COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

Rapid Business Change and ABL Productivity

16 March 2011

Thomas Mercer-Hursh, Ph.D.

Experts tell us that the rate of change in business today has increased dramatically. This creates demands for rapid software development and quick changes to existing software. The compelling story for ABL as a fourth generation language (4GL) has always been productivity. However, the productivity advantage ABL once enjoyed over third generation languages seems not to be as great as it was 20 years ago. I review issues in ABL productivity leading to a proposal for Progress Software to explore Model-to-Code translation techniques as a means of substantially boosting ABL productivity and responsiveness.

Increasing Rate of Change

Lately, the theme of increasing rate of business change has dominated PSC publications and conference presentations. This rate of change creates a need for productivity, both in the speed of creation of new applications and, perhaps even more importantly, in the speed with which existing applications can "nimbly" respond to changing requirements and needs. This capacity is further reflected in the theme of Responsive Process Management. Naturally, PSC has a number of products which facilitate building such nimble, responsive systems, not the least of which is ABL itself.

At the recent Global Partner Conference, the theme of rapid change was reinforced in several ways. One provoking remark[1] was that Java, C# and similar languages were a dead end because they were not sufficiently productive to meet the needs for rapid change which characterized modern business[2]. However, the persistence of COBOL is sufficient lesson that Java and C# are not about to disappear. Clearly, a dramatic short term retreat from the use of these languages is unlikely because there is just too much entrenchment, too much momentum for the majority of development sites to make a rapid switch. It does, however, provocatively underscore the importance of productivity in responding to modern business needs.

Those commenting on the need for rapid change typically do not propose specific languages or tools as the solution, but rather suggest that the solution lies in the use of multiple tools. One example is Progress | Savvion in which Business Process Modeling (BPM) designer allows defining processes without coding and a BPM server executes these models. Some coding is likely for some process steps to provide interfaces to databases and other application software, but this coding is a tiny fraction of what would be needed to provide such process automation entirely through custom code. Consequently, processes can be developed quickly and be easily modified, providing the needed

[1] By John Rymer of Forrester in Grow Fast or Become Extinct.

[2] See http://blogs.forrester.com/mike_gualtieri/10-11-23-java_is_a_dead_end_for_enterprise_app_development for a blog on this subject and http://blogs.forrester.com/mike_gualtieri/11-02-03-explained_java_is_a_dead_end_for_enterprise_application_development for a link to a slide deck further explaining the ideas.

productivity and responsiveness. As a bonus, monitoring of the process execution provides valuable feedback for process improvement.

While several such tools undoubtedly have an important rôle in meeting the needs of modern business, there is still a major need for writing business applications. What can address the need for high productivity and nimble response in traditional application development?

Background

From its earliest days in the 1980s, ABL's allure has been its productivity. Progress Software, then Data Language Corporation, was one of few companies which championed the advantages of fourth generation languages for increasing programmer productivity. It was the only company which succeeded in creating a 4GL good enough to write entire applications in the 4GL without resorting to coding difficult areas in C or other 3GL. Other 4GLs could get high productivity producing those parts of the application which could be written in the 4GL, but lost productivity when having to write 10-15% of the application in C in order to fulfill requirements. Not only did this C code greatly impede easy maintenance, but overall productivity was substantially reduced because the hardest parts of the application were still coded the old way. Thus, these languages were the worst case of the Pareto principle[3], since the productivity gains were realized in the part of the application that didn't take a lot of work to create anyway and the part of the application which was hard work was done the same as it always was[4].

Naturally, languages have evolved since the 1980s. ABL has become less 4GLish (adding language constructs for finer grained control, e.g., to support the ABL GUI for .NET). Traditional 3GL languages have gained higher level structures like data sets which provide a greater degree of abstraction from the computational implementation. Perhaps more importantly, substantial evolution has occurred in tool support of 3GLs and in creation of libraries and frameworks to support these languages. Any code in a standard library or framework is code that does not have to be written, not only reducing the new code necessary to implement any requirement, but often providing some of the most difficult and time-consuming code. Thus, 3GL average productivity has risen relative to ABL. While studies are rare and many dubious, anecdotally this is a shift from perhaps a 10X productivity advantage in the 1980s to a 3X productivity advantage now.

Thus, one naturally asks the question of whether there is some development which would restore the relative productivity of ABL and, more importantly, help make ABL a language which provides the necessary productivity and nimble responsiveness needed to meet the needs of modern business.

ABL Productivity

The remarks at the Partner Summit resonated strongly with me since programming productivity has been a major theme in my career since 1979 when I created my first 4GL to speed development of business applications in AlphaBASIC[5]. In 1983, I developed and marketed a code generation tool for the OASIS CONTROL Toolkit. This quest for productivity was one of the main lures of the Progress product for me in 1984. In the early 1990s I created a tool called Specification-Driven Development (SDD) which produced ABL code from specification files and ABL code snippets.

[3] The Pareto Principle is the original formulation of what is often referred to as the 80/20 rule, i.e., in software, it takes 20% of the time to write 80% of the application and 80% of the time to write the other 20%.

[4] For example, a 5X gain in the 4GL portion would only reduce the 20% to 4% while the 80% would remain fixed, so the total reduction was only from 100% to 84% of the original time.

[5] AlphaBASIC was a business oriented language for use on the AlphaMicro platform.

