SPE

Society of Petroleum Engineers

SPE 028255

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

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

Abstract

The buzz-phrase of the 90’s in the petroleum 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. For a software developer, either within an oil company or with a software vendor, the concept of “buy don’t build” can apply to software toolkit components and can expedite the development of an application as well as reduce future support requirements.

This paper presents several software tools and the process by which they were evaluated for use in a commercial petroleum engineering application (DeskTop VIP). It highlights the tendency in the software development process to underestimate the complexity of the development process, as well as to underestimate the value of the services provided by a software tool. Ultimately, the decision of “buy don’t build” should be an economic decision. As a slogan, it reminds us that whenever one considers building or developing new software, one should also consider the possibility that buying off-the-shelf software could cost less in the long run and bring a product to completion quicker.

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.

One of the major problems in making sound business decisions with regard to buying versus building software is the difficulty of estimating the cost and duration of the development and maintenance activities. Most organizations and individuals in the software development business can recount endless horror stories of software development projects gone awry. Studies have shown that typical software development estimates are significantly less than what is eventually required for the development and maintenance task.

This paper focuses on the economics of buying versus building software components. It is presented from the viewpoint of a commercial software application vendor concerned with deciding how to obtain the software tools necessary for the application development. Several specific examples of software tools and the process by which they were evaluated for use in a petroleum engineering application (DeskTop VIP) are presented. For many components where off-the-shelf tools provide the required functionality the decision to buy 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 thought and analysis are required.

Much of the process discussed will be applicable whether one is concerned with software components or with complete applications. Although the perspective is presented from that of a software vendor, most of the discussion of software tools should be applicable to an internal development effort as well.

Buy versus Build for Software Components

Today's software applications are typically built from many software components or tools. For many of those components it is taken for granted that they will be bought. Almost no application software developer today would build operating systems, compilers, windowing systems, or base level user interface libraries. Those tools have become industry standards and are built for the mass market making them so inexpensive that no economic analysis is required to determine that buying them is far preferable to building a specialized tool. Other components such as editors, development environments, documentation tools, and license managers are also so inexpensive that building such components would be many times more costly. As components become intertwined with the application and closer to the software development task, the economics become less self evident and the "not-invented-here" syndrome starts to bias the developers towards building instead of buying. This bias can be extremely expensive and very risky in many cases.

References at end of paper

The Economics of Building Software

When evaluating the buy versus build economics for a software project it is important to consider all aspects of the software project from its initial conception to its final obsolescence, commonly referred to as the software life cycle. In the case of a software development project, the generally recognized phases of the software life cycle are:

- Requirements
- Specification
- Design
- Code and Debug
- Testing
- Documentation
- Training
- Operation and Maintenance

One of the major problems with performing an economic analysis of the buy versus build question is the degree of accuracy in the work estimation process for software development tasks. Figure 1 shows an industry survey by DeMarco[1,2] of actual efforts versus estimated effort. From the graph it is clear that estimation has been an inexact science with an overwhelmingly bias towards underestimation. In fact, the data would suggest that projects in general took twice as much effort as was estimated·

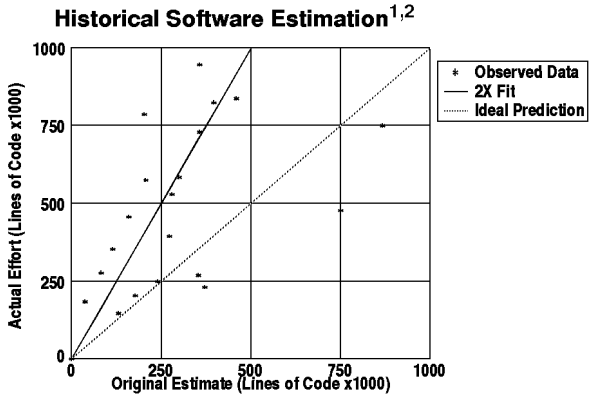

Figure 1

The problem with software development estimation is exacerbated by the ease with which one can neglect critical components of the software life cycle from the overall work estimate. In severe cases where the "not-invented-here" syndrome has taken over, one looks just at the coding phase and grossly underestimates the true cost of the total development effort. Figure 2 from Zelkowitz[3,4] shows the importance of viewing the total picture. Not only is coding a relatively small component of the initial product development, it is totally dwarfed when the huge costs of software maintenance are included.

