the supplier, the supplier dismisses the accountability (see ⟨Get | p, [UNLOCK]⟩). If there is still some work left from the client, the supplier will execute it. In case of failure, the supplier will not safe the exception, since it is no longer accountable (and therefore the client can never hold the supplier accountable). When there is no work left and the supplier is no longer accountable, the supplier forgets any exception it might have stored before (see **upon** $\neg accountable \wedge requests = ()$).

This exception mechanism allows exceptions to be lost, for example by never querying a processor after giving it some asynchronous work. Sometimes it is vital that no exception ever gets lost, and this is the purpose of the $safemode$. In case the safemode is used, the client will not only query for exceptions on synchronous calls, but also whenever it finishes executing its own feature body (see Feature_body_end). By doing so, the supplier will never be in a situation where it discards an exception, and no exceptions will be lost. The safemode is a slight variant of the original accountability mechanism, and can be turned on or off by the developer.

```
1  upon event ⟨Initialize⟩ do
2  |  locked := false; client := Void; dirty_towards := (); requests := ();
3  upon event ⟨Feature_application_start | f, {q_1, ..., q_n}⟩ do
4  |  forall the q_i ∈ {q_1, ..., q_n} do
5  |  |  trigger ⟨Send | q_i, [LOCK]⟩;
6  |  end
7  upon event ⟨Feature_call | f, q_i, is_synchronous⟩ do
8  |  trigger ⟨Send | q_i, [GET_REPORT]⟩;
9  |  wait ⟨Get | q_i, [REPORT, s]⟩;
10 |  if s = fail then
11 |  |  rescue;
12 |  end
13 |  trigger ⟨Send | q_i, [FEATURE_LOG, f]⟩;
14 |  if is_synchronous then
15 |  |  trigger ⟨Send | q_i, [GET_REPORT_WAIT]⟩;
16 |  |  wait ⟨Get | q_i, [REPORT, s]⟩;
17 |  |  if s = fail then
18 |  |  |  rescue;
19 |  |  end
20 |  end
21 upon event ⟨Feature_application_end | f, {q_1, ..., q_n}⟩ do
22 |  forall the q_i ∈ {q_1, ..., q_n} do
23 |  |  trigger ⟨Send | q_i, [UNLOCK]⟩;
24 |  end
25 upon event ⟨Get | p, [LOCK]⟩ do
26 |  locked := true; client := p;
27 upon event ⟨Get | p, [FEATURE_LOG, f]⟩ do
28 |  requests := requests • f;
29 upon event ⟨Rescue_clause_end⟩ do
30 |  dirty_towards := dirty_towards ∪ {client};
31 |  requests := ();
32 upon event ⟨Get | p, [GET_REPORT]⟩ do
33 |  if p ∈ dirty_towards then
34 |  |  trigger ⟨Send | p, [REPORT fail ]⟩;
35 |  else
36 |  |  trigger ⟨Send | p, [REPORT success ]⟩;
37 |  end
38 upon event ⟨Get | p, [GET_REPORT_WAIT]⟩ such that requests = ()
   do
39 |  if p ∈ dirty_towards then
40 |  |  trigger ⟨Send | p, [REPORT fail ]⟩;
41 |  else
42 |  |  trigger ⟨Send | p, [REPORT success ]⟩;
43 |  end
44 upon event ⟨Get | p, [UNLOCK]⟩ do
45 |  locked := false;
```

**Algorithm 1:** Asynchronous exception mechanism, EiffelSoftware implementation

```
1  upon event ⟨Initialize⟩ do
2  |  locked := false; accountable := false;
3  |  failed := false; requests := ();
4  upon event ⟨Feature_application_start | f, {q_1, ..., q_n}⟩ do
5  |  forall the q_i ∈ {q_1, ..., q_n} do
6  |  |  trigger ⟨Send | q_i, [LOCK]⟩;
7  |  end
8  upon event ⟨Feature_call | f, q_i, is_synchronous⟩ do
9  |  trigger ⟨Send | q_i, [FEATURE_LOG, f]⟩;
10 |  if is_synchronous then
11 |  |  trigger ⟨Send | q_i, [HOLD_ACCOUNTABLE]⟩;
12 |  |  wait ⟨Get | q_i, [REPORT, s]⟩;
13 |  |  if s = fail then
14 |  |  |  rescue;
15 |  |  end
16 |  end
17 upon event ⟨Feature_body_end | f, {q_1, ..., q_n}, in_safe_mode⟩ do
18 |  if in_safe_mode then
19 |  |  forall the q_i ∈ {q_1, ..., q_n} do
20 |  |  |  trigger ⟨Send | q_i, [HOLD_ACCOUNTABLE]⟩;
21 |  |  |  wait ⟨Get | q_i, [REPORT, s]⟩;
22 |  |  |  if s = fail then
23 |  |  |  |  rescue;
24 |  |  |  end
25 |  |  end
26 |  end
27 upon event ⟨Feature_application_end | f, {q_1, ..., q_n}⟩ do
28 |  forall the q_i ∈ {q_1, ..., q_n} do
29 |  |  trigger ⟨Send | q_i, [UNLOCK]⟩;
30 |  end
31 upon event ⟨Get | p, [LOCK]⟩ do
32 |  locked := true; accountable := true;
33 upon event ⟨Get | p, [FEATURE_LOG, f]⟩ do
34 |  if ¬failed then
35 |  |  requests := requests • f;
36 |  end
37 upon event ⟨Rescue_clause_end⟩ do
38 |  failed := true; requests := ();
39 upon event ⟨Get | p, [HOLD_ACCOUNTABLE]⟩ such that
   requests = () do
40 |  if failed ∧ accountable then
41 |  |  failed := false; trigger ⟨Send | p, [REPORT fail ]⟩;
42 |  else
43 |  |  trigger ⟨Send | p, [REPORT success ]⟩;
44 |  end
45 upon event ⟨Get | p, [UNLOCK]⟩ do
46 |  accountable := false;
47 upon ¬accountable∧ requests = () do
48 |  locked := false; failed := false;
```

**Algorithm 2:** Asynchronous exception mechanism, accountability mechanism with and without safemode

# Chapter 4

# Implementation

This chapter discusses how SCOOP and the various exception mechanism are implemented. We use high-level descriptions, if you are interested in the code, have a look at the developer guide in Section 8.2.

## 4.1 EiffelStudio SCOOP Implementation

SCOOP uses a *chains over queues system*, or COQS in short. To understand the implementation of the exception mechanism, it is vital to know how a COQS works.

EiffelSoftware implemented a simple COQS system, defined as follows: A COQS is defined as a finite set $Q_1, Q_2, \ldots, Q_n$ called the *queues*, such that there exists a set of non-empty lists, called the *chains* with the following properties:

1. Every element of every queue belongs to exactly one chain.
2. Every element of every chain belongs to exactly one queue.

Every SCOOP processor $P_i$ has a queue $Q_i$ attached to itself. The chain elements of $Q_i$ hold the work that processor $P_i$ must execute.

Figure 4.1 shows a COQS as it is used in SCOOP: P as well as R would like to assign some work to Q. Since the calls from P and R should not interleave, two chains are necessary. P creates a chain with itself as *head* and Q as a *dependant*. R does the same. P and R can now assign some work to Q: P places the two calls *f()* and *g()* on Q while R places the call *x(3)*.

Processors execute a processor loop, as shown in Algorithm 3. A processor gets the top chain node of its queue (for Q this would be the node containing *f()* and *g()*), then starts to work on the chain node. A processor only exits its current chain node if the head of the chain *closes* the chain (for Q, P would be the head). The head of a chain closes a chain only if it executed all feature requests of its chain node. Figure 4.2 shows the same processors, after a small

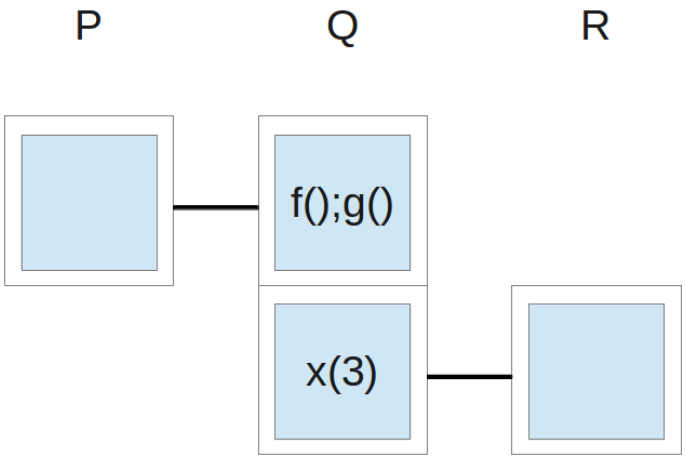


Figure 4.1: P and R place calls on Q

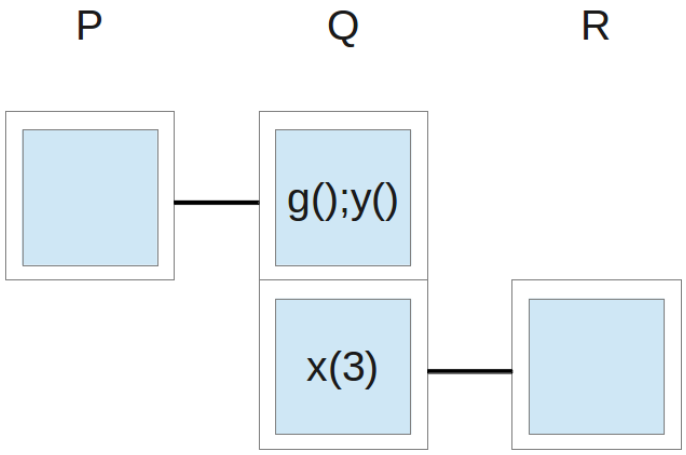


Figure 4.2: Q began working on its top chain node, P placed another call on Q

amount of time has passed: Q has executed the call *f()*, and P has placed another call on Q. Note that the call from P will be executed before the earlier call of *x(3)* from R. This is necessary since otherwise the calls from P and R would interleave.

