Case Study: An Investigation into Factors Affecting Component Selection Decisions In Component Based Software Development

Wes J. Lloyd
Computer Science, Colorado State University
Fort Collins, Colorado, USA 80521
wlloyd@acm.org

Abstract: Component based software development (CBSD) has recently been considered to be the next major software development paradigm that promises significant software quality improvements and productivity gains. In CBSD, software developers focus primarily on the assembly of preexisting software components in order to realize the required functionality of the system under development. One of many challenges associated with component-based development is the process of selecting the best component to realize required functionality. In order to make better component selection decisions, software engineers could benefit from the application of software engineering measurements to quantitatively measure and assess which component(s) are best for a given development scenario. This case study proposes a hypothetical development scenario and then proceeds to evaluate several components that could be used to meet the functional requirements. Quantitative measurements of size and performance are used to assess how well the components perform. How the attributes of complexity, understandability, and ease of integration can be measured is the topic of this research.
Key Words: CBSD, CBSE, component selection, understandability, component complexity

1. Introduction

Component based software development (CBSD) has been termed as a new silver bullet for software engineering. CBSD allows software developers to build software systems by selecting and integrating pre-existing software components. These software components encapsulate specific functions through a set of interfaces in a black box like fashion. Available components for integration can come from a variety of sources, from in-house built component repositories consisting of domain-specific components, to commercially available components developed by third party software vendors. By using preexisting software components a large percentage of the functional requirements for a software system can be implemented more rapidly because the functionality is already built into the existing components. Component based software development focuses on the process of assembling together components to implement the majority of the functional requirements for the system. CBSD promises to deliver higher quality software systems in less time because a good percentage of the code required to implement the system's functionality is assumed to be already complete and tested. In making this promise the assumption is made that preexisting software components will have already been thoroughly tested and will be of higher quality than any software that could be built from the ground up for implementing a systems' functional requirements.

Initially CBSD seems to be a silver bullet for revolutionizing software development, however further inspection reveals challenges and complications that increase the costs, but not the

benefits of using CBSD techniques. [2] Complications include: components with excessive defects and low quality, too much overhead and unneeded functions within the component, inadequate component functionality, unsatisfactory performance, difficulty understanding component interfaces, integration testing of components, business instability of component vendors, licensing issues, integration difficulties among components, and maintainability challenges with component based systems, especially in the case where component source code is unavailable.

Among the development challenges of CBSD we also have the problem of component selection. Selecting the proper component to meet functionality requirements is a complex task. First a component must be validated to see that it meets a minimum set of requirements. Available component options, found in repositories or from the commercial software marketplace must be evaluated to see if they meet the minimum set of functional requirements for the project. Once more than one potential component has been identified the best component should be selected for use in the software system. There are many factors that can influence the component selection choice. In many ways component selection is similar to software design decisions. Various factors can influence the decision and each must be considered and weighed into the ultimate selection decision.

In this research the topic of component selection is considered. First the common factors that may effect component selection are discussed. The goal of this research is twofold. This research seeks to make the component selection process easier by using software measurement techniques to provide quantitative data for component selection decisions. In addition to making component selection easier, a second goal is to improve component selection. In the end the desire is to choose the best components that will integrate well into a software system's overall design, and that will have the highest longevity throughout the software's maintenance life cycle.

This research proposes an example scenario that a software developer may face when making a component selection decision. Basic software requirements are specified and then a case study analysis of potential components is conducted. Software metrics are used to measure aspects of the components. The component data is analyzed in order to understand the importance of several internal component attributes and possible interesting relationships among them. Finally from this initial investigation several directions are suggested which may garner further research.

2. Approach

2.1. An Example Scenario

This research proposes a common development scenario that demonstrates the problem of component selection. Consider that an application developer is designing the graphical user interface (GUI) for an application that requires a user to supply a date. The developer wants to show a monthly calendar from which the user can navigate through months and years until finding and selecting the desired date.