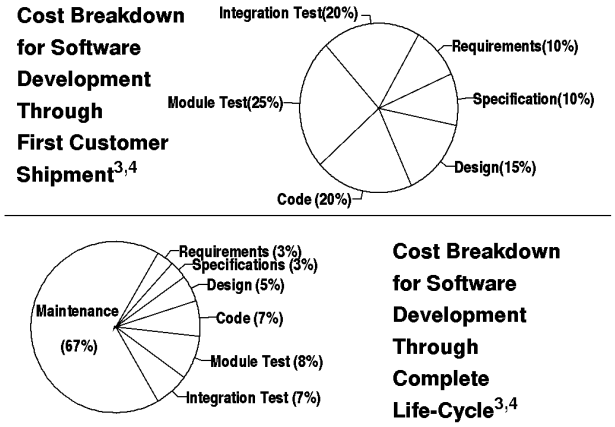

Figure 2

A similar perspective is provided by Brooks[5]. He poses the question that if two people coding in a remodeled garage can make astonishingly useful programs in short order, then why haven't all programming teams been replaced by dedicated garage duos? His answer is that what these duos have generated is a program, complete in itself, ready to be run by the authors on the system on which it was built. In order to be generic, tested, documented and thus become a program product, Brooks estimates it will cost at least three times as much. The original program is also isolated from other products. To become integrated and part of a programming system, Brooks estimates the cost to be at least three times as much as the original effort. For the program to be truly useful, both the productization and the systems integration are required which therefore costs nine times as much as the original program. This relationship is depicted in Figure 3.

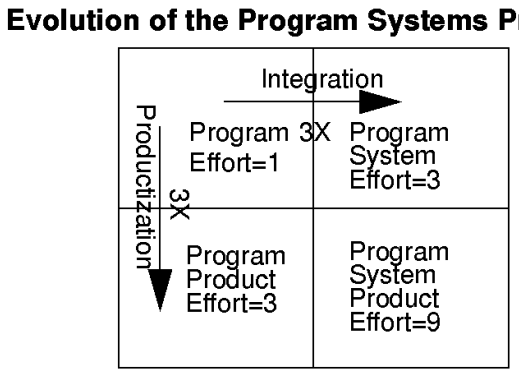

Figure 3

Risks in the estimation procedure are magnified by the increasing complexity of today's software projects. As software programs live out their life cycle they are generally replaced by more complicated and significantly larger

programs. Data from several sources[6,7,8] showing the increase in code size from one generation of software to the next is plotted in Figure 4. There is a strong nearly linear correlation between lines of code in one generation and lines of code in the next generation. A factor of four appears to be a good approximation for relatively large projects while smaller projects may grow even larger.

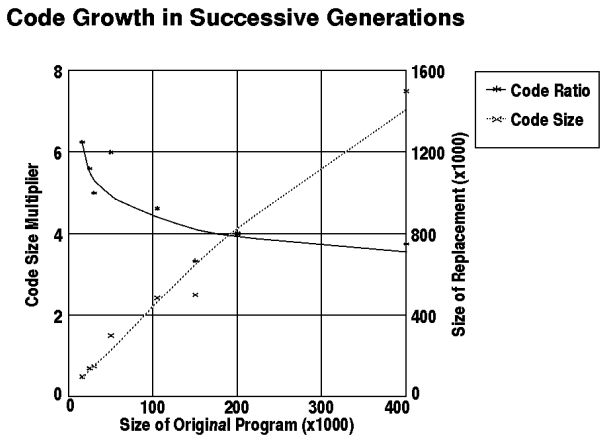

Figure 4

This four-fold increase could be worse than it seems. Studies[5,6,9,10,11] have proposed relationships between programming effort in man-time and the size of the program in lines of code.

