A New Way of Error Handling

Peter Götz

December 2010

Abstract

The problems of common error handling constructs such as error return codes or exceptions are investigated based on examples in different commonly used languages such as C, C++ and Java. The Google Go language is used as an example of a modern language which does not use exceptions. While error return codes give a precise way of handling errors it is usually too verbose and obfuscates the business logic. Exceptions, although at first glance seem like an elegant solution, lead to sloppy error handling or to code littered with try-catch-finally blocks as stated also in the Google Go FAQ. A new way of error handling that works around these issues by introducing new constructs is proposed.

1 Introduction

Since the advent of exceptions as a language construct in programming languages it was praised as the holy grail of error handling by many people. It was preferred over the primitive use of error return codes as used in low level languages like C, because of its cleanness.

But there are downsides of exceptions. The most obvious is that it is always hard to decide if something is truly *exceptional* or if it is an *expected* condition which we can handle with a conventional if-statement. The authors of the highly regarded C++ Boost libraries for example therefore state, an appropriate question to whether or not to use an exception is: “Can I afford stack unwinding here?”, which means that there is a performance decrease when using exceptions.

The designers of Google’s Go language go even further and argue, that exceptions often lead to code convoluted with try-catch-finally statements and that the question if a situation is exceptional or not is highly subjective. They therefore did not include exceptions in the language, but rely on multiple return values and a new constructs called “defer, panic and recover”.

In this paper I will not argue about when to use a construct specifically made for error handling like exceptions, or when to use a conventional if-statement. I will rather argue that the concept of exceptions is in itself a flawed way of handling errors. And while error return codes are a better way of handling errors, they are both too verbose and also not safe enough.

In order to explain the shortcomings of today’s common error handling constructs I will revise the different approaches in different common languages like C, C++ and Java and I will also show an example how Google’s Go language approaches the problem. Afterwards a formal discussion shall lead to a new concept of error handling which is presented in the last section of the paper.

2 Case Study

Throughout this paper I will use a simple function as an example which represents an operation consisting of several steps. I will then show different methods used traditionally for error handling.

Now let us have a look at this operation:

```
void operation()
{
    A a = set_up_a();
    B b = set_up_b(a);
    C c = set_up_c(b);

    do_some_work_with(b);
    do_more_work_with(c);

    clean_up_c(c);
    clean_up_b(b);
    clean_up_a(a);
}
```

The code is very clean and the business logic is easy to spot: we acquire three resource which depend on each other, do some work on the last two and finally release or clean up the resources in the end. This operation could be something as common as acquiring a lock, then opening one file, then another file, reading or writing the two files, then closing them and finally releasing the lock.

Unfortunately the code as shown does not handle errors at all. What happens when a file could not be opened? The operation will continue in an undefined state. Since this is not acceptable in a non-trivial program we need to handle those special cases.

2.1 Error Handling in C

The code as shown above would work in C. Even if each function such as `set_up_a` returned 0 in case of error or `do_some_work_with` returned immediately with return code -1 if given an invalid parameter, the function `operation` would run and return no error. This is one of the most critical problems with error handling in C using error return codes: the programmer can (accidently) ignore errors and might only find out when it is too late.

Therefore in theory, in C one common way to do proper error-handling is by using `goto` statements as shown below:

```
int operation()
{
    int status = SUCCESS;
    int rc;

    rc = set_up_a(&a);
    if (FAILED(rc)) {
        status = ERROR_A;
        goto SETTING_UP_A_FAILED;
    }

    rc = set_up_b(a, &b);
    if (FAILED(rc)) {
        status = ERROR_B;
        goto SETTING_UP_B_FAILED;
    }

    c = set_up_c(b);
    if (FAILED(rc)) {
        status = ERROR_C;
        goto SETTING_UP_C_FAILED;
    }

    rc = do_some_work_with(c);
    if (FAILED(rc)) {
        status = ERROR_DOING_SOME_WORK;
        goto DOING_WORK_FAILED;
    }

    rc = do_more_work_with(c);
    if (FAILED(rc)) {
        status = ERROR_DOING_MORE_WORK;
        goto DOING_WORK_FAILED;
    }

DOING_WORK_FAILED:
    clean_up_c(c);

SETTING_UP_C_FAILED:
    clean_up_b(b);

SETTING_UP_B_FAILED:
    clean_up_a(a);

SETTING_UP_A_FAILED:

    return status;
}
```

This code correctly handles all error situations by reacting on return codes returned by the low-level functions, setting the high-level return code to be returned and making sure all necessary cleanup code is called.

