

Paper for the 16th Annual Midwest Computer Conference:

The submitted manuscript has been created by the University of Chicago as Operator of Argonne National Laboratory ("Argonne") under Contract No. W-31-109-ENG-38 with the U.S. Department of Energy. The U.S. Government retains for itself, and others acting on its behalf, a paid-up, nonexclusive, irrevocable worldwide license in said article to reproduce, prepare derivative works, distribute copies to the public, and perform publicly and display publicly, by or on behalf of the Government.

Need for Multiple Approaches in Collaborative Software Development

David J. LePoire

**DePaul University / Argonne National Laboratory
ANL 900:C-30, 9700 S. Cass Ave, Argonne IL, 60439
dlepoire@anl.gov / 630-252-5566**

Paper for the 16th Annual Midwest Computer Conference:

Need for Multiple Approaches in Collaborative Software Development*

David J. LePoire

**DePaul University / Argonne National Laboratory
ANL 900:C-30, 9700 S. Cass Ave, Argonne IL, 60439
dlepoire@anl.gov / 630-252-5566**

Abstract

The need to share software and reintegrate it into new applications presents a difficult but important challenge. Component-based development as an approach to this problem is receiving much attention in professional journals and academic curricula. However, there are many other approaches to collaborative software development that might be more appropriate. This paper reviews a few of these approaches and discusses criteria for the conditions and contexts in which these alternative approaches might be more appropriate. This paper complements the discussion of context-based development team organizations and processes. Examples from a small development team that interacts with a larger professional community are analyzed.

Problem

The need to share software and reintegrate it into new applications is a difficult but important problem. The component-based model for collaborative software development has serious limitations that depend on the context in which it is developed and used. Component-based development is useful only if the components are reused. To be reused, the components must be created at the correct granular level, be flexible enough for customization, not be flexible that they require an inordinate number of specifications or implement a resource-wasting general algorithm, and be communicated and accepted by a community of developers. As software development proceeds, the decision must be made to 1) look for already developed (internal or third-party) components, 2) develop the customized code from scratch, or 3) develop a more general component that will satisfy both the current project and potential future needs.

There is no single solution. The decision strongly depends on the context of the project, the development teams, and their collaborative environment. The effort involved in the first option includes research to find, evaluate, and estimate cost. Sometimes the component is very rich and has a language of its own. If so, the development team requires a learning curve to integrate the component. If the component is complex, the compatibility of the component with the project might not be known until much research and prototyping are done. If the team will likely reuse the component, it might make sense to acquire and document the knowledge. The effort for custom development does not include the learning period but does include more extensive development time,

without recovering this effort later in reuse. The effort for the general component development includes both the development effort required to satisfy the project requirements plus the effort necessary to generalize, document, and communicate the component.

User interface components were the first components to be widely used and accepted by a large community. Third-party vendors for Visual Basic led the development after the innovative RAD interface development tool was released in the early 1990s. These components can be relatively simple to use and create (e.g., specialized text boxes) or very complex (e.g., graphics, report, and interactive mapping components). However, the user interface components usually solve some set of common problems faced by software developers and therefore have a wide market.

Components for modeling and web services are a bit more difficult to use and generate because their functionality is not as defined and general. For example, in modeling situations, there are many levels of assumptions and granularity levels. The application of a general modeling component in an application that requires only a limited set of features might generate unacceptable development overhead and also run-time behavior that is not very efficient.

Approach

In a series of recent Association for Computing Machinery (ACM) *Communications of the ACM* editorials in the “Business of Software” column, Phillip Armour (2000, 2001a, 2001b) has called for a multiplicity of approaches to be used by software development teams. These approaches correspond to the level of knowledge that the team has. Based on his “Levels of Ignorance” model, these teams are:

- Tactical teams: Everything is known (e.g., configuration management tasks).
- Problem-solving teams: The question is known, but the solution is not (e.g., software debugging).
- Creative teams: The questions are not known, but the process to derive the questions is known.
- Learning teams: No process to derive the questions is known (i.e., the team “does not have a suitably efficient way to find out they don’t know that they don’t know something”).

