The following paper was originally published in the
Proceedings of the USENIX 1996
Conference on Object-Oriented Technologies
Toronto, Ontario, Canada, June 1996.

Pattern Languages for Handing C++ Resources in an Exception-Safe Way

Harald Mueller
SIEMENS

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org

Pattern Languages for Handling C++ Resources in an Exception-Safe Way

Harald M. Müller

Siemens AG Austria

harald.mueller@siemens.at

Abstract

Using exception handling in C++ can lead to severe problems with dynamic objects and other resources—dangling pointers, memory leaks etc. By using a small set of patterns (collected into a pattern language "Exception-safe C++ objects"), these problems can be avoided. However, it turns out that these pattern make use of "resource management", which poses a lot of questions in itself. For the solution of these problems, a second, somewhat larger set of patterns (collected into a pattern language "Responsibility management under exception handling") is provided.

1. Introduction

Using exception handling in C++ can lead to severe problems with dynamic objects and other resource—dangling pointers, memory leaks etc. A number of articles have dealt with these problems (e.g. [1], [2], [3], [7]), and some solution patterns have been provided. Here, we collect all these solutions and organize them into a pattern-like way for easy application (and possible criticism).

The article is organized as follows: Section 2 provides three patterns for solving fundamental problems of exception handling. The last of these patterns requires correct resource management for all resources in a program, which is a non-trivial problem in itself. Therefore, section 3 goes on to provide patterns for a generalization of resource management, called responsibility management. Before dealing with "good patterns", a bad one is also shown as an example of how *not* to do it. After that, various patterns for managing responsibilities are provided, depending on the services provided by the environment (e.g., is a garbage collector available or not) and on the complexity of how an object might be managed by other objects (e.g. is there always a single manager for each object?). One important case that requires three patterns of its own occurs if the responsibility for a resource is to be passed around among several managers, yet, there must always be exactly one responsible at any time. The patterns dealing with this case are described in section 3.4.2.

The patterns in this article are presented in the Alexandrian form described in [8]; at a few places, a "motivation" section or an "alternative name" section is included additionally.

2. Exception-safe C++ objects

2.1. Problem space

Exception handling changes the nature of algorithms considerably: When an exception is thrown deep in the call stack, it is possible that many functions are suddenly interrupted in the middle of a computation (indeed, this is the rationale for exception handling). This means that those functions will not be able to fulfill the duty they were designed for, which could mean that data structures are no longer in a usable state. A simple example of such an event is the following: Consider a class for storing Gregorian dates:

```
struct GregorianDate {
   string month;
   unsigned day;
   unsigned year;
};
```

Assume that an object of this type stores the date June 30th, 1996. The following sequence of operations tries to change this to July 31st, 1996:

```
day = 31;
month = "July";
```

If an exception is thrown in the assignment to `month` (e.g. due to lack of memory) and `month` keeps its original value "June", we suddenly have stored a bad date—"June 31st, 1996". If the date values are used to index a table later on, it might well be that the program crashes*.

The problem is most serious with objects that cannot even be deleted after an exception interrupted an operation on them (see the BoundedStack example in section 2.4.1).

*If `month` does not keep its original value, but contains garbage afterwards, the situation is certainly only worse.

The following text presents some patterns that explain how to deal with this sort of problems. The fundamental problem discussed here was presented in [2], the solutions stem from [7].

2.2. Terminology

Resource

A resource is a single object or a tightly coupled group of objects that live longer than a single operation (this means that temporary objects are usually not viewed as resources). A resource usually must be created (or "acquired") and released (or "destroyed") by distinct operations.

An example of a resource consisting of a tightly coupled group of objects is the `argv` and `argc` parameters to the `main()` function.

Remark: Although resources are the basic sort of thing dealt with in this article, we give only a vague definition of this term. The reason is that it is not the case that something in a program is or is not resource, but rather that many things can be *viewed* as resources.

For resources that consist of a single C++ object, we will in the following equate resource release and object destruction. However, there might be resources around that provide different operations for resource release (a simple example are C files, which are released by a call to `fclose()`).

Resource invariant

For most resources, not all combinations of states of sub-resources make correct states of the compound resource. For example, the `GregorianDate` struct defined above might have the value

```
day == 42
month == ""
year == -1
```

However, this is not a valid Gregorian date. The same is true for the state

```
day == 29
month == "February"
year == 1995
```

Obviously, there is a condition restricting the values of the sub-resources so that only those values that "make sense" in the application are possible. We call this condition the "resource invariant".

For `GregorianDate`, this is a condition that connects the value of `day` to the name of the `month` and the value of `year` (because of leap years). Moreover, the date must be larger than October 15th, 1582, the day when Pope Gregor introduced the new calendar in (most of) Italy.

At many places in this article, we will use a general Stack class template to describe problems and solutions to them (the declaration of this template is given in the appendix). A few important parts of the invariant of this class are:

- `nelems` is not larger than the size of the array pointed to by `vec`
- `top <= nelems`
- `top >= 0`

The "State Scale"

As mentioned in the introduction, the big problem with exception handling is that the interruption of a computation may leave some resources in a state not acceptable for following operations. However, the "badness" of the state may be of a varying degree—e.g. it might be so bad that the resource cannot even be released; or it could merely have retained its previous state, which means that it is no longer correct, but certainly still usable. The following defines a partioning of the possible states into different categories; the main discrimination is based on the resource invariant introduced above.

