Change Of Authority and Thread Safe Interface goes `synchronized`

Philipp Bachmann. Revision: 1.11 *

December 16, 2005

Abstract

Programming languages with deterministic destruction of instances like C++ allow for an idiom called Resource Acquisition is Initialization [Str98, pp 388–393], [Str94, pp 495–497]. This technique provides many advantages. Applications are e.g. the Object as Sole Owner pattern [Car96] and the Scoped Locking idiom [SSRB02d]. Instances of classes implementing Resource Acquisition is Initialization the way Scoped Locking does, i.e. referencing a resource they do not own, can neither be copied nor assigned. This particularly means that they can not be returned by value from functions.

The Change Of Authority idiom is shown in detail in Chapter 1. This idiom enables instances of these classes to be returned by value from functions and thus overcomes the above limitation. It also helps reduce the cost of temporaries generated by the compiler.

The usefulness of this idiom is shown by examples mainly from the concurrency domain. So one example is an extension to Scoped Locking.

In Java there is the `synchronized` method qualifier and in C# / .NET you can set the `synchronized` attribute in the metadata of a method. Both add the respective method to the Thread Safe Interface [SSRB02e]. In C++ such a language feature does not exist. The code shown in Chapter 2 makes the convenience of the Java resp. C# / .NET keywords mentioned available to C++. It is an application of the idiom shown in Chapter 1. Thus the implementation of the Thread Safe Interface design pattern becomes substantially simplified.

Contents

What's new here is that there's nothing new here.

Brian Foote [MRB98, p ix]

Chapter 1

The Change Of Authority idiom

This chapter describes the Change Of Authority idiom, an enhancement for classes implementing the Execute–Around Object design pattern [Hen00, pp 4–7], a common application of the Resource Acquisition is Initialization technique [Str98, pp 388–393], [Str94, pp 495–497]. The idiom also is a generalization of Change of Ownership as implemented e.g. with C++ `std::auto_ptr<>`.

1.1 Also known as

Move Constructors [Ale03], [HDA02]

1.2 Intent

Resource Acquisition is Initialization–guards usually are not copyassignable and thus can not be returned by value from functions. This idiom shows, how to implement guard classes, such that functions can return such guard instances without destroying guarantees. It is also useful to reduce the cost of temporaries generated by a compiler.

1.3 Example

Within the community of those who implement concurrent systems in C++ or other programming languages with deterministic destruction of instances the Scoped Locking idiom [SSRB02d] is well established. This purpose of this idiom is to couple the period during which a lock is being held to the lifetime of an automatic variable.

Listing 1.1: The Scoped Locking idiom according to [SSRB02d, pp 361–362]

```
class Thread_Mutex_Guard {
  Thread_Mutex &lock_;
  // No copy allowed, therefore private and declared only
  Thread_Mutex_Guard(const Thread_Mutex_Guard &);
  // No assignment allowed, therefore private and declared only
  Thread_Mutex_Guard &operator=(const Thread_Mutex_Guard &);
public:
  explicit Thread_Mutex_Guard(Thread_Mutex &lock) : lock_(lock) {
    lock_.acquire();
  }
  ~Thread_Mutex_Guard(void) {
    lock_.release();
  }
};
```

Thus on instantiation of `Thread_Mutex_Guard` a lock gets acquired, and on leaving the surrounding block—most often the body of a function—the lock gets released regardless of how the block is actually being left, be it by `return` statements, be it by an exception thrown.

Access to an entity `size`, which is shared among different threads of execution, can be serialized as follows:

Listing 1.2: Using Scoped Locking

```
// Shared among multiple threads
static Thread_Mutex lock;
// Shared among multiple threads and protected by "lock"
static size_t size;
...
// Potentially called concurrently
void setSize(size_t sz) {
  Thread_Mutex_Guard guard(lock);
  size=sz;
} // On automatic destruction of "guard" "lock" will
  // reliably be released again.
```

Because the copy constructor is `private`, instances of `Thread_Mutex_Guard` can not be copied. This prevents bad surprises elaborated on in detail below in Listing 1.7.

Instances not being copyable can not be returned by value from functions, however. Therefore e.g. adding the following template member function to the `Thread_Condition` Wrapperfacade [SSRB02f, pp 66–67] will not work with standard Scoped Locking:

Listing 1.3: A convenience member function returning `Thread_Mutex_Guard`

```
class Thread_Condition {
  Thread_Mutex &mutex_;
  ...
public:
  ...
  // void wait(void); renamed to
```

```
7   void wait_i(void); // not part of the
8                      // Thread Safe Interface
9   // Convenience template member function
10  template< class Predicate >
11  Thread_Mutex_Guard wait_if(Predicate p) {
12    Thread_Mutex_Guard guard(mutex_);
13    while(p())
14      wait_i();
15    return guard;
16  }
17 };
```

This convenience member function periodically checks the condition, and if it is not fulfilled yet, then the simpler overload is being called. Thus this convenience member function catches the most important Use Case [FS00, pp 35–42] associated to condition variables and casts this into an Inversion of Control member function configurable by means of a Strategy [GHJV96m]. This generalizes Convenience Method [Hir97]. The class `boost::condition` of the Boost.Threads library [Kem] provides a similar convenience member function.

If we could manage `Thread_Mutex_Guard`s to be returned by value from functions, though still not copyable, the above convenience member function became possible, and we could use it as follows:

Listing 1.4: Using `Thread_Condition::wait_if<>()`

```
1  class Message_Queue {
2    class IsFull : public std::unary_function< void,bool > {
3      const Message_Queue &outer_;
4    public:
5      explicit IsFull(const Message_Queue &outer) : outer_(outer) {}
6      bool operator()(void) const {
7        return outer_.full_i();
8      }
9    };
10   friend class IsFull;
11   const IsFull isFull_;
12   ...
13   size_t max_messages_;
14   mutable Thread_Mutex monitor_lock_;
15   Thread_Condition not_full_;
16   Thread_Condition not_empty_;
17 public:
18   enum { MAX_MESSAGES = ... };
19   explicit Message_Queue(size_t max_messages =MAX_MESSAGES)
20     : isFull_(*this),
21       max_messages_(max_messages),
22       not_full_(monitor_lock_),... {
23     ...
24   }
25   Message_Queue(const Message_Queue &rhs)
26     : isFull_(*this),           // Don't let the copy of
27                                 // the predicate refer to
28                                 // the wrong queue
29       max_messages_(rhs.max_messages_),
30       not_full_(monitor_lock_), // Condition variables can't
```

```
31                                  // be copied
32       ... {
33     ...
34   }
35   void put(const Message &msg) {
36     Thread_Mutex_Guard guard(not_full_.wait_if(isFull_));
37     const bool wasEmpty(empty_i());
38     put_i(msg);
39     if(wasEmpty)
40       not_empty_.notify_all();
41   }
42   ...
43 };
```

Here an alternative implementation of the `Message_Queue` class [SSRB02b] was shown in a sparse way. This code fragment especially shows the notational simplicity of the new implementation of `Message_Queue::put()`. This simplicity on usage outweighs the disadvantage of an increased Surface–to–Volume Ratio of `Thread_Condition`, which should be kept to a minimum in general [FY98, pp 459–462], [Mey98, pp 107–111], [Str98, pp 819–821].

The rest of this chapter proposes a safe modification to `Thread_Mutex_Guard` and similar classes to do the above stuff.

1.4 Context

Languages with deterministic destruction of instances like C++ allow for a programming technique called Resource Acquisition is Initialization (RAII). A common application is the Execute–Around Object. The basic principle is to call the first member function of a pair of complementary member functions from the constructor of a utility class and the second one from the destructor. These classes are often referred to as guards.

Consider for example the following utility class:

Listing 1.5: Resource Acquisition is Initialization

```
1  class Ofstream_OpenClose_Guard {
2    std::ofstream &stream_;
3  public:
4    Ofstream_OpenClose_Guard(
5        std::ofstream &stream,
6        const char fileName[],
7        std::ios_base::openmode mode
8          =std::ios_base::out | std::ios_base::app)
9      : stream_(stream) {
10     stream_.open(fileName,mode);
11   }
12   ~Thread_Mutex_Guard(void) {
13     stream_.close();
14   }
15 };
```

Log files are a common tool for both debugging and auditing purposes. It is advisable to open a log file only for the short period something has to be logged and to close it right afterwards. This both keeps resource usage to a minimum and allows log files to be rotated by some external tool. The above class can facilitate this kind of usage pattern:

Listing 1.6: Using `Ofstream_OpenClose_Guard`

```
1 std::ofstream stream;
2 // Indicate certain errors by means of exceptions
3 stream.exceptions(std::ios_base::badbit|std::ios_base::failbit);
4 ...
5 {
6   Ofstream_OpenClose_Guard guard(stream,"application.log");
7   stream<<"a␣log␣message"<<std::endl;
8 } // On automatic destruction of "guard" "stream" will
9   // reliably be closed again.
```

If appending text to the file fails, an exception will be thrown. Even in this case the file will reliably be closed though there are no `try` and `catch` blocks—RAII and exceptions work well hand in hand.

Using RAII provides several advantages:

1. It turns explicit release statements into more convenient, implicit ones. This advantage is present in all examples shown throughout this paper.

2. It is a programming style particularly useful, where you were tempted to explicitly or implicitly call a `virtual` member function from the constructor or destructor of an Abstract Class [Aue95, Woo00], which would have resulted in trouble, because the implementation of the base class would have been called, which is probably not the intended one. Just using constructors and destructors in the Abstract Class only, it is not possible to accomplish such a late initialization and early cleanup behaviour. You will find an example for this in Listing 1.14 in Section 1.9.2.

3. Even in the advent of an exception the release operation is reliably executed. The same holds for multiple return points from functions, which is a recommended style [sin], [Fow99, pp 250–254]. This advantage is present in all examples shown throughout this paper.

4. Factoring some initialization and cleanup actions into a separate guard enables reuse of the instance affected avoiding potentially expensive, repeated destruction and construction inbetween. Especially instances representing Value Objects [FS00, pp 83–84] are good candidates for this style, as object identity does not matter with them. Examples for this advantage look similar to Listing 1.14.

5. Factoring some initialization and cleanup actions into a separate guard is necessary for states of resources, which are expensive in some sense. The duration such states persist can be limited to short periods of time this

way. As the stack provides for deterministic, immediate destruction of instances, which run out of scope, RAII can release not only memory, but even resources more limited or critical [NW00]. Garbage collection today, on the other hand, is primarily designed with memory in mind and could therefore not serve for this purpose. This fact gave birth to the Dispose pattern in C# [Mic05] and Java [AGH01, pp 228–230], [rel], see further [Hen00, pp 6–7].

