Design by Contract

Adrian Kosmaczewski

2007-08-23

Even if Design by Contract is a trademark (Eiffel Software, 2007) the idea behind it is the more general one of “defensive programming”. As software developers, we often concentrate our efforts in the main code of the application, which is the interesting part, and that provides the realization of the use cases identified during the early phases of the project.

My opinion is that defensive programming techniques lead to more secure, stable, and less bug-prone code, and that they require less documentation, since the resulting code is more “self-documenting”. Moreover, I will describe in this paper language-independent techniques, that can be used in different situations and systems, that are somehow similar to the Eiffel approach.

Design by Contract

In a previous entry in my blog, I have described the Ariane 5 rocket disaster of 1996, its cause and how the notion of defensive programming could have helped avoid it:

> 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.

(Kosmaczewski.net, 2005)

Eiffel’s idea is coherent with the concept of “Defensive Programming”. It is not a new trend in programming, but if you Google for it you will find quite a few papers describing common techniques and best practices, like the one on CodeProject.com that I provide in the references (Manderson, 2004).

The idea behind defensive programming is that you should not trust what comes from outside your code, even if it is your own code that calls other pieces of your own code; you could just as well call it “limited applied paranoia”. This is important not only for stability purposes (you do not want to call methods on a null pointer) but also for security; take for example the well known “SQL injection attacks” that happen when you trust too much the inputs of your users… a little verification helps to avoid disasters.

In Eiffel, this is done using the following syntax:

```
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
```

(Source: Kosmaczewski.net, 2005)

Of course, not all programming languages provide this kind of pre- and post-verification syntax; some ways to apply defensive programming in non-Eiffel languages, could be for example:

- Asserting for validity
- Using Aspect-Oriented Programming
- Using proper exception handling

I will give a short overview of these in the following paragraphs, comparing them with the Eiffel approach.

Asserting

“Asserting” is a common technique used in C and C++, but that can be used as well in any other language; basically it consists in using a macro or function that will raise an exception (or halt the program execution altogether) if a condition is true. Typically this is done before sending a message to a null pointer, since this mistake is a common one:

> The ANSI assert macro is typically used to identify logic errors during program development by implementing the expression argument to evaluate to false only when the program is operating incorrectly. After debugging is complete, assertion checking can be turned off

> without modifying the source file by defining the identifier NDEBUG. NDEBUG can be defined with a /D command-line option or with a #define directive. If NDEBUG is defined with #define, the directive must appear before Assert.h is included.

(MSDN, 2007)

The use of macros help to remove the asserts from the shipping code, that are usually only used during development and debugging. You do not want to ship code that discloses too much information about errors…

The situation in which asserting is useful is not uncommon in other languages; the much feared "null pointer exception" can happen in JavaScript, Ruby, C#, Java, and many other languages. In Eiffel, this would be handled in the "require" block, right before the main code execution.

Just for the record, the Apple Cocoa runtime (and in general the Objective-C language runtime) allows messages to be sent to "nil" objects (aka null pointers), without problem. However strange this might sound, this has an interesing side effect; even if the developers forget to initialize a pointer, and send a message to it, nothing will happen; the application will not crash, and this particular feat is one of the secrets of the stability of the overall Mac system. It is not that developers make less mistakes or that some magic applies; the Cocoa runtime proactively protects users from sloppy programmers, providing a more stable environment for them: the application might not do what the user want, but at least it does not crash.

Aspect-Oriented Programming

AOP can be used in this context as well, intercepting messages before and after methods execution, to verify their conformity and their correctness. The problem with AOP is that the definition of aspects is usually done in a different code file, which makes it harder to understand and maintain; this is where the Eiffel approach is the most interesting, since the "require" and "ensure" methods are somehow part of the method's signature.

However, the advantage of AOP is that the definition of poincuts is much more flexible, and one could theoretically inject code in different situations, without having to touch the original code (and thus reducing coupling and cohesion); the AOP runtime would take care of the weaving for us. Moreover, the weaving could be changed at runtime, while "require" and "ensure" conditions in Eiffel are compiled and statically linked.