After an operation finishes, the state of a resource accessed and possibly modified by this operation may be in one of the following state classes:

- Consistent state – any state fulfilling the resource invariant.
    - Correct state – as it is defined by the specification of the operation.
    - Incorrect state.
        - Same state – the same state as before the operation (if this is not the correct state).
        - Any consistent state – any other consistent state (e.g. a state like after resource creation).
- Inconsistent state – a state contradicting the class invariant.
    - Shut-down state – an inconsistent state where at least resource release is possible.
    - Very bad state – a state where potentially no operation (not even a resource release) is possible.

Resources in an inconsistent state are also called "damaged resources" or "bad resources" in the following; resources in a consistent state are also called "good resources" or "resources in a good state".

2.4. Patterns

2.4.1. "Ensure Shut-down-ability"

- **Problem**

An exception might interrupt a function modifying some resource at such a moment that the resource is no longer in a consistent state afterwards. In general, the inconsistent state might be a very bad state, which means that the resource cannot even be destroyed afterwards.

- **Forces**

1. It is necessary that no resource enters a very bad state, as every resource must eventually be released.

2. Indeed, it is preferable that all resources are as high as possible on the state scale.

3. On the other hand, it is very complicated to write all modification functions in such a way that they leave all resources only in consistent states (see e.g. the GregorianDate example in the introduction).

4. At least *some* operations must be possible on resources even if an exception was thrown during a modification.

- **Solution**

Design all resources and all modification functions on them in such a way that the resources never enter a very bad state.

Heuristic A: Do not have a value necessary for destruction depend on two or more variables ("do not do computations in destructors").

Heuristic B: Do not store values other than those a destructor expects in an object's attributes (e.g. do not use class members as "temporaries").

- **Examples**

The `GregorianDate` struct is an example of a resource that is designed according to this principle: The (trivial) destructor will work regardless of the values that have been assigned to the member variables.

An example of a class violating this rule is the `BoundedStack` class declared the appendix—a stack class with a maximum number of elements. Its member variable `current_top` always points one element above the top element, and `current_size` is the number of pushed values. The element counter `current_top` is implemented by a class `Bounded-Int` that can only represent a bounded interval of integers. Operations trying to assign a value outside this interval throw an exception.

The destructor of a `BoundedStack` object simply computes the base of the allocated array and then deletes it:

```
template <class T>
BoundedStack<T>::~BoundedStack() throw ()
{
    delete [] (current_top - current_size);
}
```

—however, this contradicts Heuristic A given above. The problems arising from this design are explained in the paragraph on "Design rationales".

- **Force resolution**

This pattern steers an intermediate way between items 1. and 2. in the forces' list on the one hand, and item 3. on the other.

It might appear that this solution is somewhat weak—one would expect that using exception handling would yield more powerful error handling policies that would also allow to repair damaged resources somehow. However, in most real applications throwing an exception means that all resources modified by a function interrupted by the exception can no longer be used. If the resource stores its information non-redundantly—which is the case for the vast majority of data structures—, it is not possible to reconstruct the correct result, anyway. So shutting down the resource and others that depend on it is typically the only possible solution.

- **Design rationale**

It is important to notice that by this pattern, we explicitly allow resources to be in bad states. One might wonder whether this is really necessary—after all, isn't it possible to keep resources in some good state all the time, e.g. by always returning to the (necessarily good) state from before the exception? Alas, it turns out that this is equivalent to a complete transaction system with full rollback capabilities—certainly not the right approach for the small resources that are customarily needed in a C++ program. For more information, see [7].

Heuristic A: For two or more variables, any modification must occur as a sequence of modification steps. If an exception is thrown by one of these steps, the compound resource consisting of all the variables might no longer be in a consistent state, hence a value computed from some of the variables might be garbage. An example for this is the following implementation of `BoundedStack::push()`:

```
template <class T>
void BoundedStack<T>::push(const T& e)
// throw (overflow_error, ... T&=const T& )*
{
   ++current_size;
   *current_top++ = e;
}
```

If the increment of `current_size` throws an exception (which is possible if we try to push more elements than the stack can hold), we end up with an incremented `current_size` value, but `current_top` has not changed, and hence the destructor will call `delete` with an invalid argument.

Heuristic B simply addresses a common misuse of member variables.

- **Consequence**

A consequence (or rather *the* consequence) of this pattern is that resources might be in bad states, i.e., states where no operation except release is possible. This means that other parts of the system that might access the resource in the future must get notice of the state of affairs so that they can avoid these accesses. That this should be done explicitly is codified in the next pattern.

2.4.2. “Hide damaged resources till destruction”

- **Problem**

A modification of a resource might fail. Use of “Ensure Shut-down-ability” means that the resource cannot be used afterwards, but only destroyed. In general, every such resource is part of some other, larger resource†. The important question now is: When, and how, is it possible to continue using the larger resource?

- **Forces**

1. Keeping all sub-resources of the larger resource always usable is not feasible (see previous pattern).

2. Throwing away each resource with a single damaged sub-resource might result in shut-down of functionalities that could still be provided.