In this sense limiting resource usage happens in Listings 1.2, 1.4, 1.6 and 1.9 and potentially in Listings 1.3 and 1.8.

The above design of `Ofstream_OpenClose_Guard` has one flaw, however: It does not prevent wrong usage as much as it could:

Listing 1.7: Bad things could happen if copy was not prohibited.

```
1 {
2   Ofstream_OpenClose_Guard guard0(stream,"application.log");
3   // Built in copy constructor called, i.e. bitwise copy
4   Ofstream_OpenClose_Guard guard1(guard0);
5   stream<<"a␣log␣message"<<std::endl;
6 } // Both "guard1" and "guard0" are going to be destroyed here,
7   // therefore "stream.release()" called twice!
```

To prevent this scenario, `Ofstream_OpenClose_Guard` should declare its copy constructor `private` and should not define it similarly as shown in Listing 1.1. This forces the compiler to reject the copy of guards. Compile time errors are preferred over run time errors. The same reasoning also holds for the assignment operator.

Using RAII guards therefore normally has the following disadvantage, too, however:

- You have to prohibit copy and assignment.

1.5 Problem

RAII guards can neither be copied nor assigned. If copy and assignment were allowed, cleanup would be performed as many times as there were guards referring to the same resource. From a conceptual point of view RAII guards are Reference Objects [FS00, pp 83–84], because they internally associate another instance.

This means in particular, that you can not return guards from functions. This is a big limitation, as it means, that you can not factor code which initializes the guard into a separate function. This wish is related to the intent behind the refactorings Extract Method [Fow99, pp 110–116], Replace Parameter with Method [Fow99, pp 292–294], and Replace Constructor with Factory Method [Fow99, pp 304–307] and can lead to better encapsulation.

1.6 Forces

- Initialization and clean up should be factored in a separate class implementing RAII.
- RAII guards are Reference Objects and thus must neither be copied nor assigned.
- An instance which can not be copied can not be returned by value from a function.
- Return by value from a function is a very special case of copy as the instance copied always will run out of scope immediately afterwards.
- Return by value is preferred over non `const` reference parameters, because it is more usual.
- The Counted or Detached Counted Body idiom [Cop00, pp 173–179], [Lak96, pp 429–434], [Str98, pp 841–845] resp. Shared Ownership [Car96] can be used for resource management and results in relatively cheap copy and assignment operations. But if we know in advance, that a reference count was either 0 or 1 at any time, which especially holds for temporaries, then even this idiom is overkill.
- On return from functions by value the copy operation does not need to be thread safe with respect to the instance being copied.
- Temporaries are as expensive as the respective copy constructors. Therefore return by value can be expensive especially if no optimizer jumps in.
- Function signatures should clearly express guarantees and missing guarantees.

1.7 Solution

If you need to forbid copy and assignment, then think twice whether it would be better to allow for moving copy and moving assignment instead. Moving means with classes implementing the Resource Acquisition is Initialization technique in particular to move the authority over the instance referenced from the right hand– to the left hand–side. After the move on destruction of the right hand–instance nothing will happen any more. A moving assignment additionally will first call the same operation as the destructor would have called to the instance referenced by the left hand–side before the move takes place.

A moving copy and a moving assignment differ from a usual copy and a usual assignment in so far as the former violate the postcondition that both instances which participated in copy resp. assignment evaluate as equal afterwards. The

Table 1.1: Classes, responsibilies, and collaborations

AbstractResource	
Set of complementary member function interfaces	

(a) AbstractResource

Client	
Calls	GuardGenerator
Moves on stack	Guard

(b) Client

ConcreteResource	
Implements	AbstractResource

(c) ConcreteResource

Guard	
Ensures state	AbstractResource
Passes authority	Guard
Takes authority	Guard

(d) Guard

GuardGenerator	
Returns by value	Guard

(e) GuardGenerator

right hand–side becomes modified and in general remains only destructible afterwards [Ale03].

A first sketch of the solution is shown in Table 1.1. The client uses a Guard to get the promise, that AbstractResource has a certain state during the lifetime of the Guard. As long as the Guard exists, the client can safely access the AbstractResource. A Guard can pass its authority over the state of the AbstractResource to another Guard both on moving copy and on moving assignment.

1.7.1 Participants

AbstractResource An instance of a class prescribing a pair of complementary member functions to be implemented by concrete classes. Such classes are sometimes referred to as resources. You do not need access to the source code of this class.

Client Client code ensures that AbstractResource has a certain state. To do so, it calls GuardGenerator to get a Guard.

ConcreteResource An instance of a class implementing or overwriting the `virtual` member functions declared by AbstractResource. You do not need access to the source code of this class.

Guard A class referencing the AbstractResource instance and applying calls to its pair of member functions on construction and destruction. This idiom especially describes the collaboration between two instances of the Guard:

- An instance constructed with AbstractResource as reference parameter.

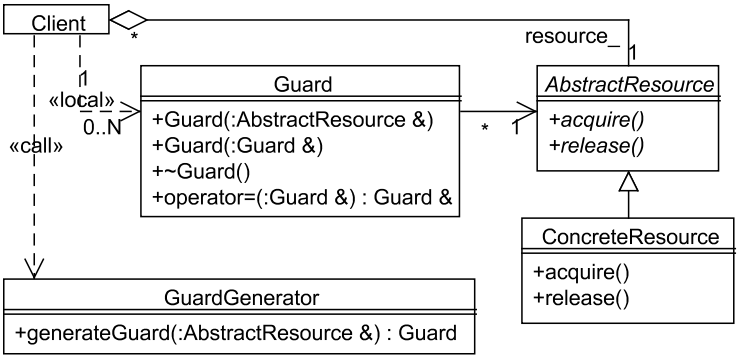


Figure 1.1: Class diagram illustrating return by value with moving copy

- Another instance the first one is moving copied or moving assigned to.

GuardGenerator A class with a member function which returns a Guard by value.

Figure 1.1 sketches the participants and their relations to each other. The class diagram basically provides information that this idiom still works, if Guard does not depend on ConcreteResource, and that AbstractResource is shared among multiple Clients.

Note that it has been left open, whether a single Client can guard an instance of AbstractResource multiple times at the same time. Though in most cases N must not exceed 1, there are AbstractResources, which you can define a Guard for with arbitrarily large N.

1.7.2 Dynamics

The core of this idiom is its dynamics. Instead of prohibiting copy and assignment of Reference Objects, allow for a *moving* copy and a *moving* assignment.

The dynamics of returning by value with moving copy is shown in Figure 1.2. The period between return from the *GuardGenerator::generateGuard()* member function and the point in time when *:Guard* runs out of scope is the window during which *:Client* can safely perform the appropriate actions. This sequence diagram shows the worst case without help from a clever optimizer. An optimizer can perform two optimizations here: It is likely, that the instance *[temporary]* can be optimized away, because the instance *:Guard* can be constructed with *local* as an argument, thus taking the role of *[temporary]* [BM00, pp 72–75]. This optimization is not possible in case of a moving assignment instead of copy, however, because in this case *:Guard* has already been constructed before. The other kind of optimization is known as Return Value Optimization and means, that *local* can be skipped, thus constructing the returned instance

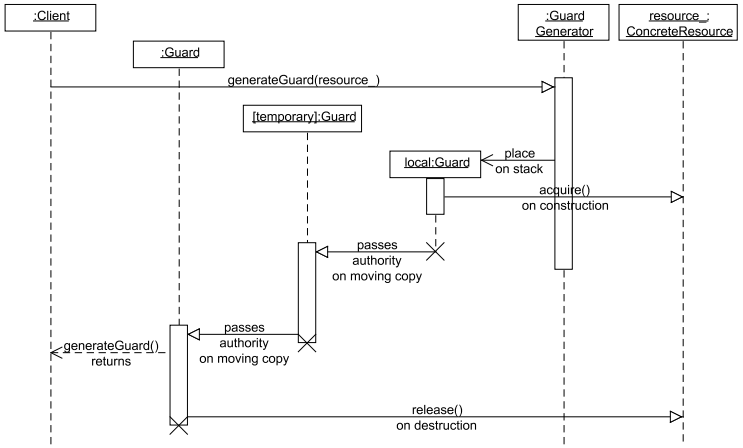


Figure 1.2: Sequence diagram illustrating return by value with moving copy without any optimization

of the class *Guard* into *[temporary]* [BM00, pp 59–65], [HDA02]. This optimization is possible both on moving copy and on moving assignment. A compiler may combine both optimizations and thus construct the returned instance of *Guard* directly into *:Guard*. Following sequence diagrams will contain these optimizations mainly for readability purposes.

1.7.3 Rationale

This section illustrates, how the Change of Authority idiom solves the problem stated.

We want to be able to return guards from functions, but as guards are Reference Objects, there is neither a copy constructor nor an assignment operator. The basic insight is, that there is no need for the total contract copy and assignment provide. A weaker contract still suffices to return guards from functions. It suffices to implement only a moving copy resp. a moving assignment instead, because the right hand side is thrown away right after the return anyway on leaving the scope given by the function body.

1.8 Resulting Context

The idiom proposed is a way to combine the advantages of the RAII technique with return by value. It also provides a means to reduce the cost of temporaries.

Consider the following function which uses the example from Listing 1.5.

Listing 1.8: Function which generates an instance of `Ofstream_OpenClose_Guard`

```
1 Ofstream_OpenClose_Guard generateCurrentOpenClose_Guard(
```

```
                                std::ofstream &stream,
                                const char fileName[],
                                std::ios_base::openmode mode
                                  = std::ios_base::out
                                    |std::ios_base::app) {
  const std::time_t now(std::time(0));
  const std::tm *const timeStruct=std::gmtime(&now);
  Ofstream_OpenClose_Guard guard(stream,fileName,mode);
  stream<<timeStruct->tm_year+1900
        <<timeStruct->tm_mon+1
        <<timeStruct->tm_mday
        <<":␣";
  return guard;
}
```

This way we are able now to automatically produce log files prefixing each line with the current date in the format `20051215:` without adding another constructor to `Ofstream_OpenClose_Guard` (which might not be possible anyway due to the lack of source code):

Listing 1.9: Using `generateCurrentOpenClose_Guard()`

```
{
  Ofstream_OpenClose_Guard guard(
    generateCurrentOpenClose_Guard(stream,"application.log"));
  stream<<"Paper␣finally␣submitted␣to␣PLoP."<<std::endl;
} // On automatic destruction of "guard" "stream" will
  // reliably be closed again.
```

Some consequences are listed below, that may be given consideration to by people who apply this idiom.