$$\text{Effort} = K \cdot size^{b}$$

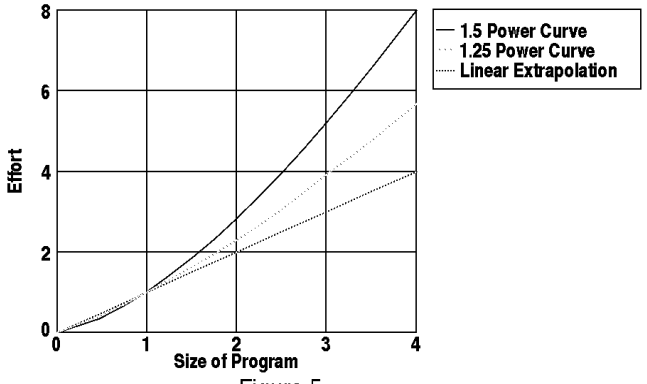

Figure 5

These studies have *b* ranging from anywhere from 1.05 to 1.75. Figure 5 shows the danger of linearly extrapolating the previous development effort to obtain an estimate for the new effort. Since the previous program size was one fourth of the expected new size, linear interpolation would give four times more effort. Assuming a value of *b*=1.5, however, suggests it would more likely require eight times the effort, an underestimation by a factor of 2.

The inherent risk of underestimation should be factored into the total direct development costs. Ultimately, the high cost of software development provides a significant incentive to look for alternatives.

The Economics of Buying Software

Purchasing off-the-shelf software components can provide several economic benefits. If the requirements are met by off-the-shelf packages, typically the acquisition cost is low as the cost of development is shared by many customers. Risk is reduced since much of the inaccuracy inherent in the estimation of new development effort is replaced by bounded expenses.

Just as with software development, acquired software components have a life-cycle and associated costs at each stage. From the point of view of the purchaser these phases are:

- Requirements
- Specification
- Evaluation
- Acquisition
- Integration
- Enhancement
- Testing
- Training
- Operation and Maintenance

The requirements and specification phases are generally independent of whether one is buying or building. Certainly once one starts into the evaluation phase and determines that one of the requirements cannot be met, it is a good reality check to determine if this requirement is really necessary. Often software development tasks are driven by the 80/20 rule: 20% of the product ends up costing 80% of the total effort. Thus if one can live with 80% of the product it can be obtained for 20% of the cost. The reality check is do you really need the last 20% and is it really worth five times more? This is especially true when buying software, since the acquisition cost is typically only a fraction of the total cost invested in the product by the vendor.

The differences in the buy versus build software life cycles are accompanied by differences in the cost equations. The cost of the evaluation phase includes the man-time for the evaluation plus any out of pocket expenses required to complete the evaluation. It is the acquisition phase that is the most commonly associated with buying software since it is the point where license fees are paid and where the commitment to a vendor relationship is made.

During the integration, enhancement, and testing phases the software is fit into the buyer's environment to accommodate their specific needs. Costs can be incurred as in-house man-

time or can be contracted from the vendor. Typically a buyer is asking for future problems if they undertake any enhancements themselves, unless the vendor is willing to incorporate those enhancements into future releases. In fact, it is often beneficial to insulate the applications from direct utilization of the vendor toolkit in order to reduce dependency on the vendor and to provide future migration paths. Once the software is in place, usability of the software will require training. Training courses can be very expensive to prepare and present. Fortunately, most vendors will be able to contract to provide training thus sharing the training costs among several customers. After the applications have been developed, redistribution costs may be incurred once they are deployed. Lastly, while the applications are undergoing their life cycles, the software components are also being enhanced and maintained incurring maintenance costs.

Hidden Costs

Beyond the direct costs of buying or building, there are several hidden costs to be considered. On the build side, perhaps the most significant hidden cost is the opportunity loss as a result of a delay in the time-to-market for the application. A significant time period may be required to build a component delaying the deliverability of the end product as a consequence. When an off-the-shelf component is ready to be used immediately or with minimal enhancements, this can be an important differentiation. Other hidden costs result if the development activity is not in the developer's core business area as expertise and specialization are focused towards a non-core area. This specialization often requires training to support the development activity. Since the expertise is often not transferrable to the company's core business area, the career paths of the specialized developers are limited. Not surprisingly, this perpetuates the "not-invented-here" syndrome as developers begin to defend their turf.

Buying software also brings some hidden costs. The most common hidden cost associated with buying software is the cost of working through bugs in the vendor product. Bugs and poor software reliability can be costly to the user. However, this cost should not be overestimated since in reality it exists for internally developed software as well. The good news when buying is that the vendor has the responsibility to fix the bugs. The bad news is that they may not consider it to be anywhere near as critical as the buyer knows it to be. This is just one of the reasons that a good relationship with the vendor is very important. Other hidden costs of buying can come from buying into a generic off-the-shelf package. Often the necessity to have a generic tool ends up creating an environment which creates incompatibility with other components or brings hidden baggage with it. Portability, optimization, and hardware requirements are also potential costs to be considered.