Algorithm 3 shows the basic behaviour of a processor: A processor grabs its next chain node and proceeds to work on it. While a processor works on a chain node, it queries the chain node for the top feature request stored, and saves the request as the *current* feature request. The processor will then *execute* the current feature request. Executing a feature request means executing each call one by one. If a call has a separate target the client will place it as a feature request on the supplier. If a separate call is also synchronous, the client waits for the supplier for an answer (wait by necessity).

It is important to distinguish between chain nodes and actual feature requests: A chain node is placed on a queue and can contain any number of

feature requests for a processor to execute. A feature request is always part of a chain node. This constraint makes sure that processors working on a chain cannot be disturbed, and no two clients interleave work on a supplier.

## 4.2 EiffelStudio Exception Mechanism Implementation

Algorithm 3 works as long as no organized panic happens. However, if asynchronous exceptions should be handled the algorithm needs to be extended. Algorithm 4 shows the changes needed to implement the EiffelSoftware exception mechanism (*work_on_chain* was not changed and thus omitted). The client will look if the supplier is dirty towards it before placing a call as well as after waiting for the response from a placed separate synchronous call. Processors encountering an exception will look up their current client, and set themselves dirty towards it.

## 4.3 Accountability Mechanism Implementation

Algorithm 5 shows the processor loop when using the accountability mechanism with and without safemode (again, *work_on_chain* was not altered and is therefore omitted). If the accountability mechanism is used, chains receive a new flag called *accountable*. If a chain is accountable or the safemode is turned on, the involved processors will store exceptions, and otherwise throw them away. Chains are accountable when created, and become unaccountable whenever the client releases its locks. This accountability mechanism makes sure that processors only store exceptions that have a chance to be later propagated back to the client that caused them. Processors have a new flag called $failed$. Processors start unfailed, and become failed if they enter an organized panic. Processors become unfailed whenever they propagate an exception back to the client or when the client releases its lock. Thus the mechanism ensures that the supplier is always unfailed when it is locked by a client.

If the safemode is turned on processors will query all their suppliers before releasing locks, thus preventing exceptions to be lost.

## 4.4 Testing

We developed many test cases in order to verify the exception mechanism performs as it should. In total, we developed 32 programs that test one or multiple aspects of an exception mechanism. You can find all these programs online when building our solution, see Section 8.2.1.

For very basic functionality testing, we developed six programs that deal with exception handling. These programs are of an academic nature, and do

```
1  Algorithm processor_loop()
2      initialization;
3      while not processor.is_redundant do
4          current_chain_node := processor.queue.top;
5          current_chain := current_chain_node.entire_chain;
6          while not (current_chain.is_closed ∧ current_chain_node.is_empty)
           do
7              work_on_chain (current_chain_node);
8              if processor = current_chain.head ∧
               current_chain_node.is_empty then
9                  current_chain.close
10             end
11         end
12         processor.queue.pop;
13     end
1  Procedure work_on_chain(Chain_node node)
2      if node.top != Void then
3          current_feature_request := node.top;
4          node.pop;
5          if current_feature_request.needs_new_locks then
6              Create a new chain, spanning all processors which need to be
               locked.
7          end
8          execute (current_feature_request);
9      end
1  Procedure execute(Feature_request req)
2      while req.body.top != Void do
3          current_call := req.body.top;
4          req.body.pop;
5          if current_call.target = processor then
6              Execute locally
7          else if current_call.is_lock_passing_call then
8              place_call(current_call, current_call.target); –Places the call (as
               a feature request) on the target processor
9              Wait for result, and periodically check for callbacks.
10         else if current_call.is_separate_synchronous_call then
11             place_call(current_call, current_call.target);
12             Wait for result (do not check for callbacks).
13         else
14             place_call(current_call, current_call.target);
15             –Do not wait, call is asynchronous
16         end
17     end
```

**Algorithm 3:** Processor loop without support for exceptions

```
1  Algorithm processor_loop()
2      initialization;
3      while not processor.is_redundant do
4          current_chain_node := processor.queue.top;
5          current_chain := current_chain_node.entire_chain;
6          while not (current_chain.is_closed ∧ current_chain_node.is_empty)
           do
7              work_on_chain (current_chain_node);
8              if processor = current_chain.head ∧
               (current_chain_node.is_empty ∨
               processor.is_in_organized_panic) then
9                  current_chain.close;
10             end
11             if processor.is_in_organized_panic then
12                 processor.set_dirty_towards(current_chain.head);
13                 Wake up current_chain.head; –The client may be waiting
                   for a result, it must be woken up.
14                 Break; –Exit while loop, continue without organized panic.
15             end
16         end
17         processor.queue.pop;
18     end
1  Procedure execute(Feature_request req)
2      while req.body.top != Void do
3          current_call := req.body.top;
4          req.body.pop;
5          if current_call.is_separate_call ∧ current_call.target.is_dirty_towards
           (processor) then
6              throw exception;
7          end
8          if current_call.target = processor then
9              Execute locally –This can throw an exception
10         else if current_call.is_lock_passing_call then
11             place_call(current_call, current_call.target); –Places the call (as
               a feature request) on the target processor
12             Wait for result, and periodically check for callbacks.
13         else if current_call.is_separate_synchronous_call then
14             place_call(current_call, current_call.target);
15             Wait for result (do not check for callbacks).
16         else
17             place_call(current_call, current_call.target);
18             –Do not wait, call is asynchronous
19         end
20         if current_call.is_separate_synchronous_call ∧
           current_call.target.is_dirty_towards (processor) then
21             throw exception;
22         end
23     end
```

**Algorithm 4:** Execute with support for the EiffelStudio exception mechanism

```
1  Algorithm processor_loop()
2      initialization;
3      while not processor.is_redundant do
4          current_chain_node := processor.queue.top;
5          current_chain := current_chain_node.entire_chain;
6          while not (current_chain.is_closed ∧ current_chain_node.is_empty)
           do
7              work_on_chain (current_chain_node);
8              if processor.is_in_organized_panic ∧
               (current_chain.is_accountable ∨ safemode) then
9                  processor.set_failed;
10                 Wake up current_chain.head; –The client may be waiting
                   for a result, it must be woken up.
11             end
12             if processor = current_chain.head ∧
               current_chain_node.is_empty then
13                 if safemode then
14                     Wait for all suppliers to finish, then query if any are
                       failed. If one or more are failed, throw exception.
15                 end
16                 current_chain.close;
17                 current_chain.set_unaccountable;
18             end
19         end
20         processor.queue.pop;
21     end
1  Procedure execute(Feature_request req)
2      while req.body.top != Void do
3          current_call := req.body.top;
4          req.body.pop;
5          if processor.is_failed then
6              –Only execute calls if the processor is not failed
7              continue; –Go to the next iteration of the while loop.
8          end
9          if current_call.target = processor then
10             Execute locally –This can throw an exception
11         else if current_call.is_lock_passing_call then
12             place_call(current_call, current_call.target); –Places the call on
               the target processor
13             Wait for result, and periodically check for callbacks.
14         else if current_call.is_separate_synchronous_call then
15             place_call(current_call, current_call.target);
16             Wait for result (do not check for callbacks).
17         else
18             place_call(current_call, current_call.target);
19             –Do not wait, call is asynchronous
20         end
21         if current_call.is_separate_synchronous_call ∧
           current_call.target.is_failed then
22             throw exception;
23         end
24     end
```

**Algorithm 5:** Processor loop with support for the accountability mechanism

not illustrate a real-world problem. You can find these programs in the *Maude Tests* folder in our delivery.

We developed another six programs to test basic functionality, but this time we tried emulating common problems, such as trying to read a non-existent file. These programs can be found in the *Small Tests* folder.

A testing process that only tests small parts of the overall solution is not sufficient. We changed two basic SCOOP examples to rely on exceptions instead of other functionality. You can find these programs in the *SCOOP modified Examples* folder. We implemented every example four times, once for every exception mechanism. The exception mechanisms have functional differences that make it impossible to write code that works with all four of them.