1.8.1 Pros and Cons

The Change of Authority idiom has the following benefits:

1. *Return guards from functions possible.* Functionality can be factored into functions now, even if they should return guards. The context around the construction of guards can be as complicated as the generation of any other entity within an application. A guard could model a running thread pool for example, and the number of threads the pool should consist of might have to be read from a configuration file first. It was a bad idea to introduce a new constructor for this special case taking a file name. Instead the number should be read from the file and then be fed to an already existing constructor that takes the number as an integral parameter, and both steps should be grouped together in a function.

2. *Information hiding.* On return from a function the caller only gets a guard—the instance the guard refers to can be completely hidden within the guard. In the case of mutual exclusion locks this might be seen as an advantage. There are cases, however, where RAII guards are also

implemented as Proxies (see Section 1.10.1 and [Hen00, pp 8–14]). In this case access is unified at least.

3. *Low runtime overhead.* An alternative to Change of Authority was reference counting as done in the Counted or Detached Counted Body idiom resp. Shared Ownership. But with the latter a counter has to be managed and—worse—memory has to be allocated for it on the free store, which is bad both in terms of runtime overhead and in terms of possible exceptions.

4. *Nothrow Guarantee.* As also indicated below in Section 1.9 moving copy and moving assignment fulfill the Nothrow Exception Safety Guarantee [Sut00a] [Hen00, p 2]. This means, that no exceptions will ever be thrown on either of these two operations. The reason for this is that these member functions only operate on data types which do not throw by definition during copy and assignment. The Nothrow Guarantee is a prerequisite to implement assignment operators which are Strongly Exception Safe [Sut00f] [Hen00, p 2].

5. *Temporaries made cheaper.* This idiom is especially well suited to optimize the cost induced by temporaries. Temporaries are a very special kind of data different from free store, heap, and stack. This is a recurring topic, and related solutions have been suggested so far [Ale03], [HDA02]. There are also idioms especially concerned with temporaries [Lak96, p 621], [Mey99, pp 114–117], [Sut00g], and more sophisticated tools like Expression Templates [VJ03b], [CE00, pp 464–465, 680–708] have been proposed to avoid temporaries. In the special case of return by value from a function the advantage is, that the usual deep copy performed on returning can safely be turned into a moving copy, because the local variable becomes automatically destroyed immediately after the copy anyway.

The Change of Authority idiom has the following liabilities:

1. *Two step initialization considered harmful.* Advantage 4 on Page 8 means, that you can use Change of Authority to factor some of the initialization and cleanup code from the constructor and destructor of a class into a guard class which accompanies the stripped class. If your class is usable even without the second initialization step already being performed, this is a good solution. This is the case, if the class represents a Null Object [Woo98] as long as there is no corresponding guard. If it is not possible to implement two step initialization this way, the advantage has to be carefully traded against potential errors caused by the usage of instances only initialized partially before, however.

2. *In need for guard classes.* For each Abstract Class you want a guard for you have to implement a companion class. There are classes you may even want to have multiple guard classes for.

   This issue might be solved by generative programming techniques like C++ templates, though this is far from trivial in the general case, as the

configuration effort of such templates can equal the effort to build a RAII class from scratch. This configuration effort furthermore obfuscates the generation of the RAII classes, because it means to use some Inversion of Control / Strategy / Policy. Usage would also become obfuscated by the fact, that such a template would have a unified constructor signature or at most a limited set of different constructors, which likely does not match the needs of a concrete template instantiation exactly. In short, as this disadvantage can not completely be resolved, it has to be traded against the advantages.

3. *Danger of confusing move with usual copy.* Not looking at the signatures of the respective constructors or assignment operators there is no easy, natural way to guess what is actually going on. The syntax of using moving copy resp. assignment does not differ from the syntax of copy resp. assignment you are used to. This can lead to surprises like [Mey01, pp 40–43].

1.8.2 The `std::auto_ptr<>` way of Change of Ownership

The following discussion strives for three goals: Firstly it shows another motivation for Change of Authority. Change of Authority is not only moving copy and moving assignment, but also a generalization of Change of Ownership. Secondly it emphasizes the best known example for Change of Authority. After reading this section the reader thirdly will have seen the consequences of delegating `AbstractResource::init()` to the caller.

The C++ standard class template `std::auto_ptr<>` takes a pointer to an instance and takes strict ownership over this instance on construction. On destruction of the instance `delete` is called. To protect its users from bad surprises due to automatic type conversions the constructor is declared `explicit`, thus it doesn't participate in such conversions. Otherwise situations might occur, where the owning entity is not obvious any more resulting in multiple owners and multiple destruction of the same instance. In other words, an instance of `std::auto_ptr<>` is an Object as Sole Owner [Car96], which is an application of the Resource Acquisition is Initialization technique. That is the reason for the first part of the name, `auto`. Instances of `std::auto_ptr<>` are primarily intended to be used as automatic variables and as class attributes.

With `std::auto_ptr<>` advantage 1 on page 8 means to turn free store into stack semantics. So even in cases, where you absolutely cannot create instances on the stack, as is the case e.g. with the Factory family of design patterns [GHJV96a, GHJV96g] or with Virtual Constructors [Cop00, pp 184–185], [Cop92, pp 140–160, 288–295], [Str98, pp 452–453] and Prototypes [GHJV96k] resp. Exemplars [Cop92, pp 279–306], which both are special incarnations of Abstract Factories in fact, `std::auto_ptr<>` reestablishes the convenience of stack semantics and lets the use of polymorph pointer types feel like fundamental types, which is a general goal of the C++ type system [Str94, pp 285–326]. Advantage 3 on page 8 means here, that even if during the usage of this guard

an exception gets thrown and remains uncaught within the block which defined the guard before, then the destructor of the guard will reliably be called and no memory leak will result.

`std::auto_ptr<>` implements moving copy and moving assignment, i.e. additionally to the construction mentioned above it also supports a second kind of Change of Ownership. This is a specialization of Change of Authority. The creation of an instance is placed to the responsibility of the client. This makes it possible to hide the dynamic type of the `AbstractResource` completely from the `std::auto_ptr<>` instance. In other words, an `AbstractResource::init()` member function has no relevance to this guard.

This has two consequences for people who want to design classes similar to `std::auto_ptr<>`.

Firstly, it has to be decided, what to do with the instance of `AbstractResource` just created, if on construction of the Change of Ownership–container an exception gets thrown. With `std::auto_ptr<>` this is a non–issue, as its constructor involved here does not throw. But with counting pointers this is an issue, as they allocate memory within their respective constructor for the reference counter, which may fail. `boost::shared_ptr<>` [ACD] takes ownership even in the case of an exception, i.e. after the constructor returns with an exception the instance given as an argument has already been destroyed. This is necessary to guard code like `boost::shared_ptr< int > i(new int)` against potential memory leaks—otherwise you could not safely create the instance within the argument list of the constructor. But this decision implies, that even in case of an exception a side effect [Mey97, pp 748–764] will happen, which somewhat contradicts the goal of Strong Exception Safety.

Secondly, the flexibility gained by delegating the task of `AbstractResource::init()` to the client also means, that you may want to provide a way to configure the behaviour of `AbstractResource::clear()` in a more powerful way than possible by polymorphism. While `std::auto_ptr<>` does not allow for that, `boost::shared_ptr<>` does. With the latter you can even manage e.g. fields allocated with `new[]` or instances placed into shared memory therefore.

The second part of the name, `ptr`, expresses, that `std::auto_ptr<>` is a smart pointer. It is a Proxy [GHJV96l, BMR+00b] [Hen00, pp 8–11] granting access to the instance aggregated by means of its member `operator->()`. This way an instance of `std::auto_ptr<>` lets the user e.g. call member functions of `AbstractResource` as if it was a pointer.

`std::auto_ptr<>` further more provides templated member functions and operators which let it reflect potential type relationships between classes provided at instantiation of the `std::auto_ptr<>` template. Given that `A` and `B` are classes related to each other—e.g. `A` inherits from `B`—, `std::auto_ptr< A >` and `std::auto_ptr< B >` are therefore related, too. This increases the flexibility of moving copy and moving assignment and mimics the object oriented flexibility with entities of `A *` and `B *`.

1.8.3 Exception safety requirements of `AbstractResource`

Destructors must not throw exceptions. One reason for this is, that if an exception becomes thrown somewhere the stack is being unwound until an appropriate exception handler catches the exception thrown. Stack unwinding means, that destructors will be called. If a destructor throws during this procedure, you will end up with two exceptions in parallel. The second exception basically indicates, that it proved impossible to unwind the stack, which in turn means, that there is no clean way anymore to terminate the application. Not being able any more to gracefully react upon exceptions [NW00] or even to cleanly stop your application if errors have occured should be restricted to really fatal errors, you cannot write useful exception handlers for anyway [Str98, p 393], [Str94, p 492].

To get the destructor of the guard class proposed non–throwing, the following assumption has to be made: `AbstractResource::clear()` must not throw, if and only if the last member function of the pair `AbstractResource::init; clear()` called was `AbstractResource::init()`. So there is no need to require `Abstract Object::clear()` not to throw at all. This promise can not be expressed in terms of C++ exception specifications. A way to reliably achieve this prerequisite is not to declare both the `AbstractResource::init()` and `AbstractResource:: clear()` `public` and the RAII class to be a `friend` of the class it refers to. This requires access to the source code of `AbstractResource`, however.

1.9 Implementation

In this section two examples will be presented. First the open example from Section 1.3 is going to be solved in depth. The second example illustrates a Change of Authority guard suitable to solve a potential problem with `MQ_Scheduler`.

A third subsection proposes an idiom which might accompany the Change of Authority idiom.

1.9.1 Example Resolved

The example taken from [SSRB02d] and already mentioned in Section 1.3 can be changed such that instead of prohibiting copy and assignment it fully supports the ideas layed out throughout this paper.

The original example with moving copy and moving assignment added is shown below.

