SPE 030886

Brief: Buy don’t Build - What does that Mean for a Software Developer?

Todd Little, SPE, M. Ahsan Rahi, Craig Sinclair, Western Atlas Software

Summary

The buzz-phrase of the 90’s for the petroleum software industry has become “Buy, don’t Build”. For an end user in an oil company, this generally means acquiring application software rather than developing it internally. The concept of “Buy don’t Build” can also apply for a software developer. Purchasing software toolkit components can expedite the development of an application as well as reduce future support requirements.

Introduction

Recently, the desire to reduce costs within the E&P industry has led several companies to investigate the significant expenses related to software development costs. The overwhelming conclusion of these investigations begot a slogan for the 90's: Buy, don't Build.

“Buy, don’t Build” conflicts with the industry’s prevalent "not-invented-here" mentality (“I know what I really want so I can do it myself and do it better.”). This mentality breads a re-invention process that can be very costly given the complexity of today's software. Additionally, this complexity makes it difficult to estimate the cost and duration of the development and maintenance activities, thereby introducing significant risks. Almost anyone in the software business can recount horror stories of projects gone awry. One way to reduce the software development complexity is to utilize software tools.

Our concern as a commercial software application vendor is deciding how to obtain the software tools necessary for the application development. For many components where off-the-shelf tools provide the required functionality, the decision to buy these tools is easy. In other cases where what is needed is truly novel, the decision to build may be obvious. It is the area in between where technical and economic analysis is required.

Buy Versus Build Economics

Building Software: One of the major problems with economic analysis of the buy versus build question is the accuracy of the software development work estimation. DeMarco[1] (Figure 1) shows an overwhelming bias of underestimation or work effort with a factor of two being fairly common.

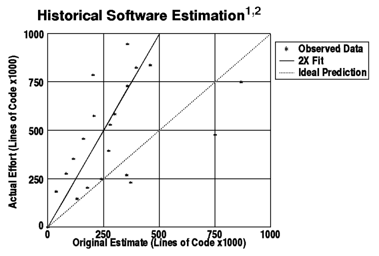

Figure 1

Inaccuracies in the estimation procedure are amplified by increasing complexity. Next generation software is roughly four times larger than its predecessor and the exponential relationship between code size and development effort magnifies this inherent estimation risk. Ultimately, the uncertainty and high cost of software development provides a significant incentive to look for alternatives.

Buying Software: Often, off-the-shelf components can be purchased which meet most requirements. The purchase cost will usually be much less than internal development since development cost are effectively shared by many customers. Risk is also reduced since the inaccuracy inherent in the effort estimation is replaced by bounded expenses.

Hidden Costs beyond the direct costs of buying or building may be important. On the build side, the opportunity loss as a result of a delay in the time-to-market for the application can be significant. On the buy side, generic tools often end up creating an incompatible environment or bring hidden baggage with them. Portability, optimization, and hardware requirements should also be considered.

Hidden Benefits can often be reaped by buying software. The product has been designed by experts and has improved through customer feedback. This greatly increases the usability in other projects and improves the likelihood of meeting future unforeseen needs.

Some Specific Case Studies

C++ Base Class Tools: In our early C++ development we utilized classes for lists, arrays, and strings that had been developed by a member of our group during university days. We incurred no cost building the software, only maintenance costs. Since a limited number of developers were utilizing the tools we decided productization was not necessary.

We recently re-evaluated the internal tools. With more developers needing to use the tools, productization had become a significant issue. In the meantime, the Rogue Wave Tools.h++ library had emerged as an industry leader. After a quick evaluation we determined that this library was far richer in functionality, designed much better, and was fully documented. Just documenting our existing toolkit would far exceed the acquisition cost. As in this case, the economic rationale favoring the buying of base tools is typically overwhelming.

3D Graphics: Our 3D applications required several fundamental features:

- Fast interactive display.
- Software rendering to X-terminals.
- Scalable hardcopy to CGM/PostScript.
- Portability to most UNIX platforms.
- Interoperability with X windows.
- Compatibility with Motif.
- Efficient memory utilization

We were unwilling to accept a significant performance degradation relative to our existing SGI-GL prototype. Ideally, we wanted an industry standard interface which would handle our data efficiently.

Our search found no single solution which met our needs. We therefore approached the problem differently. Instead of looking for a single solution we decided to combine the interactive performance of GL with the hardcopy and X support of HOOPS. The applications would be shielded by an object oriented middle level interface. This hybrid approach of buying and building minimized costs, while providing the required flexibility.