Hidden Benefits

In addition to the costs, it is also useful to consider some of the additional hidden benefits that can be reaped by buying software. An important consideration is that generally the purchased software is in the vendor's core business area and therefore the vendor has developed a specialization in that area. As a result the product has been designed by experts and the vendor has improved the product to meet many objectives by responding to customer feedback. Generally, the vendor has completed both the productization and integration processes. This greatly increases the probability that the component can be reused in other projects. It also improves that chance that the product will be able to meet future unforeseen needs. The productization step should not be underestimated. Often, just the value of the documentation justifies the cost of the acquisition.

Some Specific Case Studies:

C++ Base Class Tools: Buy Generic Tools

When we started C++ development in 1990 our group acquired generic public domain C++ classes. These base classes provided facilities for doing lists, arrays, and strings. One member of our group had been active in the development of the tools during his university days and had used them substantially. As a result, there was no base cost in building the software and there was only a minimal learning curve required to use the classes. Since only a limited number of developers were utilizing the tools it was decided that real productization was not necessary.

We recently we went back and reevaluated 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 the library was far richer in functionality, designed much better, and was fully documented. Since it was a mass marketed toolkit, the acquisition cost was almost insignificant. Just documenting our existing toolkit would far exceed the acquisition cost. The fact that the Rogue Wave tools are becoming a de facto standard within the industry provides a base from which to derive specialized classes which can be shared between different development efforts.

One hidden cost that was considered when looking at the Rogue Wave tools was the cost of migrating legacy applications to the new tools. The current applications are based on the old C++ toolkit classes which are not compatible with the Rogue Wave classes. It was determined that this will require minimal migration of the applications to take place over time. The migration is minimal because the migration can be done in stages through the use of the Rogue Wave templates which allows our old applications to incorporate the new tools without having to remove the old class hierarchy. All new development can be based entirely on the Rogue Wave tools.

The economic rational favoring the buying of base tools is typically overwhelming. Mass marketability and standardization of the base tools has driven acquisition and total costs down to the point that it would be impossible to cost effectively justify building such tools.

3D Graphics: Buy and Build if Necessary

Also in 1990 we began to identify our needs in the 3D graphics area. We had previously developed a prototype 3D application utilizing Silicon Graphics GL graphics library. This prototype was being well received in the marketplace although we were constantly being asked questions about portability and hardcopy. Both of these areas were weaknesses with the GL API. As a result we began to search the marketplace for an alternative 3D graphics library.

Our specification for 3D applications required several fundamental features:

- Fast interactive display.
- Software rendering to X-terminals.
- Scalable hardcopy to CGM and PostScript.
- Portability to common UNIX platforms.
- Interoperability with the X window system.
- Compatibility with the Motif user interface.
- Efficient memory utilization

We were unwilling to suffer a significant performance degradation relative to our existing GL prototype. Ideally, we wanted a clean, standard graphical interface which would stay prominent within the marketplace.

The alternatives considered were GL, PHIGS, HOOPS or an environment like AVS. Although GL provided the fast interactive display and was compatible with the X window system, it had limited portability and had no hardcopy or software X terminal support.

A commercial PHIGS implementation marketed by Liant called Figaro was first looked at. The retained mode nature of PHIGS did not appear to be well suited for our dynamic applications. Furthermore, it was not well integrated with the X window system and Motif and at the time Figaro had no true 3D hardcopy support.

HOOPS from Ithaca Software was a retained mode library similar to PHIGS that was the next tool evaluated. The major differentiation between HOOPS and Figaro was that HOOPS supported hardcopy to PostScript and CGM. It also had software rendering to an X terminal and was compatible with the X window system and Motif. Again, however, the retained mode nature of HOOPS severely limited the interactive performance and greatly increased the memory utilization requirements.

Environments like AVS, Info Explorer and Advanced Data Explorer were also considered. Although they were good prototyping tools, none of them supported scalable hardcopy and all of them used excessive amounts of resources.

