Deferred Cancellation

A behavioral pattern

Philipp Bachmann
Institute for Medical Informatics and Biostatistics
Clarastrasse 12
CH–4058 Basel, BS
Switzerland
bachlipp@web.de

ABSTRACT

People who design their own pool of WORKER THREADS [33, pp 290–298] or processes have to consider how to shut down the WORKERs again or how to dynamically adapt the number of WORKERs to varying load. Especially with regard to application termination you may have the choice between an immediate destruction of the pool and a more graceful shutdown. The pattern proposed helps to portably implement such termination and load adaptation mechanisms that assume you voted for the second choice. The main area of application are the internals of ACTIVE OBJECTS [40] and similar designs that delegate work to a pool of threads or processes to execute service requests asynchronously from their actual invocation.

For the pattern proposed we identified usage examples in popular existing applications or libraries.

Both a real world example and sample code accompany the pattern presentation. This sample code is in C++.

The presentation of the pattern follows the style well known from [11] and [44]. This pattern is based upon other patterns. Typographic conventions for references to other patterns are similar to [3]. A Glossary provides thumbnails of many of these patterns.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; D.2.11 [**Software Engineering**]: Software Architectures—*Patterns*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Programming by contract*; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement—*Portability*; D.4.0 [**Operating Systems**]: General—*Microsoft Windows NT, UNIX*

PLoP '08, October 18–20, 2008, Nashville, TN, USA.

General Terms

Design

Keywords

Patterns, Destructor, Actor, Reliability, Portability

> Sir, my need is sore.
> Spirits that I've cited
> My commands ignore.
>
> JOHANN WOLFGANG VON GOETHE: The Sorcerer's Apprentice [23]

1. INTENT

Safely shut down pools of WORKER THREADS [33, pp 290–298] or processes without resource leakages and premature rollback of transactions. The design proposed aims at portability.

2. EXAMPLE

Consider order processing in a restaurant. The customer places his or her order with the waiter, the face of the restaurant to its guests.[1] At the interface between the dining area and the kitchen, which is also the interface between the waiters and the kitchen staff, an orders holder is located. Figure 1 shows a nice example. The waiter attaches the order notice to one of the spring holders, and the next free member of the kitchen staff takes one of the notices queued and prepares the meal.

Now do not look at the whole order processing—instead let us focus on the following two scenarios only:

Cancellation What happens when the restaurant is about to close? Is the orders holder operated differently than during normal service hours? Customers who successfully placed their orders and perhaps waited for some period of time expect to get their food and drinks even though the waiters reject new orders now. When can the kitchen staff leave off work?

Adaptation to Decreasing Load What happens when the noon rush is over? There are fewer customers around now, and they order less demanding food and drinks than for lunch. So some of the cooks can take

[1] This seems to be a common example for ACTIVE OBJECT [40]. One of several sources is [35, p 20].

Figure 1: An orders bar at a restaurant interfaces between waiters and kitchen staff. Note that the capacity is bounded by the finite number of spring holders. Photograph: Thomas Hoof Produkt GmbH [56]

> a break and probably take their own lunch now. Is the orders holder operated differently than during the rush? When can some members of the kitchen staff finally take a break or leave off work? Do customers notice the reduced capacity of the kitchen?

This pattern addresses these scenarios and proposes a general solution to both. It investigates similarities and differences between them.

3. CONTEXT

You manually implement a pool of WORKER THREADS or processes instead of using a ready–made building block provided by some library. This pool is going to be used as an implementation detail in an ACTIVE OBJECT. ACTIVE OBJECTs decouple the invocation of services from their actual execution. ACTIVE OBJECTs manage their own pool of Units of Execution [34, pp 217–221], and the actual execution of the services requested is carried out by these Units. All Units run the same code, it's the data that differs. An example of an ACTIVE OBJECT is a webserver. More sophisticated examples, that also fulfill real–time constraints, are reactive systems.

The code executed by the Units of Execution basically consists of an infinite loop that contains a blocking system call to temporarily yield execution of a Unit until a client asynchronously requests another service from the ACTIVE OBJECT. MONITOR OBJECTs [43] and REACTORs [52, 50, 45] represent these blocking system calls to the Units of Execution within ACTIVE OBJECTs.

Your product should fulfill certain reliability constraints: For example, the application you build probably requires long uptimes.

4. PROBLEM

How to implement destruction of ACTIVE OBJECTs?

ACTIVE OBJECTs need to be shut down when the application receives a termination request. On construction ACTIVE OBJECTs create a pool of Units of Execution. These Units actually make ACTIVE OBJECTs active. But how to recall the Units again once unleashed? Different from e.g. pools of Units of Execution in certain parallel numerical applications, that loop through a number of timesteps known in advance and only may need some coordination with each other, the Units in ACTIVE OBJECTs receive their work from the outside, and the amount of work is not known *a priori*. In some parallel numerical applications the Units can terminate on their own, in ACTIVE OBJECTs they need to be commanded to do so.

The Units of Execution fulfill useful purposes during their lifetime, and the fulfillment of their respective duties takes some time. Some Units may find themselves in the middle of the execution of a transaction, when they receive the recall. Some Units may have acquired expensive resources like memory or even more expensive ones like mutual exclusion locks when they are commanded to terminate. The creation of files is also a form of resource acquisition. Some applications create files for locking purposes for example and write their process identifier into the file. The creation of files externalizes the resource STATE [22], however, and an abrupt termination of the Units may leave such files dangling, which gives a false impression of the state of the application then.

On the other hand, some STATE is virtually expected to be externalized just before termination of an application to allow ACTIVE OBJECTs to seamlessly resume work when they start again. So how to give the Units supposed to terminate an opportunity to still write their STATE to disk?

ACTIVE OBJECTs may have a backlog of work—should this

backlog be simply discarded on destruction of the respective ACTIVE OBJECT? In case you implement the destruction in this way you may need to solve another problem: How do the clients that produced this backlog notice this situation to actually become able to recover from it?

Additionally there might be the need to dynamically adapt the degree of concurrency within the ACTIVE OBJECT to varying load, which means both increasing and decreasing the number of Units of Execution, the latter being a similar case to application termination. As the application is supposed to run uninterruptedly for long periods of time, you do not want resource leaks to emerge from these adaptation mechanisms.

Some programming languages and operating systems assist with the solution of the problem more than others (see e.g. Sections 10.3 and 10.4). Some do not even provide system calls to assist with the solution of these problems. So particularly it is a challenge to *portably* destroy ACTIVE OBJECTS.

5. FORCES

- Resource leaks are evil at runtime, because they accumulate. They are still ugly at termination, consider dangling lockfiles, files with the id of the (now gone) process, or still open connections to other systems, for example.
- Application termination needs to be at least as reliable as startup.
- Application termination is expected to happen quickly once demanded.
- The higher the level of abstraction, the better the portability.
- The way its termination is implemented and the contracts between an ACTIVE OBJECT and its clients [37, pp 331–438] agreed upon on accepting the respective service requests must match.
- While blocking in a system call a Unit of Execution can not even terminate.
- Especially Units of Execution blocking in certain system calls are good candidates to cancel.

Simply unplugging the computer reliably and quickly terminates your ACTIVE OBJECT—but what if it has written a file that is going to be interpreted as a vital sign by the ACTIVE OBJECT the next time it is about to startup, so it finally refuses to start because it assumes there is already another one? An application developer could also be tempted to let the Units simply check a flag whether to process new work or quit—but what happens in a low load situation when there is no new work? The check passed hours ago and the Unit blocks waiting for new work.

6. SOLUTION

Add a special exception class to your code. Identify those calls the concurrent Units of Execution may infinitely block in. Change the implementation of these calls as follows: Let producers throw an instance of the exception class on calling a respective function if and only if a certain flag is set. Let consumers throw on calling a respective function if and only if both the same flag is set and the backlog is empty. On termination set this flag and unblock the Units of Execution, both within the same THREAD SAFE INTERFACE [47] member function.

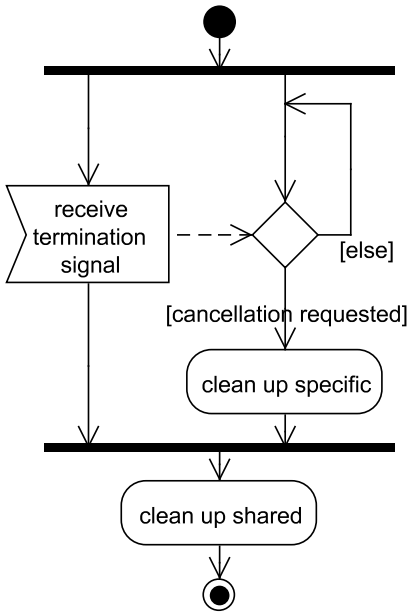

Figure 2: Activity diagram illustrating ACTIVE OBJECT with DEFERRED CANCELLATION. For the sake of readability the concurrency inside was limited to one Unit of Execution.

This solution can be adapted to also apply for the problem of adaptation of the number of Units of Execution to varying load: In this case do not notify producers and let a finite number of consumers throw on calling a respective function if and only if a certain flag is set, but do not take the backlog into account in this test.