We implemented three more large projects to test very specific weaknesses of an exception mechanism. The idea was to see what effort a developer would have to undertake to make a certain program still work even with the "wrong" exception mechanism. You can find these files in the *Large Tests* folder. Again every project is implemented four times, once for every exception mechanism.

# Chapter 5

# Evaluation

This chapter presents an evaluation of the exception mechanisms. The mechanisms were evaluated for usability, expressiveness and performance. An exception mechanism is considered *expressive* if it can be used to model many different types of problems without considerable workarounds by the developer. *Usability* is defined as how easily a developer can use the mechanism.

We implemented the following examples on every exception mechanism, and use them to gauge the expressiveness and usability of the exception mechanism.

1. Worker-aggregator, see Listings B.1 and B.2.
2. Filesystem, see Listings B.3, B.4 and B.5.
3. Producer-consumer, see Section 8.2.1 for building the solution.
4. Barbershop, see Section 8.2.1 for building the solution.

The *producer-consumer* as well as the *barbershop* example have one central server that is shared by multiple users. In the *producer-consumer* example, the central server is a bounded buffer that is used by both producers and consumers. Everyone tries to lock the shared buffer, and then either produce or consume an element. The buffer throws an exception if someone tries to consume from it even though it is empty or if someone tries to insert an element even though it is full. The example demonstrates the benefit of a failed supplier becoming unfailed: If the buffer would no longer accept calls from a producer or consumer, then the system would deadlock. Since there can be only one instance of the buffer, a client cannot simply create a new buffer and continue working on it.

In the *worker-aggregator* example, several workers are given some asynchronous tasks, but the original client will never again interact with them. Instead, a second client will later query the workers. The idea is that the first client is unable to handle exceptions from the tasks, and instead the second client should receive the exceptions. We came up with this example when we tried to find programs that would reveal weaknesses in exception mechanisms.

The *filesystem* example tries to combine the difficulties from the other examples. Several workers share a central service, which they use to access files. However, the workers must be able to query the service in a non-blocking way. Since SCOOP blocks whenever a result is not available, the workers can't simply query the service. Instead, the service accepts work, but then sends any answers to a separate answer register. The workers query the answer register, which will immediately return the file contents, an empty string (if no content is available currently) or an exception (if there was an exception when reading the file). This example is not handled *correctly* by any of the exception mechanisms, some workarounds are always required. That makes the example a good opportunity to study how easily the exception mechanism can be adapted to fit a situation it was not originally intended for.

## 5.1 EiffelSoftware Exception Mechanism

### 5.1.1 Usability and Expressiveness

From a theoretic point of view, the EiffelSoftware exception mechanism is quite easy to use. It is clear who will receive exceptions, and at first it looks quite robust. The implementation of the mechanism has problems when it comes to certain types of SCOOP programs however:

1. Since the supplier saves an exception indefinitely, this can lead to some problems for the client: A client might not always be able to handle all possible exceptions from earlier calls to the supplier. For example, a client might use the same supplier for multiple purposes. It will place a workload on the supplier, and later try to place another workload on the client, then receiving an exception. Developers using the current implementation must therefore always make sure that any exception the supplier might throw from earlier workloads can always be handled. A program showing this problem can be found in Listing 5.1.

2. Another problem of the supplier indefinitely saving exceptions is that it can never become unfailed. This means for the client that once the supplier has an exception, it can never use the supplier again. The client must recreate the entire supplier again, adding a lot of complicated code. It might not even be possible to simply create a new supplier, for example if the supplier uses the singleton pattern. The singleton pattern dictates that at most one instance of the class can exist, and no further instances may be created.

3. It is also possible to lose exceptions. For example, a client places a call on the supplier, and later a second client queries the supplier for the result. In that example, the first client never again interacts with the supplier, so the exception is never propagated and thus lost. A program showing this can be found in Listings B.1 and B.2

The EiffelSoftware exception mechanism performs well in systems where a client does not need to interact with a failed supplier again. Generally every

Listing 5.1: *Eiffel*: Past exceptions class

```
1  note
2    description: "Clients must be able to handle any prior
         exceptions from their suppliers."
3
4  class PAST_EXCEPTIONS
5
6  create
7    make
8  feature
9    make
10     local
11       l_worker: separate WORKER
12     do
13       create l_worker.make
14       a (l_worker)
15       b (l_worker)
16     end
17
18   a (a_worker: separate WORKER)
19     do
20       a_worker.async_call_fail  --This places an
             asynchronous call on the worker, which will later
             fail.
21     rescue
22       --Since there is no further interaction with a_worker,
             this rescue clause is never used
23     end
24
25   b (a_worker: separate WORKER)
26     do
27       a_worker.sync_call    --This places a synchronous call
             on the worker, which would complete normally. But
             due to the earlier failing call, the worker will
             instead return an exception here!
28     rescue
29       --This rescue clause must be able to handle the
             exception that was caused by the failing call
             placed on a_worker in a(). A behaviour like this
             makes the program unnecessarily complicated and
             hard to follow.
30     end
31
32 end
```

program where a failed supplier can be discarded can be modeled using this exception mechanism. A good example would be the *worker-aggregator* program, if the aggregator and root class were rolled into one.

To summarize: The mechanism is easy to use, and developers will quickly learn how to effectively use it. Developers will however face difficulty when using the mechanism to construct larger systems: The client must be able to handle all prior exceptions from a supplier, so developers must add many safeguards, increasing code complexity. Interactions between clients and suppliers can be modeled if no exceptions occur, or if after the exception is propagated the client does not have any further business with the supplier. In the case that the client *does* need further communication with a dirty supplier, then the developer must add code to the client, allowing it to construct a new supplier. This issue can greatly increase code complexity or might even be impossible to solve.

## 5.2 Accountability Mechanism without Safemode

### 5.2.1 Usability and Expressiveness

The accountability mechanism requires developers to think in terms of locks: They must know which processor currently locks which other processors, and they must understand that exceptions disappear as soon as these locks are lifted. It is possible that a developer does not wish to work with an exception mechanism that allows exceptions to be lost; For these cases, the accountability mechanism without safemode is clearly a bad fit, and this led to the implementation of the accountability mechanism with safemode, which is evaluated in Section 5.3. Additionally, developers must understand the difference between synchronous and asynchronous calls, since asynchronous calls will never return exceptions.

Exceptions can be forgotten by processors because of design by contract: Even if a supplier fails, it must be able to satisfy its invariant. In other words, it must be able to receive some further work, and thus should be able to forget an exception. If a client releases the lock of the supplier, it can no longer interact with it in any way, and thus it is safe for the supplier to forget the exception. Forgetting exceptions gives some additional guarantees to clients, namely that a processor is unfailed whenever a lock on it is acquired. This means clients do not have to handle exceptions that were caused earlier.

The accountability mechanism without safemode also has problems when it comes to certain SCOOP programs:

1. The mechanism that the supplier forgets any exceptions as soon as the client releases the lock can have some problematic side effects. It can happen that a client locks a supplier, just to place some asynchronous calls on it. In these cases, exceptions will be lost. An example of this behaviour can be found in the *producer-consumer* program: The buffer returns exceptions whenever *produce* or *consume* is not immediately possible. Since exceptions are lost as soon as the lock is released, producers must add

a synchronous call if they want to receive exceptions. Otherwise, not all elements the producers try to insert into the buffer actually end up in the buffer, so in the end the consumers still try to *consume* further elements, while the producers have long stopped producing. This means the program never terminates.

2. It is possible that the client performing the call on the supplier is not actually able to handle exceptions caused by that call. For example, a client places a call on the supplier, and later a second client queries the supplier for the result. In that example, the first client never again interacts with the supplier, so the exception is never propagated and thus lost. A program showing this can be found in Listings B.1 and B.2.

If a system requires a failed supplier to be able to take on new work, then this exception mechanism works very well. For example, *producer-consumer* as well as *barbershop* are two examples where this approach works very well, albeit with minor changes.

In conclusion, the accountability mechanism is slightly harder to grasp the first time it is used, but it pays off by allowing developers to write cleaner code. They no longer have to take indefinitely failed processors as well as exceptions from the past into consideration when coding. This model allows SCOOP processors to truly behave like objects with contracts by leveraging the principles of design by contract. Developers should find this model more intuitive and easier to use, once they have grasped the finer nuances of it.

## 5.3 Accountability Mechanism with Safemode

The accountability mechanism with safemode was invented to be an alternative to the accountability mechanism without safemode. It allows developers to easily code some of the SCOOP programs that are difficult to code when using the accountability mechanism without safemode.

If developers know how the accountability mechanism without safemode behaves, they will be able to use this mechanism without problems. One main point to be remembered is that at the end of the client's body, the client must wait for all suppliers. This means that a client has to carefully plan which jobs it places on its suppliers and which jobs it executes itself. Otherwise, performance will suffer.

This altered mechanism is not without problems, though:

1. Since the accountability mechanism with safemode dictates that at the end of the client body the client will wait for all suppliers, this complicates long-lived entities. When working with those entities, the client is always waiting for its suppliers. An example for this can be found in listing 5.2. This forces the client to lock long-lived entities once, and then pass their locks around to stop them from being unlocked. For these programs, consider using the accountability mechanism without safemode.