XY Plotting Tools: A logical starting point for our evaluation of XY plotting tools was to consider utilizing our 3D tools library. In fact, a functional prototype application was built using the 3D tools, but it lacked the polished look that we desired. We estimated that at least 6 months would be required to develop a minimal plotting tool, thus providing incentive for us to investigate purchasing a tool.

XRT/Graph from KL Group was a mature product that was in wide industry use for primarily business applications. It was limited to two vertical axes and it did not provide CGM output capabilities, both critical requirements. A good example of the 80/20 principle was exhibited when we inquired into the possibility of adding these features the tool; adding these features would cost about 5 times the cost of the base software.

In comparison to XRT/Graph, the PlotXY widget by INT was primarily targeted to the needs of the petroleum industry providing multiple axes, and CGM and PostScript hardcopy.

Upon completing the technical evaluation, we continued with the economic analysis. Figure 2 compares the timelines of buying versus building. In the case of buying, there would be start-up time associated with training. Once the tool was understood, the application development could proceed. This significantly shortened the time-to-market. Building required the design and development of a toolkit before the application development could get underway, and continued maintenance of the toolkit would rob valuable developer time from the application development

From a cost standpoint, the acquisition cost was far less than the expected burdened development costs. Furthermore, the risks associated with the development estimation and probability of future ongoing maintenance costs greatly favored the decision to buy.

Conclusion

The major driving forces in the buy versus build economics come from increasingly complex software development tasks. For many reasons, software expenses have traditionally been underestimated. This underestimation will probably continue, greatly increasing the risks associated with software development. These high costs and risks give a great incentive to consider alternatives such as buying off-the-shelf software.

Our direct experiences showed that the economic rationale favoring buying C++ base class tools was overwhelming. Documentation and productization costs for our internal tools would far exceed the purchase cost. For 3D graphics tools, no off-the-shelf package was considered acceptable. However, a combination of two tools was a viable alternative and we thus chose a hybrid "buy and build" alternative. This approach minimized the costs associated with building while providing the flexibility required to meet our needs. With regard to XY plotting tools, it would have been easy for the "not-invented-here" syndrome to set in and for us to have developed our own tools. In this case, "Buy don't Build" encouraged us to analyze the situation to determine the best alternative. Our analysis showed both that the direct development cost of building would exceed that of buying, and that there would be a significant hidden cost generated by the delay to market. The benefits of a generic, productized tool would also encourage future reuse and probable cost savings.

Sometimes internal development is necessary in order to meet requirements. If an off-the-shelf package can meet most of the requirements, then "Buy don't Build" can be a significant cost saver.

References

1. DeMarco, T., "1978-1980 Project Survey, Final Report," New York, NY, Yourdon Inc., 1981.

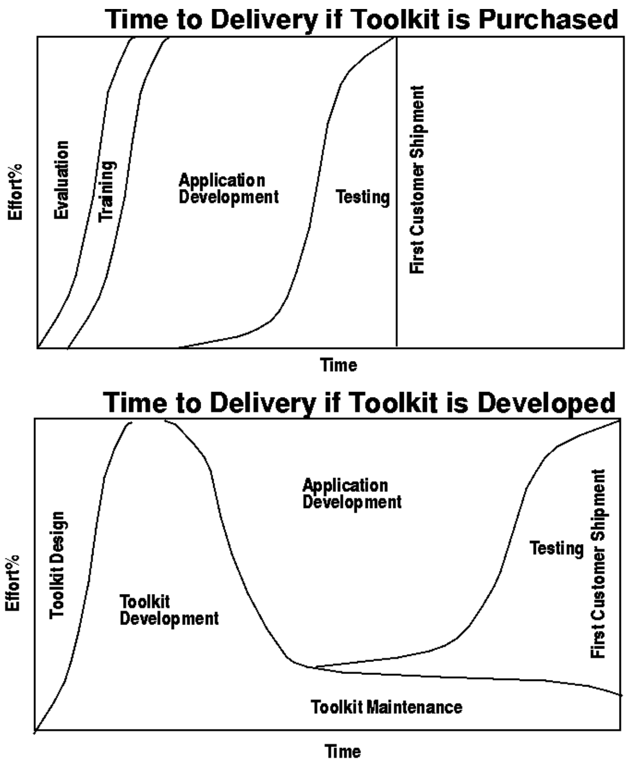

Figure 2