Notice that this code abstracts the errors that can happen within `operation` to the caller. Low-level errors are translated into high-level errors which the caller of `operation` can understand. This is a very important aspect and we will come back later to it when discussing exceptions.

The advantage of this type of error handling is that we precisely react on error conditions.

But of course this code has some huge disadvantages. It is hard to spot the business logic. And this even though not even loops or other control structures for the business logic are used. With this approach it's also a lot more complicated to realize functions that return a value, because this return value must now be passed as an out argument to the function.

The error handling shown here using return codes is the traditional method used before the concept of exceptions was introduced in programming languages.

2.2 Error Handling in C++

The verbosity of error handling as shown above led to the development of a new way of error handling: exceptions.

The C++ language implements the concept of exceptions as a language construct. It is different than the one used in Java due to their different nature of handling instances of classes. In C++, whenever we work with resources, we use the RAII idiom to make sure the resource gets released independent of any error conditions. For our example operation from above we would define three RAII classes:

```
class A {
    A() {
        set_up_a();
    }

    ~A() {
        clean_up_a();
    }
};

class B {
    B(A a) {
        set_up_b(a);
    }

    ~B() {
        clean_up_b();
    }
};

class C {
    C(B b) {
        set_up_c(b);
    }

    ~C() {
        clean_up_c();
    }
};
```

Then we could write our function as follows:

```
void operation()
{
    A a;
    B b(a);
    C c(b);

    do_some_work_with(c);
    do_more_work_with(c);
}
```

Again the function is very clean and the business logic is easy to spot. In case an exception is thrown in one of the low-level functions `do_some_work_with` or `do_more_work_with` all resources are automatically released through the destructors of our RAII objects.

However there is a major and a minor drawback. The major drawback is, that this version does not abstract the low-level errors. E.g. if the low-level function is a file open call, which fails and throws a `CouldNotOpenFileException`, the caller of `operation` is confronted with handling an exception he might not even be aware of could happen in `operation`. The caller should not be expected to know all the exceptions that can happen in `operation`. He should only be required to handle exceptions that belong to the `operation`'s interface, done by comment in case of C++ (the use of exception specifications is not recommended). Of course one could argue that indeed the caller of operation should not handle a low-level exception and just let it go through until it's finally caught somewhere in `main()`, logged into some log file and the program terminated with internal error return code. But then the question is: what is the purpose of exceptions propagating up the stack in the first place? We will get back to this question once we discussed the remaining examples.

A minor drawback is, that the code in `operation` is not very explicit about cleanup. In some cases this is fine, e.g. when the RAII object seems very natural in use like an open file. In other cases it is not as natural to use a RAII object and it feels unnatural to use one. RAII objects are also lots of coding overhead.

Now that we realized that we would like to give the caller a high-level exception we can translate the exceptions as follows:

```
void operation()
{
    try {
        A a;
        B b(a);
        C c(b);

        do_some_work_with(c);
        do_more_work_with(c);
    } catch (const SetupErrorA& e) {
        throw HighLevelErrorA;
    } catch (const SetupErrorB& e) {
        throw HighLevelErrorB;
    } catch (const SetupErrorC& e) {
        throw HighLevelErrorC;
    } catch (const SomeLowlevelError& e) {
        throw SomeHighLevelError;
    }
}
```

The advantage now is that the business logic is still easy to spot, because we separated all the error handling from the business logic. The resources are still automatically released and the caller only gets high-level errors which he understands.

There are few drawbacks though:

1. We don't know where `SomeLowLevelError` comes from and therefore cannot translate it differently into a high-level error. The only way to solve this problem is a nested `try-catch` block.
2. With the code as shown, no recovery is possible.