Listing 5.2: *Eiffel*: Long lived entities class

```
1  note
2    description: "Long-lived entities with safemode."
3
4  class LONG_LIVED_ENTITIES
5
6  create
7    make
8  feature
9    make
10     local
11       i: INTEGER
12       l_worker: separate WORKER
13     do
14       from
15         i := i
16       until
17         i > 5
18       loop
19         create l_worker.make
20         start_work (l_worker)
21       end
22     end
23
24   start_work (a_worker: separate WORKER)
25     do
26       a_worker.long_work  --This asynchronous call is
             intended to give the worker some work, while the
             application can continue. But since the safemode is
              turned on, the application actually has to wait
             until the worker has finished, meaning this code is
              just as fast as sequential code.
27     end
28
29 end
```

2. It is possible that the client performing the call on the supplier is not actually able to handle exceptions caused by that call. For example, a client places a call on the supplier, and later a second client queries the supplier for the result. In that example, the first client is forced to handle the exception, even though it has no way of doing so.

This exception mechanism works best with programs where an omitted exception could be fatal. A good example would be *producer-consumer*: If the accountability mechanism without safemode is chosen, then extra care must be taken that no exceptions are lost (synchronous query from the producers). With the safemode on, this extra code is no longer needed.

## 5.4 Performance Analysis

Performance is an important aspect of any program. We wanted to gauge how fast the exception mechanisms were when compared to one another.

We present micro-benchmarks showing the performance of individual aspects of the system. The different exception mechanisms made it impossible for us to run meaningful macro benchmarks. We thought about implementing a problem multiple times, once for every exception mechanism. The resulting benchmark would have been unfair though: If we choose a problem that favors one exception mechanism over the other, then any implementation using the other exception mechanisms is less performant. The EiffelSoftware exception mechanism cannot work with a failed supplier, so the only examples left allow failed suppliers to be discarded. The accountability exception mechanism was explicitly constructed to work with failed suppliers. So either way our macro benchmarks would be biased.

The benchmarks were executed under Linux Mint, on an Acer Aspire V3-771G. This laptop features an Intel Core i7 at 2.4 - 3.4 GHz, as well as 16 GB RAM and a 120 GB SSD. Note that SCOOP is not aimed at high-performance parallel machines. It is still important to get an overview of the performance of our system, for example to assess if the benefit of added flexibility is not outweighed by slower performance.

The aim of benchmarking is to investigate the relationships between different exception mechanisms. This investigation can be used to identify bottlenecks in the system. Timings were obtained using the C *gettimeofday* command. This command returns the seconds as well as the microseconds since the Epoch. All benchmark code was inserted by hand to ensure accuracy. The benchmark programs are a number of Eiffel programs specifically designed to test the individual aspects being timed.

You can find the results in Table 5.1. The first column holds the name of the benchmark. Next to it is a shorthand notation of what exception mechanism was used. *Accountability* refers to the accountability mechanism with safemode. *EiffelSoftware* refers to the EiffelSoftware exception mechanism. The column *repetitions* contains how often we measured. You can find the *mean* of all results in the next column. The last two columns contain the 95% and 99% percentile respectively.

The first two rows are the results of the *sync call* benchmark. Synchronous calls are placed by a client on a supplier, with the client waiting for the answer. No exceptions are thrown. This benchmark shows the overhead incurred when using an exception mechanism that is not used. The accountability mechanism is slightly worse off against the Eiffelsoftware implementation by around 6%.

The second pair of rows shows the *async call* benchmark. Here the client places asynchronous calls on the supplier, and then continues without querying the supplier. None of these calls cause exceptions either. Since no changes were made to how asynchronous calls are placed, this outcome was expected.

The next pair shows the time taken to *lock* a supplier processor. There were no changes to this functionality, so an even outcome was expected.

| Benchmark | Mechanism | Repetitions | Mean | $q_{0.95}$ | $q_{0.99}$ |
|---|---|---|---|---|---|
| Sync call | EiffelSoftware | 200k | 18.37 $\mu$s | 4 $\mu$s | 9 $\mu$s |
| Sync call | Accountability | 200k | 19.60 $\mu$s | 7 $\mu$s | 12 $\mu$s |
| Async call | EiffelSoftware | 100k | 0.52 $\mu$s | 1 $\mu$s | 5 $\mu$s |
| Async call | Accountability | 100k | 0.54 $\mu$s | 1 $\mu$s | 4 $\mu$s |
| Locking | EiffelSoftware | 100k | 0.78 $\mu$s | 1 $\mu$s | 6 $\mu$s |
| Locking | Accountability | 100k | 0.77 $\mu$s | 1 $\mu$s | 6 $\mu$s |
| Safemode sync | Accountability | 100k | 62.49 $\mu$s | 64 $\mu$s | 75 $\mu$s |
| Local exception | EiffelSoftware | 100k | 12.03 $\mu$s | 43 $\mu$s | 76 $\mu$s |
| Local exception | Accountability | 100k | 2.09 $\mu$s | 3 $\mu$s | 8 $\mu$s |
| Propagate exception | EiffelSoftware | 100k | 100.82 $\mu$s | 178 $\mu$s | 273 $\mu$s |
| Propagate exception | Accountability | 100k | 17.30 $\mu$s | 22 $\mu$s | 29 $\mu$s |

Table 5.1: The measured results of various benchmarks.

The seventh row shows the synchronisation step when a client finishes executing its feature body. This step only occurs if the safemode is used. The client has to query all suppliers; In this benchmark, there was only one supplier to be queried. The time this step takes might seem quite high, but the step involves the client querying the supplier, and the supplier sending an answer. If an application using the safemode is correctly optimized, this step should not occur too often.

Rows eight and nine show the time taken to for a *local* organized panic. A local panic is measured from the time the exception is raised until the executing processor sets itself dirty or failed. Processors using the accountability mechanism handle exceptions quite fast, by simply storing the exception and setting themselves failed. A Processors using the EiffelSoftware exception mechanism has to do some more work, for example it has to find out its current client, and then set itself dirty towards that client.

The last pair shows how fast an exception can be propagated between processors. This exception propagation includes the time taken for the supplier to have a local organized panic. We chose this measurement because it nicely demonstrates the entire subprocess of receiving, handling and propagating exceptions. Again the accountability mechanism is significantly faster; The client simply checks the failed flag of the supplier, and immediately knows if there was an exception.

To summarize: The accountability mechanism is faster when confronted with exceptions, whereas the EiffelSoftware implementation has a slight advantage if no exceptions are thrown. We believe that the accountability mechanism can reach the same performance as the EiffelSoftware mechanism if it is improved in the future.

You can find these files in the *Benchmarks* folder when building the solution,

see Section 8.2.1.

# Chapter 6

# Improvements

This chapter presents improvements which we developed after the evaluation phase. It explains how we added another mode for the accountability mechanism as well as how we preserved the type of asynchronous exceptions.

## 6.1 Accountability Mechanism Never Forget

As stated during the evaluation of the accountability mechanism with and without safemode, there are still some SCOOP programs that require the developer to manually propagate exceptions. We extended the accountability mechanism further, by adding a mode which we call *never forget*. This new mode can be used to model problems that cannot be modeled when using the accountability mechanism or the Eiffelsoftware exception mechanism.

This mode behaves like the accountability mechanism without safemode, except no exceptions get deleted when locks are released.

### 6.1.1 Evaluation

Since the accountability mechanism never forget does not forget exceptions on lock releases, this brings two important changes:

1. If a client only places asynchronous calls on a supplier, the supplier can become failed, and it will not forget it. If then at some later point a second client locks the supplier and queries it, the second client will receive any exception that was stored earlier, just as intended.
2. This very behaviour also has an important downside: A client can no longer be sure that a supplier is unfailed whenever it is locked. It is now possible that a client will get an exception from a supplier that it can neither handle nor that it caused. Developers will have to remember this downside.

Only use the accountability mechanism never forget when a situation like worker-aggregator (see Listings B.1 and B.2, compare to Listing 6.1) arises. Otherwise, use the accountability mechanism with or without safemode.

### 6.1.2 Performance

The accountability mechanism never forget is a variant of the original accountability mechanism without safemode when it comes to performance. Usually they are on par, but there are some exemptions to that rule. If there are a lot of occasions where the accountability mechanism without safemode would have lost exceptions, then using the new accountability mechanism never forget can lead to a big slowdown, since the exceptions may no longer be lost, but instead propagated and handled.

## 6.2 Exception Types

When evaluating the different exception mechanisms, we found an additional problem that developers will have to face. The type with which exceptions are passed between processors is always changed to a generic *developer exception*. In Eiffel there are many exception types, such as *operating system failure*, or *precondition violation*. This creates a problem, as the example web-exceptions shows: The root class (Listing 6.2) delegates work to the *web supplier* (Listing 6.3), and later queries the supplier. The supplier can either return a result, a *timeout exception* or a *404 exception*. Before our improvement, the supplier either returns a result, or simply a *developer exception*.