R1: Component must allow for the user to select a date.
R2: Component must display a one-month calendar that can be navigated to show different months and years and used for date selection.
R3: Component must supply a date back to the application. The specific format of the date data can vary because the component can use a wrapper to convert the date into the application's desired data format.
R4: Component must show a preselected date or if no date has been preselected the current date upon initialization.

Table 1- Basic Calendar Component Requirements

In this study we use Java as the language and technology platform for study. Four Java calendar components were identified that meet the minimum functional requirements as stated in table 1. The requirements were kept simple for a few reasons. The focus of this investigation was to evaluate the components themselves, not to certify if they met a complex set of requirements. Secondly by keeping the requirements loose it was more likely that there would be many readily available components to consider in the study. Requirements were also kept simple because this is was an initial exploratory study. This research proceeds to apply measurements to the available components in order to determine which component may be the best component that should then be selected for use in the hypothetical software system.

In typical development scenarios such as the example scenario proposed, the software developer typically will make a decision based on the qualitative evidence about which component seems best. Knowledge from prior experience, testimonials from other developers, and ad hoc inspections of the components may be used to help the developer make the final selection decision.

2.2. Background

Selecting the proper component for a component-based system is a recognized problem in CBSD. [1] [2] [4] Formal specification can help certify that components meet a given set of requirements [3], but if there exists more than one viable component that meets minimum functional requirements, such as in our example scenario, the formal specifications can not be used to make the component selection decision. Work is needed to help "evaluate the quality of a component" and "its complexity" [1]. In many cases quality and complexity may be the leading factors driving the decision process in component selection. Other factors that could influence component selection include: speed of the component, size of the component, number of unwanted features, usability, price, and maintainability. The stability of the component vendor is another potential influencing factor. [2] If the organization that developed and supports the software component dissolves, then technical support and bug fixes may not be possible especially if component source code is not available.

The factors influencing component selection can be grouped into two categories: Internal and External component attributes. Internal attributes are concerned with the internal structure and implementation of the component. These attributes include details such as component size, number of interfaces, complexity of the component, and component performance. External attributes deal with non-implementation specific component details. Things such as component cost, business viability of the component vendor, availability of source code, licensing requirements, and availability and quality of documentation are considered external component attributes. The external attributes do not deal specifically with the implementation of the

component, but they can adversely affect the component selection process. Table 2 presents a listing of factors affecting the component selection process.

Attributes Affecting Component Selection	
Internal Attributes	*External Attributes*
Component Size	Component Cost
Number of Interfaces	Business Viability of Component Vendor
Component Complexity	Licensing Requirements
Component Performance	Documentation Quality
Component Usability	Source Code Availability
Number of unrequired features	Quality of Developer Support
Component Understandability	
Ease of Integration	

Table 2- Attributes that affect the component selection process

It is difficult to develop analytical methods for the assessment of the external attributes effecting component selection. These attributes deal with business aspects and the software developers often evaluate these attributes qualitatively before making a final component selection decision. However it should be possible to quantitatively assess the internal attributes effecting component selection. It should be possible to derive at least some measures of internal attributes such as component complexity, interface size, component size, and performance.

Four common groups of component selection criteria were identified in [7]. They include functional requirements, quality, business concerns, and relevant software architecture. Of these criteria, functional requirements, quality characteristics (such as reliability, maintainability, portability), and software architectural issues (such as operating system constraints and communication mechanisms between modules) are mostly considered to be internal attributes relating to the implementation of the component.

General component selection processes are presented in [6] [7] [8] [11] and [9]. Kontio presents a basic process that begins with a component search based on looking for components meeting the primary functional requirements. [6] [7] After a search is conducted a screening process is done to eliminate components that do not meet particular evaluation criteria unique to the needs of the development project. Once some components have been screened out of the selection decision the remaining candidates are evaluated based on detailed project specific criteria. The use of a weighted scoring method is applied to rank the performance of the candidate components for the final selection decision.