Listing 1.10: Change of Authority added to Listing 1.1

```
15 class Thread_Mutex_Guard {
16   Thread_Mutex *lock_;
17 public:
18   explicit Thread_Mutex_Guard(Thread_Mutex &lock) : lock_(&lock) {
19     lock_->acquire();
20   }
```

```
21   ~Thread_Mutex_Guard(void) {
22     if(lock_)
23       lock_->release();
24   }
25   // Moving copy
26   Thread_Mutex_Guard(Thread_Mutex_Guard &rhs) throw()
27     : lock_(rhs.lock_) {
28     rhs.lock_=0;
29   }
30   // Moving assignment
31   Thread_Mutex_Guard &operator=(Thread_Mutex_Guard &rhs) {
32     if(rhs.lock_!=lock_) {
33       if(lock_)
34         lock_->release();
35       lock_=rhs.lock_;
36       rhs.lock_=0;
37     }
38     return *this;
39   }
40 };
```

Both moving copy and moving assignment indicate by their respective signatures, that they only grant a weaker contract to its clients than a usual copy or assignment would: They do not promise not to modify the right hand side. In contrast to Listing 1.1 `Thread_Mutex_Guard::lock_` is a pointer and not a reference for two reasons independent from each other here:

- References are copyable, but not assignable—but the latter is necessary to implement assignment of `Thread_Mutex_Guard` instances.
- Pointers can point to nothing, i.e. `0`. This indicates here, that authority is lost.

The implementation of a moving assignment has to guard against self assignment, because the right hand side in the general case would also have been changed, which has to be handled as a special case here, therefore. The implementation simply returns the unchanged instance then. The same branch is going to be executed if both instances affected already lost authority—but this is just an optimization.

The constructor taking the reference to the lock is declared `explicit` not to contribute to automatic type conversions. This protects the user from bad surprises with locks acquired by accident and reflects the fact, that a guard is a different kind of thing than the instance it refers to.

Later versions of the C++ standard prohibit rvalues and thus temporaries from being bound to non `const` references [Str94, pp 105–106]. This means, that the usage of the naïve implementation shown on Page 18 will be rejected by a conforming compiler and therefore not work as expected. It needs to be slightly extended as shown below:

Listing 1.11: Adding a helper class to make `Thread_Mutex_Guard` compile

```
41 class Thread_Mutex_Guard {
```

```
42    ...
43  protected:
44    struct self_ref {
45      Thread_Mutex_Guard &guard_;
46      explicit self_ref(Thread_Mutex_Guard &guard) throw()
47        : guard_(guard) {}
48    };
49  public:
50    ...
51    // Another moving copy
52    Thread_Mutex_Guard(self_ref rhs) throw()
53      : lock_(rhs.guard_.lock_) {
54      rhs.guard_.lock_=0;
55    }
56    ...
57    operator self_ref(void) throw() {
58      return self_ref(*this);
59    }
60  };
```

The trick here is to encapsulate the reference by an instance which is returned and copied by value. This implementation is well known from `std::auto_ptr<>` from the C++ standard library.

One might ask why not to take the usual signature of copy construction and assignment and to cast the `const` of the argument away to implement the move. This was a bad alternative to the solution proposed as the moving character of these operations remains hidden from the interface (compare with [Lak96, p 612]). Further more casting away `const` can lead to undefined behaviour.

1.9.2 `Thread_Pool_Operation`: Change of Authority, Composite, and Iterator combined

Change of Authority can be combined with other idioms and patterns. This implementation example suggests a simple thread pool, thus also implementing the Composite pattern [GHJV96i]. The Iterator pattern [GHJV96h] is used to enable clients to access the threads the pool consists of. The code is as follows.

Listing 1.12: A pool of threads executing a function object each shared among them

```
1   extern "C" {
2     void *svc_run(void *);
3   }
4
5   class Thread_Pool_Operation {
6   public:
7     typedef std::vector< thread_type > pool_type;
8     typedef typename pool_type::const_iterator const_iterator;
9     typedef typename pool_type::size_type size_type;
10    struct command_adapter
11      : public std::unary_function< void,void > {
12      virtual ~command_adapter(void) {}
13      virtual void operator()(void) =0;
```

```
14    };
15  private:
16    template< class Command > class command_proxy
17      : public command_adapter {
18      Command &command_;
19    public:
20      explicit command_proxy(Command &command) : command_(command) {}
21      void operator()(void) {
22        command_();
23      }
24    };
25    std::auto_ptr< command_adapter > command_;
26    pool_type pool_;
27    void joinPool_(void) throw() {
28      for(pool_type::reverse_iterator in(pool_.rbegin());
29          pool_.rend()!=in;
30          ++in)
31        joinThread(*in);
32    }
33  public:
34    template< class Command >
35    Thread_Pool_Operation(size_type number_of_threads,
36        Command &command)
37      : command_(new command_proxy< Command >(command)) {
38      pool_.reserve(number_of_threads);
39      try {
40        for(size_type i(0);number_of_threads>i;++i)
41          pool_.push_back(createThread(svc_run,*command_));
42      }
43      catch(...) {
44        joinPool_();
45        throw;
46      }
47    }
48    ~Thread_Pool_Operation(void) {
49      joinPool_();
50    }
51    Thread_Pool_Operation(Thread_Pool_Operation &rhs) throw()
52      : command_(rhs.command_) {
53      pool_.swap(rhs.pool_);
54    }
55    Thread_Pool_Operation &operator=(Thread_Pool_Operation &rhs) {
56      if(rhs.command_.get()!=command_.get()) {
57        joinPool_();
58        pool_.clear();
59        pool_.swap(rhs.pool_);
60      }
61      command_=rhs.command_;
62      return *this;
63    }
64    const_iterator begin(void) const {
65      return pool_.begin();
66    }
67    const_iterator end(void) const {
68      return pool_.end();
69    }
70    ...
```

```
71 };
72
73 void *svc_run(void *arg) {
74   assert(arg);
75   (static_cast< Thread_Pool_Operation::command_adapter >(arg))();
76   return 0;
77 }
```

Note some details of this implementation:

- The Command does not need to support polymorphism. The injection of the Command is being performed statically. This differs e.g. from `java.lang.Runnable` and `java.lang.Thread` in Java, where the code is injected by subclassing implementing resp. overwriting the `run()` method [AGH01, pp 253–259].

- `Thread_Pool_Operation::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 exception.

- The existance of `std::vector<>::swap() throw()` is a true sign, that `std::vector<>` implements the Handle / Body idiom [Cop92, pp 58–62], [Lak96, pp 352–353], also known as the Pimpl idiom [Sut00b] and the Bridge pattern [GHJV96e].

- `std::auto_ptr<>` and `std::vector<>`, the template classes of both attributes, support Change of Authority out of the box. The first one does so exactly as proposed in this paper while with the second one an arbitrary instance can be `swap()`ped for an empty one [HDA02]. Thus the implementations of both moving copy and moving assignment of `Thread_Pool_Operation` can simply delegate their jobs. `std::auto_ptr<>` especially guards itself against self assignment.

- The thread Command remains a reference; each thread refers to the same instance. This design decision lets the client easily share data among the threads of a pool simply by adding attributes to the appropriate Command. Access to these may need to be serialized within `Command::operator()()`, however.

- `Thread_Pool_Operation::command_adapter` is an Adapter [GHJV96b], that serves for the purpose to provide a non templated way to apply the Command. This is necessary because it is assumed that the multithreading library of the operating system has a C programming interface and thus is agnostic about C++ templates. The Adapter has an arbitrarily large number of implementations. To each `Command` corresponds exactly one template instantiation of `Thread_Pool_Operation::command_proxy<>`,

which is a Proxy. This idea is similar to the idea behind the implementation of the External Polymorphism pattern [CSH98]. The mapping of a `Command` to its Proxy is performed by the templated constructor of `Thread_Pool_Operation` within its initialization list.

From a more abstract point of view the interaction between the constructor and the adapter mentioned bewares the class `Thread_Pool_Operation` from being a template class with `Command` as its parameter. Similarly to the deleter argument on construction of `boost::shared_ptr<>` instances this keeps the type relationship between thread pools intact even if they execute different Commands.

- `thread_type`, `createThread()`, and `joinThread()` wrap the respective operating system specific types and functions. `createThread()` translates error codes into exceptions. `joinThread()` must not throw if applied to a valid thread identifier not to let `Thread_Pool_Operation::joinPool_()` throw, which in turn ensures that `Thread_Pool_Operation::Thread_Pool_Operation()` does not throw.

  From a design point of view `thread_type` is a Future type, and the role of the Rendezvous function is taken by `joinThread()` [SSRB02a, pp 413, 417,423–430,435–436]. `Thread_Pool_Operation::pool_` is a Composite Future instance, and `Thread_Pool_Operation::joinPool_()` is the appropriate Rendezvous member function. Thus `Thread_Pool_Operation` turns asynchrony into synchrony.

- The destructor of `Thread_Pool_Operation` joins the threads. This ensures that no data is still being referenced by some threads after destruction of the instance of `Thread_Pool_Operation` simply as there are no such threads at that time anymore. This is especially important in the case of exceptions thrown by the thread that created an instance of `Thread_Pool_Operation` which still has not run out of scope. Related issues are known as the Orphaned Thread bug pattern [All02b]. This item corresponds to advantage 3 on page 8.

- The technical `self_ref` stuff has been skipped here as in Listing 1.10. It was almost the same as in Listing 1.11.

An application example for this thread pool is the `MQ_Scheduler` class proposed by [SSRB02a, pp 425–427]. `MQ_Scheduler` implements the Command design pattern [GHJV96c]: The code to be executed is the default implementation of the `virtual` member function `MQ_Scheduler::dispatch()`, the destination instance the command is applied to is the instance of `MQ_Scheduler` itself. Because the `MQ_Scheduler` must rely on the procedural multithreading library of the operating system, the Command pattern is indirectly implemented using the Strategy design pattern: The `extern "C"` function[1] `svc_run()` is the

[1] In the original paper it is a `static` member function of `MQ_Scheduler`. This decision was better in terms of encapsulation, but does not work with all multithreading libraries. POSIX Threads' `pthread_create()` e.g. requires the Strategy to be `extern "C"`.

Strategy, which turns itself into a Command by executing object oriented code supplied as its argument. As thread creation is performed by the constructor of `MQ_Scheduler`, the Command will be executed indirectly multiple times on construction of `MQ_Scheduler`. You might run into trouble, if you inherit from and overwrite its `MQ_Scheduler::dispatch()` member function. It is likely that the default implementation of this member function will be executed at least by some threads, which start "too fast", and not the implementation of the specialization; so by means of the keyword `virtual` configurability is being announced which actually does not exist. There are no timing problems, there are programming errors only, however. A RAII guard, which will start the threads in its constructor and may also join them in its destructor, will fix this.

With the help of our new class `Thread_Pool_Operation` thread pool management can be factored out of `MQ_Scheduler` for example. Doing so results in an example for a guard, where N of Figure 1.1 can be arbitrarily large. This also serves as an example, where two step initialization is not dangerous because an instance of `MQ_Scheduler` without any guards reduces to Null Object. The modified `MQ_Scheduler` is shown below.