The client will have no way of knowing whether the supplier has encountered a timeout or a 404 error. But in this case, such a distinction is vital: If a timeout occurred, the client should simply print a quick message to the user and retry issuing the command. In case of a 404 error the client must show an error message to the user and abort. There is no chance a call that produced a 404 error will succeed the next time it is attempted.

We changed the implementation of SCOOP in such a way that the actual exception is saved, ready to be passed to the client. When the time comes, the SCOOP runtime environment throws the correct exception to the client, and doesn't have to create a *developer exception* from scratch.

This should enable developers to work with exceptions in a more efficient way, they will immediately know what type of exception was passed around.

Listing 6.1: *Eiffel*: Modified aggregator class

```
1  note
2    description: "The aggregator class (modified for never
         forget), which is used to aggregate the results of all
         workers."
3
4  class AGGREGATOR
5
6  create
7    make
8  feature
9    make
10     do
11       create workers.make
12     end
13
14   receive_worker (a_worker: separate WORKER)
15     do
16       workers.put_front (a_worker)
17     end
18
19   collect
20     --Collect results from all the workers.
21     do
22       from
23         workers.start
24       until
25         workers.after
26       loop
27         receive_results (workers.item)
28         workers.forth
29       end
30     end
31
32   receive_results (a_worker: separate WORKER)
33     local
34       b: BOOLEAN
35     do
36       b := w.get_result --This call now yields the exception
              caused ealier by the root class.
37     rescue
38       --And here we can handle the exception.
39     end
40 feature {NONE}  --Internal
41   workers: LINKED_LIST[separate WORKER]
42 end
```

Listing 6.2: *Eiffel*: Exception type demo root class

```
1  note
2    description: "This root class interacts with the web
         supplier."
3
4  class EXCEPTION_TYPE_DEMO
5
6  create
7    make
8  feature
9    make
10     local
11       l_web_supplier: separate WEB_SUPPLIER
12     do
13       create l_web_supplier.make
14       work (l_web_supplier, "http://www.example.com/index.
             html")    --Works on first try, get response
15       work (l_web_supplier, "http://www.far-away.cn/index.
             html")      --Timeout, will work after a while
16       work (l_web_supplier, "http://www.unreachable.com/
             doesnotexist.html") --404 Error
17     rescue
18       --If this rescue clause is reached, the program is
             aborted. This should only happen if a 404 exception
              is encountered.
19     end
20   work (a_web_supplier: separate WEB_SUPPLIER; a_str: STRING
       )
21     local
22       response: BOOLEAN
23       l_exception_404: EXCEPTION_404
24       l_exception_timeout: EXCEPTION_TIMEOUT
25     do
26       response := a_web_supplier.serve_request (a_str)
27       print ("Gotten response. Everything is fine.%N")
28     rescue
29       l_exception_404 ?= (create {ISE_EXCEPTION_MANAGER}).
             last_exception
30       l_exception_timeout ?= (create {ISE_EXCEPTION_MANAGER
             }).last_exception
31       if l_exception_timeout /= Void then
32         print ("Timeout received, simply trying again...%N")
33         retry
34       elseif l_exception_404 /= Void then
35         print ("404 exception received, not handling that
               error.%N")
36       else
37         print ("Unknown error received, not handling it.%N")
38       end
39     end
40 end
```

Listing 6.3: *Eiffel*: Web supplier class

```
note
  description: "The web supplier tries to take URLs and
      deliver responses."

class WEB_SUPPLIER

create
  make
feature
  make
    do
      counter := 1
    end
  serve_request (a_url: separate STRING): BOOLEAN
    local
      l_url: STRING
    do
      create l_url.make_from_separate (a_url)
      --For simplicity we do not actually make web requests,
          but deliver the answer directly.
      if l_url.is_equal("http://www.far-away.cn/index.html")
          and counter <= 3 then
        counter := counter + 1
        exception_timeout.raise
      elseif l_url.is_equal("http://www.unreachable.com/
          doesnotexist.html") then
        exception_404.raise
      else
        Result := True
      end
    end
feature {NONE}
  counter: INTEGER
  exception_404: EXCEPTION_404
    once
      create Result
    end
  exception_timeout: EXCEPTION_TIMEOUT
    once
      create Result
    end
end
```

# Chapter 7

# Duels

One major problem with the locking of SCOOP processors is that while a client locks a supplier, no other processor may interfere. The idea that locks are always held until the client releases them greatly complicates SCOOP programs where one processor should have a higher priority than the others. Duels allow a processor to be unlocked prematurely.

So far developers had two options: Either accept these delays and try to work around them as best as possible or simply make all processors release and reacquire all locks in short intervals. The first solution complicates code. The second approach is also not really a good fit, since it leads to contention of the locks, and therefor performance drops. It also does not guarantee that the important SCOOP processor can acquire the necessary locks.

There is a solution to this problem, however: The duel mechanism allows a client that currently holds the lock for a supplier (called a *holder*) to prematurely release that lock to some other processor that would like to acquire the lock (called the *challenger*). Since the holder and the challenger will fight for the lock, this mechanism is called a *duel mechanism*. It was first mentioned in [3], page 999+.

The mechanism allows a SCOOP processor to have both a holder behaviour as well as a challenger behaviour. The holder behaviour can either be *retain* or *yield*. *Retain* means that locks are not released and is activated by default. *Yield* enables the processor to yield its locks, meaning others can acquire them even if the current processor has not regularly released them. The challenger behaviour can either be *wait_turn*, *demand* or *insist*. *Wait_turn* is enabled per default and means that the challenger will patiently wait until the locks become regularly available, even if the holder is willing to give them up earlier. *Insist* tries to interrupt the holder, but if the holder retains its locks, the challenger waits. Lastly, *demand* is a "now or never" tactic: The challenger tries to interrupt the holder, but if that fails the challenger gets an exception. This is intended for real-time systems, where it is unacceptable for the challenger to wait.

The following table shows all possible conflicts as well as their outcome. Underlined options are activated by default.

| Challenger Behaviour ⇒<br>Holder Behaviour ⇓ | *wait_turn* | *insist* | *demand* |
|---|---|---|---|
| *retain* | Challenger waits | Challenger waits | Exception in challenger |
| *yield* | Challenger waits | Exception in holder's routine | Exception in holder's routine |

## 7.1 Implementation

The implementation was closely integrated into the existing SCOOP runtime system. Every SCOOP processor can simply execute the features *set holder behaviour retain*, *set holder behaviour yield*, *set challenger behaviour wait turn*, *set challenger behaviour demand* or *set challenger behaviour insist* at any place and time to set the corresponding behaviour.

Processors will no longer apply calls from clients that lost their lock. The interesting part about the duel mechanism is how locks are acquired, see Algorithm 6. Whenever a processor would like to lock a supplier, it will query the behaviour of the current holder, and then act accordingly. If there is no current holder, then the lock is acquired immediately.

```
   input: A processor supplier which is to be locked
 1 challenger_behaviour := processor.challenger_behaviour;
 2 current_holder := supplier.client;
 3 if current_holder /= Void ∧ supplier.locked then
 4 |   holder_behaviour := current_holder.holder_behaviour;
 5 |   if (holder_behaviour = retain ∧ challenger_behaviour = wait_turn) ∨
   |   (holder_behaviour = retain ∧ challenger_behaviour = insist) ∨
   |   (holder_behaviour = yield ∧ challenger_behaviour = wait_turn) then
 6 |   |   Wait until supplier.locked is false, then try again
 7 |   else if holder_behaviour = retain ∧ challenger_behaviour = demand
   |   then
 8 |   |   throw exception;
 9 |   else if (holder_behaviour = yield ∧ challenger_behaviour = demand)
   |   ∨ (holder_behaviour = yield ∧ challenger_behaviour = insist) then
10 |   |   current_holder.cancel_lock –Makes sure the current holder has lost
   |   |   the lock for the supplier;
11 |   |   Acquire lock;
12 |   end
13 else
14 |   Acquire lock;
15 end
```

**Algorithm 6:** Acquiring a lock, executed by the challenger processor

## 7.2 Evaluation

The duel mechanism works as intended. It allows developers to easily implement models that call for processors which yield their locks prematurely. The mechanism has further potential, which is discussed in Section 10.2.

Listings 7.1 and 7.2 give a nice example of how the duel mechanism can be used: The *boss* will issue constant work to the *secretary receptionist*. There will also be a *customer* that comes in and interrupts the *secretary receptionist* every once in awhile. In order for the *customer* to be able to interrupt, the *boss* will *yield* the locks it holds and the *customer* will *insist* upon obtaining locks.

## 7.3 Testing

We developed seven additional tests to test the functionality of the duel mechanism implementation. You can find these programs in the *Duel Tests* folder when building our solution, see Section 8.2.1.

The tests for the duel mechanism consist of six small programs that test the base cases of two processors dueling with each other. The seventh program is the real-world *Secretary Receptionist* example, taken out of [3], page 1001. In this program, a *Boss* and a *Customer* duel for the *Secretary Receptionist*'s attention. The boss has a lot of work for the secretary, but every now and then a customer enters with a quick query. Since customers are important and often impatient, the boss *yields* in favor of a customer, so that the secretary can temporarily turn into a receptionist and serve a customer's request. As soon as the customer is served, the boss tries to assign work to the secretary again.

