# About code and eggs - excuse me?

Adrian Kosmaczewski

2005-04-20

The purpose of this article is to show that the current trends in software development owe a lot to ancient mindsets, and that some good old Object-Oriented Programming (OOP) programming constructs are no longer accepted in modern business development scenarios. It serves also as an introduction to Service-Oriented Architectures (SOA), putting it into context of what has been done in the past, and what can be done in the future with it.

## Procedural vs. OOP

Since the OOP programming paradigm appeared, tons and tons of books and articles have been written about the advantages of it, against "older" ways used to structure the code of your application.

Let's face it: traditional compilers (not those targeting virtual machines such as Java or .NET) produce binaries, executables targeting some platform, no matter what programming language you choose, be it procedural or object-oriented. At the end, this binary code does not care whether it was pumped up using OOP or procedural techniques; the resulting binary code may be different (and usually is) but the processor does not care about it: it's just plain vanilla executable binary code.

By the end of the seventies almost all serious system programming was done in C; this language is sometimes referred to as "portable assembly" and is available in every single development platform on Earth. It has become a de facto standard language, used to create anything from operating systems, word processors, spreadsheets, you name it. It is the language that others have copied, both in terms of look and feel and capabilities (every "curly bracketed" language over there has borrowed something from C, be it C++, Objective-C, Java, JavaScript, C# or PHP). Kernighan and Ritchie can be proud.

C is a procedural language like classic ALGOL-family languages (Pascal, Basic, FORTRAN, etc), which means that you concentrate primarily in "how" you do things, rather than "what" is being done (or does it). In C you basically write kitchen recipes: you specify the ingredients, you assume the existence of at least one cook, you tell him for how long she or he should heat those ingredients, and voilà, you've got your omelette.

This is what programming was, until the seventies.

Alan Kay of the Xerox PARC changed everything with Smalltalk and the OOP paradigm; in any typical OOP language you no longer focus in the recipe itself, but you start "modelling your problem domain". This last phrase means it all: you start describing the classes of objects that interact (the cook, the ingredients, the pans, the resulting omelette) and then you describe the interactions among them (pan holds omelette, omelette contains eggs, cook breaks eggs, etc). Instead of focusing in the recipe, you focus in the theatre play that will ultimately deliver your omelette.

Let me tell you something that I learnt by experience: both approaches, OOP and procedural, are useful, none is "better" by any absolute means than the other. Each has its strengths and weaknesses, and you have to consider the context in which each applies better than the other; the important things to consider are the tradeoffs that you are ready to accept while building your solution, the size of it, and the underlying platform that you use to build it. Each of these parameters usually clearly determines which approach to use.

I hope that you get my point; I do not want to fuel the flames of another religious war, but this is what software development is all about.

20 years have passed since Stroustrup created C++, and now we are in good shape to look backwards and see what the industry has generated:

- Lots (and lots and lots) of project failures;
- Incredible (but far, far fewer) success stories;
- A thousand different methodologies;
- The dot-com boom;
- Google.

And among all of these stories, best practices emerged out of good and bad experiences. It is now possible to find great books about software development, telling what you should and should not do in every situation of most development projects. One does not have the right of not knowing what happened before anymore.

## Back to C

In plain C you would "break the egg" doing something like this:

```
Egg egg;
break_egg(&egg);
Omelette *omelette = cook_omelette(&egg);
```

You define a variable that points to an instance of an Egg structure, then pass it as reference to a C function that will provide the egg-breaking logic. So far so good. Here there is a clear separation between the data and the procedure, but the existence of the Cook is supposed, inferred, but never disclosed (nor needed, at least in this case).

The first reaction, when translating this code to OOP, would be to represent the above C snippet into this C++ code:

```
Egg * egg = new Egg();
egg->breakEgg(); // "break" is a reserved word, after all
Omelette * omelette = new Omelette(egg);
omelette->cook();
```

In this case, the egg-breaking logic is located inside the Egg class. We might have even used the same C code (maybe not, but C++ is a superset of ANSI C, after all), providing that you replace every instance of the "&egg" parameter variable by a "this" pointer, and you would have your code up and running in no time. The same is valid for the Omelette class.

But OOP is more than a way to produce fancy code; OOP was one of the first mainstream industrial solutions to one of the worst problems in software development: source code maintenance. Source code is not something that, once compiled, disappears forever buried inside your source control system: source code is an evolutionary asset, that might as well become a huge liability if you do not take care of it from the very beginning in your design, and this is true in any programming language, be it procedural or not.

Thus, OOP must be used to create code that can be maintained; by different people, at different times, in different places. Source code must be flexible enough to allow modifications, fixes and refactoring, and strong enough to represent a complete problem domain.

Let's say that we're now in 1990: what if we want to model the whole kitchen? What if we want to model multithreaded cookers doing several omelettes at the same time? Say that your client, thanks to the success of "Omelette 1.0" (released in 1987 for MS-DOS) decides to grow up his team of cookers and want to model different egg-breaking methods (say, the French, the Brazilian and the Thai methods). What do you do?

You must do one of the most important things in software development: decouple functionality. Omelette 2.0 for Windows 3.1 would have a new object model (and hopefully you have read "The Mythical Man-Month" and won't suffer the second system syndrome). Of course, you lose backwards code compatibility (and you should refactor your application completely, which could quickly become a huge problem…) but as you will see, the benefits are worth the change.