A first sketch of the solution is shown in Table 1. Figure 2 sketches the activity of ACTIVE OBJECTS and their destruction. The design of the life cycle phases prior to destruction is not discussed in this pattern, so there are no activities related to these phases in the diagram. Section 11 refers to patterns that are concerned with such activities.

6.1 Participants

ActivationList Queues service requests to ActiveObject until there is a Unit of Execution from PoolOfUnitsOfExecution ready to finally execute the call. ActivationList implements the MONITOR OBJECT pattern, because it is shared among the Units of Execution of LocalClient and PoolOfUnitsOfExecution, and it uses synchronization depending on its STATE. Note that this participant does need to be explicit in case of ACTIVE OBJECT and some HALF–SYNC / HALF–ASYNC server designs [51, 41], but can be left implicit in case of LEADER / FOLLOWERS designs [42], [53, pp 754–756]. ActivationLists can be disabled and indicate this by means of an ActivationListDisabled exception.

ActivationListDisabled Exception indicating not to expect any services from ActivationList.

Table 1: Class–Responsibility–Collaboration Cards

ActivationList	
Hands service requests from	ActiveObject
over to	PoolOfUnitsOfExecution
First unblocks...	RemoteClient and members of PoolOfUnitsOfExecution
...then throws	ActivationListDisabled

(a) Activation List

ActivationListDisabled	
Thrown by	ActivationList

(b) Activation List Disabled

ActiveObject	
Creates	all members of PoolOfUnitsOfExecution
Delegates service requests from	RemoteClients
to	PoolOfUnitsOfExecution
by means of	ActivationList
Disables	ActivationList
Joins	all members of PoolOfUnitsOfExecution

(c) Active Object

LocalClient	
Creates	ActiveObject
Destroys	ActiveObject

(d) Local Client

PoolOfUnitsOfExecution	
Takes service requests from	ActivationList
and executes them.	
Reacts upon	ActivationListDisabled

(e) Pool of Units of Execution

RemoteClients	
Request services from	ActiveObject
React upon	ActivationListDisabled

(f) Remote Clients

ActiveObject Decouples a service request from its actual execution. So service requests are handled asynchronously. ActiveObjects delegate work to a PoolOfUnitsOfExecution it owns.[2] It indirectly hands work over to the pool by means of the intermediate ActivationList. ActiveObject may implement a non–trivial scheduling policy. On destruction ActiveObject first disables ActivationList and then joins the members of PoolOfUnitsOfExecution.

LocalClient The LocalClient owns ActiveObject. First it creates an ActiveObject, and later destroys it.

PoolOfUnitsOfExecution Each Unit takes member function requests out of ActivationList to execute them on behalf of the ActiveObject. The Units handle ActivationListDisabled exceptions. A Unit is called Servant in the original ACTIVE OBJECT pattern.

RemoteClients RemoteClients request services from ActiveObject.

Figure 3 sketches the participants and their relations to each other.

6.2 Dynamics

At startup LocalClient creates ActiveObject, which in turn creates ActivationList and PoolOfUnitsOfExecution. The details were described in [40, p 425] and [6, pp 24–25].

This mechanism can be adapted to the case that by means of some metric LocalClient gets forced to increase the size of PoolOfUnitsOfExecution.

During the lifetime of ActiveObject RemoteClients send service requests to ActiveObject. ActiveObject reifies the requests as COMMANDs [19] and hands the COMMANDs over to ActivationList, its COMMAND PROCESSOR [9]. This also was described in detail before. Here the patterns ACTIVE OBJECT, HALF–SYNC / HALF–ASYNC, and LEADER / FOLLOWERS differ from each other.

The termination of ActiveObject starts with a person or a parent process sending a termination signal to LocalClient. LocalClient in turn disables ActivationList. ActivationList sets a flag and wakes up all Units from PoolOfUnitsOfExecution blocking in a member function of ActivationList to receive new work. Before returning from the member function each Unit checks the flag set by the LocalClient Unit of Execution before. Because it is set, the member function is left now by means of throwing ActivationListDisabled. Each Unit is given the chance to release resources and then terminates. LocalClient then joins the Units of Execution, i.e. it waits for all of them to terminate. Now LocalClient releases its own resources including resources shared among PoolOfUnitsOfExecution and terminates.

The latter mechanism can be adapted to the case that by means of some metric LocalClient gets forced to reduce the size of PoolOfUnitsOfExecution.

The dynamics of DEFERRED CANCELLATION is shown in Figure 4.

[2]For the sake of this paper ActiveObjects include implementations of the HALF–SYNC / HALF–ASYNC and the LEADER / FOLLOWERS pattern, because regarding to cancellation all three are very similar to each other. For simplicity reasons we merged the participants Proxy (a PROXY [21, 12]) and Scheduler of the original ACTIVE OBJECT pattern here.

6.3 Rationale

This pattern is about controlled cancellation of Units of Execution by another Unit of Execution.

On cancellation the ActivationList treats producers and consumers differently: While producers are immediately sent an exception, cancellation of consumers is deferred until the list is empty—which surely happens because production stops. The rationale behind this is to respect the contract between producers that once successfully placed a service request with ActiveObject and the ActiveObject that the request gets eventually executed at all. Here a first deferral happens. On adaptation to varying load, however, producers never get notified. Therefore the backlog will not necessarily clear. So for the desired number of consumers to terminate they must be notified unconditionally instead.

The solution proposed provides for well defined cancellation points: Upon disabling the member functions of ActivationList are going to be unblocked and signal this special situation by throwing an exception. So the Units from PoolOfUnitsOfExecution receive this exception only if they call a member function on the shared ActivationList. It is important to understand, that therefore any transaction that does not involve calls to ActivationList is never affected by premature rollback. Furthermore, even after catching such an exception the Unit is given the chance to clean up. More generally, resources that are both acquired and again released between calls to ActivationList by a Unit from PoolOfUnitsOfExecution do not need special consideration regarding to cancellation; all other resources acquired by a Unit from the pool can be released in a `finally` block (DISPOSE pattern [5, pp 228–230], [38], [1]) or using the Resource Acquisition is Initialization technique [55, pp 388–393], [54, pp 495–497], also known as the EXECUTE–AROUND OBJECT design pattern [25, pp 4–7].

Each Unit from PoolOfUnitsOfExecution is not simply being forced to immediately die. It is given the chance to release resources. Here a second deferral can take place.

The suggested design consists of joinable Units of Execution, not of detached ones. So LocalClient can wait until the last Unit from PoolOfUnitsOfExecution has terminated, before resources shared among the Units and itself are going to be released. If you go for detached Units of Execution instead, combine THREAD SAFE INTERFACE with one of the following smart pointer patterns: COUNTED or DETACHED COUNTED BODY idiom [16, pp 173–179], [32, pp 429–434], [55, pp 841–845] or SHARED OWNERSHIP [15] to protect ActivationList and other shared resources from premature release (see Section 8.2.2 for an example).

Note that the call to ActivationList to disable its operations is asynchronous, too. LocalClient can continue to work with virtually no delay. Later it can rendezvous [40, pp 417,428–430] with the exit statuses of the Units by joining them.

7. RESULTING CONTEXT

Blocking calls in PoolOfUnitsOfExecution were identified. There is a mechanism in place to transmit cancellation requests from LocalClient to the members of PoolOfUnitsOfExecution. All Units of Execution implement means to react upon cancellation requests by releasing resources and finally quitting.

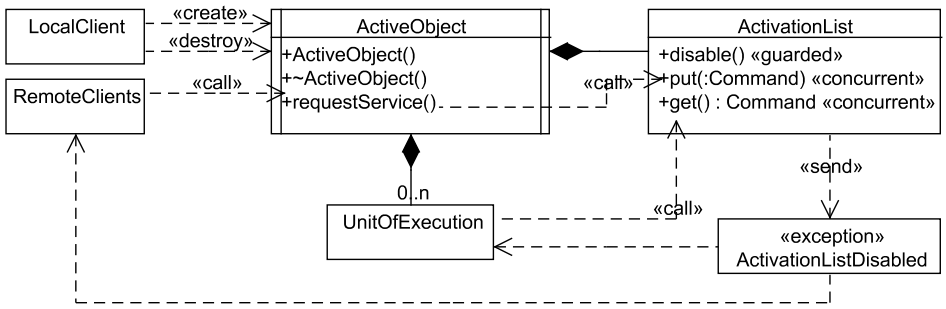

Figure 3: Class diagram illustrating DEFERRED CANCELLATION

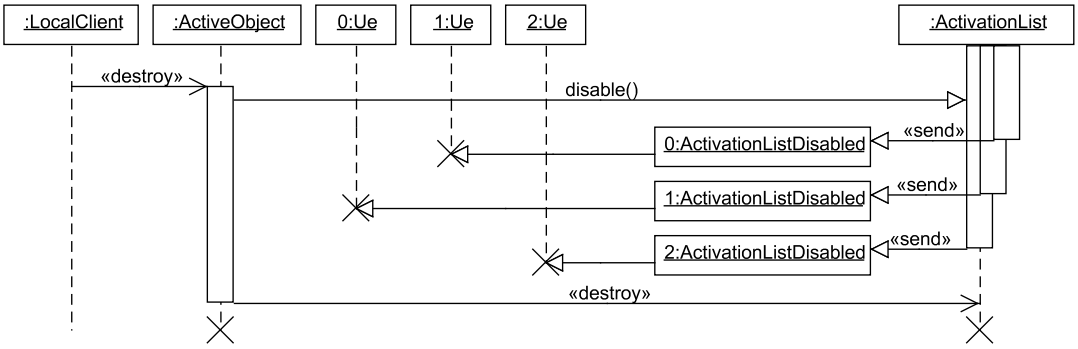

Figure 4: Sequence diagram illustrating DEFERRED CANCELLATION

7.1 Pros and Cons

The DEFERRED CANCELLATION pattern has the following benefits:

1. *Well–defined cancellation points.* The control flows of the working Units of Execution get interrupted at clearly defined calls.

