Evolution in Action – Building Up to a Service-Oriented Architecture

Because this column has to be written several months in advance, I am writing it at the 2007 Systems and Software Technology Conference (SSTC) in Tampa Bay, Florida. Talk about a bunch of geeks[1]! I mean this in a nice way, of course (because I know I *am* one).

Seriously, the SSTC had some great exhibits this year. Most of the exhibits, of course, displayed software products designed to help you develop and maintain your systems, which leads me to the topic of this column: Service-Oriented Architectures (SOAs). SOAs are loosely defined as *an environment made available as independent services that can be accessed without knowledge of their underlying platform implementation*[2]. Interoperability – what a concept. You see, it's all evolutionary, because….

… IN THE BEGINNING was machine language. You remember?

BALR R14, R15
USING *,*?

The problem was that machine language was so closely tied to the machine that it wasn't even transportable across different machines from the same vendor. Thus, programs that ran on an IBM 1401 with a specific memory configuration would not even transport to another differently configured 1401. What we needed was…

… HIGH-LEVEL LANGUAGES. FORTRAN, COBOL, RPG, and eventually, languages like C and Standardized are transportable (within limits) from one machine to another. Which leads to….

… PARAMETER PROBLEMS. If a C program passed two parameters (such as an integer and a real), but the receiving subprogram read them as a real followed by an integer, it would try and work and usually fail because the data was interpreted incorrectly. When I taught at the U.S. Air Force Academy back in the '80s, we used Pascal and always taught that parameters had to agree (between caller and callee) in number, order, and type. Early on, software engineers found out that about 75 percent of all errors occurred not directly in the code but in the code interfaces. So we tried to emphasize to students to *always* check the number, order, and type of parameters. Of course, telling them to check their parameters was not as good as …

… ENFORCED PARAMETERS AND STRONGLY TYPED LANGUAGES, Ada (and its successors), C++ (to an extent), and Java. Want to pass four parameters that consist of two reals and two floats? Then let the compiler and run-time environment *enforce* the parameter number, order, and type. If you tried passing them in the wrong order, then the program would return an error and not attempt to convert incorrect data. In fact, if you accidentally try and pass a floating point number as an integer (and perhaps lose precision), the compiler/runtime environment will also prevent that, giving you a better chance at correct programs and preventing accidental parameter type mismatches. However, as programs grew larger and larger (and systems of systems evolved) the interfaces between multiple subprograms became larger and larger (with more and more parameters), which leads to…

… STANDARDIZED LIBRARIES. Why bother to pass a zillion parameters to a handwritten user display procedure (which took lots of time to write), when a completely pre-written Graphical User Interface was available? All you have to do is use an object-oriented language and development environment, spend a little time researching which services and libraries are available, and then inherit/instantiate the code you need. Don't write it – REUSE IT! However, to make standardized libraries and reusable code/services efficient and cheap, we really needed…

… STANDARDIZED OPERATING SYSTEM SERVICES. Back in the '80s and '90s, you couldn't trust the operating system (OS) code. You used the OS to load your program, but then you wanted to write your own memory management, real-time scheduling, and other critical services. However, as OSs became more and more standardized, they provided more and more services. In fact, what you wanted was a reasonably secure and reliable set of services that would let you not just reuse code, but also let you utilize reusable services. Which is why we need…

… SERVICE-ORIENTED ARCHITECTURES. It's all about saving time and money. High-level languages are cheaper than assembly language. Standardized OSs give you services that are cheaper than *roll your own*. Reusing code saves you time and money. So why not have take advantage of *independent services* with defined interfaces that can be called to perform their tasks in a standard way, without the service having foreknowledge of the calling application, and without the application having or needing knowledge of how the service actually performs its tasks. SOAs are an evolutionary step above standardized OS services, and can be regarded *as a style of information systems architecture that enables the creation of applications that are built by combining loosely coupled and interoperable services*[2]. All of the good software engineering buzzwords that we have tried to teach over the last 20 or so years are inherent in an SOA: loosely coupled, interoperable, abstractions, and enforced interfaces.

There you have it – engineering evolution in a nutshell. By using an SOA, you will achieve savings in both development cost and time. As a single example, interprocess communication and network interface issues will have been solved in an SOA (and when they need updating, the SOA will be updated, not your program!). It's just like plagiarism but without the moral dilemma.

Why do you want to reinvent the wheel when there are already perfectly good wheels that somebody else has already built?

— David A. Cook, Ph.D.
The AEgis Technologies Group, Inc.
dcook@aegistg.com

Notes

1. Well, where else could you find an audience that appreciates the following? "Obviously, God LOVES the C programming language, because in Genesis 1, verse 2, it clearly states that the earth was without form, and it was VOID." And, for you FORTRAN programmers: "God is REAL (unless explicitly declared INTEGER)."
2. Wikipedia <http://en.wikipedia.org/wiki/service-oriented_architecture>.