Listing 1.13: Reengineering the constructor of `MQ_Scheduler` from [SSRB02a, p 425]

```
class MQ_Scheduler : public std::unary_function< void,void > {
  Activation_List act_queue_;
public:
  explicit MQ_Scheduler(size_t high_water_mark)
    : act_queue_(high_water_mark) {}
  void insert(Method_Request *);
  // virtual void dispatch(void); renamed to
  virtual void operator()(void);
  Thread_Pool_Operation generateThread_Pool_Operation(
      Thread_Pool_Operation::size_type number_of_threads) {
    return Thread_Pool_Operation(number_of_threads,*this);
  }
};
```

The constructor does not spawn any worker threads any more.

A client now can safely derive from `MQ_Scheduler` and overwrite even its `MQ_Scheduler::operator()()` without resulting in trouble. Let us say, there is such a child class called `Specialized_Scheduler`. Then usage is as follows:

Listing 1.14: Using `MQ_Scheduler`

```
Specialized_Scheduler scheduler;
// Let's go
const Thread_Pool_Operation
  gasoline0(scheduler.generateThread_Pool_Operation(42));
...
// Increase size of the worker pool
const Thread_Pool_Operation
  gasoline1(scheduler.generateThread_Pool_Operation(10));
```

```
...
// Decrease size of the worker pool
for(Thread_Pool_Operation::const_iterator ci(gasoline1.begin());
    gasoline1.end()!=ci;
    ++ci)
  cancelThread(*ci);
...
// Shut down the worker pool
for(Thread_Pool_Operation::const_iterator ci(gasoline0.begin());
    gasoline0.end()!=ci;
    ++ci)
  cancelThread(*ci);
// Now "Thread_Pool_Operation::~Thread_Pool_Operation()"
// won't block infinitely
```

The flexibility to dynamically increase and decrease the size of the thread pool is possible only because a single instance of `MQ_Scheduler` can be associated with an arbitrary number of instances of `Thread_Pool_Operation`.

Thread cancellation is used here only as an example to illustrate the Iterator interface.

Thread cancellation is not discussed in detail here because it is not portable. The `cancelThread()` placeholder works here only satisfactory if the platform provides a function similar to POSIX' `pthread_cancel()`, which has two important properties: The cancellation points where the target thread is about to finish at are well defined, configurable and extensible, and resource management can be performed on cancellation [Ste99, pp 187–192]. Both does not hold e.g. for MS Win32 `TerminateThread()`.

1.9.3 The Reset Idiom

Sometimes a way is needed to reset the Change of Authority guard. Either a `reset()` member function is introduced for this purpose as it exists with `std::auto_ptr<>`, or the following idiom can be used.

Listing 1.15: The Reset idiom

```
Guard guard(abstractResource);
const GuardedType cp(proto);
{
  const Guard reset(guard);
}
// no need for protection any more here,
// as "proto" not used any longer
cp.doSomething();
...
```

The purpose of this idiom is to allow the instance of `guardedType` to be copied (and not default constructed and then assigned) during the lifetime of the `guard`, but to use the copy later without the `guard`. Resetting authority can be dangerous, therefore this idiom looks so ugly. That is the same design principle as with the new `*_cast<>()` operators in C++: Dangerous actions should still be

possible, but they should syntactically be that complex that they are unlikely to be used thoughtless [Str94, pp 395–398]. This is an advantage of the Reset idiom over a simple `reset()`.

1.10 Variants

Change of Authority can be combined with other idioms and patterns. Some of these cases are documented in this section.

1.10.1 Change of Authority and Proxy combined

If the class implementing Change of Authority also implements the Proxy design pattern, then the user must use the instance with caution after it participated in a move on the right hand–side as the object reference is invalid afterwards. Proxy behaviour should be assumed only for the period the Change of Authority guard also holds authority over the instance referenced, therefore. In other words, the only guarantee granted to the Change of Authority instance after it lost authority is that it is still destructible.

1.10.2 Change of Authority and Composite combined

Combining Change of Authority and Composite lets the guard manage more than one resource at a time and occasionally pass the authority over all resources to another guard. An example for this case has been layed out in Section 1.9.2.

1.10.3 Change of Authority and Iterator or Visitor combined

Change of Authority and Iterator or Visitor [GHJV96d] combined is to guards implemeted as Composites (see Section 1.10.2), what Change of Authority and Proxy combined (see Section 1.10.1) is to guards managing a single resource. The example presented in Section 1.9.2 also applies for this case. As with the combination with Proxy care must be taken not to let guards return Iterators resp. to let guards accept Visitors after loss of authority over the items stored in the Composite.

1.10.4 Change of Ownership

As discussed in Section 1.8.2, Change of Ownership sometimes is implemented as a special case of Change of Authority.

1.11 Known Uses

1.11.1 `std::auto_ptr<>`

The template class `std::auto_ptr<>` implements Change of Authority and combines this idiom with Proxy. It is an example for the variants described in Sections 1.10.1 and 1.10.4. `std::auto_ptr<>` is a good choice for the return of functions which belong to the Factory family of design patterns.

1.11.2 The `swap()` member function

The C++ standard Composites implemented by means of the Bridge design pattern declare a `swap()` member function each. This bidirectionally changes authority over the implementation. Strong Exception Safety in general is expensive with Composites, because the Memento design pattern [GHJV96j] likely needs to be applied, which in turn requires a deep copy. Therefore the standard C++ algorithms operating on the Composites of the C++ standard library only provide for the Basic Exception Safety Guarantee [Sut00e, pp 47–48] [Hen00, p 2]. With help of the `swap()` member function the user still has the freedom to turn this guarantee into the Strong Guarantee [Abr00], though. To syntactically come closer to the symmetry inherent in `swap()`, the algorithm `std::swap<>()` can be used instead of the `swap()` member function.

To really support moving copy and moving assignment for all classes supporting `swap()`, a templated Proxy can be built similarly to the example shown in Section 1.9.2.

1.12 Related Patterns

Many design patterns have already been mentioned in this paper. The patterns most related to Change of Authority are repeated here.

Abstract Class is controlled by the Change of Authority guard.

Bridges sometimes are equipped with Change of Authority by means of a `swap()` member function.

Composite, Iterator, Proxy, and Visitor can help with the implementation of Change of Authority as shown in Section 1.10.

The reference counting employed in Counted or Detached Counted Body can be an alternative for Change of Authority. While with Change of Authority the authority of the guard instance over a certain state the AbstractResource can take is strict and exclusive and therefore like the Composition relationship in UML, reference counting leads to a relationship similar to Aggregation, i.e. the state of AbstractResource persists until the last guard runs out of scope then.

Resource Acquisition is Initialization is the foundation of Change of Authority.

Simplicity is not an end in art,
but we usually arrive at
simplicity as we approach the
true sense of things.

Constantin Brancusi

Chapter 2

The Thread Safe Interface design pattern goes `synchronized`

After reading this chapter, you will have seen an application of the Change of Authority idiom introduced in Chapter 1. An alternative and general implementation of the Thread Safe Interface design pattern [SSRB02e] [Hen00, pp 21–22] will be the result.

2.1 Context

You are about to design an application from your own and foreign libraries. The application design prescribes multiple concurrent control flows to be implemented by means of threads or processes.

2.2 Problem

Access to an instance shared among different concurrent control flows should be made threadsafe[1], i.e. the observable state of the instance must not be corrupted by any means during concurrent access. In other words, serial equivalence must be guaranteed. How to turn the interfaces of classes, especially of legacy classes which are not threadsafe, into threadsafe ones?

[1] For simplicity of description only threads are talked about here—the same applies to processes, too.

2.3 Forces

- Programmers take two roles: They use existing code and they provide their work as a library (compare to [Str02, Ven04]). Your code will get old, older than you imagine today [FY98]. Therefore it is not enough that *its inventors* know how to safely use their code—in the long run the code will be used by others like a library and should meet the quality criteria demanded on libraries.
- The compiler is programmer's friend. Libraries should use the capabilities of the programming language and the compiler to guide users of the library to correct usage.
- Locking is expensive. The appropriate pair of calls is expensive, and locking limits scalability.
- You may need serialization, you may need synchronization, too.
- Though turning an unsafe class into a threadsafe one is straightforward, it still has to be done for each class in question. `synchronized` is nice in Java—is there an equivalent in other programming languages, too?
- Threadsafe Interfaces are related to their unsafe counterparts in a non–symmetrical way. A Thread Safe Interface can be used as an unsafe one, but not vice versa.
- Serialization and synchronization can be thought of as concerns which crosscut business logic [KLM+97].
- Simplicity matters.

2.4 Solution

Add Change of Authority to Alexandrescu's `LockingPtr<>` [Ale01]. Add an instance of `Thread_Mutex` and a member function to your class `A` declared `volatile`, which otherwise only has non–`volatile` member functions. Let this new member function return an instance of `LockingPtr< A >` by value which refers to both `*this` and the lock just added. Declare each instance of `A` `volatile` which is to be shared among different threads.

2.4.1 Rationale

The `LockingPtr<>` combines Scoped Locking [SSRB02d] with Proxy [GHJV96l, BMR+00b] [Hen00, pp 8–14], but differently from Section 1.10.1: The Proxy does not provide access to the instance of `Thread_Mutex` it refers to, but instead it also references the instance access should be made threadsafe to, and this is given access to by the Proxy while casting away `volatile`.