He asserts that most software development projects are usually a mix of these teams for different aspects of the project at different times. Processes should be developed for moving from one level of ignorance to the next by capturing the mundane tasks. For example, learning teams attempt to construct some common models and language to develop a process to find the questions. Creative teams take this process and apply it to the specific application to derive the appropriate questions. Problem-solving teams can develop processes for handling a set of questions, and finally tactical teams can implement the specific solutions.

The multiple levels of processes are designed to store knowledge at the appropriate level without impinging on creativity. The above analysis assumes that the project teams are somewhat isolated when they work. In other situations, a loose confederation of software teams might be collaborating (or competing) to develop software tools to facilitate discrete functionality and further integration. In these situations, it becomes more difficult to develop standardization and therefore know the questions. Without knowing the questions, a greater flexibility is needed to address changing applications and functions.

Other approaches for collaborative development include open sourcing of integrating templates, wrapping legacy software packages with XML and database interfaces, and constructing visual programming platforms. These approaches will be reviewed within the context of the following set of criteria:

- Flexibility
- Software dissemination
- Support for collaboration
- Ability to support legacy software
- Development costs
- Quality assurance (QA)
- Maintenance issues (life-cycle development).

The open source approach does not necessarily mean that the source for a set of methods is made available so others can contribute to the next version. It can also mean that a template or example applications are developed with a set of components being integrated together with scripting. These templates solve part of the problem of the software sharing approach when it only involves components. The templates shorten the learning curve by allowing the developers to search and slightly modify the templates that most closely match their application without fully understanding the details of the component methods and properties.

Wrapping legacy codes and adding the ability to communicate with them through standard databases or XML enables quick exploration of potential uses while technology flexibility is maintained. Existing engineering models have traditionally been developed by using legacy languages such as FORTRAN. While they might be structured, they typically are not object-oriented nor database driven, but instead are tightly integrated with flat file data stores. This has led to a plethora of software that seems to solve very similar problems. Organizations have begun to question this approach, and alternatives have been explored to interconnect legacy models while their performance, user's satisfaction, and most importantly their QA, are maintained. The QA for many of these codes is derived for the length of the code use and the variety of cases to which end-users have applied them.

Some approaches for handling the wrapping of these software packages include parsing out the separable modules and reintegrating them into a generic object framework. Another compatible approach is to construct a metadatabase of the models' needs along

with options for controlling the model options and assumptions. This last method leads to a very generic wrapper that then can be easily specialized into objects or components for different modeling needs. The metadatabase maintains the engineering models' connection between the computation model and data. It also allows for a facilitated construction of the user interface and ensuring consistency in data communication among the integrated models.

Yet another technique for sharing software is by constructing a visual programming interface that allows a predefined and limited connection of components. These techniques have visual appeal for simplicity but usually eventually lead to cumbersome limitations due to the assumptions that were integrated into the visual programming platform. A tool from Sun for Java (JavaStudio) was highly hyped but dropped within about a year due to lack of interest.

Results

These approaches were explored through concrete examples and context-specific evaluations. These examples and discussions led to some general criteria to consider when considering an appropriate approach to collaborative software development that could be used within an information systems or software engineering curriculum or development team guidelines (Table 1).

In the environmental field, modeling plays a critical role in connecting current data and knowledge with predictions of future events and environmental states. Environmental problems are quite challenging to solve because of the complex relationships among many contributing factors, both natural and man-made (Constanza et al. 1993). Moreover, these problems need to be addressed not only by environmental engineers and regulators but also by concerned members of the public and nongovernmental organizations. Their demands on environmental modeling often conflict because predictions need to be accurate yet easily understood, communicated, and explored. The increasing complexity of environmental codes also places a demand on the end user, who must translate the real environmental problem into the conceptualization allowed by the model and its options. Information on assumptions and options must be conveyed to the user to ensure that the model is applied and interpreted correctly. Open communications about the model, interface, and data components would enable software applications to be more easily developed.

In a visual integration framework, development and testing are divided into two levels: (1) development of modules and (2) end-user integration and implementation through a single visual programming framework. This framework works as long as it is flexible enough to meet various needs. However, it is very difficult to leverage new technology within the framework, since the user-interface, data manipulation, and modeling connections are already specified and implemented. This system can facilitate the exploration of a specific environmental problem by a single end user but can cause difficulties for a user community that is trying to follow a regulatory process. Also, the

burdens of model integration and application are on the end user. (Frames 1.1 and GoldSim [Whelan et al. 1997; Whelan and Nicholson 2001] are examples of this type of system.)