3. It is very dangerous to rely on the promise of other program parts that the damaged sub-resource will not be accessed until destruction of the whole resource.

- **Solution**

a. If the sub-resource’s value is part of the resource’s invariant, the complete resource must be viewed as damaged and therefore abandoned (if the sub-resource’s value is not consistent in itself, the condition of the larger resource cannot possibly be true).

b. Also if the exception thrown might have left values from the invariant in an inconsistent state (although each single value is still usable), the complete resource must be viewed as damaged and abandoned.

c. If the sub-resource’s value is not part of the resource’s invariant, explicitly hide the damaged sub-resources from all further accesses *except* those in the destructor of the resource.

- **Examples**

Case a. Assume that we repair the `BoundedStack` class shown in section 2.4.1, e.g. by providing a separate data member pointing to the base of the allocated array. Now look at the `push()` operation: If an exception is thrown by the increment of `current_top`, we must assume that this sub-resource is damaged. Hence, we must abandon the complete `Bounded-Stack` object.

Case b. The `push` operation for the stack class must enlarge the internal array if it is full. This can be accomplished e.g. as follows:

```
template <class T>
void Stack<T>::push(const T& e)
// throw (bad_alloc, ...T(), ...T&=const T&)
{
   if (top == nelems) {
      nelems *= 2;
      T *new_buf = new T[nelems];
      for (unsigned i=0; i<top; i++)
         new_buf[i] = vec[i];
      // ... delete vec
      // ... replace it by new_buf
   }
   // ...
}
```

If one of the assignments in the for loop throws an exception, `nelems` will already have been doubled, but the memory pointed to by `vec` is still the old one.

---

*For an explanation of this notation, see footnote in the appendix, section A.2.

†In the last consequence, we can even view all the resources in a program as a single program-wide “program resource.”

Hence the part of the invariant saying that nelems is not larger than the size of the array is no longer true, and the complete stack object has to be abandoned.

Case c. At its end, the push operation has to assign a value to an element of the stack's internal array, e.g.

```
template <class T>
void Stack<T>::push(const T& e)
// throw (bad_alloc, ...T(), ...T&=const T&)
{
    // ...
    vec[top++] = e;
}
```

If the assignment fails, the stack is no longer usable afterwards: The top element may be in a bad state, hence no operation except destruction is possible. However, the value of the top element certainly is not relevant for the functioning of the stack, hence it will not appear in the invariant of a stack resource. Thus, it should be possible to continue the use of the Stack object.

To this end, we must hide the damaged element from further use. A simple remedy is to reduce nelems to top so that all methods (except the destructor!) now believe that there are fewer elements allocated (watch out for the increment of top; it must now be postponed until it is clear that the assignment did work):

```
void Stack<T>::push(const T& e)
// throw (bad_alloc, ...T(), ...T&=const T&)
{
    // ...
    try {
        vec[top] = e;
        top++;
    } catch (...) {
        nelems = top;
        throw;
    }
}
```

This course of action might be worthwhile for example when the stack is used for recalling commands that a user typed at some earlier time. For such an application, it might make more sense to lose some input than to shut down the complete user interface*.

- **Force resolution**

The damaged sub-resources are no longer available for any functionality the larger resource provides. However, all functionalities not using the damaged parts are still available.

- **Consequences**

It is a necessity to design explicitly the various "restricted functionalities" of a resource, i.e., which operations are still available after various sub-resources went bad.

2.4.3. "Resource management is necessary"

- **Motivation**

The previous pattern showed what to do if part of a resource entered an inconsistent state. However, certain resources must never even go into an *incorrect* state, as we cannot afford to live without them. For these resources, the sequence of operations manipulating them must *always be correct*.

The most outstanding example of such a resource is the dynamic heap. Here, we require for example that for each successful new operation, a corresponding delete must be executed somewhen later. As exceptions may interrupt statement sequences quite forcefully, it is not immediately clear how this requirement might be fulfilled. If one operates the heap naively in the presence of exception handling, throwing an exception might easily interrupt sequences manipulating the heap, thereby leading to memory leaks and/or double deletes (see [2]).

The pattern "Hide damaged resources till destruction," which promises to deal with damaged parts of a resource, does not work for the global heap: It would require us to "hide" "damaged parts" of the heap (e.g. where a double delete or a memory leak has occurred) from further access. However, this is definitely not possible in C++ (and would probably not make much sense anyway, as the available space in the correctly maintained part of the heap would get smaller and smaller until nothing was left).

The Stack::push() method explained in the previous pattern already showed how code that is correct without exception handling will be incorrect when exceptions might get thrown. It contains a section for replacing the internal buffer by a larger one:

```
if (top == nelems) {
    nelems *= 2;
    T *new_buf = new T[nelems];      (a)
    for (unsigned i=0; i<top; i++)
        new_buf[i] = vec[i];         (b)
    // ... delete vec
```

*But of course, the user should be told about this event in the exception handler (the catch clause).

```
        // ... replace it by new_buf
}
```

As we do not know anything about the parameter type `T`, we must assume that its assignment operation might throw an exception. If this happens in the assignment to `new_buf[i]` at (b), the memory allocated at (a) will leak. This simple example shows that we must put some more effort into managing resources like memory allocated on the dynamic heap.