Kunda and Brooks offer a similar component selection processes that involves first defining the selection criteria, second identifying the possible component candidates, and third making an evaluation of the candidates for the purpose of selecting the best components. [8] Both Kontio's and Kunda's processes parallel very closely and both suggest the use of the Analytic Hierarchy Process (AHP) for evaluating component information and data for decision making purposes. [10]

Each of the selection processes identified proposes a lifecycle process for finding, evaluating, and choosing components, but none is specific about quantitative measurement techniques to measure the internal aspects of the components. Internal aspects such as ease of integration and the component's impact of maintainability on the software system are important criteria that should be measured and factored into component selection decisions. Case studies presented in [6] [7] [8] [11] are largely based on hands-on, qualitative evaluations of components. Little quantitative measurement of the components was conducted in these studies. However Kunda interestingly points out "the best way of evaluating COTS products is through experimentation within the operating environment in which the product will be used". Such experimentation could include architectural testing of internal component properties such as performance, and ease of integration.

2.3. Component Selection Questions

From a study of the component selection problem in component based software development, several interesting questions arise which merit investigation. Since this is an exploratory study these ideas are proposed in an informal question form. A formal statement of hypotheses based around these questions could be the basis of future research that investigates these ideas on a more formal and grandeur scale.

Q1- Do larger components with more features inherently perform slower than their smaller, more compact counterparts? (*Complexity relation to performance*)

Q2- Does component complexity negatively affect the understandability of the component? (*Complexity relation to component understandability*)

Q3- Is a component that is hard to understand (has low understandability) require more time and effort to integrate into the software system being developed? (*Complexity relation to ease of integration*)

Q4- Will developers make more errors while integrating hard to understand software components (components with low understandability) in software systems being developed? (*Complexity relation to quality of integration*)

The primary approach of this case study is to investigate the relationships between component complexity and several of the internal attributes which effect component selection decisions namely: component performance, component understandability, and ease of integration. This study aims to explore these questions stated above, to gain insight that can then potentially lead to future, more formal investigations.

3. Results

In this case study four calendar components were selected for evaluation as potential components that met the requirements described in the development scenario. Each of the components at the surface level seemed to meet all of the basic functional requirements.

The following assessments and observations were gathered about the components in this study:

- Complexity measurements were made based on size measurements (complexity of component)
- Integration difficulty was observed by building sample applications to test components. (ease of integration)
- Time to instantiate calendar objects was recorded in sample applications. (complexity of component)

3.1. Component Complexity

The complexity of each component was measured by collecting size data using the Understand for Java measurement tool. This tool required source code in order to make measurements. Source code was not available for three of the four components in the study. In these cases a Java decompilation tool, JODE (Java Optimize and Decompilation Environment) was used to generate source code from the Java byte code. Data for attributes such as: number of Lines of code (LOC), number of methods, compiled file sizes, source code file sizes, and number of instance variables was gathered for the components in the study. Size data for the components in the study is seen in table 3. Table 4 shows the source size of the primary class file for each of the calendar components. The primary class file is the class file that is invoked in the client code to actually interact with the component. Each calendar component's implementation used a varying number of additional classes that, a client application may or may not have to reference depending on the operations required in the integration and use of the component. For the sample applications only the primary classes were referenced.

Component	Compiled (jar) File Size	Lines of Code	Method Count	Number of Instance Variables	Number of classes in Jar file
Component A AWT	167598 bytes	1979	203	109	17
Component B Swing	55160 bytes	1283	132	90	11
Component C Swing	74608 bytes	900	63	27	68
Component D Swing	68310 bytes	131	17	6	19

Table 3 - Size of components

Component	Source Code Size
Component A AWT	69086 bytes
Component B Swing	44947 bytes
Component C Swing	22444 bytes
Component D Swing	7301 bytes

Table 4 - Calendar Class Source Sizes

Another measure of complexity in the study was to measure the time required to initialize and display the component in a simple Java application. It would seem that a simple component should initialize quickly and be displayed more rapidly than a complex component. For each sample application the same base framework and layout application was used. The only differences between each sample application were the lines of code required to integrate the

calendar component and display it. The average initialization time for each of the components is shown in table 5.

Component	Average Initialization time (ms)
Component A AWT	945*
Component B Swing	1204*
Component C Swing	485
Component D Swing	1225

*= Component had a drop down display that did not show a month view of the calendar until explicitly requested by user action.