Quite a large set of tools is being developed to further separate the roles of modelers and integrators and the four components (data, models, interface, and connection). Some model integration tools include the Argonne National Laboratory (ANL) DIAS system (Sydelko et al. 1999) and the U.S. Environmental Protection Agency (EPA) MIMS system. These tools offer a system of utilities for model integration and data communication. The DIAS tool is based on the concept of using models to provide methods for a higher-level conceptualization of an object. This allows both new development and the wrapping of existing models. However, there are many other ways to accomplish this wrapping and object integration with commercial tools (J2EE, ColdFusion [Forta 1998], Microsoft [MS] .NET [Hollis and Lhotka 2001]) that might not supply the same utility support but allow a flexible integration with commercial components.

One commercial system that seems to have a good approach to using templates as an integrator of components is the Environmental Systems Research Institute ArcIMS system (ESRI 2001), an Internet-based system for supplying geographic information system (GIS) maps and data. The main map-rendering application is deployed on a server. The developer works with this service and is supplied with a default set of tools to develop a user interface for the manipulation and display of the maps. The interface components are object-oriented but written in a client scripting language (JavaScript). This allows the component provider (ESRI) to provide a flexible template to the integrator to customize the user interface for the end user. The ArcIMS services are based on the ESRI MapObjects; however, the package also supports connectors to these components via a series of techniques (e.g., servlets, ASP, ColdFusion) and generates template applications in Javascripting of the components. The Javascript is organized so that the end user can quickly explore the template and customize the application without a deep understanding of the component model. This ability to see the source code that integrates the components has led to rapid application development with this set of tools without a large training investment.

To demonstrate the leveraging of existing codes through wrapping and communication through XML and databases, an environmental code for determining environmental risks was wrapped, used for a slightly different purpose, and connected to a new user interface that included GIS and visualization components (LePoire et al. 2001). Furthermore, the results can be made accessible on the Internet through a simple web browser interface, giving users easy access to the model, data, and visualizations. This technique allowed a quicker response and more flexibility than the traditional component development approach, because all the inputs and outputs of the model were available through a standard database connection. The wrapping was performed on the FORTRAN code and placed into a Visual Basic object as a DLL. Connections between the Visual Basic object and the other applications were made through ColdFusion scripting.

Variations of this technique were used to connect two existing environmental risk assessment codes. Data dissemination through an XML web service into codes was also demonstrated.

Discussion

Four different approaches to collaborative software development have been explored, including the component-based model. The model for matching a software process to a software development team's needs was used as a template for making a similar matching between the collaborative model and the team.

This work resulted in identifying two dimensions of the software: flexibility and integration level. Flexibility relates to the amount of standardization that can be supported in the application domain. If the domain is not mature, innovation and prototyping can be facilitated with templates and customizable scripting for final integration. Lower-level functionality can be wrapped with the latest technology, supported with database or XML connections. If the domain is more mature, with standards, then the integration can be accomplished with a visual programming interface. The lower-level functions can be captured in components. However, this more stable technique is not only affected by domain maturity but also by technology maturity.

A review of uses of these techniques in the domain of environmental risk analysis has shown the advantages and disadvantages of the techniques. Experience has demonstrated the need to try to match the collaborative software development done by disparate organizations in this maturing domain.

References

Armour, P.G., 2000, "The Five Orders of Ignorance," *Communications of the ACM*, Vol. 43, No. 10, October.

Armour, P.G., 2001(a), "The Laws of Software Process," *Communications of the ACM*, Vol. 44, No. 1, January.

Armour, P.G., 2001(b), "Matching Process to Types of Teams," *Communications of the ACM*, Vol. 44, No. 7, July.

Constanza, R., et al., 1993, "Modeling Complex Ecological Economic Systems," *BioScience* 43, Sept.

ESRI, 2001, *ArcIMS* 3, Redlands, Calif., <http://www.esri.com/software/arcims/>, accessed Sept. 2001.

Forta, B., 1998, *The ColdFusion Web Application Construction Kit*, Que, Indianapolis, Ind.