Both of these first two points would require a nested try-catch block:

```
void operation()
{
    try {
        A a;
        B b(a);
        C c(b);
        try {
            do_some_work_with(c);
        } catch (const SomeLowLevelError& e
            ) {
            do_some_recovery();
        }
        do_more_work_with(c);
    } catch (const SetupErrorA& e) {
        throw ErrorA;
    } catch (const SetupErrorB& e) {
        throw ErrorB;
    } catch (const SetupErrorC& e) {
        throw ErrorC;
    } catch (const SomeLowlevelError& e) {
        throw WorkError;
    }
}
```

So the advantage of having separated the error handling from the business logic is only a superficial one. In reality it just says that our error handling is not precise anymore or that we cannot react on specific conditions in case that is necessary.

Of course we could avoid the nested `try-catch` block and argue that the error in `do_some_work_with` is not truely exceptional and instead use the return code variant. But that does not really solve the issue and we just go back to error return codes as in C.

3. This is a more subtle error. What if the implementation of the low-level functions is changed and they can now throw an additional exception? It would slip through `operation` and there would be no error translation anymore. Of course we can add something like a `catch(const std::exception& e)` or `catch(...)` in every function we write that is similar to `operation` which then translates into a high-level exception. But that would pollute the code with error handling constructs even more.

2.3 Error Handling in Java

In Java due to its nature of not having destructors, the cleanup must happen in the finally block. That is, we could do something like this:

```
void operation() {
    A a;
    B b;
    C c;
    try {
        a = set_up_a();
        b = set_up_b(a);
        c = set_up_c(b);
        do_some_work_with(c);
        do_more_work_with(c)
    } catch (SetupErrorA e) {
        throw new ErrorA();
    } catch (SetupErrorB e) {
        throw new ErrorB();
    } catch (SetupErrorC e) {
        throw new ErrorC();
    } catch (SomeLowlevelError e) {
        throw new WorkError();
    } finally {
        clean_up_c(c);
        clean_up_b(b);
        clean_up_a(a);
    }
}
```

Although declaring our variables at the beginning before the `try-catch-finally` block is more code than we actually would like to have, and we have again the problem of not being able to recover from errors in `do_some_work_with`, the real issue with this example is in the `finally` block: if an exception is thrown in `set_up_b`, we will end up in the `finally` block, which calls all cleanup code. All cleanup methods must be implemented in a way that they do nothing in case the object was not set up yet. The other solution would require three nested `try-catch-finally` blocks which would destroy the whole purpose of cleaner code.

2.4 Error Handling in Google Go

Google Go is not a commonly used language (yet). The approach to error handling its designers used, however, is interesting in this discussion. They "believe that coupling exceptions to a control structure, as in the `try-catch-finally` idiom, results in convoluted code". That is what we have seen in the previous section.

Their approach to solve the problem is by

1. using the language's ability to return multiple values.
2. introducing a new keyword `defer` to handle cleanup code.
3. introducing a panic mode which is similar to an unhandled exception which propagates up the stack.

The ability to return multiple values simplifies the usage of error return codes, because errors can simply be returned alongside other return values. That avoids out arguments and provides a clean way of calling functions.

However, checking the error code in a following `if`-statement is again more verbose than we would wish.

More critical though is, again, the possibility of (accidently) ignoring the returned error.

The new `defer` statement reminds of C++'s RAII idiom, although it is much shorter in use and more explicit. This is a welcome improvement.

It is arguable what is better: having cleanup code right next to where the setup code is located. Or having cleanup code where it is actually executed, i.e. at the end of a function. Go's designers chose the first one. I believe the second one is better.

The following two paragraphs quote the explanation from the Go blog about the panic mode:

“`panic` is a built-in function that stops the ordinary flow of control and begins panicking. When the function `F` calls `panic`, execution of `F` stops, any `defer`red functions in F are executed normally, and then `F` returns to its caller. To the caller, `F` then behaves like a call to `panic`. The process continues up the stack until all functions in the current goroutine have returned, at which point the program crashes. Panics can be initiated by invoking `panic` directly. They can also be caused by runtime errors, such as out-of-bounds array accesses.