2. *Opportunity granted to release resources.* After receiving a cancellation request the workers can release resources acquired before, be it memory, be it open files, be it any other resource.

3. *Portability.* This pattern works on all platforms that allow for handcrafting the MONITOR OBJECT pattern, the core of ActivationList.

4. *Cooperation.* As pointed out by ANTHONY WILLIAMS, one of the authors of the current Boost.Thread implementation, using exceptions to force termination gives Units from PoolOfUnitsOfExecution great flexibility to react upon such requests [59].

The DEFERRED CANCELLATION pattern has the following liabilities:

1. *Cooperation.* Benefit 4 can also turn into a liability. Each Unit from PoolOfUnitsOfExecution can react upon ActivationListDisabled by ignoring it. As LocalClient only terminates after all Units have terminated, a malicious Unit has the power to veto against termination.

2. *Shutdown can take quite some time.* Even if all Units cooperate, the shutdown of the application takes the time it takes to first shutdown all Units from PoolOfUnitsOfExecution and then shutdown the main Unit. So especially if you can tolerate resource leakages, the application of this pattern might be too much of a good thing.

Additionally to these general pros and cons we identified the following implementation specific ones.

The implementation technique of the DEFERRED CANCELLATION pattern shown has the following liability:

3. *There might be too few cancellation points.* In case of some HALF–SYNC / HALF–ASYNC or all LEADER / FOLLOWERS server designs the blocking system calls are `select()` (a REACTOR) or `accept()`, which you have less control over than regarding to the condition variables aggregated by MONITOR OBJECTS, the core of ActivationList. A workaround is as follows: The controlling Unit of Execution sends a network packet to the port the application listens to. This unblocks the LEADER. Then both the LEADER and the FOLLOWERS may be treated as described above.

8. IMPLEMENTATION

We present two examples in this section. First the restaurant example from Section 2 gets resolved. Then we generalize from the special example and present code that supplements the code shown in the ACTIVE OBJECT pattern.

8.1 Example Resolved

This example focuses on two scenarios. We present a solution for each of these scenarios:

Cancellation The waiters will let the customers know that the restaurant will close soon. They reject any new

orders, so no new notes get attached to the orders holder. Any orders accepted will still be processed by the kitchen staff, until the orders holder is empty. After serving the meals to the customers there is time for a final clean up of the kitchen to make it ready for the next day to come. The worker who leaves last turns off the lights and locks the door.

Adaptation to Decreasing Load The capacity of the kitchen can be adapted to the lesser amount of orders than during the noon rush. Those members of the kitchen staff that are supposed to take a break or leave off work do not simply drop everything. Instead they hand over their operations to their remaining colleagues. This case is similar to the closing of the restaurant in that the concurrency present in the kitchen gets reduced, but it is different from this scenario in that at best the customers do not notice anything of the now decreased capacity of the kitchen. They can order as usual. The orders holder gets operated by both the waiters and the kitchen staff as during the noon rush, only the throughput is different.

In both scenarios a certain kind of deferral takes place: The employees involved do not simply go home when the clock strikes, but some clean up or handover takes place before. This ordered phase out makes these scenarios quite different from really catastrophic emergencies like the collapse of the restaurant building.

8.2 Sample Code

Depending on the platforms the application is required to work on an implementation might rely on features of the operating system to shut down (see Section 10). If this is not possible, you need a way to intercept blocking calls. The following two Sections conform to the description of the pattern in that it steps into blocking calls not at WRAPPER FACADES [49] of condition variables, the lowest possible LAYER [10] of abstraction, but at the higher–level MONITOR OBJECT.

The code is presented as self–contained as possible. Therefore we had to decide on the nature of the Units of Execution: Here we consider threads. To allow for a quick identification of the core of this pattern the most important entities have been underlined, and a comparison between code artifacts and the participants is given in Section 8.3.

8.2.1 Cancellation

The code in Listing 1 shows a very simple ACTIVE OBJECT taken from [40, p 425] without sophisticated scheduling: Every service request gets executed as soon as there is a free Unit of Execution available. Only the cancellation aspect is shown in detail. Other aspects are discussed in [40, p 425].

Listing 1: Cancellability added to scheduler of ACTIVE OBJECT

```
extern "C" {
  void *svc_run(void *);
}

struct Method_Request {
  virtual ~Method_Request();
  virtual void call() =0;
};

struct Message_Queue_Disabled {};

class Message_Queue {
  ...
  std::size_t max_messages_;
  mutable Thread_Mutex monitor_lock_;
  Thread_Condition not_empty_;
  Thread_Condition not_full_;
  volatile bool isActive_;
  Method_Request *get_i();
  void put_i(Method_Request *);
public:
  enum { MAX_MESSAGES = ... };
  explicit Message_Queue(std::size_t
      max_messages =MAX_MESSAGES
  ) : max_messages_(max_messages),
      not_empty_(monitor_lock_),
      not_full_(monitor_lock_),
      isActive_(true),... {
    ...
  }
  Message_Queue(const Message_Queue &rhs)
    : // Condition variables cannot
      // be copied:
      not_empty_(monitor_lock_),
      not_full_(monitor_lock_),
      isActive_(rhs.isActive),... {
    ...
  }
  ...
  bool empty_i() const;
  bool emptyAndEnabled_i() const {
    return empty_i() && isActive_;
  }
  bool full_i() const;
  bool fullAndEnabled_i() const {
    return full_i() && isActive_;
  }
  void disable() {
    Thread_Mutex_Guard
      guard(monitor_lock_);
    isActive_=false;
    not_full_.notify_all();
    not_empty_.notify_all();
  }
  // Transfers ownership
  Method_Request *get() {
    Method_Request *result=0;
    {
      Thread_Mutex_Guard
        guard(std::move(
          not_empty_.wait_if(
            boost::bind(
              &Message_Queue
                ::emptyAndEnabled_i,
              this,
              _1
            )
      )));
      // Note the difference to put().
      if(empty_i())
        throw Message_Queue_Disabled;
      const bool wasFull(full_i());
      result=get_i();
      if(wasFull)
        not_full_.notify_all();
    }
    return result;
  }
  // Transfers ownership
  void put(Method_Request *msg) {
    Thread_Mutex_Guard
      guard(std::move(not_full_.wait_if(
        boost::bind(
```

```
          &Message_Queue
            ::fullAndEnabled_i,
          this,
          _1
        )
      )));
    // Note the difference to get().
    if(!isActive_)
      throw Message_Queue_Disabled;
    const bool wasEmpty(empty_i());
    put_i(msg);
    if(wasEmpty)
      not_empty_.notify_all();
  }
};

class MQ_Scheduler {
  typedef std::vector< thread_type >
    pool_type;
  Message_Queue act_queue_;
  pool_type pool_;
  void joinPool_() {
    for(pool_type::reverse_iterator
          in(pool_.rbegin());
        pool_.rend()!=in;
        ++in)
      joinThread(*in);
  }
public:
  MQ_Scheduler(std::size_t high_water_mark,
               std::size_t
                 number_of_threads)
    : act_queue_(high_water_mark) {
    pool_.reserve(number_of_threads);
    try {
      for(std::size_t i(0);
          number_of_threads>i;
          ++i)
        pool_.push_back(
          createThread(svc_run,&act_queue_)
        );
    }
    catch(...) {
      act_queue_.disable();
      joinPool_();
      throw;
    }
  }
  ~MQ_Scheduler() {
    act_queue_.disable();
    joinPool_();
  }
  // Transfers ownership
  void insert(Method_Request *
                method_request) {
    act_queue_.put(method_request);
  }
};

void *svc_run(void *arg) {
  // Set all signals blocked in the
  // current thread's signal mask
  ...
  assert(arg);
  Message_Queue *act_queue
    =static_cast< Message_Queue * >(arg);
  while(true)
    try {
      // Block until the queue is
      // not empty
      std::auto_ptr< Method_Request >
        mr(act_queue->get());
      mr->call();
    }
    catch(const Message_Queue_Disabled &) {
      break;
    }
    catch(...) {
    }
  return 0;
}
```

Note some details of this implementation:

- `MQ_Scheduler::insert()` acts asynchronously. Here it does not associate a call to it with a Future [40, pp 413, 417,423–430,435–436] that would allow the client to check whether the request has been executed or is still pending. Therefore on cancellation all requests stored in `Message_Queue` still get processed before the pool of WORKER THREADS gets destroyed.