Hollis, B., and R. Lhotka, 2001, *VB.NET Programming with the Public Beta*, Wrox Press, Birmingham, U.K., Feb.

LePoire, D.J., et al., 2001, *OpenLink: A Flexible Integration System for Environmental Risk Analysis and Management*,” ANL/EAD/TM-114, Argonne National Laboratory, Argonne, Ill., Oct.

Sydelko, P.J., et al., 1999, “A Dynamic Object-Oriented Architecture Approach to Ecosystem Modeling and Simulation,” *Proceedings of the 1999 American Society of Photogrammetry and Remote Sensing (ASPRS) Annual Conference*, Portland, Ore., May 19-21.

Whelan, G., et al., 1997, *Concepts of a Framework for Risk Analysis in Multimedia Environmental Systems*, report by Pacific Northwest National Laboratory, Richland, Wash., Oct.

Whelan, G., and T. Nicholson (editors), 2001, *Proceedings of the Environmental Software Systems Compatibility and Linkage Workshop*, Draft, Pacific Northwest National Laboratory, Richland, Wash., June.

TABLE 1 Criteria for Collaborative Software Approaches

Criteria	Component-Based	Database, XML, and Wrapping	Templates and Scripting	Visual Programming Framework
Description	Functionality is built from the bottom up. Although the component can support multiple interfaces, the granularity requirements can not be easily changed.	Development and testing are divided into three levels: modules, integration, and end use. This approach allows module reuse and swapping and provides the ability to develop flexible end-user interfaces and data management.	Templates are developed for a set of basic functions. These examples can be customized by using standard scripting languages.	Development and testing are divided into two levels: (1) modules and (2) end-user integration and implementation through a single visual programming framework.
Maintainability	Customization must be done to ensure proper communication between components. While the components and integrations are maintained separately, the integrator is dependent on the component developer to supply appropriate interfaces.	Modules are maintained by the developers. Standards are agreed upon and followed in module and data specification. Integration can be done in a number of ways, depending on the requirements.	This is easy to script but difficult to document since a function might be implemented throughout many files.	The framework must have one specified standard. All codes must go through the standard to be incorporated.
Dissemination	Modules can be distributed with the integrated application. Later modules can be maintained on distributed servers.		The templates can be distributed, but the customization is difficult to share.	Framework and modules are installed separately.
Validation and verification (V&V) and QA	Each module maintains its own V&V. Applications are connected to the modules by integrators who ensure assumptions are appropriately compatible for the application V&V.		Templates can be subject to QA. The customization is more difficult to test.	Modules can be V&V'd, but V&V of the integration process is up to the end user.
Flexibility	Components can be interchanged if their interfaces are compatible. However, specializing components might result in large performance losses.	Modules can be added, substituted, and modified with flexible connections to other modules. This practice allows for flexibility in both the module level and the integration level.	Great flexibility is achieved because the source code is available and is functional without modification.	Modules can be added as long as they fit the framework's fixed structure. Modules cannot be flexibly integrated for other potential integrating frameworks.
Use of legacy software	Components could be developed from wrapped software; however, it is difficult to divide functionality of existing codes without access to the source code.	Legacy code can be "wrapped" for use with other codes. Some functions can be called separately. A modularized version of the model would be more flexible.	Scripting can be used to connect components or wrapped legacy codes.	It is difficult to incorporate legacy code without a large effort to modularize it to conform to the framework's fixed structure.

Support for cooperation

Development is very efficient if the scope of functions and integration applications are well known.

Modules and data can be shared for different applications. Different applications can be constructed with the shared modules to accommodate the different requirements of the agencies.

Templates can be a good way to share code in a rapidly changing environment.

All agencies can develop their own modules, but they must conform to the framework structure. It may be difficult to construct one structure to satisfy the needs of all agencies and organizations.

Development costs

Development is efficient when the components will be reused in similar situations. This occurs when the problems are well defined.

Modularization and structural flexibility lead to efficient reuse and development of modules while maintaining an efficient user interface.

Development costs are small at first, but the maintenance and dissemination of customized templates lead to inefficiencies.

Modularization leads to more efficiency, but effort can be expended on conforming the modules to a structure that is not efficient and effective.