“`recover` is a built-in function that regains control of a panicking goroutine. `recover` is only useful inside `defer`red functions. During normal execution, a call to `recover` will return `nil` and have no other effect. If the current goroutine is panicking, a call to `recover` will capture the value given to panic and resume normal execution.”

3 General Discussion of Error Handling

Besides all these shortcomings of existing error handling techniques as shown above, there is a general problem with the concept of exceptions. I would like to discuss this in the following.

Exceptions in their very nature do not describe *where* or *when* an error happened, but only that a certain *type* of error happened. Catching an exception never tells me where this exception came from unless the exception type is unique and thrown only at one specific line in the code, which is called only in one specific execution path. In all other cases, let’s say e.g. a `CouldNotOpenFileException` was thrown somewhere, and I catch it only in main because I forgot to catch it where I actually should have caught it, there is no way to properly handle that. It is because *an error is only meaningful in a context.*

So the mechanism of automatic progagation that exception provide is not useful. It violates the principle of abstraction, because high-level functions must understand what happens in low-level functions.

So my conclusion on how error handling must happen in reality is:

When an error happens in a function and it cannot handle it, it must be reported to the caller. If the caller cannot handle it, the caller must translate it into an error which the caller’s caller can understand and report it again. This of course is exactly the principle of error return codes as used in C. The problem with the C implementation, however, is, that we can (accidently) ignore the error return code. But that leads to undefined state, something we never want in a program and as soon as that happens we can as well terminate and print a stack trace.

Exceptions cannot be ignored. And if not swallowed via `catch(...)`, they eventually lead to some reaction in case of an erroneous condition. That is good. But the problem is, as described, that when propagated up more than one layer of abstraction, it is useless, because we don’t know the context they were thrown in. Since each stack frame represents a layer of abstraction, we never want an exception to propagate more than one stack frame up to its caller.

So what exactly should happen when a function fails and returns and its caller does not handle the failure? A logical approach would be to directly terminate the program and print a stack trace. Here is why: something that propagates up the stack automatically, and is eventually caught in main and logged, as is done commonly when using exceptions, actually leads to undefined state. Because we do not know *when* and *where* this exception was thrown.[1]

Why is this logical approach a bit extreme? Because programming errors can happen. But especially in server applications that should not mean that the whole server terminates. And usually undefined state is not as fatal as it is in theory.

A mechanism like the panic mode in Google’s Go language would be a adequate compromise between being strict enough about errors while still being practical enough for everyday use. The panic mode should rarely be used and only at very high level, like e.g. the http-request dispatcher loop in a webserver. There is probably no use for panic mode in a client application. We should instead let the application peacefully die and print the stack trace.

4 A New Concept of Handling Errors

In the following I will describe a new concept and syntax of error handling. It consists of three new constructs:

[1]In Java of course we could examine the exception and learn about the stack trace and recover accordingly, but this is not a solution for proper error handling.

1. a fails_with statement in the signature of a function, which tells the caller of this function the type of error it can expect when calling this function. The function itself can fail by using the fail_with statement.[2]

2. acquire and release which are an easy way for setup and cleanup code.

3. an or_on_error statement that gives a caller the possibility of reacting on a failed function call.

All keywords were chosen carefully to avoid confusion with already existing concepts, but still precisely explain what they mean.

4.1 A Simple Example

Let us start with our usual function operation to see what it looks like:

```
void operation() fails_with HighLevelErr {
    acquire a = set_up_a() or_on_error
       fail_with HighLevelErr("Set up a");
    acquire b = set_up_b(a) or_on_error
       fail_with HighLevelErr("Set up b");
    acquire c = set_up_c(b) or_on_error
       fail_with HighLevelErr("Set up c");

    do_some_work_with(b) or_on_error
       fail_with HighLevelErr("Some work");
    do_more_work_with(c) or_on_error
       fail_with HighLevelErr("More work");

    release clean_up_c(c);
    release clean_up_b(b);
    release clean_up_a(a);
}
```

Let us have a closer look to the new language constructs and its keywords.