- `Message_Queue::disable()` uses SCOPED LOCKING [46] to acquire and unconditionally release the lock again, an application of Resource Acquisition is Initialization and the EXECUTE–AROUND OBJECT design pattern.

- `Thread_Condition` and `Thread_Mutex` are WRAPPER FACADEs that turn operating system specific imperative interfaces into object oriented ones without impacting performance. WRAPPER FACADEs still give the compiler the opportunity to inline the respective member functions.

- `Message_Queue::{get(),put()}` make use of condition variables equipped with move semantics proposed for future revisions of the C++ standard [27] to become able to return SCOPED LOCKING guards by value from `Thread_Condition::wait_if<>()`, a CONVENIENCE METHOD [28] similar to what Boost.Thread offers. Move semantics can be implemented today with help of the CHANGE OF AUTHORITY idiom [6].

- `MQ_Scheduler::joinPool_()` is incomplete in that it does not call `pool_.clear()`. It is an implementation helper only and is therefore declared `private`.

- The templated constructor calls `std::vector<>::reserve()` to prevent `std::vector<>::push_back()` to throw an `std::bad_alloc` exception. If `thread_type` is a plain old data type in the sense of the C++ standard [4, 3.9.10, 12.8.6], then its built–in copy constructor will never throw. In this case `std::vector<>::push_back()` throws no exception at all.

- `thread_type`, `createThread()`, and `joinThread()` wrap the respective operating system specific types and functions. `createThread()` translates error codes into exceptions.

  From a design point of view `thread_type` is a Future type, and the role of the Rendezvous function is taken by `joinThread()`. `MQ_Scheduler::pool_` is an instance of a Future container, `MQ_Scheduler::joinPool_()` being its Rendezvous function.

- The main loop of the threads in `svc_run()` is in fact a simple example of the LEADER / FOLLOWERS design pattern: The threads line up to become a LEADER, i.e. they are FOLLOWERS. Only one thread at any

one time can take new work out of the queue, i.e. it takes the role of the LEADER. On returning from `Message_Queue::get()` `Message_Queue::monitor_lock_` is being released, thus implicitly a new LEADER gets designated. The former LEADERs can process their work packages concurrently then.

Listing 2 shows how the parts proposed above work together. No emphasis was put on where the service requests actually originate from, i.e. on an useful interface to RemoteClient.

Listing 2: How to use `MQ_Scheduler`

```
namespace {
  std::jmp_buf env_;
} /* (anonymous) namespace */

extern "C" {
  static void SIGTERMDisposition_(int sig)
  {
    longjmp(env_,1);
  }
}

int main() {
  signal(SIGTERM,SIGTERMDisposition_);
  MQ_Scheduler sched;
  std::auto_ptr< Method_Request > mr;
  if(!setjmp(env_)) {
    while(true) {
      // Very rough sketch...
      mr.reset(new Concrete_Method_Request);
      sched.insert(mr.release());
    }
  }
  return EXIT_SUCCESS;
} // On automatic destruction of "sched"
  // first its workerswill terminate
  // after having caught a
  // "Message_Queue_Disabled" exception.
  // Then they are going to be joined
  // before "Message_Queue_Disabled::
  // act_queue_" is destructed.
  // After all, "MQ_Scheduler" vanishes.
```

In an even simpler setting the main thread can simply block in `pause()` until a signal comes in as described e.g. in [53] instead of the above signal handler and its interaction via `longjmp()` and `setjmp()` that have to be used with caution to avoid resource leaks. `mr` was defined outside the `setjmp()` block to avoid it not to be destructed in case of termination.

Figure 5 sketches the participants and their relations to each other. The dynamics of the entities in this example is shown in Figure 6. Note that `Thread_Condition` automatically deactivates the associated `Thread_Mutex` on blocking and activates it again before returning in `Thread_Condition::wait()`.

8.2.2 Dynamic Adaptation to Varying Load

This section extends the example with a mechanism to dynamically shut down some Units of Execution to adapt their number to decreasing load. Reasons for the need for this dynamic adaptation are e.g.

- Threads consume resources even if they block, so they should be limited in number to the expected load.
- Many ACTIVE OBJECTs are I/O bound, so the performance can benefit from more threads than there are processor cores.

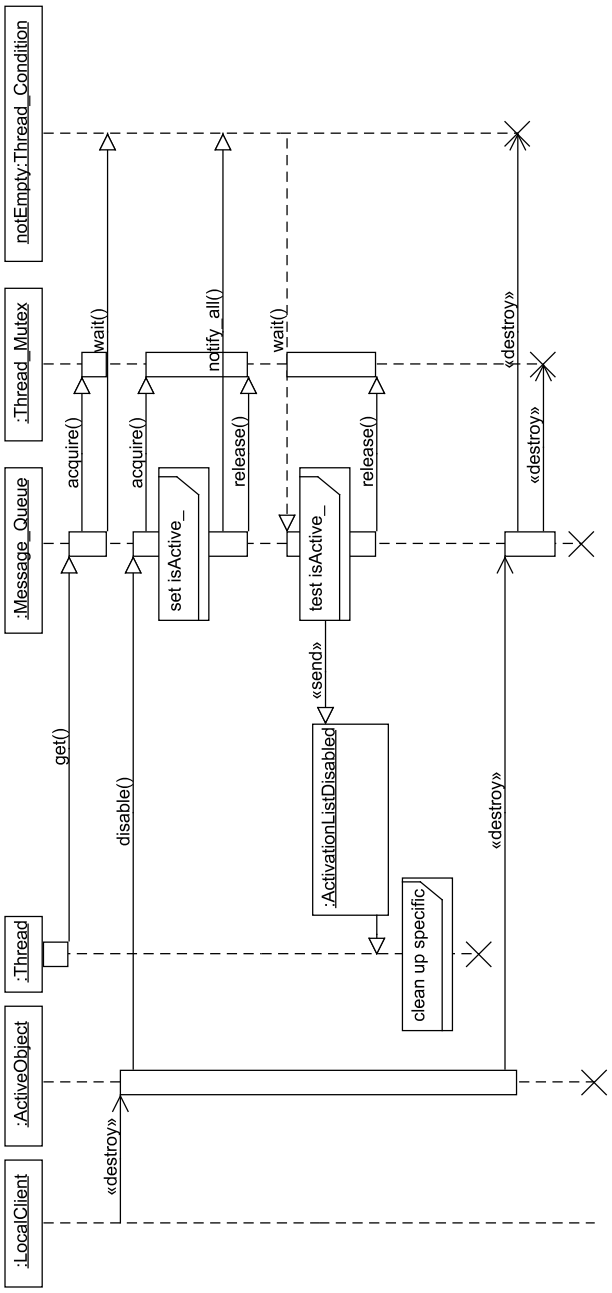

Figure 6: Sequence diagram matching Listings 1 and 2. For the sake of readability the concurrency inside was limited to one thread.

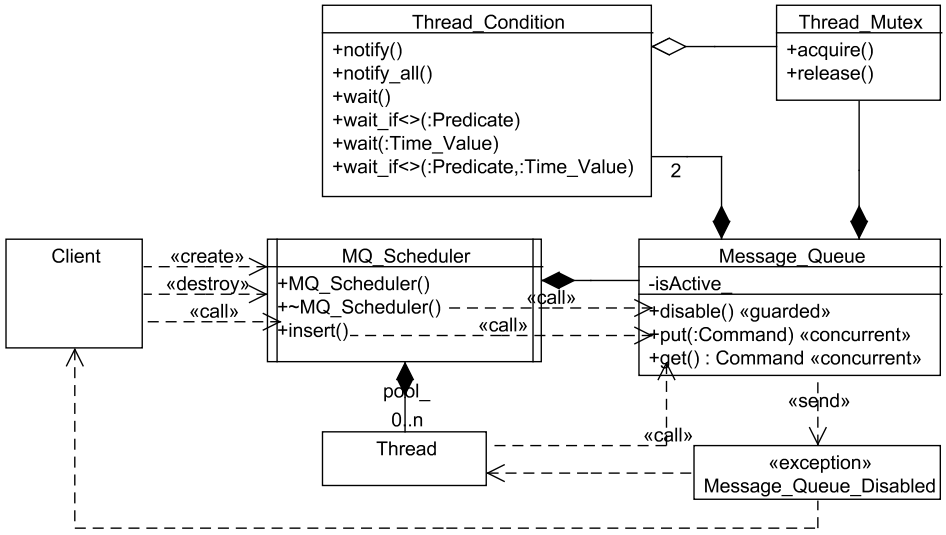

Figure 5: Class diagram matching Listings 1 and 2.

- It is hard to estimate the load up–front. Furthermore, the load likely varies.