Table 5 - Initialization Time of Components

3.2. Ease of Component Integration

Short sample applications were written to test each component. Each sample application initialized and displayed the component. The sample application also reported the currently selected date when the selected date changed. The sample application formally tested the previously stated component requirements to certify that the component did indeed meet the minimum set of requirements. Each component was found to meet the minimum stated requirements with different degrees of style and flair. In general, developing the sample application was simple and straightforward. The number of lines required to initialize and configure each calendar component were counted and their totals appear in table 6.

Component	Integration Size (LOC)
Component A AWT	5 loc
Component B Swing	3 loc
Component C Swing	3 loc
Component D Swing	4 loc

Table 6 – Integration Code Sizes (Glue Code)

The following are some observational notes from the development of the sample applications to test each of the components:

- (3) Components were based on the swing class library, where the other component was based on the AWT class library.
- (2) Components returned the currently selected date as a Java Date object. (1) Component returned the currently selected date as a Java Calendar object, which then a method could be used to acquire a Java Date object. Finally another component returned the currently selected date as a text string.

An easy to integrate component would preferably return data in standardized formats, such as for a calendar the Java Date object. Requiring the client application to perform trivial manipulations of returned data increases the amount of integration/glue code that needs to be written.

- All components supported adding a listener to detect when the currently selected date(s) changed.
- (3) Of (4) components supported multiple selection of dates by default

Maintenance activities are likely to require feature enhancements to the overall application. The best components may be components that have many features that could later be utilized by the client application as user enhancement requests are addressed.

- (2) Components presented the date selection as a drop down list. When the drop down was not engaged these components only took up the space of a text field saving valuable screen real estate. (1) Of the components offered a calendar drop-down style component within the jar library, and the other component, the smallest component, did not support this functionality.

Some of the calendar components evaluated were shipped in a library file that included many additional components and classes. Component D included an entire separate display library, which was used to give the user interface distinct display characteristics. Many of the additional classes and components supplied in the libraries were extra and were not necessary even for an application with advanced calendar display requirements. Although having additional functionality is useful, in general including these libraries bundled with the calendar components seemed to add more complexity and make the calendars harder to understand. These components included larger sets of documentation and in general they seemed more intimating from the viewpoint of an integrator.

With respect to interface size component A had (203) methods in the primary calendar class file. Although the design of the component may have included more encapsulation and information hiding, from an integrators standpoint the larger interface of this component is more intimidating that component D's (17) method interface. It is interesting to note that component A actual initialized in less time than did the simpler component D. Component D was in general a very small component that largely reused parts of the Java swing library. Each date was displayed as a separate Jbutton object in component D. This design makes Component D perform significantly slower than other components even though the overall size of component D is small. This observation is a reminder that in addition to interface complexity and size, the design and implementation of internal details of the component can also adversely effect performance.

4. Discussion

The first proposed question asked whether larger, more complex components perform slower than smaller, simpler counterparts. Although this simple relationship makes sense in reality the factors affecting performance can vary greatly. While components A and B were the most complex component in terms of interface size, their overall initialization times were not significantly different than that for components C and D. However both components A and B only displayed the calendar as a drop down list. The initialization time measurement did not consider the time required to pull down the list and display the month view of the calendar. Components C and D showed a month view of the calendar upon initialization. When components A and B are forced to show the month view their initialization time increased by 1000-2000 ms. Furthermore the initialization time of the component also seemed to depend upon specifics of its design and implementation. For example Component C which is significantly larger than Component D in LOC, actually performed significantly faster because its design used faster native Java graphics for drawing the calendar rather than relying on Java's built-in swing classes for rendering. From observations in this study the assertion can be made that the larger components did seem to

perform slower in this case study, but more factors than just the interface size and component size were involved.

The second question cannot be answered from the data collected in this study. From the experience of writing the sample applications, more complex components seemed less intuitive and more intimidating on the surface, but further empirical investigation is needed to analyze this relationship.