SDD was designed to result in no-compromise applications and resulted in very high levels of productivity and very stable and easily modifiable applications as well. During one large project the productivity of SDD for new application development was tested at a rate in excess of 1000 lines of integration test ready ABL code per programmer per hour, though admittedly with a less structured analysis process than I would use today.

Model-Driven Architecture and Translation

In recent years this personal quest for productivity has turned toward Model-Driven Architecture (MDA), the name given by the Object Management Group[6] (OMG)), which defines the standard for UML, to generating application code from UML models. Some practitioners prefer to speak of "translation", as in translation of a model to code as being a more intuitive name for the process. While one generally thinks of MDA or translation as producing code from a model, the technology also includes Model-to-Model translation, in particular, creating a more platform-specific form of a model from a more platform-independent form, e.g., creating the model related to a particular UI platform.

There are several ways in which MDA enhances developer productivity. First, obviously, there is a substantial part of any application for which the implementation is entirely predictable from the combination of the functional requirements and the current development model. It is exactly this kind of code generation which SDD was created to exploit. While some up front effort is required to create the translations to produce the current development patterns from the model; after this investment is made, the definition of the model takes a small fraction of the time required to code the function by conventional means. More importantly, perhaps, the existence of a model implies up front analysis which decreases the likelihood of design flaws and increases the fit to requirements … including the clarification of those requirements before any code is produced. Improving fit to requirements reduces wasted time modifying a completed program to fit requirements which were not clear or explicit at the time the program was originally written.

Perhaps more important, however, is the impact on changes. If requirements change, one can quickly change the model and "push the button" to produce a fresh copy of the application. Not only is this extremely rapid, but the resulting code is highly stable and consistent—one of the big benefits I discovered with code from the SDD technology. Because of this stability and predictability, testing requirements are greatly reduced. In effect, one tests the translator initially and then much of the test of a new function is fulfilled by the pre-existing test of the translator because the translator can be relied upon to produce consistent results.

Even more dramatic is the impact when there is a change in the implementation. If one decides on a new feature or bug fix or change in operation, all that needs to be changed are the translation rules. Depending on the change, this can vary from trivial to significant; but even when significant, it is many orders of magnitude less work than having to rewrite every relevant part of the application. Change the translation, "push the button", and one has a whole new application which incorporates the new feature—typically at a tiny fraction of the effort that would be required to have manually implemented the change in conventional code.

If the change is more substantial, e.g., a new client type or a major change in architecture, then the savings in effort can be even more dramatic. While revisions to the translations take effort, the effort is more on the scale of figuring out what change needs to happen to a few instances of the impacted

[6] http://www.omg.org/

existing code than it is on the scale of applying that change to every instance of that structure throughout the code. Even something as sweeping as a change in client technology is a process of revising one set of translations that already covers all cases and then applying that set to all instances rather than having to consider each instance directly. Even a change in the target language for the client is a modest effort compared to manually rewriting the client code.

Indeed, one of the very attractive aspects of this technology is that a unified model can readily produce both ABL server code and non-ABL client code, if that is what a particular shop would like. One can readily produce multiple client code sets in different technologies if such is desired.

One should emphasize that this ability to produce new code from a revised model is a major watershed difference from one-time code generators in which code is created originally and then manually maintained going forward. If it is desired to change the implementation in some way, the only advantage one has with one-time generation code is the uniformity of pattern which might help in applying substitution tools. With true regenerability, the model can evolve repeatedly and the code reflects completely the on-going principles of design and implementation.

Conclusion

Providing the ability to generate code using MDA techniques is a key opportunity by which Progress Software can provide substantially higher productivity in production of new code and dramatically facilitate nimble revisions in response to changes in business requirements. This capability will provide an important market advantage for the users of its products and a marketing advantage in the high productivity levels which can be claimed[7].

PSC's acquisition of Savvion helps to underscore this conclusion because, within the sphere of business process management, Savvion does something strongly related to MDA. Models of business processes are constructed in a design tool and automatically converted to executable form. As there may be a need for manual interaction in this conversion for some tasks, Savvion is not 100% Model-to-Code, but the ability to regenerate and continuously modify existing models is strongly related to the advantages to be found with MDA.

I propose that Model-to-Code translation represents a major opportunity for increased productivity in ABL and one that is very much in tune with developments for other languages. Implementing such translation and making it available to ABL developers can boost productivity and increase quality and nimbleness, improving competitiveness with other development tools and enhancing the fit of OpenEdge to the rapidly changing requirements of modern business. This productivity advantage is not a minor issue of a few percentage points, but often can be as much as 10X[8].

An investment in Model-to-Code technology will return more substantial gains in core ABL productivity and responsiveness than any other possible investment[9].

---

[7] It should be noted that translation technologies are already in use in 3GL environments where they provide high levels of development productivity and nimble response to change. However, this usage is most common in certain specialized applications such as real time environments or telephony systems so there is still a marketing advantage for ABL in providing translation technology for enterprise business applications. Indeed, it is compelling to provide that productivity in order not to be outcompeted by 3GL systems using translation.

[8] Personal communication from H.S. Lahman, a 20 year veteran of Model-to-Code translation in complex real time applications.

[9] See *A Path to Model-to-Code Translation in ABL* for a description of issues and plans to achieve these productivity gains.