Listing 2.1: `LockingPtr<>` according to [Ale01]

```
template< class T > class LockingPtr {
  T &t_;
  Thread_Mutex &lock_;
  // No copy allowed, therefore private and declared only
  LockingPtr(const Thread_Mutex_Guard &);
  // No assignment allowed, therefore private and declared only
  LockingPtr &operator=(const LockingPtr &);
public:
  LockingPtr(volatile T &,Thread_Mutex &lock)
    : t_(const_cast< T & >(t)), // As there's no "volatile_cast<>"
                                // in C++, "const_cast<>" casts
                                // away both cv-qualifiers "const"
                                // and "volatile".
      lock_(lock) {
    lock_.acquire();
  }
  ~LockingPtr(void) {
    lock_.release();
  }
  T &operator*(void) const volatile throw() {
    return t_;
  }
  T *operator->(void) const volatile throw() {
    return &t_;
  }
};
```

`LockingPtr<>` can be equipped with Change of Authority semantics exactly as shown in Section 1.9.1. The recipe of Section 2.4 then results in the following code:

Listing 2.2: Thread Safe Interface restructured

```
class ThreadSafeInterface {
  mutable volatile Thread_Mutex lock_;
  // Some data, e.g.
  size_t size_;
public:
  typedef LockingPtr< ThreadSafeInterface > locking_ptr;
  typedef LockingPtr< const ThreadSafeInterface > const_locking_ptr;
  ...
  void incrementSizeBy(size_t s) { // not part of the
    size_+=s;                      // Thread Safe Interface
  }
  size_t getSize(void) const { // not part of the
    return size_;              // Thread Safe Interface
  }
  locking_ptr generateLockingPtr(void) volatile {
    return locking_ptr(*this,lock_);
  }
  const_locking_ptr generateConstLockingPtr(void) const volatile {
    return const_locking_ptr(*this,lock_);
  }
};
```

As an instance should be declared `volatile` if it is intended to be potentially accessed by multiple threads at the same time according to Alexandrescu's idea, these threads can only access its `volatile` member function, which returns a reflexive `LockingPtr<>`. Because this is a specialization of Scoped Locking, the acquisition of the lock member is coupled to the construction of the `LockingPtr<>`, while destruction implies that the lock will be released immediately. During the lifetime of the instance of `LockingPtr<>` the access even of the unsafe interface of `A` can be considered safe therefore, which is manifested in the cast done by the `LockingPtr<>`. So the client now has the full view on the interface of `A`. See [Cop92, pp 26–28], [Lak96, pp 605–606].

2.5 Resulting Context

Now your class `A` exposes two interfaces: A threadsafe one and an unsafe one. The latter is only visible if `A` is not shared among different threads. The Thread Safe Interface is visible regardless of whether `A` requires threadsafe handling or not. So threadsafety is a static property of a class, and thread unsafe access will be rejected by the compiler.

A limitation of the solution proposed results from the fact that there is only one `volatile` keyword, and therefore this solution can only manage one mutex lock per instance. If your classes to add a Threadsafe Interface to allow for finer grained locking, then the original implementation applies.

Furthermore only such classes could be treated as proposed where serialization suffices and no changes of the interface are required to gain threadsafety. For all other cases refer to e.g. the Monitor Object design pattern [SSRB02b]. Synchronized queues for example need both mutual exclusion locks, i.e. serialization, and condition variables, i.e. synchronization, must not expose `front()` and `pop()` in two separate functions to its clients and can not let `pop()` return by value for the sake of Exception Safety [Sut00c, pp 47–48], see further [Hen02], thus must publish an interface different from single threaded queues like C++ `std::queue<>` [Sut00c].

Classes granting access to their implementation to a client by means of member functions returning by reference are not very suitable for the solution proposed, too, as indirect access through a `LockingPtr<>` can not change the return type of member functions. These member functions would have to return `volatile` references on threadsafe access not to indirectly cut holes into threadsafety. Compare with [Lak96, pp 607–612].

As a special case of this issue it should be considered to support Visitors [GHJV96d] instead of Iterators [GHJV96h] with Thread Safe Composites [GHJV96i] because then the Composite has full control over the traversal due to the Strategy [GHJV96m] resp. Inversion of Control nature of the Visitor design pattern. An Iterator is a vehicle to access or modify the internal state of a Composite during the whole lifetime of the Iterator. Once the Composite issued an Iterator, it has no control over it any more. Different from Visitors, in a concurrent environment Iterators therefore can lead to race conditions.

2.6 Implementation

The solution proposed in Section 2.4 can be cast into a single class template `Serialized<>`, thus adding a Thread Safe Interface to the whole series of classes characterized in Section 2.5.

Here an additional generalization is being proposed: To further reduce the impact of locking on scalability, readers / writer locking is used. While `ReadLockingPtr<>` calls `ReadersWriter_Mutex::readAcquire()` on construction and takes the instance of `ThreadUnsafeInterface const`, `WriteLockingPtr<>` calls `ReadersWriter_Mutex::writeAcquire()` and takes the instance of `ThreadUnsafeInterface` non `const`. Note that `Serialized<>::generateConstLockingPtr()` and `Serialized<>::generateLockingPtr()` differ by name and not only by the `const` qualification. Clients can explicitly assume the role of readers even if they could have non `const` access to the instance of `Serialized<>`.

The statics is shown in Figure 2.1, the dynamics in Figure 2.2. Example code is as follows:

Listing 2.3: Combining Change of Authority and `LockingPtr<>`

```
template< class ThreadUnsafeInterface > class Serialized
  : public ThreadUnsafeInterface {
public:
  typedef ThreadUnsafeInterface value_type;
  typedef WriteLockingPtr< Serialized > locking_ptr;
  typedef ReadLockingPtr< const Serialized > const_locking_ptr;
private:
  mutable volatile ReadersWriter_Mutex lock_;
  Serialized &assign_(const Serialized &rhs) {
    value_type::operator=(rhs);
    return *this;
  }
  Serialized &assign_(const volatile Serialized &rhs) {
    const_locking_ptr lptrrhs(rhs.generateConstLockingPtr());
    return assign_(*lptrrhs);
  }
public:
  Serialized(void) {}
  Serialized(const Serialized &rhs) : value_type(rhs) {}
  Serialized(const volatile Serialized &rhs)
    : value_type(*rhs.generateConstLockingPtr()) {}
  // Intentionally not "explicit"
  Serialized(const value_type &tui) : value_type(rhs) {}
  Serialized &operator=(const Serialized &rhs) {
    if(&rhs==this)
      return *this;
    return assign_(rhs);
  }
  Serialized &operator=(const volatile Serialized &rhs) {
    if(&rhs==this)
      return *this;
    return assign_(rhs);
  }
```

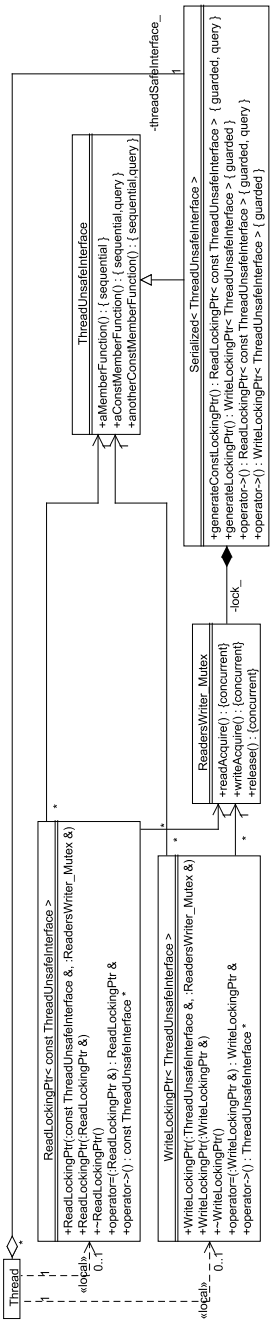
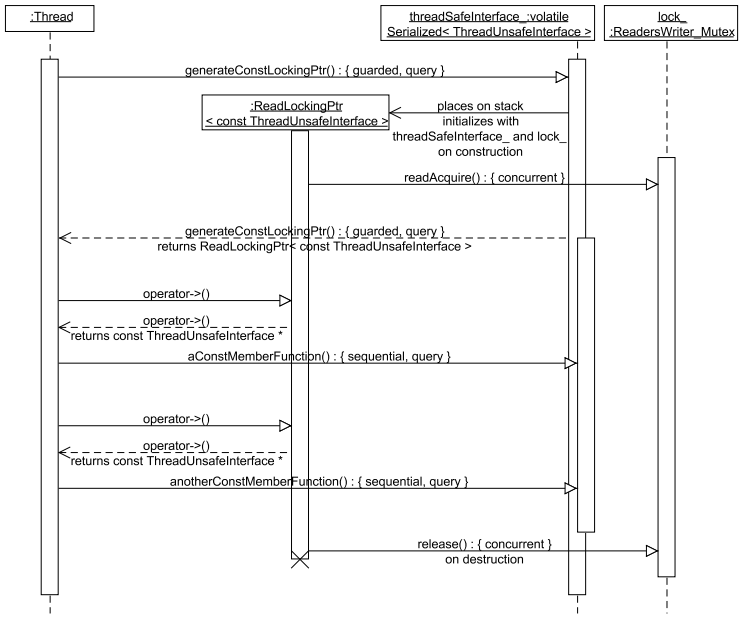
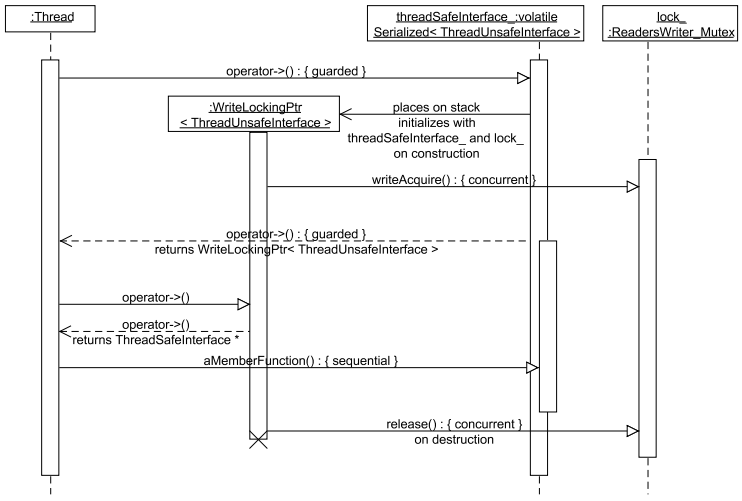


Figure 2.1: Class diagram illustrating the `Serialized<>` class template



(a) One atomic call to multiple { `sequential,query` } member functions



(b) The usage of the convenience member `operator->()`

Figure 2.2: Sequence diagrams illustrating the `Serialized<>` class template

```
34    volatile Serialized &operator=(const Serialized &rhs) volatile {
35      if(&rhs==this)
36        return *this;
37      locking_ptr lptr(generateLockingPtr());
38      return lptr->assign_(rhs);
39    }
40    volatile Serialized &
41    operator=(const volatile Serialized &rhs) volatile {
42      if(&rhs==this)
43        return *this;
44      locking_ptr lptr(generateLockingPtr());
45      return lptr->assign_(rhs);
46    }
47    locking_ptr generateLockingPtr(void) volatile {
48      return locking_ptr(*this,lock_);
49    }
50    const_locking_ptr generateConstLockingPtr(void) const volatile {
51      return const_locking_ptr(*this,lock_);
52    }
53    // Convenience functions
54    locking_ptr operator->(void) volatile {
55      return generateLockingPtr();
56    }
57    const_locking_ptr operator->(void) const volatile {
58      return generateConstLockingPtr();
59    }
60  };
```

The main two member functions of this class template are `Serialized<>::generateLockingPtr()` and `Serialized<>::generateConstLockingPtr()`. If readers / writer locks are available, their difference will pay off.