Just for the example, I will show a bit of Ruby on Rails code that shows a class that provides a "hook" for before-execution methods; these methods are called automatically by Rails before any execution:

```
# The administration functions allow authorized users
# to add, delete, list, and edit products.(...)
```

```
#
# Only logged-in administrators can use the actions
# here. (...)

class AdminController < ApplicationController

    before_filter :authorize

    # List all current products.
    def list
    @product_pages, @products = paginate :product,
                                        :per_page => 10
    end

    # Show details of a particular product.
    def show
    @product = Product.find(@params[:id])
    end

    (...)

end
```

The important part of the code above is the `before_filter :authorize` line, that tells Ruby to automatically call the `authorize` method before executing the other methods. The `authorize` method belongs the to the `ApplicationController` class, and the `before_filter` hook is defined in the `ActionController::Base` class, that's part of the Ruby on Rails framework. Rails uses the dynamicity of Ruby to "detect" a call to a method, and intercepts it to inject the desired behavior before the execution of that method. Similar hooks exist for post processing.

It is interesting to note that such mechanisms can also be implemented in static, compiled languages using class hierarchies, where base class' methods call virtual methods of subclasses, similar to how ASP.NET processes pages.

Proper Exception Handling

Modern object-oriented runtimes, such as Java, .NET, Ruby and Cocoa use "exceptions" to handle errors. Exceptions are objects that are "thrown" whenever some unexpected condition is met during runtime to the method callers up in the stack, until some method "catches" it (hopefully someone will). This objects carry a whole meaning about the error, and are much more explicit than the C / C++ approach of returning numeric codes (HRESULTs, anyone?). By looking an exception, maintainers can see the context of execution of the code, and find the bugs that have created (or allowed) it to happen.

Even if the idea is interesting, it is also known that Exception handling is an expensive way to handle errors; for each exception thrown, runtimes have to walk the stack and look for possible handlers. This operation is expensive (Sintes, 2001) and that is why exceptions should not be used to control program flow.

The canonical example to show this is the following: instead of writing this (pseudo) code

```
try
{
    openFile(fileName);
}
catch (FileNotFoundException e)
{
    doSomething(e);
}
```

one could write the following, functionally similar, but more "defensive" code:

```
if (fileExists(fileName))
{
    openFile(fileName);
}
else
{
    notifyProblem();
}
```

If the above code is part of a performance-sensitive system, such as a web application, and the "file not found" situation happens more often than not, then a lot of processing power could be saved by just introducing this small change in the implementation.

In the case of Eiffel, the "contract" for the above "openFile" method would not only include the file name, but also a "require" block stating that the file must exist before any processing. This way, all calls to openFile would be safe, and as a result, clients would not need to check that fact before calling the method. Less code, cheaper to maintain, and more stable.

Conclusion

Defensive programming is a mind paradigm; you can apply them in any language, be it procedural of object-oriented, and provides stronger code, resistant to changes and self-documenting.

References

Eiffel Software, "Building bug-free O-O software: An introduction to Design by Contract(TM)", [Internet] http://archive.eiffel.com/doc/manuals/technology/contract/

(Accessed June 22th, 2007)

Kosmaczewski, Adrian, “The Exception to the Rule”, February 15th, 2005 [Internet] http://kosmaczewski.net/2005/02/13/the-exception-to-the-rule/ (Accessed June 22th, 2007)

Manderson, Rob, “Defensive Programming”, August 6th, 2004, [Internet] http://www.codeproject.com/gen/design/defensiveprogramming.asp (Accessed June 22th, 2007)

MSDN, “assert (Visual C++ Libraries)”, [Internet] http://msdn.microsoft.com/library/en-us/vclib/html/_CRT_assert.asp?frame=true (Accessed June 22th, 2007)

Sintes, Tony, “Does exception handling impair performance?” [Internet] http://www.javaworld.com/javaworld/javaqa/2001-07/04-qa-0727-try.html (Accessed June 22th, 2007)

Thomas, Dave & Heinemeier Hansson, David, “Agile Web Development with Rails”, The Pragmatic Programmers, 2005, ISBN 0-9766940-0-X, sample source code taken from http://pragmaticprogrammer.com/titles/rails1/code.html (Accessed June 22th, 2007)