Listing 7.1: *Eiffel*: Boss class

```
1  note
2    description: "The Boss will periodically issue work to the
         secretary, but it will allow the secretary to be
        interrupted by customers."
3  class
4    BOSS
5  create
6    make
7  feature
8    make (sr: separate SECRETARY_RECEPTIONIST)
9      do
10       secretary_receptionist := sr
11       placed_work := 0
12
13       set_holder_behaviour_yield  --The BOSS is willing to
             yield locks to challengers that either insist or
             demand locks.
14     end
15   live
16     do
17       from
18
19       until
20         placed_work >= 100000
21       loop
22         issue_work (secretary_receptionist)
23       end
24     end
25   issue_work (sr: separate SECRETARY_RECEPTIONIST)
26     --Will work with the secretary until interrupted
27     local
28       b, interrupted: BOOLEAN
29     do
30       if interrupted then
31         -- BOSS has been interrupted, will try to regain
               lock.
32       else
33         from
34
35         until
36           placed_work >= 100000
37         loop
38           b := sr.do_work
39           placed_work := placed_work + 1
40         end
41       end
42     rescue
43       interrupted := True
44       retry
45     end
46   secretary_receptionist: separate SECRETARY_RECEPTIONIST
47   placed_work: INTEGER
48 end
```

Listing 7.2: *Eiffel*: Customer class

```
1  note
2    description: "The customer will try to interrupt the
         receptionist in order to get some quick info."
3  class
4    CUSTOMER
5  create
6    make
7  feature
8    make (sr: separate SECRETARY_RECEPTIONIST)
9      do
10       secretary_receptionist := sr
11       placed_work := 0
12
13       set_challenger_behaviour_insist --The CUSTOMER will
             insist upon locks, thus it will interrupt lock
             holders willing to yield locks
14     end
15   live
16     do
17       from
18
19       until
20         placed_work >= 100
21       loop
22         interrupt (secretary_receptionist)
23         wait  --Wait for some time, to simulate the customer
               only coming in once in awhile
24       end
25     end
26   interrupt (sr: separate SECRETARY_RECEPTIONIST)
27     --Just a quick query
28     local
29       b: BOOLEAN
30     do
31       b := sr.do_work
32       placed_work := placed_work+1
33     end
34   secretary_receptionist: separate SECRETARY_RECEPTIONIST
35   placed_work: INTEGER
36 end
```

# Chapter 8

# Guides

## 8.1 User Guide

This user guide is intended for developers looking to make use of the various SCOOP exception mechanisms. It is not intended for developers looking to change parts of the implementation. If you wish to change the implementation, have a look at the developer's guide in Section 8.2.

### 8.1.1 Choosing your Exception Mechanism

Should you wish to choose or change the exception mechanism used in your project, there is an easy way to do this. You can also change the exception mechanism of a project later, by using the same few steps:

1. Open EiffelStudio and highlight your project.
2. Now click on the "Edit Project" button; a new window will pop up.
3. On the left of the new window, select the "Advanced" entry in the list.
4. On the right you will now see several configuration options appear. Look for the two saying "SCOOP Safemode" and "SCOOP Never Forget".
   (a) If you wish to use the accountability mechanism without safemode or never forget, simply set both of these entries to "False".
   (b) If you wish to use the accountability mechanism with safemode, set "SCOOP Safemode" to "True", and "SCOOP Never Forget" to "False".
   (c) If you wish to use the accountability mechanism with never forget, set"SCOOP Safemode" to "False", and "SCOOP Never Forget" to "True".
5. Click on the "OK" button on the lower right. The window will close.

6. Now check the "Clean" checkbox. Also select "Compile" from the "Action" dropdown list. This forces a recompile. You should always recompile your project after changing the exception mechanism.

There is no way to currently set the exception mechanism when creating a new project. The exception mechanism will default to the accountability mechanism without safemode. If you wish to create a new project with a different exception mechanism, first create the project with the default exception mechanism, then close EiffelStudio. Now follow the instructions above to select your exception mechanism.

### 8.1.2 Quick Recap of Some Important SCOOP Facts

When using any of the available accountability mechanisms, there are a few important things that should always be remembered:

1. A client can only place calls on a supplier if the client holds a lock of the supplier.
2. While a client holds a lock on a supplier, no other SCOOP processor may place calls on the supplier or the client. So if possible, do not hold these locks longer than necessary.
3. Commands that the client places on the supplier allow the client to continue without waiting. These calls are asynchronous. Commands are calls which do not return a result, and which do not have any of the client's held locks as arguments.
4. Queries that the client places on the supplier make the client wait until the supplier answers. These calls are synchronous. Queries always have a return value.
5. If the client places a lock-passing call on the supplier, then the client must also wait for the supplier. As such, these calls are synchronous, no matter whether they are commands or queries. Lock-passing calls use one or more of the locks the client holds as an argument.
6. Whenever possible, handle exceptions in the supplier, without passing them to the client. This is faster and allows the client to continue undisturbed. If you can't handle exceptions in the supplier, make sure the client has the corresponding exception handling necessary to handle them.

## 8.2 Developer Guide

This developer guide is intended for developers who want to change the behaviour of the SCOOP exception mechanism.

### 8.2.1 Where to Find and Build the Solution

It is required that you have a working version of EiffelStudio (not EVE) installed on your system. You can download EiffelStudio from

http://www.eiffel.com/products/studio/

We used EiffelStudio 7.3.92665. You may want to get a version as close as possible to this release, or your results might not exactly match ours. Once you have EiffelStudio set up and working you must download EVE. Our code only extends EVE, and cannot run on its own. You can find a tutorial about EVE here:

https://trac.inf.ethz.ch/trac/meyer/eve/wiki

Our code is based upon revision 92789, and we recommend getting exactly that revision. There may be errors when merging EVE with our code files if you use a different version, proceed with caution. After you have downloaded EVE, download our extension:

http://se.inf.ethz.ch/student_projects/florian_besser/

Our source code is delivered in the form of a patch. Please extract the downloaded archive, move to your EVE Src directory and then use this command to merge EVE with our patch:

```
patch -p0 -i path/to/thesis.diff
```

Then continue compiling and running EVE as described in the tutorial.

### 8.2.2 SCOOP Framework

We observed some interesting implementation details of which you should be aware:

1. The features of *ISE SCOOP MANAGER* are executed at a time where you cannot create new objects. If you need any new objects, for example some strings to write stuff to a logfile, you have to create them on the C side and not in Eiffel!

2. Chains as well as various other structures are represented as arrays. If you want to know what an entry in the array symbolizes, search the class *ISE SCOOP MANAGER* for constants which end in *_index*, such as *request_chain_accountable_index* with value 5. These constants will tell you what a specific array element represents. In the example, the 6th entry (arrays are zero-based here) tells you whether the chain is accountable or not.

3. If you want to add or remove entries from these arrays, you can usually find a constant that holds the array size. Search the class *ISE SCOOP MANAGER* for constants which end in *_index_count*, such as *processor_meta_data_index_count*. If you change such a constant, all arrays that use it will have the correct number of entries.

4. The class *ATOMIC MEMORY OPERATIONS* provides some helper features for setting and changing variables atomically. If you need some examples how to use the functionality, have a look at *flag_chain_accountable* in *ISE SCOOP MANAGER* for setting values or look at *is_chain_accountable* in the same class for reading them.

5. There is a large C side to the Eiffel Framework. Many features use direct C code and are not written in Eiffel. There is a gap between C and Eiffel, which is bridged by callbacks. An example would be the feature *scoop_task_callback* in class *ISE SCOOP MANAGER*: The C side will call this feature whenever tasks are to be completed. From the Eiffel side these calls magically appear, and you cannot easily trace from where the callback came. So if you do not find any Eiffel code that invokes a certain routine, you have to search the C side for a point where a callback to Eiffel is made.

6. Nearly every tasks that the SCOOP runtime system has to perform, such as adding a call to a processor or creating a chain, will start with a callback from the C side to the feature *scoop_task_callback* in class *ISE SCOOP MANAGER*. This is a good entry point to see the workflow on the Eiffel side.

### 8.2.3 Accountability Mechanism without Safemode

There are three main places where parts of the exception mechanism are implemented, all of them inside the class *ISE SCOOP MANAGER*:

1. *scoop_application_loop*: This loop is executed by every processor. The processor finds its current chain, and then executes this loop to work on said chain. In the loop, the supplier gets the oldest call that was placed on it and tries to execute it. Additionally, this feature handles any communication with the client, should the need arise. Our changes focus on the *rescue* clause of this feature: Here, we implemented the behaviour that exceptions are only stored if the chain is accountable.

2. *signify_end_of_request_chain*: This feature is executed by the client to close a chain. In the implementation by EiffelSoftware this feature just sets the chain status to *closed*, and nothing more. We amended this feature to close the chain and set it unaccountable. The client also deletes any exceptions the supplier processors might have stored here.