From a more abstract point of view HALF–SYNC / HALF–ASYNC server designs turn asynchrony into synchrony, which on the one hand yields a programming model easier to deal with than event–driven I/O strategies, but on the other hand results in the need for dynamic load adaptation.

Several aspects need to be refined in order to do so: We want to cancel some, but not all threads of PoolOfUnitsOfExecution. As all of these threads execute the same code, it is reasonable to assume that the code does not need to cancel a particular thread—all we need is a way to cancel a certain number of threads from the pool regardless of their identity[3]. So the cancellability of `Message_Queue::get()` got changed in a way that the number of threads to send the exception to can be specified.

Furthermore, we can now cancel getting work from `Message_Queue` independently from putting new work to the queue—only the first operation needs to be cancelled during dynamic adaptation to decreasing load, because the RemoteClients must not be affected by this internal operation. Note, however, that both the member function `Message_Queue::disable()` and the destructor `Message_Queue::~Message_Queue()` shown above are supposed to also be part of the refined code—they are not shown here in order not to duplicate code.

The code in Listing 3 shows the modified `Message_Queue`.

Listing 3: Message queue with refined cancellability

```
class Message_Queue {
  ...
  volatile std::size_t consumers2Cancel_;
public:
  explicit Message_Queue(std::size_t
      max_messages =MAX_MESSAGES
  ) : ...,
      consumers2Cancel_(0),... {
    ...
  }
  ...
  bool emptyAndEnabled_i() const {
    return     empty_i()
            && isActive_
            && 0==consumers2Cancel_;
  }
  void disableGet(std::size_t
      consumers2Cancel) {
    Thread_Mutex_Guard guard(monitor_lock_);
    consumers2Cancel_=consumers2Cancel;
    not_empty_.notify_all();
  }
  // Transfers ownership
  Method_Request *get() {
    Method_Request *result=0;
    {
      Thread_Mutex_Guard
        guard(std::move(
          not_empty_.wait_if(
            boost::bind(
              &Message_Queue
                ::emptyAndEnabled_i,
              this,
              _1
            )
      )));
      // Note the differences to put().
      if(consumers2Cancel_)
      {
        --consumers2Cancel_;
        throw Message_Queue_Disabled;
      }
      if(empty_i())
        throw Message_Queue_Disabled;
      const bool wasFull(full_i());
      result=get_i();
      if(wasFull)
        not_full_.notify_all();
    }
    return result;
  }
};
```

Simply cancelling some of the threads from the pool re-

[3]It might be wise to refine this strategy to improve CPU cache affinity. This is beyond the scope of this example. See the original LEADER / FOLLOWERS paper for details on the LIFO follower promotion protocol.

gardless of their identity also means that it becomes difficult to both join the threads that have gone and to maintain a list of ids of still active threads later to join. Therefore we use detached threads here and emulate `joinThread()` by our own mechanism similar to what David R. Butenhof once suggested [13]. The heart of this is a threadsafe incarnation of the DETACHED COUNTED BODY idiom or SHARED OWNERSHIP (see Section 6.3), shown in Listing 4.

Listing 4: Smart pointer that both protects shared data and acts as a Future

```
class Message_QueueHandle {
  class Count {
    // No copy allowed, therefore
    // private and declared only
    Count(const Count &);
    // No assignment allowed, therefore
    // private and declared only
    Count &operator=(const Count &);
  public:
    std::size_t count_;
    mutable Thread_Mutex lock_;
    Thread_Condition unique_;
    Count() : count_(1), unique_(lock_) {}
  };
  Message_Queue *rep_;
  Count *count_;
public:
  explicit Message_QueueHandle(
      Message_Queue *rep
  ) : rep_(rep), count_(new Count) {}
  Message_QueueHandle(const
      Message_QueueHandle &rhs
  ) : rep_(rhs.rep_), count_(rhs.count_) {
    atomicIncrement();
  }
  ~Message_QueueHandle() {
    if(0==atomicDecrement()) {
      delete count_;
      delete rep_;
    }
  }
  Message_QueueHandle &operator=(const
      Message_QueueHandle &);
  Message_Queue *operator->() const {
    return rep_;
  }
  std::size_t atomicIncrement() {
    Thread_Mutex_Guard
      guard(count_->lock_);
    return ++count_->count_;
  }
  std::size_t atomicDecrement() {
    Thread_Mutex_Guard
      guard(count_->lock_);
    if(1==--count_->count_)
      count_->unique_.notify();
    return count_->count_;
  }
  void waitUnlessUnique() const {
    Thread_Mutex_Guard
      guard(count_->lock_);
    while(1!=count_->count_)
      count_->unique_.wait();
  }
};
```

Instances of this class protect an instance of `Message_Queue` from premature destruction. As shown below, `Message_QueueHandle::waitUnlessUnique()` makes instances of this class Futures.

The scheduler only needs slight adaptations, whereas the main thread loop now periodically tests whether to cancel or start threads—the respective metric is only sketched in the following code. Listing 5 shows the remaining code fraction.

Listing 5: Scheduler and thread main loop

```
class MQ_Scheduler {
  Message_QueueHandle act_queue_;
  void joinPool_() {
    act_queue_.waitUnlessUnique();
  }
public:
  MQ_Scheduler(std::size_t high_water_mark,
      std::size_t number_of_threads
  ) : act_queue_(
        new Message_Queue(high_water_mark)
      )
  {
    std::size_t i(0);
    try {
      for(;number_of_threads>i;++i)
      {
        act_queue_.atomicIncrement();
        createDetachedThread(svc_run,
          &act_queue_);
      }
    }
    catch(...) {
      // Creation of (i+1)th thread failed
      // to happen.
      act_queue_.atomicDecrement();
      act_queue_.disableGet(i);
      joinPool_();
      throw;
    }
  }
  ...
  // Transfers ownership
  void insert(Method_Request *
                method_request) {
    act_queue_->put(method_request);
  }
};

void *svc_run(void *arg) {
  // Set all signals blocked in the
  // current thread's signal mask
  ...
  assert(arg);
  // Copy the smart pointer
  Message_QueueHandle act_queue(
    *static_cast< Message_QueueHandle * >
       (arg)
  );
  act_queue.atomicDecrement();
  while(true)
    try {
      if(tooManySpareThreads)
        act_queue->disableGet(
          superfluousSpares
        );
      // Block until the queue is
      // not empty
      std::auto_ptr< Method_Request >
        mr(act_queue->get());
      if(tooFewSpareThreads)
        // Increase number of Units of
        // Execution
        // (beyond the scope of this paper)
      mr->call();
    }
    catch(const Message_Queue_Disabled &) {
      break;
```

```
    }
    catch(...) {
    }
  return 0;
} // On automatic destruction of
  // "act_queue" its reference counter will
  // be reliably decremented. After all
  // worker threads terminated, the main
  // thread holds the only reference to the
  // Activation List and therefore knows
  // that all other threads were gone.
```

Additionally to the comments on the code in the section before note some details of this implementation:

- As the instance of `Message_QueueHandle` must be pas sed by pointer to `svc_run()` it is important to temporarily manipulate its reference counter.
- The main thread waits until it is the only one that holds a reference to the contents of `Message_QueueHandle`. To implement waiting, a condition variable is used within `Message_QueueHandle`.

The usage of the modified scheduler does not differ from the first example.

8.3 Relationship of Examples and Participants

The restaurant example layed out in Sections 2 and 8.1 and the code examples shown in Section 8.2 map to the participants defined in Section 6.1 as shown in Table 2.

9. VARIANTS

The level of abstraction chosen for the interception of blocking calls can vary. So a variant is to equip condition variables with an operation to disable their operation and to let all their potentially blocking operations throw if and only if they have been disabled. If portability is an issue, then choosing as high a level as suggested can make life easier.

Another variant is to make cancellability a property of a Unit of Execution instead of a class with blocking operations.

The decision to first let the Units empty the queue before terminating is well–founded, but nevertheless arbitrary. So a variant was to replicate CORBA interfaces qualified `oneway` [39, pp 15–17], that do not guarantee that the service requested is ever going to be executed. Another variant was to dump the backlog to disk, terminate immediately afterwards and read it in again the next time the ACTIVE OBJECT starts. But as serialization can fail, this may be not as reliable as the solution proposed in the first place. Furthermore an according implementation was more complex.

This pattern can be combined with more aggressive ways of cancellation: To overcome liability 1 a timeout may be introduced such that PoolOfUnitsOfExecution is forced to immediately die if it did not cooperatively terminate within a certain time frame. The difficulty is the heuristic choice of a timeout large enough to only affect uncooperative Units.

As its name already suggests, an implementation according to the poison pill approach lets the consumers die, but does not consider producers [34, pp 145–146].

10. KNOWN USES

Examples of DEFERRED CANCELLATION can be found in existing software.

10.1 ACE

The ADAPTIVE Communication Environment (ACE) provides `ACE_Message_Queue<>`, a synchronized queue, that can be disabled similar to the one proposed above. Its blocking member functions do not throw if the queue has been disabled. Instead they return the special value `ESHUTDOWN` then.