Thus, this would be the equivalent, more maintainable code:

```
Cook * cook = new Cook();
Egg * egg = new Egg();
cook->breakEgg(egg);
Omelette * omelette = cook->cookOmelette(egg);
```

Is it correct? Is it better? And in which context? Is it so "illogical" to pretend that an egg could break itself? What does "logical" mean in this context, any-

way? The discussions are endless, and I've seen every possible answer. Each OOP person will come up with its own experience and insight, and will try to convince you that her or his position is right and that others are wrong.

One can safely argue that the binary code produced by both approaches is essentially the same, and produces the same results during runtime (even if the first is slightly more efficient just because you don't have to create a "Cook" object in the heap to take care of the egg-breaking process).

My opinion is that the second C++ code is better, for a number of reasons:

- It is a more suitable view of the world; in reality, eggs don't usually break by themselves without external help; it also describes the cook, and this class might as well be used in other parts of the application for other tasks.
- Since it is a more "classic" view of the world, the code is more maintainable (the computer industry has always had important turnover rates, people come and go and you must assume that today's code will be maintained by tomorrow's people…)
- The Egg class becomes more a "data" class, without further complexity; the Cook class holds the knowledge of all the needed processes to create an Omelette.

Finally, remember the classic tradeoffs that appear when using OOP technologies:

Performance

- Usually OOP techniques are not suitable for high performance systems because of the payload needed for resolving polymorphic methods, or for heap allocation and deallocation; that's why even today, hardcore systems are coded in assembly or in C (operating systems, device drivers, embedded controllers, etc). For business applications, the primary concern regarding performance has to do with the scalability of the systems when deployed in large organizations, where they can potentially be used by thousands of users simultaneously. Special attention must be paid to stress-testing critical systems before deploying them in production scenarios.

Final binary size

- Java and .NET both try to overcome this trade-off using the concept of virtual machines, with large pre-installed class frameworks; in this case, the resulting binaries are small, since they reuse the logic contained in the framework; at the same time, they provide greater portability and solve memory-management issues off-the-box as well.

Bigger model complexity

- As Robert Glass points out, a 25% problem complexity increase holds a 100% overall solution complexity increase. Be careful when you add a class to a model, and be sure that you need it.

## New millennium, new paradigms

So your client has been very happy with version 2.0 of the Omelette software, and later you provided to him Omelette 95 (it used C++ COM components to encapsulate broken eggs), Omelette.NET (developed in managed C++, providing server-based, scalable, multithreaded & distributed egg-breaking components shared by both managed Windows and Web applications) and now it's time to prepare for the Omelette SOAP API. That's it, it's time for the Service-Oriented version of Omelette.

Hey, your product has really gone far; now you will provide broken eggs throughout the Internet. By the way, you are happy to have chosen a "real" programming language such as C++ and an encapsulation mechanism such as COM during the nineties, rather than a pure Visual Basic solution :) which might have proven rather difficult to port to .NET… but that's another story.

And moreover… you are now happy to have an evolutionary object model. Because now you will be sending SOAP-encapsulated eggs as messages over the Internet, and your service will just receive them, and break those eggs using that high performance, secret algorithm that you shall not show to your competitors. Can you imagine how to translate

```
Egg * egg = new Egg();
egg->breakEgg();
```

into a Service-Oriented architecture? How would you do now if your Egg class had the logic in it? How would it be serialized into a SOAP message? The answer is, you can't do that. SOAP messages only hold data, while the egg-breaking code is stored in your web service.

So, you need decoupling the data and the procedure. Quite the opposite of what many OOP advocates said back in the eighties, right? The egg does not break by itself; it will be broken, as in the real world. And now we will handle it to someone over there to break it for us in a highly specialized manner.

This decoupling pattern is commonly used in all message-oriented architectures: it means the decoupling of the object that contains only data (something like a C struct) and another object that holds the logic (something like an object holding C functions). No longer are both entities contained inside the same structure, at least not from this high-level point of view.

And thus, the circle is closed: we have returned to a slightly more complex procedural system, that can be implemented (in a synchronous fashion) using the following code:

```
OmeletteService::Proxy * proxy = new OmeletteService::Proxy();
OmeletteService::Egg * egg = new OmeletteService::Egg();
proxy->BreakEgg(egg);
OmeletteService::Omelette * omelette = proxy->CookOmelette(egg);
```

At the end of the SOAP call, our `egg` variable contains an `Egg` instance with the "broken" flag set to true, and maybe some more information inside.

And the Cook? Well the Cook is inside the OmeletteService, somewhere on the network; it's like you had a central kitchen, that only expects from you to provide some basic material (the ingredients) and they will provide you with a delicious Omelette instance. The interaction between the client code and the service becomes a workflow, a set of rules that can be easily monitored and maintained, and even better: you can now offer Omelettes to any system that talks SOAP…

## Conclusion

Somehow, the last code snippet (the SOAP call) is extremely similar to the C code shown above. And we are using the most advanced technologies! This is confusing and marvellous at the same time; we have gone through the whole circle of development, from pure procedural, to pure OOP, to a mix of both that enables the writing of more complex, distributed applications.

The new paradigm of the Service-Oriented Architecture holds the promise of information exchange, high level component reuse at network level, interoperability with otherwise incompatible systems, and stateless, loosely-coupled, message-based components, while using the primary decomposition of code and data that was at the core of procedural programming mindsets.

Remember:

- Decouple data and procedures;
- Think messages;
- Stress-test your solutions;
- Reduce dependencies everywhere;
- Don't fear modelling and documenting your solution until you and your team feel comfortable with the object model;

Lots of things to keep in mind but remember: you can't do an omelette without actually breaking some eggs!