- **Problem**

In the presence of exceptions, each sequence of function calls can potentially be interrupted at any place, which seems to make guaranteed sequences of function calls impossible. However, for some resources (like the global heap), the sequence of operations applied to them—including the parameters supplied—must always be correct. How can this be accomplished?

- **Forces**

1. Correctness is essential, i.e. the invariants for such resources must be true all the time.

2. Complexity of code should be kept to a minimum.

- **Solution**

Have a strategy of explicitly assigning and transferring *responsibilities* that also works if exceptions are thrown. A *responsibility* here means a duty to execute a certain operation at some time in the future.

- **Examples**

For the most important operation to be executed "at some time in the future," namely resource release, some of the possible strategies include:

- Delegate release responsibilities to a central system (usually called garbage collector). This strategy is usually not adopted in C++ programs.
- Arrange for possibility that memory may be doubly deleted without doing any harm. Also this strategy is usually not adopted in C++ programs.
- Pass responsibility for resource release (especially destruction of objects) around so that there is always exactly one *owner* of the object responsible for the release [6]. This is the standard strategy adopted in C++ programs.

- **Design rationale**

The idea behind this solution is that although exceptions can forcefully interrupt statement sequences (and thereby bypass necessary function calls), they cannot change data that are used to store responsibilities. Hence, explicit responsibility management is less vulnerable to possible exceptions.

It might seem that this "solution" is not more than a general "hint." However, the text tries to balance between the two forces that on the one hand, it should be possible to have different strategies how to enforce correct handling of resources, but on the other hand it should be concrete enough to give some guidelines for evaluating and/or designing such strategies.

In some sense, the solution stresses the fact that it is more important to *have* a strategy than *which* strategy is adopted (as long as it is a correct one).

- **Force resolution**

The forces mentioned in the "Forces" section are not immediately dealt with. The reason is that the various possible strategies have different trade-offs. The sections describing concrete strategies will give more information about these trade-offs.

- **Consequences**

It must be admitted that this pattern creates problems of its own: First, strategies fulfilling the conditions of the solution must be developed. Second, one has to decide *which* strategy should be used. Both problems are non-trivial. Because of this, we will view the *solution* given here as a *problem* in itself in the rest of the article, for whose solution we will again provide a set of patterns. These patterns will explain the standard strategies that can be used for responsibility management under exception handling and provide guidelines for selecting among them.

3. Responsibility management under exception handling

3.1. Problem description

- **Motivation**

See section 2 of this article.

- **Problem**

Have a strategy of explicitly assigning and transferring *responsibilities* that also works if exceptions are thrown. A *responsibility* here means a duty to execute a certain (secure*) operation at some time in the future.

---

*See the following section on terminology.

• **Forces**

1. Correctness is a non-trivial requirement for these patterns in itself, therefore it is mentioned here.

2. The code complexity necessary to implement the responsibility management logics should be as small as possible. This is mainly due to the requirement that the testability, maintainability, documentability, and understandability of the code should not get worse when exception handling is introduced—after all, this language feature is intended to make it *simpler* to deal with exceptional events in a program.

3. The patterns should allow to write programs as efficient as without special responsibility management.

4. In some cases, it is important that resource release or any other operation passed to a responsibility holder occurs at a specific time; in other cases, it is only necessary *that* the operation be executed. The trade-offs between the earlier requirement for determinism and the possibility of indeterminism are another force on a possible solution.

5. Sometimes resources have the same lifetime as their creator*, i.e., the responsibility for any operation on them can always reside with the creating block. However, "interesting" resources usually live much longer than their creating block is active—here some sort of responsibility transfer will be necessary.

3.2. Terminology

Secure operation – An operation that does not throw an exception. In C++, secure functions are marked by an empty exception specification `throw()`.

Responsibility – A duty to execute some operation in the future. In this article, we assume that these functions are secure operations. For the most important operation needing responsibility management, namely resource release (in many cases accomplished by a destructor), this is a realistic assumption.

Responsibility holder – An object that has a responsibility. Usually, the operation to be executed is placed in the destructor of the responsibility holder, as the destructor is (almost) certainly bound to be executed at some time (this is a small pattern of its own). [6] uses the term "owner" instead of responsibility holder.

Resource management class – A class explicitly designed for responsibility holders (in contrast to the use of e.g. simple pointers for holding responsibilities). One example of such a class (actually a class template) is the `auto_ptr` type defined in the C++ standard.

Creator of a resource – The first responsibility holder that initially creates a resource.

In this discussion, the creator of a resource is always some block, i.e., some sequence of statements.

In [6], also objects and classes are seen as creators if they get control over the resource "quick enough." In the presence of exception handling, this view could be dangerous. Consider a constructor for some class `X`:

```
X::X() throw (X_problem)
{
   member = new Resource;
   // ... some operations that possibly
   // ... throw an X_problem exception
   // ...
}
```

If one assumes that the allocated `Resource` object is under object responsibility, one would expect that no additional actions are necessary in the constructor to release `member` in the case of an exception. However, this is not true: In C++, the destructor is not executed if an exception is thrown in the constructor; hence, resources allocated in the constructor must also be released there.

Because of this, we assume here that the `Resource` object is placed under the responsibility of the enclosing block—whose responsibility it is to release the resource in case of an exception. In order to avoid a differentiation between constructors and other functions, we adopt the view that the creator is always the block enclosing the resource acquisition.