None of the alternatives met our requirements. This forced us to do one of two things: reconsider our requirements or consider building. Although we were willing to reconsider our requirements, none of the products met even a minimal subset of our requirements. On the other hand, creating a fast, portable, clean interface which could render to high speed interactive displays and produce scalable hardcopy is a large undertaking. The development would require continual investment of resources due to the evolving nature of the graphical industry. Timing also presented a critical problem. By the time the graphical layer would have been developed there was a real danger that the window of market opportunity for the graphical applications would be lost. Alternatively, co-developing the tools and applications would have also introduced stability problems for the developers. As the graphical layer was updated the applications would need to be continually reworked to reflect changes in the tools. Development of the applications would be adversely impacted when key components of the graphical layer did not exist or were only in a protoype stage. Costs overruns and delays were a serious risk in this scenario making it difficult to manage the costs and deliver the applications to the market.

With no single solution available it was decided to approach the problem differently. Instead of looking for a single solution we decided to break the graphical interface into three separate layers, low level drivers, a middle level graphical interface and high level graphical applications. Since it was not feasible to create the low level graphical drivers GL was selected for interactive display and HOOPS was selected for hardcopy. By combining the strengths of the two low level graphical libraries the functionality required for our 3D graphical applications could be realized. The applications would be shielded from the differences by the middle level interface. An objected oriented approach[12] was used to build this layer as depicted in Figure 6. Objects provided a

convenient way to create an implementation that could handle immediate mode graphics (i.e. GL) and a display list system (i.e. HOOPS) transparently. Instead of building a complete graphical system only an interface was built. This hybrid approach of buying and building minimized the costs associated with building while providing a flexible system to work around the hidden inflexibility costs associated with the off-the-shelf software.

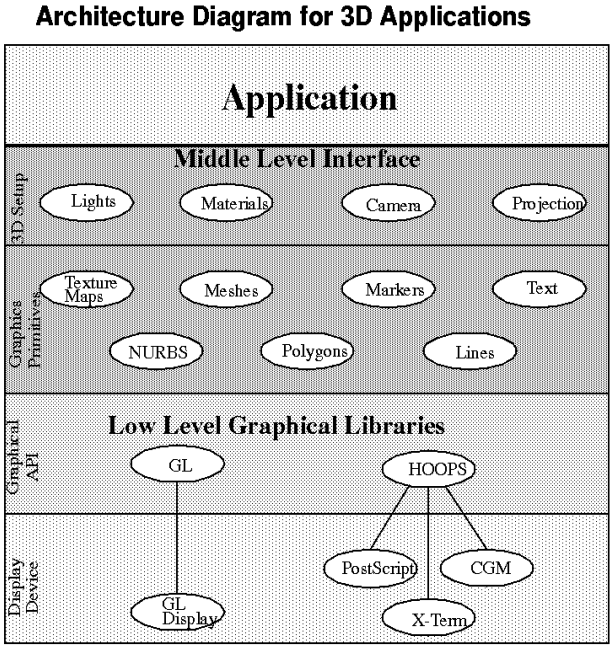

Figure 6

XY Plotting Tools: Analyze, then Buy

In 1993 we began to formulate our requirements for an XY plotting tool in order to develop a successor to a legacy reservoir simulation plotting application[13]. Market trends dictated that the XY plotting tool had to be very tightly coupled with the X window system and with Motif. Again, both PostScript and CGM hardcopy would need to be supported. Additionally, customer comments about the legacy plotting application indicated that the tool should be able to support multiple vertical axes. Since we now had a successful 3D application, it was also important that the plotting package integrate with it.

A logical starting point for our evaluation was to consider building the plotting toolkit on top of the existing 3D tools library. In fact, a functional prototype application was built using the 3D tools. This prototype lacked the polished look that we desired and we estimated that at least 6 months would be required to develop a minimal plotting tool. Our estimate was for just a minimal tool and did not include to effort to make it general or to productize it. There was enough incentive for us to search the market to investigate purchasing a plotting tool.

XRT/Graph was a mature product that was in wide industry use. It offered some useful convenience features such as an automatic legend, a built in time axis, and a single X resource setting to display the same plot as an area graph, XY plot, bar chart, pie chart, etc. However, our analysis of the features suggested that this was a product whose target market was primarily business applications and as such it was deemed less suitable for our purposes. For example, the time axis was restricted to dates between 1970 and 2038 which is not a reasonable constraint when dealing with reservoir production data. Also, the XRT/Graph product was limited to two vertical axes, one on either side, and it did not provide CGM output capabilities. The downside of XRT/Graph's maturity and wide customer base was that no influence could be exerted towards the enhancement of the product. In fact, a good example of the 80/20 principle was exhibited when we inquired into the possibility of adding multiple Y axes and CGM to the tool. It turned out that the cost to add these features would be about 5 times the cost to acquire the base software.

