The Exception to the Rule

Adrian Kosmaczewski

2005-02-13

The rainy, later snowy, Swiss weather of this weekend has made me just stay at home, and take the chance to try out a few things: that's how I made my first steps in Ruby and WebObjects. These two deserve an article of their own.

However, surfing on CNN I found an article about Ariane 5, which successfully took off this weekend. That made me remind of the 1996 accident, in which one of the first (if not the first, can't remember) Ariane 5 rockets just exploded 40 seconds after takeoff.

I just did not know that the reason of this was a software error… a bug!

A quick Google on "Ariane" brought up this interesting document. In this page one can find the explanation of the software error that caused the explosion and subsequent loss of (hang on) half a billion dollars (uninsured); that is, 500 million bucks! And the cause of it is (prepare your compilers) an unhanded exception due to a floating-point conversion error!

Credits: http://www.uni-stuttgart.de/akamodell/index.eng.html (Click to enlarge)

> *The exception was due to a floating-point error: a conversion from a 64-bit integer to a 16-bit signed integer, which should only have been applied to a number less than 2^15, was erroneously applied to a greater number, representing the "horizontal bias" of the flight. There was no explicit exception handler to catch the exception, so it followed the usual fate of uncaught exceptions and crashed the entire software, hence the on-board computers, hence the mission.*

Incredible, huh? It seems to be the most expensive software bug ever, hopefully not causing any human casualties. The explanation that follows in that article is of greater interest and I highly recommend it.

This took me to the Eiffel language; I had heard about it but didn't know any of its characteristics, nor seen any sample code. Now it seems that Eiffel introduces a language-level syntax that allows what the Eiffel guys name "Design by Contract (TM)" (watch your step, it's a trademark). In spite of the marketing blah blah that goes around these fancy words, I was caught by the definition

and the syntax of Eiffel, that takes the definition of a method to another level. Take a look at this:

```
method_name (parameter_name: INTEGER): INTEGER is
require
    parameter_name <= some_maximum_value
    -- more conditions, if needed...
do
    -- code of the method here
ensure
    -- postconditions that must always be met
    -- no matter what happens, here
end
```

As shown in the previous example, an Eiffel method can define "require" and "ensure" blocks that contain pre- and post-conditions that must be met by the method before any processing of the "do" block. Really neat. What happens if these conditions are not met? Well, you've got your exception popping off the stack. Meet the conditions, and your method will exit cleanly and silently.

That's your "contract": take out the "do" block from your method, and you will see a couple of lines that describe a handshake contract between your method and any client that uses it. Even non-engineers can see it and understand what's going on, without having to deal with "how" things are implemented.

And how does this relate to our day-to-day activities? Of course neither me nor my colleagues work on Eiffel (yet). But, we do create software, we do handle exceptions (do we?), and we do lose money, credibility and faith every time there's an unhanded exception in our software. These exceptions cost us a lot: that "Design by Contract (TM)" thing can positively help us, just by rethinking the way we build our software.

Here's my idea: even if we don't use Eiffel, our good-old Algol-related languages can be used in pretty much the same way as Eiffel behaves, but of course it requires some of our own brainy CPU time. The trade off is simply a much clear interface that fits into a higher level, architectural view of the system, explicitly stating the valid ranges of execution for our code, and helping out in setting unit and integration tests.

Consider what an "exception" is: it is simply not the rule. For example, if a method must read the contents of a file in a remote network location, several things can happen:

- The network connection may not open;
- The network connection may close unexpectedly in the middle of the transfer because of some proxy between our code and the file server;
- The file might not be there at all;
- The file might be there but might not be read because of security or sharing reasons;

- Etc…

Seen like this, it seems like a miracle that our routine works at all, right? Well, even with all those problems, our method should take a string (the filename) and return a stream of bytes (the binary contents of the file) no matter what happens in the middle. The rest, simply has nothing to do with the "normal" workflow of the application: those situations are handled as exceptions. Excuse me, I will repeat and slightly change the last sentence to make the point: those situations must be handled as exceptions.

This way, we have defined the contract for our method: it should expect some kind of string (thus a simple regular expression could help us tell if the string is a valid filename -without forbidden characters- or not) and should return some byte stream (hopefully encoded in a way that our hardware and software can read it!).

Our code, the interesting part of it, can then trust its environment and perform its operations in the safest possible way.

Eiffel's capabilities not only deal with methods but also with classes as a whole: it allows class developers to define "invariants", that is, conditions (such as boundaries) that class fields should respect at any time during the lifetime of an instance. These "invariants" can also be inherited by the subclasses, thus providing not only behavior and structure inheritance but also validation rules inheritance. This is an extremely powerful concept, that I wish C# had built-in from the beginning (in fact I would trade .NET generics for some time of Eiffel-like capabilities…!).

To achieve similar results in .NET, a couple of years ago I was involved in the development of an execution runtime that we called "DataServices". It allowed to decorate method signatures with .NET Attributes, having a special infrastructure perform pre- and post- processing on the method input and output. The underlying idea in Eiffel and DataServices was the same, and it's really close to the ultimate goal of AOP (Aspect Oriented Programming): provide processing infrastructures that allow to create bigger, reusable class frameworks.

Using DataServices, you could define methods like this:

```
[Log()]
[RequiresRole(Role.Admin)]
[Database(ConnectionType.SQLServer)]
public bool CreateRecord([RegExp("[a-z]*")] string name, [Range(1, 99)] int age)
{
    // just open the connection and insert, no further checking needed!
}
```

As you can see the CreateRecord method contains .NET attributes that define the valid ranges of execution for the parameters, and has some other attributes that define pre- and post-conditions to be checked prior to execution. This allowed us to centrally manage a quite large framework (~20 classes, ~100 quite

complex methods) and centralize the debugging, security, logging and parameter checking routines into a single location.

I think that we achieved a similar goal that the one promoted by Eiffel's designers, while theirs is much, much more elegant :)

I can only recommend watching the following presentations on the Eiffel website, which will give you a better overview of what's Eiffel about: http://www.eiffel.com/developers/presentations/dbc/partone/player.html?slide= http://www.eiffel.com/developers/presentations/dbc/parttwo/player.html?slide=

By the way, I have also asked for my trial copy of EiffelStudio, I got curious about it :)

Credits: the QuickTime video linked in this article comes from a CNN.com article dating from 1997.