3. *log_call_on_processor*: The client uses this feature to log a call on a supplier. The feature is quite complicated and does a lot more than the name suggests. This feature also handles the entire execution of callbacks on the client as well as when exactly the client checks if the supplier is failed! In our implementation asynchronous calls never yield exceptions for the client. Additionally, the client checks after a synchronous call whether the supplier is failed, and so it receives the exception directly.

### 8.2.4 Accountability Mechanism with Safemode

Everything covered in the section above also applies to the accountability mechanism with safemode. This section presents the changes made from the accountability mechanism without safemode, to give you a good overview on how you could change the safemode implementation.

At the end of the client feature body, there is a query to all suppliers. The client uses a direct callback from C into the *ISE SCOOP MANAGER*.

Here is how the callback was achieved on the C side:

1. We implemented a new SCOOP task for the end of a feature body: We defined the task *scoop_task_feature_body_end* both in *ISE SCOOP MANAGER* as well as in *eif_macros.h* (since the C side must also know what the task ID is). We created a new C macro called *EIF_FEATURE_BODY_END* in *eif_macros.h* to be used for the callback.

2. We modified the Eiffel classes *BYTE CODE* as well as *STD BYTE CODE* which are used to generate C code from Eiffel. If you do a search for *safemode* you will find all the modifications which we made. All these modifications make sure that the callback is injected just at the end of the client's feature body. Of course, that additional code is only injected if safemode is turned on.

And these are the changes we made to the *ISE SCOOP MANAGER* to handle the callback:

1. We updated *scoop_manager_task_callback* so that the new task ID is recognized. Whenever a processor receives the feature body end task ID, it executes a call to *feature_body_end*. This feature grabs the client's current chain, which contains all suppliers. It then sends a signal to every supplier to finish up and report and waits for an answer. Once an answer is received, the client checks for any exception from the suppliers. If an exception occurred, the exception is raised on the client.

2. We also had to make sure that no exceptions were lost before a callback to *feature_body_end* was triggered. So we additionally modified the features *signify_end_of_request_chain* and *scoop_application_loop*.

3. In *signify_end_of_request_chain* we added a conditional that if the safemode was enabled, no exceptions may be dropped.

4. In *scoop_application_loop*, there were two changes. First, we added a conditional in the rescue clause so exceptions are always stored if safemode is enabled. Secondly, we had to implement a mechanism that allows the client to wait until the supplier is finished. We added a conditional for synchronisation: If a supplier finishes all its work while safemode is enabled, then it will wait for a special signal from the client to report any exceptions.

### 8.2.5 Duel mechanism

We implemented new features that the developer can use in his code to set holder and challenger behaviour. These features involve callbacks, so we implemented the functionality to handle these callbacks in *ISE SCOOP MANAGER*. This class also contains our changes for using the behaviour set by the developer.

The following changes allow setting a processors behaviour:

1. In the class ANY we implemented *set holder behaviour retain*, *set holder behaviour yield*, *set challenger behaviour wait turn*, *set challenger behaviour demand* and *set challenger behaviour insist*, which directly use macros defined in *eif_macros.h*.
2. These macros automatically execute a callback with new SCOOP task IDs *scoop_task_holder_behaviour* or *scoop_task_challenger_behaviour* (defined in *eif_macros.h*) to the class *ISE SCOOP MANAGER*.

We then changed the implementation of *ISE SCOOP MANAGER* as follows:

1. We changed *scoop_manager_task_callback* so that the new task IDs are recognized. Depending whether holder or challenger behaviour is set, *set_holder_behaviour* or *set_challenger_behaviour* is executed.
2. The SCOOP processors did originally not have any flags for holder or challenger behaviour, so we had to create them. We added two entries to *processor_meta_data*. These two entries are used to store the desired behaviour of the processor. The two new entries are at index *processor_holder_behaviour_index* and *processor_challenger_behaviour_index*, respectively.
3. We implemented the logic of obtaining locks in *request_processor_resource*. The challenger uses the supplier's chain to find the current lock holder. The challenger then extracts its own challenger behaviour as well as the holders holder behaviour, and proceed accordingly.
4. There was a slight problem interrupting the holder: It is not possible to throw an exception to a processor, it is only possible to let the processor discover it. To implement this, we either require every processor constantly check for new exceptions (which is a big performance hit) or have the processor only check occasionally (meaning there is a race condition that two clients might access the same supplier). The solution was to add a new status for chains. They can now be in a state called *lock lost*. If the holder loses its lock, then the holder's chain receives the status *lock lost*.
5. To make sure that the holder actually sees the changed chain state, we changed the implementation of *scoop_application_loop*. The processor queries the chain state before applying any features: If the chain state is *lock lost*, the processor throws an exception to itself.

### 8.2.6 Integrating the Accountability Mechanism into EiffelStudio

In order for developers to be able to select which exception mechanism to use, we had to change the Eiffel project definition, the Eiffel parser and a few auxiliary files.

These changes were necessary to change the project definition:

1. *CONF VALIDITY*: Changes here allow us to parse project files which contain a flag for safemode or never forget.
2. *CONF TARGET*: By adding features here we can set if a project uses the safemode or never forget system.
3. *CONF CONSTANTS*: This file contains the name and the description for the safemode and never forget system.
4. *CONF INTERFACE NAMES*: This file contains the actual entries to the EiffelStudio interface.

And these changes were applied to the parser:

1. *TARGET PROPERTIES*: This file contains the flags for the safemode and never forget system on the Eiffel side.
2. *SYSTEM I*: This file contains new features to set if a project uses the safemode or never forget mechanism.
3. *LACE I*: This class is used to read out a project, then input any settings such as the safemode or never forget entries into the system using the features from *SYSTEM I*.

There were some changes to other files necessary:

1. *eif_project.h*: This file contains the flags for the safemode or never forget system on the C side.
2. *eif_project.c*: And this file contains the documentation for the flags, and makes them globally available.
3. *AUXILIARY FILES*: This file contains C calls to set the two flags defined in *eif_project.h*.
4. *ISE SCOOP MANAGER*: The processors query if the safemode or never forget system ar on here (at runtime), and cache the result to improve efficiency.

# Chapter 9

# Related Work

Compton et al. [5] present a first implementation of a runtime system for SCOOP. The runtime system discussed in the paper includes a simple exception mechanism that treats any asynchronous exception as fatal. Thus the problem of transferring exceptions between processors is simply solved by halting the entire system.

Morandi et al. [1] first defined the accountability mechanism. Their specification is the basis for the implemented accountability mechanism. The paper also includes a discussion about different asynchronous exception mechanisms and can be used to get an overview of the subject.

Morandi et al. [2] defined the operational semantics of SCOOP in their paper. The paper does not discuss exception mechanisms, but instead gives a formal definition of how SCOOP should behave.

# Chapter 10

# Conclusions and Future Work

## 10.1 Conclusions

*Implementation*: We successfully implemented the accountability mechanism both with and without safemode as specified in [1]. The implementation is integrated into EVE.

*Evaluation*: We evaluated all flavours of the accountability mechanism as well as the implementation done by EiffelSoftware.

*Evaluation programs*: We wrote several SCOOP programs to evaluate the different accountability mechanisms. These programs highlight different problems for different exception mechanisms.

*Improvements*: We improved the accountability mechanism by adding another mode to it, called never forget. The new mode allows developers to model some SCOOP programs that could not be modeled using the other modes of the accountability mechanism. Additionally, developers now always receive the original exception, which should help them debug their applications.

*Duel mechanism*: The duel mechanism requires a well-defined underlying exception mechanism. Since we implemented several exception mechanisms, we built upon them and added the duel mechanism on top. With this mechanism several processors can now duel for locks, which allows the premature transfer of locks from one processor to another. This simplifies the process for developers to write SCOOP programs where reaction time as well as performance is important.

## 10.2 Future Work

Our work on different exception mechanisms and the duel mechanism has uncovered some yet unanswered questions. This section will briefly discuss work

that requires further investigation.

### 10.2.1 Asynchronous Callbacks

Callbacks happen whenever a processor A passes its own lock to another processor B, and then B places a callback on A. According to the formal model of SCOOP [2], there can be no *asynchronous* callback, it should instead always be *synchronous*. However, EiffelSoftware implemented this behaviour differently, allowing asynchronous callbacks. In the example above: A receives an asynchronous callback from B, goes into an organized panic but B never queries A. Instead, B finishes its job, and then returns all locks to A. The question is: What should A do now? It might not be possible for A to handle an exception which was caused by a callback from B, and B has finished its job and therefore cannot handle any exceptions.

We tried to solve this issue by having processors that receive callbacks behave just like ordinary suppliers. In other words, if they get an exception, they store it, and return it whenever the client tries to query them. Should the client return all locks to them, they forget the exception. But it is clear that our way of handling this problem is but one of many, and may not be optimal.

### 10.2.2 Can Processors Become Unfailed?

Another interesting question is whether a processor is unfailed. In the implementation by EiffelSoftware the answer is simply *no*. The accountability mechanism allows a processor to become unfailed in two ways: Either if the lock of that processor is released or if the processor manages to propagate the exception.