In comparison to XRT/Graph, the PlotXY widget by INT was a younger, less developed product. It was primarily targeted to

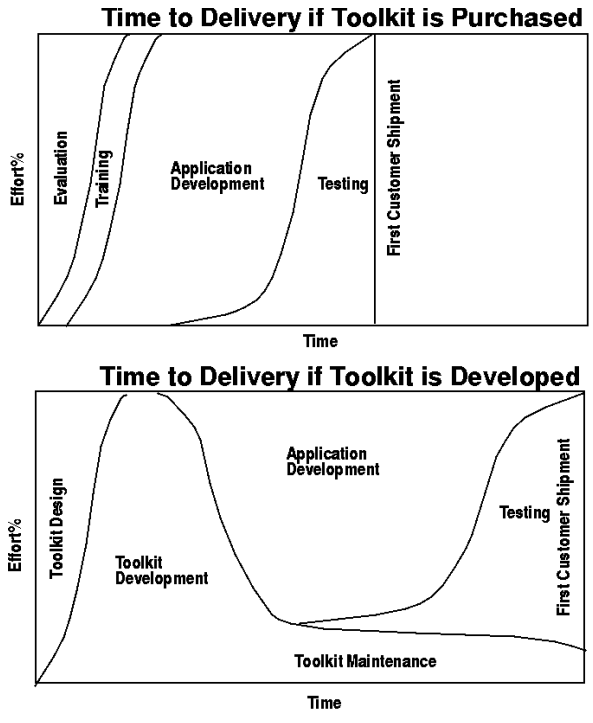

Figure 7

the needs of the petroleum computer industry. Not being already entrenched in the marketplace with this product, INT was willing during initial testing and evaluation to incorporate some of our suggestions, such as rotating label and annotation on vertical axes, and automatic legend generation. In addition, PlotXY provided multiple axes, and CGM and PostScript.

Upon completing the technical evaluation, we continued with the economic analysis. Figure 7 compares the timelines of buying versus building. We had a fixed set of resources for the project so the total effort was essentially constant. In the case of buying, there would be start-up time associated with training and familiarization with the tool. However, once the tool was basically understood, the development of the application could proceed. This significantly shortened the time-to-market. The alternative required the design and development of a toolkit before the application development could get underway. Furthermore, continued maintenance of the toolkit would rob valuable developer time from the application development task.

From a cost standpoint, the acquisition cost was far less than the expected burdened development costs for the toolkit. Figure 8 compares the total costs of the application development for the two alternatives up through the first customer shipment. As is clear, the option to buy the toolkit was predicted to bring a product to market much quicker and at a lower total cost. Furthermore, the risks associated with the development estimation and probability of future ongoing maintenance costs greatly favored the decision to buy.

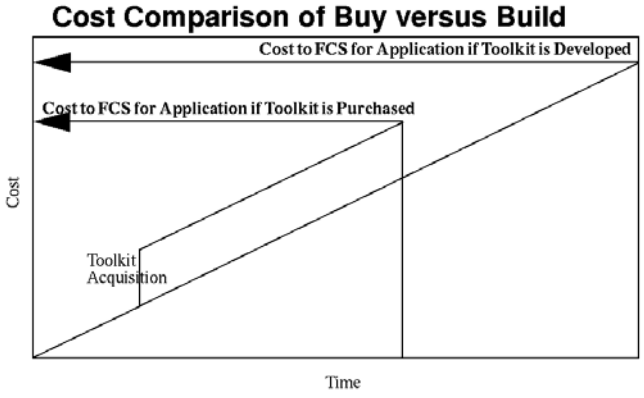

Figure 8

The decision of which tool to buy was governed largely by the technical evaluation and the degree to which we felt our future needs could be best met. From a direct cost perspective, both products were comparable, although the additional costs for enhancements to XRT/Graph made it more expensive. The maturity and reliability of XRT/Graph was a significant advantage, as we anticipated costs due to bugs in the immature INT product. In our case, however, this immaturity meant that we had more input into the direction of the product and thus had access to a more flexible tool. It was this flexibility and our concern for the integration with our 3D application that guided us towards INT. Integration costs would have been less had we chosen to build, however, the good vendor relationship that we had formed with INT allowed us to conclude that they could provide the flexibility that would meet our future integration needs. Even with the incurred costs from bugs and integration, there was a significant projected overall cost savings compared to the alternative of building a toolkit.