We will deal in the following not with the general problem of strategies for arbitrary operations, but restrict the patterns to the most prominent operation for which responsibility management is necessary, namely resource release ("resource shut-down," "resource destruction"). However, the following patterns can be used also for other operations.

Most of the following patterns could be part of Tom Cargill's "Localized Ownership" pattern language. However, they differ from the patterns given in [6] in that they explicitly provide exception-safe code patterns—which is the topic of this paper, after all.

*See also in the following section on terminology.

3.3. A Bad Pattern

- **Not a (good) solution**

The following is a possible pattern to deal with the problem of correct resource management: After each resource acquisition, a try block encloses the operations on the resource. If any of the operations fails, an unspecific catch block releases the resource and rethrows the exception so that further handlers get notice of the exception.

- **Example**

```
T *r1 = new T[...];
try {
   // ... do something with r1;
   T *r2 = new T;
   try {
      // ... do something with r1 and r2;
      // etc. for each resource allocated
   } catch (...) {
      delete r2;
      throw;
   }
} catch (...) {
   delete [] r1;
   throw;
}
```

- **Force resolution**

There is only one down-side to this pattern, namely that it has a very high code complexity: Each single resource acquisition leads to a separate nesting level and an additional catch block. However, this single problem makes the pattern virtually unusable for real programs, as the high code complexity entails many other software problems, like poor testability, maintainability, documentability, reuse etc.

For the frequent case of purely local resources, there is an additional complexity introduced, as the resource release has to be doubled:

```
T *local_r = new T[...];
try {
   // ... do something with local_r;

   delete [] local_r;
} catch (...) {
   delete [] local_r;
   throw;
}
```

3.4. Good Patterns

The following patterns for solving the responsibility management problem in the presence of exceptions are grouped into four sections:

- Patterns with a single responsibility holder - these contain “Garbage Collector” and the famous “Resource Acquisition is (Object) Initialization” pattern.
- “Sequence of exclusive responsibility holders” describes how a resource is passed from one responsibility holder to another.
- “Reference counting” and
- “Idempotent operations” give solutions where more than one responsibility holders exists for a single duty.

As all the patterns share the common problem and forces described in section 3.1 above, these are not repeated in the following sections.

3.4.1. Patterns with a single responsibility holder

3.4.1.1. “Garbage Collector”

- **Solution**

A dedicated subsystem is capable of finding and destroying all resources that may be destroyed.

- **Examples**

Concepts of garbage collection for C++ are presented in [11]. There are also a number of commercially available garbage collection systems for C++. Moreover, all languages providing garbage collection are examples of this strategy (however, this is not of much help for C++ users in the current state of affairs).

- **Force resolution**

There are four potential drawbacks to this solution:

1. C++ does not provide a garbage collector in standard implementations, therefore such a system must be acquired or implemented. However, implementing a garbage collector is a large problem in itself.

2. Many garbage collection algorithms can fluctuate widely in their consumption of processing power. For systems with narrow timing conditions, this may be a problem (however, the same can also be true for general heap algorithms, where finding a space for an ob-

ject and/or releasing memory need not take a predictable time).

3. Garbage collection is in general indeterministic, i.e. it is not possible to predict when the resource destruction actually takes place. This can be a problem if the destruction accomplishes different tasks from freeing memory, e.g., closing a window or a file.

4. Garbage collection is confined to freeing allocated memory. Other operations (like closing files, releasing allocated time slots on a communications channel etc.) are usually not handled by a garbage collector.

However, there are two possibilities how to handle such cases with garbage collection:

- The "Garbage Collector" *pattern* can be extended to serve other purposes. One example is the automatic closing of files resp. streams at the end of a C or C++ program, which could be viewed as a collection of "file garbage".
- Another possibility is to map all operations to memory release by placing the operations in a destructor call. When the garbage collector releases the memory at some point in the future, the destructor is executed, and the duty is carried out. However, this needs some sort of garbage collector that knows about C++ objects and their destructors, i.e., it must support some kind of *finalization* (see [12]).

• **Remark**

One could also see the garbage collector as a part of the language and not as a separate, identifiable responsibility holder. Under this view, one would interpret this pattern as a complete abandonment of the resource invariant*.

3.4.1.2. "Resource Acquisition is (Object) Initialization"

• **Solution**

Use an automatic (i.e., stack-based) responsibility holder object for managing the resource in the following way:

(a+†) Allocate a resource by creating a resource management object on the stack.

(b+) Use the resource.

(c) The resource is released by the destructor of the resource management class.

For an extended explanation of this pattern, see e.g. [1].

• **Alternative name**

"Resource Release is Object Destruction"

The alternative name stresses the fact that the release operation is the more problematical one. In my opinion, the pattern should be called by this name to point out its potential usefulness more clearly.

• **Force resolution**

With this solution, the resource cannot be passed outside the creating block, hence this pattern solves only a sub-case of the general problem. However, the patterns presented in the following section can be viewed as a generalization of this pattern.

• **Consequences**

Separate resource management classes are needed for each type of resource that may be necessary. The forthcoming ANSI/ISO C++ standard library will contain a template `auto_ptr<T>` that can be used as a resource management class for a single heap object of class `T`. However, it is lacking a similar class for arrays allocated on the heap (this would only have to call `delete []` instead of `delete` in the destructor). For a possible implementation of such a class, see the `auto_ptr_array` class template in the appendix.