10.2 Boost.Thread

DEFERRED CANCELLATION was recently added to the Boost.Thread [29, 58] C++ library. Cancellation is referred to as "interruption". Different from the pattern description above it's not ActivationList that can be disabled, but it is the thread itself. This is similar to Java (see Section 10.4). Interruption takes place by means of an exception thrown from well–defined interruption points, e.g. `boost::condition_variable`. The user can define additional interruption points, if necessary. The set of pre–defined interruption points is still limited to entities within Boost.Thread—there are many more blocking calls, however, e.g. in `std::fstream` or during network communications, because cancellability is a crosscutting concern [30].

Internally, THREAD–SPECIFIC STORAGE [48] is used to manage the flag that indicates interruption of a thread for it to be accessible from the interruption points.

10.3 POSIX Threads

POSIX 1003.1c compliant systems provide `pthread_cancel()`, that cancels a thread specified as an argument. This plays the role of `Message_Queue::disable()` in Section 8.2.1. Threads can be configured dynamically to react differently upon cancellation requests: They can igore them, they can be cancellable at well–defined cancellation points only ("deferred cancellation")—in Section 8.2.1 these were only `Message_Queue::{get(),put()}`—, or they can immediately exit ("asynchronous cancellation"). In both of the last two cases during the cancellation procedure so–called cleanup handlers are going to be executed that have been registered by the user before like the `finally` block referred to in Section 6.3. The default behavior is cancellation enabled at well–defined cancellation points only. The set of POSIX Threads cancellation points is extensible. The list of cancellation points defined in UNIX98 lists `accept()` as a cancellation point, POSIX 1003.1c does not. For details see e.g. [14].

Some C++ compilers automatically register destructors as cleanup handlers and so transparently support Resource Acquisition is Initialization even in the area of multithreading, that is out of scope of the current C++ standard.

10.4 Java

Similar to Boost.Thread (see Section 10.2) instances of the class `java.lang. Thread` can be set interrupted. In this case some blocking operations are left with an instance of `java.lang.InterruptedException` thrown, some with an instance of `java.nio.channels.ClosedByInterruptException` thrown, while calls to `java.nio.channels.Selector`s are simply waked up. For details see e.g. [33, pp 169–177,294].

10.5 MS Windows

MS Windows is listed here for two reasons.

With regard to threads MS Windows is a counterexam-

Table 2: Relationship of Examples and Participants

Examples		Participant
Restaurant	Code	
Orders Holder	`Message_Queue`	ActivationList
Waiter rejects any new orders; empty orders holder triggers kitchen staff to leave off work	`Message_Queue_Disabled`	ActivationListDisabled
Restaurant operations, represented by the waiters as PROXIES	`MQ_Scheduler`	ActiveObject
Restaurant manager (not considered)	`main()`	LocalClient
Kitchen staff	Section 8.2.1: `MQ_Scheduler::pool_`; Section 8.2.2: Implicit.	PoolOfUnitsOfExecution
Customers	`main()`	RemoteClients

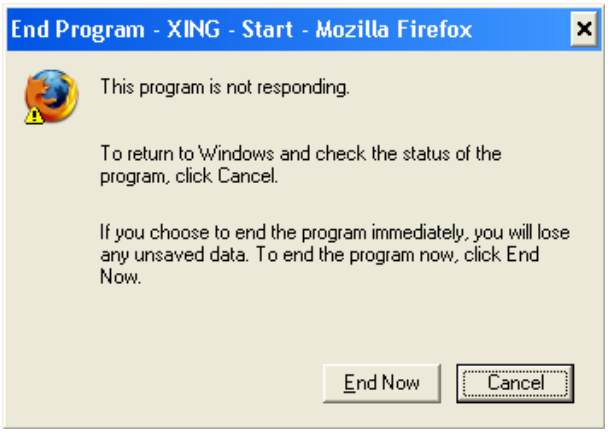

Figure 7: In case a process cancelled did not terminate within a certain time frame MS Windows NT resorts to the user.

ple. `TerminateThread()` simply kills a thread immediately without giving it a chance to clean up. There is no function similar to POSIX 1003.1c `pthread_cancel()`. Implementing cancellation of ACTIVE OBJECTS and other thread pools or implementing a mechanism that automatically adapts their sizes to varying load on MS Windows requires code similar to that of Section 8.2.1 that does not rely on safe cancellation provided by the operating system.

With regard to processes MS Windows NT implements the timeout variant referred to in Section 9. If a process cancelled does not terminate within a certain time interval, the operating system resorts do a dialog that lets the user decide whether to apply more aggressive means to get rid of the respective process. An example of these dialogs is shown in Figure 7.

11. RELATED IDIOMS AND PATTERNS

The context of DEFERRED CANCELLATION is formed by a pool of Units of Execution, e.g. WORKER THREADS. This is the basis of ACTIVE OBJECTS, HALF–SYNC / HALF–ASYNC or LEADER / FOLLOWERS designs.

DEFERRED CANCELLATION shows how to safely clean up resources specific to members of PoolOfUnitsOfExecution and shared among many Units. These resources must have been initialized before. KIRCHER / JAIN have a comprehensive collection of initialization patterns [31, pp 19–79].

The mechanism to force cancellation of the members of the pool steps in at WRAPPER FACADES or higher level abstractions like MONITOR OBJECTS.

Both the POOLING pattern and the RESOURCE LIFECYCLE MANAGER pattern give a high level view on the whole lifecycle of resources. The pattern descriptions cover the release of resources, which means cancellation in case of threads and processes [31, pp 97–110,128–146].

If static configuration of the platforms is required or there is the need to choose among different higher–level architectures, e.g. between the designs LEADER / FOLLOWERS, One Child per Client [53, pp 732–736] and One Thread per Client [53, pp 752–753], then static and metaprogramming patterns—especially STATIC ADAPTER and STATIC ABSTRACT TYPE FACTORY—can be applied additionally to build such STATIC FRAMEWORKS [7].

There are many more patterns and idioms that assist in the development of concurrent applications. SCHMIDT et al. [44] and MATTSON et al. [34] are comprehensive collections or pattern languages in this area.

Acknowledgements

Without the invaluable feedback of Berna L. Massingill, who was the PLoP shepherd of this work, this paper would not have been the way it is now.

The author would like to thank all the participants of PLoP 2008 for their contributions. Special thanks go to the members of the Writers' Workshop [18] "Processes & Services" the author participated in: Paul G. Austrem, Rob Daigneau, John Liebenau, Mark Mahoney, Geert Monsieur, Miyuko Naruse and last but not least both Lise B. Hvatum and Bobby Woolf, its moderators. Their feedback had a significant impact on the current version of the paper.

A special thanks to Thomas Hoof Produkt GmbH, Waltrop, Germany, for the use of its photograph in Figure 1. Stefan Pangritz triggered the idea for the known use shown in Figure 7.

Last but not least my thanks and love go to Cornelia Kneser, my wife, for her constant support throughout the writing of the paper.

This work was supported by the Institute for Medical Informatics and Biostatistics, Basel, Switzerland.

12. REFERENCES

[1] *Releasing resources in Java.* published online. `http://`

www.c2.com/cgi/wiki?ReleasingResourcesInJava, visited on January 9, 2007.

[2] *Preliminary conference proceedings of EuroPLoP '96*. Tech. Rep. wucs-97-07, Washington University, Department of Computer Science, 1997.

[3] Alexander, C., S. Ishikawa, and M. Silverstein: *A Pattern Language. Towns, Buildings, Construction. With* Max Jacobson, Ingrid Fiksdahl–King *and* Shlomo Angel. Oxford University Press, New York, 1977, ISBN 0-19-501919-9.

[4] American National Standards Institute, New York, NY: *ISO / IEC 14882:2003(E). Programming languages—C++. Langages de programmation—C++, International Standard*, second ed., Oct. 2003, ISBN 0-470-84674-7.

[5] Arnold, K., J. Gosling, and D. Holmes: *Die Programmiersprache Java. Deutsche Übersetzung von RederTranslations,* Dorothea Reder *und* Gabi Zöttl. Programmer's Choice. Addison–Wesley. An Imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, 2001, ISBN 3-8273-1821-1. German translation of "The Java Programming Language. Third Edition".

[6] Bachmann, P.: *Change of authority and thread safe interface goes synchronized*. In *Proceedings of the 12th Pattern Languages of Programs (PLoP) conference 2005, Allerton Park, Monticello, IL, USA*, Dec. 2005. http://hillside.net/plop/2005/proceedings/PLoP2005_pbachmann1_0.pdf, visited on January 9, 2007.

[7] Bachmann, P.: *Static and metaprogramming patterns and static frameworks. A catalog. An application*. In Yoder, J. and Johnson [60], pp. 1–33, ISBN 978-1-60558-372-3. http://hillside.net/plop/2006/Papers/ACM_Version/Static_and_Metaprogramming_Patterns_and_Static_Frameworks.pdf, visited on November 30, 2008.