Conclusion

As a general slogan, “Buy don’t Build” is very useful to remind us that building software is often much more expensive than buying off-the-shelf software. This can be true for both complete software applications as well as for software components. Beyond the slogan, the real decision to buy or build is an economic decision. Our experiences with evaluating software components have shown that while there are many times where buying software is the best choice, there are other times where it is appropriate to build the software and others where it makes sense to both buy and build.

The major driving forces in the buy versus build economics come from increasingly complex and expensive 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.

In addition to the direct costs of building or buying software, it is also worthwhile to examine other hidden costs and benefits. One major hidden cost of building software is the opportunity loss due to the delay in time-to-market. Other hidden costs can result from specialization in non-core business areas. On the buy side, hidden costs to consider are associated with software reliability, interoperability, portability, and resource utilization. Often, there are potential hidden benefits of buying into the vendor’s better expertise and specialization. This knowledge base can translate into a better design and implementation of the software component and can thus lead to a more generic and functional tool, encouraging reuse in other future products. Another key benefit to buying comes from the vendor’s productization and documentation. These benefits alone can make buying beneficial in the short term, even if building is the long term solution.

Our direct experiences, showed that the economic rational favoring buying base class tools was overwhelming. This was a case where the benefits of the buy option were significant

and the documentation by itself supported the acquisition 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 begun developing our own plotting tools. In this case "Buy don't Build" encouraged us to analyze the situation to determine the best alternative. Our prediction 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.

While our perspective is that of a software vendor, our experiences should be applicable to internal software developers as well. Differences in needs will result in slightly differing costs, but overall the software life cycle is very similar whether the point of view is from a vendor or from an internal developer.

Although we have focused our attention on the buy versus build question for software components, much of this process should be applicable to buy versus build comparisons for complete applications. The overall conclusion in today's environment is that building software can be expensive. Sometimes building is necessary in order to obtain a product which meets core requirements. If an off-the-shelf package can meet most of the requirements, then it will generally be significantly more cost effective.

References

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

2. DeMarco, T., Controlling Software Projects. Yourdon Press, Inc., New York, NY, 1982.

3. Zelkowitz, M. V., "Perspectives on Software Engineering," *ACM Computing Surveys*, June 1978.

4. Frank, W. L., Critical Issues in Software, John Wiley & Sons, Inc., New York, NY, 1983.

5. Brooks, F. P. Jr., The Mythical Man-Month, Addison-Wesley Publishing Company, Inc., Reading, MA, 1975.

6. Putnam, L. H. and W. Myers, Measures for Excellence, Yourdon Press Inc., Englewood Cliffs, NJ, 1992.

7. Stubberud, A. R, "A Hard Look at Software," *IEEE Control Systems Magazine*, Feb. 1985.

8. Stephenson, P. E. and S. Bette, "Open Architecture Computing Environment and Its Applications in Reservoir Simulation," Paper SPE 24285, presented at the 1992 European Petroleum Computer Conference, Stavanger, Norway, May 24-27.

9. Boehm, B., Software Engineering Economics, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1981.

10. Nanus, B., and L. Farr, "Some Cost Contributors to Large-scale Programs," *AFIPS Proc. SJCC, Volume 25 (Spring, 1964).*

11. Weinwurm, G. F., "Research in the Management of Computer Programming," *Report SP-2059*, System Development Corp., Santa Monica, 1965.

12. Sinclair, C., T. Little, and M. A. Rahi, "An Object Oriented Solution to an Interdisciplinary 3D Visualization Tool,", Paper SPE 27545 presented at the 1994 European Petroleum Computer Conference, Aberdeen, UK, March 15-17.

13. Little, T., D. Chien, R. Corbell, M. A. Rahi, and C. Sinclair, "Migrating Legacy Petroleum Engineering Applications to Open Systems Environments,", Paper SPE 27560 presented at the 1994 European Petroleum Computer Conference, Aberdeen, UK, March 15-17.