The acquire and release keywords should be trivial to understand: for every instruction behind the acquire keyword, there is a matching instruction behind a release keyword. The compiler must make sure, that the number of acquires and releases are the same and report an error if this condition is not fulfilled. For each acquire statement that was successfully executed the corresponding release statement will be called no matter how we return from operation.

Then there is the extended version of a function signature. The fails_with keyword denotes that in case something happens this function cannot handle, it will report a HighLevelError back to its caller. The caller can react on it exactly the same way we do it in the implementation of operation when a low-level function call fails. It is done with the or_on_error keyword. or_on_error can be used in the way as shown or it can take a parameter e which will be filled with the error that the low-level function reported. The following block can then do with e whatever it wants. For example it can use it to create a HighLevelError which is reported to the caller of operation with the keyword fail_with.

Now here are some very important things to note: the caller of a function does not have to react on an error with or_on_error and the compiler should not enforce this via error. But if an error occures and the caller of a function does not react on it, the runtime environment should immediately switch to panic mode as described in the previous section. When the panic mode is not recovered it should terminate and print a stack trace from the function that called fail_with. Here is why: If the compiler enforces this via error, it is very annoying for the developer and might lead to empty or_on_error blocks which would exactly miss their purpose. A typical developer behaviour as seen with Java's checked exception could evolve.

HighLevelErr is a conventional class or struct:

```
struct HighLevelErr {
    HighLevelErr(const string& msg)
        : message(msg) {}
    string message;
}
```

4.2 Recovering From a Failure

To see how powerful these new constructs are, let us see what we can do if we actually can recover from within operation. E.g. let us assume we know how to recover from a failure in do_some_work_with:

```
do_some_work_with(c) or_on_error
    do_some_recovery() or_on_error
        fail_with HighLevelErr("Recovery
           failed");
do_more_work_with(c) or_on_error
    fail_with HighLevelErr("More work");
```

That means we can easily provide recovery functionality if something went wrong and in case we could recover, easily continue as if nothing had happened. Note that exceptions only allow something like this with a nested try-catch block.

[2]This is very similar to the much hated checked exceptions in Java. On the one hand I must say, I do not think that the concept of checked exceptions is bad, and it seems like many people only do not like it because they do not want to write code that carefully handles errors correctly. On the other hand, the compiler should not report an error, but rather a warning if a function throws an exception that is not in its signature. In case this actually happens during runtime it should simply immediately terminate.

Of course, if several recovery steps are needed, it is easy to use a block:

```
do_some_work_with(c) or_on_error {
    do_recovery_step1();
    do_recovery_step2 or_on_error ...;
    do_recovery_step3();
}
do_more_work_with(c) or_on_error
    fail_with HighLevelErr("More work");
```

Here we assume, that only step 2 in the recovery can fail.

4.3 Reacting on the Error Type

In the new error handling as proposed there is no such thing as an error hierarchy. Every function fails with a specific error type (which would be a struct in most cases). This error type provides as much information as is needed for the function to report errors to its caller. How much information that is, is defined by the programmer. This error type strictly belongs to the interface and therefore to the signature of the function. Here is an example:

```
enum LowLevelErr {
    ErrorA, ErrorB, ErrorC
};
```

```
void func() fails_with LowLevelErr {
    do_non_failing_stuff();
    try_stuff() or_on_error
        fail_with ErrorA;
    do_more_non_failing_stuff();
    try_more_stuff() or_on_error
        fail_with ErrorB;
    do_even_more_non_failing_stuff();
    try_even_more_stuff() or_on_error
        fail_with ErrorC;
}

void operation() fails_with HighLevelErr {
  func() or_on_error(e)
    switch(e) {
      case ErrorA:
        fail_with HighLevelErr("Error A");
      case ErrorC:
        fail_with HighLevelErr("Error B");
      default:
        fail_with HighLevelErr("Some Err");
  }
}
```

In general people tend to be suspicious about `switch` statements. However in this case there is nothing wrong with it. Note that several `catch` blocks following a `try` block is equivalent to this `switch` statement. If we want error reporting on such a fine-grained level, there is no way around such a verbose treatment of errors.