This implementation contains two different copy constructors and four different assignment operators, which provide different guarantees to the instance assigned resp. copied and the instance assigned to. The way these copy constructors and assignment operators are implemented is the reason for the difference between the definition of `Serialized<>::[const_]locking_ptr` and the Figures 2.2a and 2.2b, where the Proxy grants access to the `ThreadUnsafeInterface` only instead of `Serialized< ThreadUnsafe Interface >`.

Implementing `Serialized<>` by means of the inheritance approach proposed has one disadvantage, however: As you can not inherit from fundamental types in C++, you can not serialize access e.g. to an `int` this way. This can be achieved by modifying the implementation using composition instead of inheritance, but then the relation between `ThreadUnsafeInterface` and `Serialized< ThreadUnsafeInterface >` is not represented as natural as it is with inheritance. A solution was to provide conversion operators then.

The usage is as follows:

Listing 2.4: Using `Serialized<>`

```
1  class Message_Queue {
2    typedef Serialized< std::queue< Message > > thread_safe_queue;
3    volatile thread_safe_queue impl_;
```

```
4   ...
5 public:
6   bool tryget(Message &msg) volatile {
7     thread_safe_queue::locking_ptr lptr(impl_.generateLockingPtr());
8     if(lptr->empty())
9       return false;
10    msg=lptr->front();
11    lptr->pop();
12    return true;
13  }
14  ...
15 };
```

The `Message_Queue` was taken from [SSRB02b] and already used as an example on Page 6. This scenario corresponds to Figure 2.2a.

The convenience member functions can be used to nearly hide the complexity of getting thread safe access if the client only wants to call a single member function of `ThreadUnsafeInterface`:

Listing 2.5: Using the conveneince member function

```
1 volatile Serialized< std::queue< Message > > q;
2 ...
3 q->clear();
```

This scenario corresponds to Figure 2.2b.

The convenience member functions depend on a special property of the user defined element access operator, `operator->()`, in C++ similarly to [Hen00, pp 12–14]: The compiler applies `operator->()` to the return value of a user defined `operator->()`. In this case both `Serialized<>::operator->()`s return by value and not by pointer. Therefore a call to either of them triggers a call to `{Read;Write}LockingPtr<> ::operator->()`, which in turn return a pointer to the (potentially `const`) non `volatile` instance of `Serialized<>`, which the built in `operator->()` finally applies to. Returning from an `operator->()` by value always results in a temporary. Therefore Change of Authority is an implementation technique well suited for this case to gain efficiency or even correctness.

Note that different compilers treat the temporary differently. Though the current standard limits the lifetime of the temporary until the next sequence point, some compilers still generate code which destroys the temporary not until the surrounding block was left. Subsequent calls to the convenience member functions proposed can result in deadlock then because with locks the N of Figure 1.1 is 1 if the locks are not recursive. If this is the case, introducing additional blocks will fix this issue.

2.7 Consequences

That is a notational simplicity close to methods declared `synchronized` in Java [AGH01, pp 262–266] and to methods attributed `synchronized` in C# /

.NET.

Alexandrescu's proposed use of the `volatile` cv–qualifier provoked some criticism, though. With the current C++ standard, which is agnostic about threads, the ideas layed out are a misuse of `volatile`. For a discussion see e.g. [MA04]. The assistance by the compiler gained by qualifying member functions `const` or `volatile` remains unquestioned, however. Therefore there is a proposal to adapt the C++ standard such that it is aware of multithreading [ABH+04].

Acknowledgements

Without the invaluable feedback of Berna 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 2005 for their contributions. Special thanks go to the members of the Writers' Workshop [Gab02] the author participated in: Paul Adamczyk, Tanya Crenshaw, Arvind Krishna, Jeffrey Overbey, Douglas Schmidt, John Sinnott, Tami Sorgente, and Joel Jones, its moderator. Their feedback had a significant impact on the current version of the paper.

Bibliography

[ABH+04] Alexandrescu, Andrei, Hans Boehm, Kevlin Henney, Doug Lea, and Bill Pugh: *Memory model for multithreaded C++*. Technical Report WG21/N1680 = J16/04-0120, ISO IEC JTC1 / SC22 / WG21—The C++ Standards Committee, Herb Sutter · Microsoft Corp. · 1 Microsoft Way · Redmond WA USA 98052-6399, September 2004.

[Abr00] Abrahams, David: *Exception safety in generic components*. In Jazayeri, Mehdi, Rüdiger G. K. Loos, and David R. Musser (editors): *Generic Programming. International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27–May 1, 1998, Selected Papers*, number 1766 in *Lecture Notes in Computer Science*. Springer, October 2000. <http://www.boost.org/more/generic_exception_safety.html>.

[ACD] Adler, Darin, Greg Colvin, and Beman Dawes: *Boost smart pointers*. <http://www.boost.org/libs/smart_ptr/smart_ptr.htm>.

[AGH01] Arnold, Ken, James Gosling und David 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. German translation of "The Java Programming Language. Third Edition".

[Ale01] Alexandrescu, Andrei: *Generic<programming>: volatile—multithreaded programmer's best friend. volatile–correctness or how to have your compiler detect race conditions for you*. The C/C++ Users Journal. Advanced Solutions for C/C++ Programmers, February 2001. <http://www.cuj.com/documents/s=7998/cujcexp1902alexandr/alexandr.htm>.

[Ale03] Alexandrescu, Andrei: *Generic<programming>: Move constructors*. The C/C++ Users Journal. Advanced Solutions for

C/C++ Programmers, February 2003. <http://www.cuj.com/documents/s=8246/cujcexp2102alexandr/alexandr.htm>.

[All02a] Allen, Eric: *Bug Patterns in Java*. apress, Berkeley, California, 2002.

[All02b] Allen, Eric: *The Orphaned Thread*, chapter 18, pages 151–159. In [All02a], 2002. <http://www-106.ibm.com/developerworks/java/library/j-diag0830.html>.

[Aue95] Auer, Ken: *Reusability through self–encapsulation*. In Coplien, James O. and Douglas C. Schmidt [CS95], chapter 27, pages 505–516.

[BM00] Bulka, Dov and David Mayhew: *Efficient C++. Performance Programming Techniques*. Addison–Wesley. An imprint of Addison Wesley Longman, Inc., Reading, Massachusetts · Menlo Park, California · New York · Don Mills, Ontario · Harlow, England · Amsterdam · Bonn · Sydney · Singapore · Tokyo · Madrid · San Juan · Paris · Seoul · Milan · Mexico City · Taipei, 2000.

[BMR+00a] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael 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 Auflage, 2000. German translation of "Pattern–Oriented Software Architecture. A System of Patterns".

[BMR+00b] Buschmann, Frank, Regine Meunier, Hans Rohnert, Peter Sommerlad und Michael Stal: *Proxy*, Kapitel 3: Entwurfsmuster, Seiten 263–275. In: [BMR+00a], 1998, 1. korr. Nachdruck Auflage, 2000. German translation of "Proxy".

[Car96] Cargill, Tom: *Localized ownership: Managing dynamic objects in C++*. In Vlissides, John M. et al. [VCK96], chapter 1, pages 5–18.

[CE00] Czarnecki, Krzysztof and Ulrich W. Eisenecker: *Generative Programming. Methods, Tools, and Applications*. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, first printing, may 2000 edition, 2000.

[Cop92] Coplien, James O.: *Advanced C++ Programming Styles and Idioms*. Addison–Wesley Publishing Company, Reading, Massachusetts · Menlo Park, California · New York · Don Mills, On-

tario · Workingham, England · Amsterdam · Bonn · Sydney · Singapore · Tokyo · Madrid · San Juan · Milan · Paris, reprinted with corrections december 1994 edition, 1992.

[Cop00] Coplien, James: *C++ idioms patterns*. In Harrison, Neil et al. [HFR00], chapter 10, pages 167–197.

[CS95] Coplien, James O. and Douglas C. Schmidt (editors): *Pattern Languages of Program Design*, volume 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.

[CSH98] Cleeland, Chris, Douglas C. Schmidt, and Tim Harrison: *External polymorphism*. In Martin, Robert C. et al. [MRB98], chapter 1, pages 377–390.

[Fow99] Fowler, Martin: *Refactoring. Improving the Design of Existing Code, with contributions by* Kent Beck, John Brant, William Opdyke, *and* Don Roberts. The Addison–Wesley Object Technology Series; Grady Booch, Ivar Jacobson, and James Rumbaugh, Series Editors. 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, 1999.

[FS00] Fowler, Martin und Kendall Scott: *UML konzentriert. Eine strukturierte Einführung in die Standard-Objektmodellierungssprache, deutsche Übersetzung von* Arnulf Mester, Michael Szczittnick *und* Günter Graw. Professionelle Softwareentwicklung. Addison–Wesley. An Imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, 2., aktualisierte Auflage, 2000. German translation of "UML Distilled".

[FY98] Foote, Brian and Joseph Yoder: *The selfish class*. In Martin, Robert C. et al. [MRB98], chapter 25, pages 451–470.

[Gab02] Gabriel, Richard 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.

[GHJV96a] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Abstrakte Fabrik*, Kapitel 3: Erzeugungsmuster, Seiten

107–118. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of “Abstract Factory”.

[GHJV96b] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Adapter*, Kapitel 4: Strukturmuster, Seiten 171–185. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of “Adapter”.

[GHJV96c] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Befehl*, Kapitel 5: Verhaltensmuster, Seiten 273–286. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of “Command”.

[GHJV96d] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Besucher*, Kapitel 5: Verhaltensmuster, Seiten 301–318. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of “Visitor”.

[GHJV96e] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Brücke*, Kapitel 4: Strukturmuster, Seiten 186–198. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of “Bridge”.

[GHJV96f] Gamma, Erich, Richard Helm, Ralph Johnson und John 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 Auflage, 1996. German translation of “Design Patterns. Elements of Reusable Object–Oriented Software”.

[GHJV96g] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Fabrikmethode*, Kapitel 3: Erzeugungsmuster, Seiten 131–143. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of “Factory Method”.

[GHJV96h] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Iterator*, Kapitel 5: Verhaltensmuster, Seiten 335–353. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of “Iterator”.