[8] Baker, Jr., H. C. and C. Hewitt: *The incremental garbage collection of processes*. SIGPLAN Notices, 12(8):55–59, Aug. 1977, ISSN 0362-1340. Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages.

[9] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: *Command–Processor*, ch. 3: Entwurfsmuster, pp. 277–291. In [11], 1998, 1. korr. nachdruck ed., 2000, ISBN 3-8273-1282-5. German translation of "Command Processor".

[10] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: *Layers*, ch. 2: Architekturmuster, pp. 32–53. In [11], 1998, 1. korr. nachdruck ed., 2000, ISBN 3-8273-1282-5. German translation of "Layers".

[11] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: *Pattern–orientierte Softwarearchitektur. Ein Pattern–System, deutsche Übersetzung von* Christiane Löckenhoff. Addison–Wesley. An Imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, 1998, 1. korr. nachdruck ed., 2000, ISBN 3-8273-1282-5. German translation of "Pattern–Oriented Software Architecture. A System of Patterns".

[12] Buschmann, F., R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal: *Proxy*, ch. 3: Entwurfsmuster, pp. 263–275. In [11], 1998, 1. korr. nachdruck ed., 2000, ISBN 3-8273-1282-5. German translation of "Proxy".

[13] Butenhof, D. D. R.: *Re: `pthread_join()` with a timeout?*, Feb. 1998. http://groups.google.ch/group/comp.programming.threads/msg/c12d5865e26b8b9c, visited on December 10, 2008.

[14] Butenhof, D. R.: *Programming with POSIX Threads*. Addison–Wesley Professional Computing Series; ed. by Brian W. Kernighan. Addison–Wesley Longman Publishing Co., Inc., Boston, Massachusetts, 1997, ISBN 0-201-63392-2.

[15] Cargill, T.: *Localized ownership: Managing dynamic objects in C++*. In Vlissides, J. M. *et al.* [57], ch. 1, pp. 5–18, ISBN 0-201-60734-4 / 0-201-89527-7.

[16] Coplien, J.: *C++ idioms patterns*. In Harrison, N. *et al.* [24], ch. 10, pp. 167–197, ISBN 0-201-43304-4.

[17] Coplien, J. O. and D. C. Schmidt (eds.): *Pattern Languages of Program Design*, vol. 1 of *The Software Patterns Series; ed. by* John Vlissides. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, 1995, ISBN 0-201-60734-4.

[18] Gabriel, R. P.: *Writers' Workshops & the Work of Making Things. Patterns, Poetry. . . .* Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, 2002, ISBN 0-201-72183-X.

[19] Gamma, E., R. Helm, R. Johnson, and J. Vlissides: *Befehl*, ch. 5: Verhaltensmuster, pp. 273–286. In *Professionelle Softwareentwicklung* [20], dritter, unveränderter nachdruck ed., 1996. German translation of "Command".

[20] Gamma, E., R. Helm, R. Johnson, and J. Vlissides: *Entwurfsmuster. Elemente wiederverwendbarer objektorientierter Software, deutsche Übersetzung von* Dirk Riehle. Professionelle Softwareentwicklung. Addison–Wesley–Longman, Bonn · Reading, Massachusetts · Menlo Park, California · New York · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, dritter, unveränderter nachdruck ed., 1996. German translation of "Design Patterns. Elements of Reusable Object–Oriented Software".

[21] Gamma, E., R. Helm, R. Johnson, and J. Vlissides: *Proxy*, ch. 4: Strukturmuster, pp. 254–267. In *Professionelle Softwareentwicklung* [20], dritter, unveränderter nachdruck ed., 1996. German translation of "Proxy".

[22] Gamma, E., R. Helm, R. Johnson, and J. Vlissides: *Zustand*, ch. 5: Verhaltensmuster, pp. 398–409. In *Professionelle Softwareentwicklung* [20], dritter, unveränderter nachdruck ed., 1996. German translation of "State".

[23] Goethe, J. W. von: *The sorcerer's apprentice*, 1955. http://www.fln.vcu.edu/goethe/zauber_e3.html,

visited on November 30, 2008, English translation of "der Zauberlehrling", 1797, by Edwin Zeydel.

[24] Harrison, N., B. Foote, and H. Rohnert (eds.): *Pattern Languages of Program Design*, vol. 4 of *The Software Patterns Series; ed. by* John M. Vlissides. Addison–Wesley. An imprint of Addison Wesley Longman, Inc., Reading, Massachusetts · Harlow, England · Menlo Park, California · Berkeley, California · Don Mills, Ontario · Sydney · Bonn · Amsterdam · Tokyo · Mexico City, 2000, ISBN 0-201-43304-4.

[25] Henney, K.: *Executing around sequences*. In *Proceedings of the 5th European Conference on Pattern Languages of Programs (EuroPLoP) 2000, Irsee, Germany*, July 2000. `http://hillside.net/europlop/HillsideEurope/Papers/ExecutingAroundSequences.pdf`, visited on January 9, 2007.

[26] Hewitt, C., P. Bishop, and R. Steiger: *A universal modular ACTOR formalism for artificial intelligence*. In *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 235–245, 1973.

[27] Hinnant, H. E., B. Stroustrup, and B. Kozicki: *A brief introduction to rvalue references*. Tech. Rep. WG21/N2027 = J16/06-0097, ISO IEC JTC1 / SC22 / WG21—The C++ Standards Committee, Herb Sutter · Microsoft Corp. · 1 Microsoft Way · Redmond WA USA 98052-6399, June 2006. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html`, visited on April 25, 2008.

[28] Hirschfeld, R.: *Convenience method*. In *Preliminary Conference Proceedings of EuroPLoP '96* [2].

[29] Kempf, B.: *The Boost.Threads library*. The C/C++ Users Journal. Advanced Solutions for C/C++ Programmers, May 2002. `http://www.ddj.com/cpp/184401518`, visited on May 2, 2008.

[30] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin: *Aspect–oriented programming*. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP), Finland*, no. 1241 in *Lecture Notes in Computer Science*. Springer, June 1997.

[31] Kircher, M. and P. Jain: *Pattern–Oriented Software Architecture. Patterns for Resource Management*. Wiley Series in Software Design Patterns. John Wiley & Sons, Ltd, Chichester, 2004, ISBN 0-470-84525-2.

[32] Lakos, J.: *Large–Scale C++ Software Design*. Addison–Wesley Professional Computing Series; ed. by Brian W. Kernighan. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, 11th printing may 2002 ed., 1996, ISBN 0-201-63362-0.

[33] Lea, D.: *Concurrent Programming in Java. Design Principles and Patterns*. The Java Series; Lisa Friendly, Series Editor, Tim Lindholm, Technical Editor. Addison–Wesley. An imprint of Addison Wesley Longman, Inc., Reading, Massachusetts · Harlow, England · Menlo Park, California · Berkeley, California · Don Mills, Ontario · Sydney · Bonn · Amsterdam · Tokyo · Mexico City, second ed., Nov. 1999, ISBN 0-201-31009-0.

[34] Mattson, T. G., B. A. Sanders, and B. L. Massingill: *Patterns for Parallel Programming*. The Software Patterns Series; ed. by John Vlissides. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, Sept. 2004, ISBN 0-321-22811-1.

[35] Meier, A. V.: *Networking and Concurrency Patterns. hauptseminar patterns in der softwareentwicklung*. published online, Feb. 2007. `http://andreas.anvame.net/content/download/meiera-netpatterns-slides.pdf`, visited on December 8, 2008, TUM Informatik—Lehrstuhl Brauer.

[36] Meszaros, G.: *A pattern language for improving the capacity of reactive systems*. In Vlissides, J. M. *et al.* [57], ch. 35, pp. 575–591, ISBN 0-201-60734-4 / 0-201-89527-7.

[37] Meyer, B.: *Object–Oriented Software Construction*. Prentice Hall, Upper Saddle River, New Jersey, second ed., 1997, ISBN 0-13-629155-4.

[38] Microsoft Developer Network (MSDN): *.NET framework general reference: Common design patterns. implementing finalize and dispose to clean up unmanaged resources*. published online, 2005. `http://msdn.microsoft.com/library/en-us/cpgenref/html/cpconfinalizedispose.asp`, visited on January 9, 2007.

[39] Redlich, J.-P.: *CORBA 2.0. Praktische Einführung für C++ und Java, mit einem Geleitwort von* Richard Mark Soley. Praktische Informatik; hrsg. von Ludwig Classen, Dieter Nedo und Gerhard Paulin. Addison–Wesley Publishing Company, Bonn · Reading, Massachusetts · Menlo Park, California · New York · Don Mills, Ontario · Harlow, England · Amsterdam · Milan · Sydney · Tokyo · Singapore · Madrid · San Juan · Seoul · Mexico City · Taipei, Taiwan, 1996, ISBN 3-8273-1060-1.

[40] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Active Object*, ch. 5: Nebenläufigkeit, pp. 411–443. In [44], 2002, ISBN 3-89864-142-2. German translation of "Active Object".

[41] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Half–Sync / Half–Async*, ch. 5: Nebenläufigkeit, pp. 473–497. In [44], 2002, ISBN 3-89864-142-2. German translation of "Half–Sync / Half–Async".