• **Remark**

Some other languages provide direct language means for writing statements that are guaranteed to be executed even if an exception is thrown. Examples include the `valueNowOrOnUnwindDo:` method in Smalltalk or the `finally` clause of Java.

3.4.2. "Sequence of exclusive responsibility holders"

• **Solution**

There is always exactly one responsibility holder for resource release. However, by passing around the responsibility among many responsibility holders, it is possible to keep the resource alive longer than the creating block is active.

---

*Maybe one should derive another pattern from this: "Keep invariants as few and as simple as possible."

†The + symbol means that this step may throw an exception. Steps without a + must be implemented as secure operations.

- **Force resolution**

This solution is a generalization of the "Resource Release is Object Destruction" pattern. It keeps its efficiency and determinism, however, the code complexity is higher than with all other patterns. Moreover, it produces a lot of problems of its own (these will be dealt with in following sections).

The main problem of this solution is that explicit *responsibility transfers* are necessary. This happens for instance if the creating block transfers its responsibility for some allocated memory to an object before it goes out of scope. We will use these responsibility transfers to ensure that all resources get released exactly once even in the presence of exceptions. However, this makes it necessary that the transfers themselves do *not* throw an exception, i.e., they must be *secure*—otherwise, we could no longer be sure where the responsibilities lie after a failed transfer. In almost all cases, this can be ascertained by moving around only some kind of reference value, not the resources themselves. Such values might be pointers, integral values or certain types of iterators that may be securely copied.

This solution will probably be the standard solution used in C++ programs, although in some respects it might be considered the most difficult one. It certainly places the highest burden among all solutions on the programmer.

3.4.2.1. "Responsibility transfer is swap"

- **Problem**

If a responsibility for a resource is passed from a responsibility holder A to another responsibility holder B, it might be the case that B already is responsible for some other resource. Simply discarding this responsibility is obviously not possible, as we postulated that there must always remain exactly one responsibility holder. What are we going to do with the "superfluous responsibility"?

- **Forces**

Besides the ubiquitous correctness requirement, there are two more forces on solutions for this problem:

1. It should not be required to declare and handle more responsibility holders than absolutely necessary; and

2. the code complexity should be small, as this pattern is only a stepping stone for other patterns.

- **Solution**

Swap the responsibilities between A and B.

- **Examples**

The following is an example where a resource manager called `perm` takes over the responsibility for an object of class `X` from a local resource manager. The `swap_with()` method of class `auto_ptr` (which is *not* defined in the proposed C++ standard) exchanges the internal pointers of the two objects. At first we show the wrong solution to this problem:

```
auto_ptr<X> perm;
// ...
{
    auto_ptr<X> temp;
    temp.reset(new X);
    // ...
    perm.reset(temp.release());
    // wrong: perm might already be
    // responsible for some resource!
}
```

The correct solution exchanges the responsibilities of `perm` and `temp`:

```
auto_ptr<X> perm;
// ...
{
    auto_ptr<X> temp;
    temp.reset(new X);
    // ...
    perm.swap_with(temp);
    // temp's destructor can now take
    // care of the resource for which
    // perm was responsible before, if
    // there was any.
}
```

- **Force resolution**

For once, the forces can be perfectly resolved: The pattern is obviously correct; it certainly needs the minimal number of responsibility holders; and the code complexity is really small.

- **Remark**

Swapping responsibilities does not require that one of the resources must be released afterwards. It is perfectly legal to swap two "active" resources that are both used after the swap.

The following two patterns describe two frequent applications of "Responsibility transfer is swap", namely resource replacement and acquisition of a new resource. They refine a group of “Local Ownership” patterns given in [6] by making them exception-safe.

3.4.2.2. “Resource replacement”

- **Problem**

Replace an old resource under object responsibility by a new one. This problem occurs in many basic operations, e.g. assignment (where the old member values must be replaced by new ones), in containers that automatically adjust their internal storage (e.g. a string class that enlarges its buffer if characters are added; or our running example, the Stack class template), and elsewhere.

- **Forces**

1. The resource replacement must be correct, i.e., for the old resource as well as for the new one there must always be exactly one resource manager.

2. The code complexity should be as small as possible.

- **Solution**

(a+) Allocate a local resource by creating a resource management object on the stack.

(b+) The local resource is used (usually initialized) as necessary.

(c) The responsibilities for the object’s resource and the local resource are swapped.

(d) The destructor of the resource management object releases the now local resource (the former object resource).