[GHJV96i] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Kompositum*, Kapitel 4: Strukturmuster, Seiten 239–253. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Composite".

[GHJV96j] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Memento*, Kapitel 5: Verhaltensmuster, Seiten 354–365. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Memento".

[GHJV96k] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Prototyp*, Kapitel 3: Erzeugungsmuster, Seiten 144–156. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Prototype".

[GHJV96l] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Proxy*, Kapitel 4: Strukturmuster, Seiten 254–267. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Proxy".

[GHJV96m] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Strategie*, Kapitel 5: Verhaltensmuster, Seiten 373–384. In: *Professionelle Softwareentwicklung* [GHJV96f], dritter, unveränderter Nachdruck Auflage, 1996. German translation of "Strategy".

[HDA02] Hinnant, Howard E., Peter Dimov, and Dave Abrahams: *A proposal to add move semantics support to the C++ language*. `<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1377.htm>`, September 2002.

[Hen00] Henney, Kevlin: *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>`.

[Hen02] Henney, Kevlin: *From mechanism to method: The safe stacking of cats*. The C/C++ Users Journal. Advanced Solutions for C/C++ Programmers, February 2002. `<http://www.cuj.com/documents/s=7986/cujcexp2002henney/>`.

[HFR00] Harrison, Neil, Brian Foote, and Hans Rohnert (editors): *Pattern Languages of Program Design*, volume 4 of *The Software*

Patterns Series; ed. by JOHN 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.

[Hir97] HIRSCHFELD, ROBERT: *Convenience method*. In *Preliminary Conference Proceedings of EuroPLoP '96* [wuc97].

[Kem] KEMPF, WILLIAM E.: *Boost.threads*. `<http://www.boost.org/-doc/html/threads.html>`.

[KLM+97] KICZALES, GREGOR, JOHN LAMPING, ANURAG MENDHEKAR, CHRIS MAEDA, CRISTINA VIDEIRA LOPES, JEAN-MARC LOINGTIER, and JOHN IRWIN: *Aspect–oriented programming*. In *Proceedings of the European Conference on Object–Oriented Programming (ECOOP), Finland*, number 1241 in *Lecture Notes in Computer Science*. Springer, June 1997.

[Lak96] LAKOS, JOHN: *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 edition, 1996.

[MA04] MEYERS, SCOTT and ANDREI ALEXANDRESCU: *C++ and the perils of double–checked locking*. `<http://moderncppdesign.com/-publications/DDJ_Jul_Aug_2004_revised.pdf>`, September 2004.

[Mey97] MEYER, BERTRAND: *Object–Oriented Software Construction*. Prentice Hall, Upper Saddle River, New Jersey, second edition, 1997.

[Mey98] MEYERS, SCOTT: *Effektiv C++ programmieren. 50 Wege zur Verbesserung Ihrer Programme und Entwürfe, Deutsche Übersetzung von* MICHAEL TAMM. Professionelle Programmierung. Addison–Wesley. An Imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, 3., aktualisierte Auflage, 1998. German translation of "Effective C++. 50 Specific Ways to Improve Your Programs and Designs".

[Mey99] MEYERS, SCOTT: *Mehr Effektiv C++ programmieren. 35 neue Wege zur Verbesserung Ihrer Programme und Entwürfe, Deutsche Übersetzung von* MARKUS BERINGMEIER. Professionelle Programmierung. Addison–Wesley. An Imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills,

Ontario · Sydney · Mexico City · Madrid · Amsterdam, 1999. German translation of "More Effective C++. 35 New Ways to Improve Your Programs and Designs".

[Mey01] Meyers, Scott: *Effektive STL. 50 Specific Ways to Improve Your Use of the Standard Template Library.* 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, July 2001.

[Mic05] Microsoft Developer Network (MSDN): *.NET framework general reference: Common design patterns. implementing finalize and dispose to clean up unmanaged resources.* `<http://msdn.microsoft.com/library/en-us/cpgenref/-html/cpconfinalizedispose.asp>`, 2005.

[MRB98] Martin, Robert C., Dirk Riehle, and Frank Buschmann (editors): *Pattern Languages of Program Design*, volume 3 of *The Software Patterns Series; ed. by* John 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, 1998.

[NW00] Noble, James and Charles Weir: *High–level and process patterns from the memory preservation society: Patterns for managing limited memory.* In Harrison, Neil et al. [HFR00], chapter 12, pages 221–238.

[rel] *Releasing resources in Java.* `<http://www.c2.com/cgi/wiki?-ReleasingResourcesInJava>`.

[sin] *Single function exit point–antipattern.* `<http://www.c2.com/-cgi/wiki?SingleFunctionExitPoint>`.

[SSRB02a] Schmidt, Douglas, Michael Stal, Hans Rohnert und Frank Buschmann: *Active Object*, Kapitel 5: Nebenläufigkeit, Seiten 411–443. In: [SSRB02c], 2002. German translation of "Active Object".

[SSRB02b] Schmidt, Douglas, Michael Stal, Hans Rohnert und Frank Buschmann: *Monitor Object*, Kapitel 5: Nebenläufigkeit, Seiten 445–471. In: [SSRB02c], 2002. German translation of "Monitor Object".

[SSRB02c] Schmidt, Douglas, Michael Stal, Hans Rohnert und Frank Buschmann: *Pattern–orientierte Software-Architektur. Muster für nebenläufige und vernetzte Objekte, übersetzt aus dem*

Amerikanischen von Martina Buschmann. dpunkt.verlag, Heidelberg, 2002. German translation of "Pattern–Oriented Software Architecture. Volume 2: Patterns for Concurrent and Networked Objects".

[SSRB02d] Schmidt, Douglas, Michael Stal, Hans Rohnert und Frank Buschmann: *Scoped Locking*, Kapitel 4: Synchronisation, Seiten 359–367. In: [SSRB02c], 2002. German translation of "Scoped Locking".

[SSRB02e] Schmidt, Douglas, Michael Stal, Hans Rohnert und Frank Buschmann: *Thread–Safe Interface*, Kapitel 4: Synchronisation, Seiten 383–391. In: [SSRB02c], 2002. German translation of "Thread Safe Interface".

[SSRB02f] Schmidt, Douglas, Michael Stal, Hans Rohnert und Frank Buschmann: *Wrapper–Facade*, Kapitel 2: Dienstzugriff und Konfiguration, Seiten 53–84. In: [SSRB02c], 2002. German translation of "Wrapper Facade".

[Ste99] Stevens, W. Richard: *UNIX Network Programming*, volume 2. Interprocess Communications. Prentice Hall PTR, Upper Saddle River, NJ, second edition, 1999.

[Str94] Stroustrup, Bjarne: *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. German translation of "The Design and Evolution of C++".

[Str98] Stroustrup, Bjarne: *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 Auflage, 1998. German translation of "The C++ Programming Language, Third Edition".

[Str02] Stroustrup, Bjarne: *C++ programming styles and libraries.* `<http://www.research.att.com/~bs/-style_and_libraries.pdf>`, January 2002. InformIt.com.

[Sut00a] Sutter, Herb: *Absolute Garantie*, Kapitel 3, Seite 48. In: *Professionelle Softwareentwicklung* [Sut00d], 2000. `<http://www.gotw.ca/gotw/061.htm>`. German translation of "Nothrow Guarantee".

[Sut00b] Sutter, Herb: *Compiler–Firewalls und das Pimpl–Idiom*, Kapitel 5, Seiten 119–141. In: *Professionelle Softwareentwicklung* [Sut00d], 2000. <http://www.gotw.ca/gotw/{007,015,024,-025,028,059}.htm>. German translation of "Compiler Firewalls and the Pimpl Idiom".

[Sut00c] Sutter, Herb: *Exception–Sicherheit*, Kapitel 3, Seiten 33–82. In: *Professionelle Softwareentwicklung* [Sut00d], 2000. <http://www.gotw.ca/gotw/008.htm>. German translation of "Exception Safety".

[Sut00d] Sutter, Herb: *Exceptional C++. 47 technische Denkaufgaben, Programmierprobleme und ihre Lösungen, deutsche Übersetzung von* Mathias Born *und* Michael Tamm. Professionelle Softwareentwicklung. Addison–Wesley. An imprint of Pearson Education, München · Boston · San Francisco · Harlow, England · Don Mills, Ontario · Sydney · Mexico City · Madrid · Amsterdam, 2000. German translation of "Exceptional C++. 47 Engineering Puzzles, Programming Problems, and Solutions".

[Sut00e] Sutter, Herb: *Grundlegende Garantie*, Kapitel 3, Seite 47. In: *Professionelle Softwareentwicklung* [Sut00d], 2000. <http://www.gotw.ca/gotw/061.htm>. German translation of "Basic Guarantee".

[Sut00f] Sutter, Herb: *Hohe Garantie*, Kapitel 3, Seiten 47–48. In: *Professionelle Softwareentwicklung* [Sut00d], 2000. <http://www.gotw.ca/gotw/061.htm>. German translation of "Strong Guarantee".

[Sut00g] Sutter, Herb: *Temporäre Objekte*, Kapitel 2: Generische Programmierung und die Standard–C++–Bibliothek, Seiten 23–29. In: *Professionelle Softwareentwicklung* [Sut00d], 2000. <http://www.gotw.ca/gotw/002.htm>. German translation of "Temporary Objects".

[VCK96] Vlissides, John M., James O. Coplien, and Norman L. Kerth (editors): *Pattern Languages of Program Design*, volume 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.

[Ven04] Venners, Bill: *Elegance and other design ideals. a conversation with Bjarne Stroustrup, part IV*. <http://www.artima.com/-intv/elegance.html>, February 2004.

[VJ03a] VANDEVOORDE, DAVID and NICOLAI M. JOSUTTIS: *C++ Templates. The Complete Guide*. Addison–Wesley, Boston · San Francisco · New York · Toronto · Montreal · London · Munich · Paris · Madrid · Capetown · Sydney · Tokyo · Singapore · Mexico City, June 2003.

[VJ03b] VANDEVOORDE, DAVID and NICOLAI M. JOSUTTIS: *Expression Templates*, chapter 18, pages 321–343. In [VJ03a], June 2003.

[Woo98] WOOLF, BOBBY: *Null object*. In MARTIN, ROBERT C. et al. [MRB98], chapter 1, pages 5–18.

[Woo00] WOOLF, BOBBY: *Abstract class*. In HARRISON, NEIL et al. [HFR00], chapter 1, pages 5–14.

[wuc97] *Preliminary conference proceedings of EuroPLoP '96*. Technical Report wucs-97-07, Washington University, Department of Computer Science, 1997.