[42] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Leader / Followers*, ch. 5: Nebenläufigkeit, pp. 499–528. In [44], 2002, ISBN 3-89864-142-2. German translation of "Leader / Followers".

[43] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Monitor Object*, ch. 5: Nebenläufigkeit, pp. 445–471. In [44], 2002, ISBN 3-89864-142-2. German translation of "Monitor Object".

[44] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Pattern–orientierte Software-Architektur. Muster für nebenläufige und vernetzte Objekte, übersetzt aus dem Amerikanischen von* Martina Buschmann. dpunkt.verlag, Heidelberg, 2002, ISBN 3-89864-142-2. German translation of "Pattern–Oriented Software Architecture. Volume 2: Patterns for Concurrent and Networked Objects".

[45] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Reactor*, ch. 3: Ereignisverarbeitung, pp. 197–236. In [44], 2002, ISBN 3-89864-142-2. German translation of "Reactor".

[46] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Scoped Locking*, ch. 4: Synchronisation, pp. 359–367. In [44], 2002, ISBN 3-89864-142-2. German translation of "Scoped Locking".

[47] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Thread–Safe Interface*, ch. 4: Synchronisation, pp. 383–391. In [44], 2002, ISBN 3-89864-142-2. German translation of "Thread Safe Interface".

[48] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Thread–Specific Storage*, ch. 5: Nebenläufigkeit, pp. 529–561. In [44], 2002, ISBN 3-89864-142-2. German translation of "Thread–Specific Storage".

[49] Schmidt, D., M. Stal, H. Rohnert, and F. Buschmann: *Wrapper–Facade*, ch. 2: Dienstzugriff und Konfiguration, pp. 53–84. In [44], 2002, ISBN 3-89864-142-2. German translation of "Wrapper Facade".

[50] Schmidt, D. C.: *Reactor: An object behavioral pattern for concurrent event demultiplexing and event handler dispatching*. In Coplien, J. O. and Schmidt [17], ch. 29, pp. 529–545, ISBN 0-201-60734-4.

[51] Schmidt, D. C. and C. D. Cranor: *Half–sync / half–async. An architectural pattern for efficient and well–structured concurrent I/O*. In Vlissides, J. M. *et al.* [57], ch. 27, pp. 437–459, ISBN 0-201-60734-4 / 0-201-89527-7.

[52] Schmidt, D. C. and I. Pyarali: *The design and use of the ACE reactor. An object–oriented framework for event demultiplexing*. `http://www.cs.wustl.edu/~schmidt/PDF/reactor-rules.pdf`, visited on January 9, 2007.

[53] Stevens, W. R.: *UNIX Network Programming*, vol. 1. Networking APIs: Sockets and XTI. Prentice Hall PTR, Upper Saddle River, NJ, second ed., 1998, ISBN 0-13-490012-X.

[54] Stroustrup, B.: *Design und Entwicklung von C++*. Addison–Wesley, Bonn · Paris · Reading, Massachusetts · Menlo Park, California · New York · Don Mills, Ontario · Workingham, England · Amsterdam · Milan · Sydney · Tokyo · Singapore · Madrid · San Juan · Seoul · Mexico City · Taipei, Taiwan, 1994, ISBN 3-89319-755-9. German translation of "The Design and Evolution of C++".

[55] Stroustrup, B.: *Die C++–Programmiersprache. Deutsche Übersetzung von* Nicolai Josuttis *und* Achim Lörke. Addison–Wesley–Longman, Bonn · Reading, Massachusetts · Menlo Park, California · New York · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, dritte, aktualisierte und erweiterte ed., 1998, ISBN 3-8273-1296-5. German translation of "The C++ Programming Language, Third Edition".

[56] Thomas Hoof Produkt GmbH: *Bar with spring holders, Waltrop, Germany*, 2006. `http://www.produkt-gmbh.de/de/produkte/kueche/bonleiste.html`, visited on November 24, 2008.

[57] Vlissides, J. M., J. O. Coplien, and N. L. Kerth (eds.): *Pattern Languages of Program Design*, vol. 2. Addison–Wesley. An imprint of Addison Wesley Longman, Inc., Reading, Massachusetts · Harlow, England · Menlo Park, California · Berkeley, California · Don Mills, Ontario · Sydney · Bonn · Amsterdam · Tokyo · Mexico City, 1996, ISBN 0-201-60734-4 / 0-201-89527-7.

[58] Williams, A.: *[Boost.]Thread*. published online. `http://www.boost.org/doc/libs/1_35_0/doc/html/thread.html`, visited on May 2, 2008.

[59] Williams, A.: *Thread interruption in the Boost Thread library*, Mar. 2008. `http://www.justsoftwaresolutions.co.uk/threading/thread-interruption-in-boost-thread-library.html`, visited on April 27, 2008.

[60] Yoder, J. and R. Johnson (eds.): *PLoP '06: Proceedings of the 2006 conference on Pattern Languages of Programs*, New York, NY, USA, 2006. ACM, ISBN 978-1-60558-372-3.

APPENDIX

Glossary

This section contains pattern–thumbnails and definitions of the most important terms used in this paper.

Active Object The ACTIVE OBJECT pattern proposes a design that decouples the invocation of services from their actual execution. To do so COMMANDs are going to be produced that reify the service requests. Within the ACTIVE OBJECT these COMMANDs are placed into a queue then. ACTIVE OBJECTs own a pool of Units of Execution, and each Unit periodically consumes new work from the queue. So ACTIVE OBJECTs introduce concurrency. This approach is subsumed under the WORKER THREADS pattern [33, pp 290–298], which in turn is a realization of the pattern SHARE THE LOAD [36, pp 588–589]. Calls to ACTIVE OBJECTs are asynchronously. ACTIVE OBJECTs are often combined with Futuress.[40, 26]

Command The COMMAND pattern encapsulates commands as objects. COMMANDs can be placed in a queue and are an essential ingredient for undo functionality.[19]

Future After issuing an asynchronous call to a function a client can continue to work without having to wait for the result. But at some point in time it may need to be sure that the operation initiated has completed. Futures allow for this. Futures are usually issued while calling an asynchronous member function of an ACTIVE OBJECT. A client can block in a call to a member function of the Future until the associated asynchronous call completed ("Rendezvous"). Then a client can get the equivalent of a return value from the asynchronous call.[8], [40, pp 413, 417,423–430,435–436] Basic building blocks of Futures are e.g. condition variables combined with mutual exclusion locks.

Half–Sync / Half–Async Kernels of modern operating systems act asynchronously. Operating systems offer also synchronous programming interfaces, however, because they are easier to deal with than asynchronous ones. The HALF–SYNC / HALF–ASYNC architecture pattern sheds light on the boundary between the asyn-

chronous kernel and its synchronous programming interface [51, 41].

Leader / Followers The LEADER / FOLLOWERS pattern specifies an architecture for concurrent servers. Several threads take turns accessing shared event sources, demultiplexing pending events, dispatching, and finally processing them [42], [53, pp 754–756].

Monitor Object The MONITOR OBJECT pattern synchronizes the concurrent access to an object. It keeps the state of the object well–defined by applying necessary serialization to shared data. It also controls access based on monitor conditions: In case of a queue this synchronization takes place at the two bounds of the queue: If the queue reached its maximum capacity, producers are blocked or producers receive an according return value; if the queue is empty, consumers are blocked.[43]

Portability Portability has two aspects: Portability regarding to environments or platforms means that an application requires no or only a few local changes to run on another platform than once planned for. The term platform can refer to operating system, hardware architecture or even a set of third party software the application interfaces with, e.g. a database management system. Portability in time means that an application can still be compiled after years have passed.

Reactor The REACTOR pattern describes a mechanism to demultiplex events arriving concurrently from different sources and to dispatch the requests to appropriate handlers. In a REACTOR a function blocks until an event occurs. Then the dispatching takes place.[52, 50, 45]

Resource We consider any entity a resource that takes one of two states: Released and acquired. The latter state is somehow expensive: Only one client at any one time can acquire the same resource (*access* to an acquired resource can be shared among several clients, however); some resources are limited, e.g. main memory or even more handles to open files; some can introduce scalability issues, e.g. mutual exclusion locks. Because of these properties care must be taken to reliably release resources again after use. Applications that do not properly release resources are said to leak them. Typical strategies to deal with resources are either the Resource Acquisition is Initialization technique [55, pp 388–393], [54, pp 495–497], especially the EXECUTE–AROUND OBJECT design pattern [25, pp 4–7], or the DISPOSE pattern [5, pp 228–230], [38], [1] in programming languages that do not support the first alternative. In case of not–so–expensive resources like memory garbage collection was also an option to go for, if available. See KIRCHER / JAIN for a slightly more specialized definition [31]. You will find a classification of different resources and last but not least many other useful strategies to deal with resources there, too.

Unit of Execution Unit of Execution is the umbrella term for processes and threads [34, pp 217–221]. It neither presumes a certain visibility of memory address space among several Units nor whether a hierarchy exists among the Units of Execution or not.