This topic asks some interesting questions: Can a failed processor receive work again? And if so, what conditions must be fulfilled? We believe that by using the powers of *Design by Contract* we can nicely answer these questions. If an exception occurs, the processor executes the current feature's rescue clause, restoring the object's invariant. In other words, that object can now receive further calls, even if an exception occurred. The client must still be informed of the exception, but as soon as that is done, new calls can be placed on the object.

### 10.2.3 How to Handle Holder and Challenger Behaviour for Duel Mechanism

Our approach gives every processor its own behaviour, just as described by B. Meyer in [3]. Since behaviour does usually not change often, we think this implementation is viable. While our approach works nicely and allowed us to demonstrate a proof-of-concept, developers might want more fine-grained control. For example, future implementations could make it possible that a developer can modify the holder behaviour for every lock a processor holds. A good middle way can be to allow the developer to set the desired behaviour on a per-feature basis.

### 10.2.4 Timeouts in Duel Mechanism

Some challengers might be able to wait only for a certain time, but then they either need the lock or an exception. This problem could be solved by timeouts, after which the challenger automatically tries to acquire the lock using the *insist* behaviour.

### 10.2.5 Priorities for Different SCOOP Processors

In some programs different processors have different priorities, just like threads have different priorities. Priorities for different processors would allow developers to easily design a system where locks are always released for high-priority challengers, but not for low- or medium-priority challengers. A good example would be an application that has a GUI and many different processors working on tasks. The processor responsible for keeping that GUI responsive should never have to wait for locks and thus should have a high priority.

# Appendix A

# References

[1] B. Morandi, S. Nanz, and B. Meyer. Who is Accountable for Asynchronous Exceptions? APSEC '12 Proceedings of the 2012 19th Asia-Pacific Software Engineering Conference, Volume 01, Pages 462-471

[2] B. Morandi, S. Nanz, and B. Meyer. A comprehensive operational semantics of the SCOOP programming model. Arxiv preprint arXiv:1101.1038, 2011.

[3] B. Meyer, Object-Oriented Software Construction, Second Edition, Prentice Hall, 1997.

[4] C. Cachin, R. Guerraoui, and L. Rodrigues, Introduction to Reliable and Secure Distributed Programming, 2nd ed. Springer, 2011.

[5] M. Compton and R. Walker, A Run-time System for SCOOP, Journal of Object Technology, Vol. 1, No. 3, 2002.

# Appendix B

# Code Listings

Listing B.1: *Eiffel*: Worker aggregator class

```
1  note
2    description: "This root class issues work to different
         workers, and then passes them to an aggregator for
         later querying."
3
4  class WORKER_AGGREGATOR
5
6  create
7    make
8  feature
9    make
10     local
11       i: INTEGER
12       l_aggregator: separate AGGREGATOR
13       l_worker: separate WORKER
14     do
15       create l_aggregator.make
16       from
17         i := 1
18       until
19         i > 5
20       loop
21         create l_worker.make
22         start_work (l_worker)
23         pass (l_aggregator, l_worker)
24         i := i+1
25       end
26       collect(l_aggregator)
27     end
28   start_work (a_worker: separate WORKER)
29     do
30       a_worker.async_call_fail  --This places an
             asynchronous call on the worker, which will later
             fail.
31     rescue
32       --Since there is no further interaction with a_worker,
             this rescue clause is never used
33     end
34   pass (a_aggregator: separate AGGREGATOR; a_worker:
         separate WORKER)
35     do
36       a_aggregator.receive_worker (a_worker)
37     end
38   collect (a_aggregator: separate AGGREGATOR)
39     do
40       a_aggregator.collect  --An asynchronous call to the
             aggregator, leaving the application free to
             continue.
41     end
42 end
```

Listing B.2: *Eiffel*: Aggregator class

```
1  note
2    description: "The aggregator class, which is used to
         aggregate the results of all workers."
3
4  class AGGREGATOR
5
6  create
7    make
8  feature
9    make
10     do
11       create workers.make
12     end
13   receive_worker (a_worker: separate WORKER)
14     do
15       workers.put_front (a_worker)
16     end
17   collect
18     --Collect results from all the workers.
19     do
20       from
21         workers.start
22       until
23         workers.after
24       loop
25         receive_results (workers.item)
26         workers.forth
27       end
28     end
29   receive_results (a_worker: separate WORKER)
30     local
31       b: BOOLEAN
32     do
33       b := w.get_result --This call will not yield the
             exception that the worker has encountered. Instead,
              the worker will now return whatever result is
             available, and the developer can't be sure the
             result is actually correct. The developer also has
             no way of finding out that the worker encountered
             an exception at all.
34     end
35 feature {NONE} --Internal
36   workers: LINKED_LIST[separate WORKER]
37 end
```

Listing B.3: *Eiffel*: Querier class

```
1  note
2    description: "The querier class, which places work for the
         dedicated file reader on the shared buffer, then
       queries the answer register."
3
4  class QUERIER
5
6  feature
7    live
8      do
9        from
10       until
11         False
12       loop
13         if not waiting then
14           queue_file (bounded_buffer)
15         end
16         get_results (answer_register)
17       end
18     rescue
19       retry
20     end
21   queue_file (buffer: separate BOUNDED_BUFFER[INTEGER])
22     local
23       boo: BOOLEAN
24     do
25       buffer.put (number * 1000 + counter)
26       boo := b.is_full
27       --In case any exception mechanism other than the
             accountability mechanism with safemode is used, a
             synchronous call must be placed here to receive
             exceptions!
28     end
29   get_results (ar: separate ANSWER_REGISTER)
30     local
31       l_result_from_ar: STRING
32     do
33       create l_result_from_ar.make_from_separate (ar.
             get_results ((number * 1000 + counter).out))
34       --If the EiffelSoftware exception mechanism is used,
             the answer register must never propagate an
             exception, since this would break the system. So
             the EiffelSoftware exception mechanism forces the
             developer to add more code here as well as on the
             answer register, since the answer register has to
             transfer an error message (not an exception) to the
              querier.
35       waiting := (l_result_from_ar.count = 0) --If the
             answer is "", wait further
36     rescue
37       waiting := False    --The answer was an exception, no
             need to wait further
38     end
39 feature {NONE}
40   number: INTEGER
41   answer_register: separate ANSWER_REGISTER
42   bounded_buffer: separate BOUNDED_BUFFER[INTEGER]
43   waiting: BOOLEAN
44 end
```

Listing B.4: *Eiffel*: Dedicated file reader class

```
note
  description: "The dedicated file reader executes requests
      from the queriers, and transmits answers as well as any
       exceptions to the answer register."

class DEDICATED_FILE_READER

feature
  live
    do
      from
      until
        False
      loop
        get_work (bounded_buffer)
        read_file
      end
    rescue
      retry
    end
  get_work (buffer: separate BOUNDED_BUFFER[INTEGER])
    local
      i: INTEGER
    do
      i := buffer.consume
      if not is_queued (i.out) then
        workload.put_left (i.out)
      end
    end
  read_file
    local
      file_name, line: STRING
      input : PLAIN_TEXT_FILE
    do
      if workload.is_empty then
        --No work, exit
      else
        file_name := workload.first
        create input.make_open_read(file_name)
        input.read_line
        line := input.last_string
        put_answer (answer_register, line, file_name)
        workload.remove_first
      end
    rescue
      put_answer (answer_register, "ERROR", file_name)  --
          The queriers won't be able to tell the difference
          between a file that really contained "ERROR" and an
           actual error! This shows the problem of developers
           having to do workarounds.
      workload.remove_first
    end
feature --Helpers
  put_answer (ar: separate ANSWER_REGISTER; line, file_name:
       STRING)
    do
      ar.put_answer (line, file_name)
    end
end
```

Listing B.5: *Eiffel*: Answer register class

```
note
  description: "The answer register receives any answers and
       exceptions that the deicated file reader produces.
      Queriers will query the answer register for these
      answers / exceptions."

class ANSWER_REGISTER

feature
  put_answer (line, file_name: separate STRING)
    local
      l, f: STRING
    do
      create l.make_from_separate (line)
      create f.make_from_separate (file_name)
      output.put (l, f)
    end

  get_results (file_name: separate STRING): STRING
    local
      s: STRING
      res: STRING
    do
      create s.make_from_separate (file_name)
      res := output.item (s)
      if res /= Void and then not res.is_equal ("ERROR")
          then
        Result := output.found_item --Return output
      elseif res /= Void then
        my_exception.raise      --Error encountered
        --If the EiffelSoftware exception mechanism is used,
            the answer register may not throw exceptions to
           a querier, since that would mean the querier
           could never query the answer register afterwards.
            So developers could add a special answer for
           failure, but this would create a conflict if a
           file's contents were exactly the same as the
           special answer...
      else
        Result := ""        --Data not yet received
      end
    end

feature {NONE}
my_exception: DEVELOPER_EXCEPTION
  once
    create Result
  end
output: HASH_TABLE[STRING, HASHABLE]
end
```