- **Examples**

The following two examples show the application of this pattern to the assignment and the push operation of the stack class template.

```
template <class T>
Stack<T>& Stack<T>::operator=(const Stack<T>& s)
// throw (bad_alloc, ...T())
{
    if (this != &s) {
        auto_ptr_array<T>
            new_buf(nelems=s.nelems);   (a+)

        for (unsigned i=0; i<s.top; i++)(b+)
            new_buf[i] = s.vec[i];

        new_buf.swap_with(vec);         (c)
        /* destructor of new_buf */     (d)
    }
    return *this;
}


template <class T>
void Stack<T>::push(const T& e)
// throw (bad_alloc, ...T(), ...T&=const T&)
{
    if (top == nelems) {
        nelems *= 2;
        auto_ptr_array<T> new_buf(nelems);(a+)
        for (unsigned i=0; i<top; i++)(b+)
            new_buf[i] = vec[i];
        new_buf.swap_with(vec);         (c)
        /* destructor of new_buf */     (d)
    }
    vec[top++] = e;
    // Instead, we could copy the code from
    // 2.4.2 if it should be possible to use
    // the stack object after an exception in
    // the assignment.
}
```

- **Force resolution**

By applying the “Resource transfer is swap” pattern, the pattern maintains the invariant that there is always exactly one responsibility holder for each resource release.

The complexity of the code is about the same as with the “Resource Acquisition is (Object) Initialization” pattern (in contrast to the try/catch-pattern of section 3.3).

- **Alternative name**

“Two-Step”

This name captures the fact that resource replacement (and also acquisition of a new resource—see the following pattern) always has two phases:

- a first “exception-prone” phase where the resources are acquired and used;
- a second phase of secure responsibility transfers and secure resource releases.

In many cases, operations on resources can be reordered to fit with this two-phase pattern.

However, under special circumstances, it might happen that an exception-prone operation must occur *after* the second phase. A prominent example of this is a `return` statement executing a copy constructor that might throw an exception. In such cases, somewhat more complicated patterns are necessary. For a detailed description of such a problem and its solution, see [7].

3.4.2.3. “Resource acquisition for an object is resource replacement”

• **Problem**

Allocating a new resource and putting it under object responsibility: This is the standard problem when writing a constructor.

• **Forces**

The forces are the same as for the resource replacement problem, i.e. correctness and minimal code complexity.

• **Solution**

(a+) The object’s responsibility holder is initialized to an arbitrary resource.

(b+) A local resource is allocated by creating a resource management object on the stack.

(c+) The local resource is used (usually initialized) as necessary.

(d) The resource is placed under the object’s responsibility.

(e) The destructor of the resource management object releases the arbitrary resource allocated in (a+)—this is in effect a no-op.

• **Example**

The following example shows the application of this pattern to the default constructor of the stack class template.

```
template <class T>
Stack<T>::Stack()
// throw (bad_alloc, ...T())
    : vec(0)                          (a+)
{
    auto_ptr_array<T> buf(nelems=10); (b+)
              // (c+ not necessary here)
    buf.swap_with(vec);               (d)
    /* destructor of buf */           (e)
}
```

• **Force resolution**

See “Resource Replacement” pattern.

• **Design rationale**

One could design a separate pattern for acquisition of new resources that does not need the “arbitrary resource” from step (a). However, the pattern above has the advantage that it inherits its correctness from the “Resource replacement” pattern.

Moreover, this pattern fits well with the C++ style rule that member variables should be set in the initializer list of a constructor: As shown above, the member gets the responsibility for a trivial resource in the initializer list, which is then replaced by the real resource in the constructor’s body.

The patterns described in sections 3.4.1. and 3.4.2. assume that there is only one responsibility holder. This has the advantage that it is always clear who will carry out the required operation, but it has the disadvantage that passing around responsibilities is either not possible (“Resource Release is Object Destruction”) or quite complex (“Sequence of exclusive responsibility holders”), or that the underlying system must be quite powerful (“Garbage Collector”).

The following patterns describe alternative approaches to the responsibility management problem by allowing more than one responsibility holder. From the viewpoint of exception handling, the fundamental problem of correct resource release thereby vanishes (almost): We simply give the responsibility to release a resource to a lot of responsibility holders; if the activities of one of these are interrupted by an exception, we can nevertheless be quite certain that some other responsibility holder will eventually carry out the release operation.

3.4.3. “Reference counting”

• **Solution**

Many responsibility holders take over the job to release a resource. The release happens when the last responsibility holder declares that it wants to do its duty.

For details on this well-known pattern, see e.g. [9] or [6]. It is also mentioned as a special case of the “Proxy” pattern in [10].

3.4.4. “Idempotent operations”

• **Solution**

Make resource release an idempotent operation, i.e. an operation that has no effect after the first execution. Thus, many responsibility holders may exist at one time, each of whom might carry out the release operation.

- **Examples**

An application has a central log file that is to be closed either if the user explicitly requires it, or when the application finishes. For this, a central log-file-handler could provide a function

```
Log_file_handler::close_log_file() {
    delete p_log_file;
    p_log_file = 0;
}
```

The duty to call this function is on the one hand given to a central application shut-down process. However, other processes—e.g. the afore-mentioned user command for explicitly closing the logfile—may equally call this function without giving notice to the shut-down process, as a repeated execution does no harm.

- **Force resolution**

In contrast to the reference counting scheme, with "Idempotent operations," the *first* responsibility holder that wants to carry out its duty will accomplish the resource release. It depends on the application whether this is possible or not.

Moreover, it must be still ascertained that at least one responsibility holder carries out its duty (as with reference counting).

The last problem can be overcome by putting the responsibility to a single central responsibility holder which swears to fulfilling its duties—see "Garbage Collector." We've come a full circle.

4. Conclusion

This article tried to cast a set of rules necessary for writing exception-safe code into patterns form. Widespread use of exception handling, especially in larger systems, is not yet common in the C++ community, therefore it is hoped that patterns of this kind help to deal with the problems inherent in this complex feature.

However, there are some open points that have to be addressed in the future:

At some places, implicit assumptions are not yet clearly spelled out (e.g. that it is assumed that responsibility management objects are not class members, but only local variables); at other places, explicit conditions could be weakened or left away (e.g. there seem to be some meta-patterns behind the patterns presented in 3.5.2).

Moreover, as always, the patterns certtainly need refinement by potential and actual users.

The patterns shown in section 3 are useful for resources that are designed according to "Ensure Shut-down-ability", i.e., it is not necessary that the resource itself is in some good state after an exception occurred. For resources that must have a quality as high as e.g. the global heap, i.e., where it must be possible to continue using them after an exception interrupted some operation, more powerful patterns are necessary that guard *all* modifications of all variables in a piece of code. An introduction to concepts for solving this problem can be found in [7]; shaping them into pattern form might be the contents of a future article.

5. References

[1] Stroustrup, B. The Design and Evolution of C++. Addison-Wesley, Reading, MA, 1990.

[2] Cargill, T. Exception handling: A false sense of security. C++ Report 6(9):21-24, Nov.-Dec. 1994.

[3] Carroll, M. and M. A. Ellis. Tradeoffs of exceptions. C++ Report 7(3):12-16, March-April 1995.

[4] Ellis, M. A. and B. Stroustrup. The Annotated C++ Reference Manual. Addison-Wesley, Reading, MA, 1990.

[5] Working Paper for the C++ standard, April 1995.

[6] Cargill, T. Localized Ownership: Managing Dynamic Objects in C++. Proceedings of the 2nd Pattern Languages of Programming Conference, September, 1995.

[7] Müller, H.M. 10 Rules for Handling Exception Handling Successfully, C++ Report 8(1):22-36, Jan. 1996.

[8] Coplien, J.O. Software Design Patterns: Common Questions and Answers. URL: ftp://st.cs.uiuc.edu/pub/patterns/papers/PatQandA.ps.

[9] Coplien, J.O. Advanced C++ Programming Styles and Idioms. Addison-Wesley, Reading, MA, 1992.

[10] Gamma, E. et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA, 1995.

[11] Boehm, H.J. Garbage Collection in an Uncooperative Environment. Software Practice and Experience 18:807-820, Sept. 1988.

[12] Boehm, H.J. A garbage collector for C and C++. URL: ftp://parcftp.xerox.com/pub/gc/gc.html.

A. Appendix

A.1. Declaration of GregorianDate

```
struct GregorianDate {
   string month;
   unsigned day;
   unsigned year;
};
```

A.2. Declaration of Stack<T>

```
template <class T>
class Stack {
public:
   class XPopEmpty:public domain_error {
   public:
      XPopEmpty(const string& info)
               : domain_error(info) { }
   };

   Stack(); // throw (bad_alloc, ...T()* )

   Stack(const Stack<T>&);
     // throw (bad_alloc,...T(),...T&=const T&)
   ~Stack() throw ();
   Stack<T>& operator=(const Stack<T>&);
     // throw (bad_alloc,...T(),...T&=const T&)
   unsigned count() const throw()
     { return top; }
   void push(const T& e);
     // throw (bad_alloc,...T(),...T&=const T&)
   T pop();
     // throw (XPopEmpty,...T(const T&) )
private:
   unsigned nelems;
   unsigned top;
   T* vec;
};
```

A.3. Declaration of BoundedInt and BoundedStack<T>

```
class BoundedInt {
public:
   BoundedInt(int bottom, int top) throw ()
      : bottom_(bottom)
      , top_(top)
      { }
   BoundedInt operator++()
      throw (overflow_error)
      {  if (++value_ >= top_)
            throw overflow_error(...);
         return *this;
      }
   operator int() throw()
      { return value_; }
   // ...
private:
   const int bottom_, top_;
   int value_;
};

template <class T>
class BoundedStack {
public:
   BoundedStack(unsigned maxsize)
     // throw (bad_alloc, ...T() )
      : current_size(0, maxsize)
      , current_top (new T[maxsize])
      { }
   ~BoundedStack() throw ();
   void push(const T& e);
     // throw (overflow_error,... T&=const T& )
   // ...
private:
   BoundedInt current_size;
   T* current_top;
};
```

A.4. Declaration of auto_ptr_array<T>

```
template <class T>
class auto_ptr_array {
public:
   auto_ptr_array(size_t n)
      // throw (bad_alloc, ...T())
      : ptr_(new T[n]) { }
   ~auto_ptr_array() throw ()
      { delete [] ptr_; }
   T& operator[](unsigned i) throw ()
      { return ptr_[i]; }
   T* release() throw ()
      {  T* tmp = ptr_;
         ptr_ = 0;
         return tmp;
      }
   void swap_with(T*& q) throw ()
      {  T* tmp = ptr_;
         ptr_ = q;
         q = tmp;
      }
private:
   T* ptr_;
};
```

---

[*] The `...`-notation is used to designate the set of all exceptions possibly thrown by a certain sort of expressions. Here it means "all exceptions possibly thrown by calling a parameterless constructor of `T`." This special sort of comment is used as C++ has no facility to "inherit" throw specifications from other functions.