Answering the third and fourth questions is difficult from the results of this study. The limited set of requirements for the development scenario in this study resulted in the sample applications requiring about the same amount of integration (glue code). A more complex study is required to determine how component complexity impacts time and effort of component integration. The number of lines of code for integration is relatively the same for each of the calendar components. In most cases, property configurations were optional, and because our scenario only required basic operations only a limited amount of "glue" code was required. It was possible to initialize and display all of the components with very few lines of code. However using a minimal initialization code resulted in using the default settings for most all of the calendar properties. With minimal setup component B displayed with very poor colors and had an undesirable presentation. None of the components seemed inherently more complex to interface with than others. Of a more significant interest from this study was the ease of writing the sample applications. In general working with component D seemed very intuitive due its small interface and limited number of features. On the opposite end of the spectrum was component A. In order to determine which property reported the currently selected date for component A required searching through a 161 kbyte html document. A well-designed empirical study with control variables is desired to further investigate the impact of component understandability on ease of integration.

Components that offer significantly more capabilities than what is required seem to be harder to understand and thus seemingly more difficult integrate. Negative consequences of selecting complex components could include: slower performance, more defects in integration code, increased difficulty integrating components into applications, and more complex maintenance. Although smaller components may be desirable because they have less overhead and complex features, in some cases more integration and source code may be required to integrate these very simple controls, because they lack rich functionality and actually they may require the client application to define and implement many methods which a more complex component includes. Performing integration and configuration tasks on simpler components may require more steps and lines of source code. Large components may encapsulate operations into many individualized methods, where smaller components may require many of their properties adjusted separately to achieve the same configuration or operation.

5. Conclusions

The process of component selection, that is, the process to choose the best software component to meet a set of requirements for use in a software system being developed, is a problem discussed repeatedly in component based software engineering literature. [1] [2] [4] Considerable work has been done in the CBSD community towards formally specifying component requirements so that components can be identified within a component repository. [3] [4] Several existing component selection processes exist which describe formal processes to search for, evaluate and choose

components. [6] [7] [8] [9] [10] [11] However, these processes tend to focus primarily on the process of making decisions, and not on actually quantitatively evaluating components.

Complexity of a component can be a large factor in making component selection decisions. Complexity seems to impact the ease of integration, quality of the integration and maintenance activities associated with the component because of low understandability. Several factors could impact understandability including number of methods, number of classes in a library file, number of parameters required for methods, and interdependence between methods. From this initial investigation the need to conduct further, more formal investigations is established. Well-designed empirical studies could be conducted to test the relationships between component complexity, as measured here, and the ease of component integration. Are larger, complex components always more difficult to integrate than smaller simpler ones? An empirical study could investigate the relationship between component complexity and maintenance. Are complex components better suited for use over the long haul of a software project? Is there a tradeoff between lower understandability and more difficult integration that is expected with larger complex components versus the benefits from having a more fully featured component? As software development moves towards adopting more component based development practices, future research is desired to better understand the costs and tradeoffs that go into making component selection decisions. With additional research and quantitative analysis the component selection process should easily be enhanced beyond the traditional ad hoc selection processes that are now commonplace in software engineering practice.

6. References

[1] Goulão, M., Abreu, F. B., The Quest for Software Components Quality, in Proceedings of 2002 Computer Software and Applications Conference, (COMPSAC '02), pp. 313-318, 2002.

[2] Braun, C., A Lifecycle Process for the Effective Reuse of Commercial Off-the-Shelf (COTS) Software, in Proceedings of the 1999 ACM symposium on Software reusability, Los Angeles, CA, pp. 29-36, 1999.

[3] Edwards, S., Toward Reflective Metadata Wrappers for Formally Specified Software Components, in Proceedings of the Specification and Verification of Component-Based Systems, OOPSLA Workshop, October 2001.

[4] Ghosh, S., Improving Current Component Development Techniques for Successful Component-Based Software Development, ICSR7 2002 Workshop on Component-Based Software Development Processes, Austin, Texas, 2002.

[5] Vickers, A., CBSE: Can we Count the Cost? in Proceedings of the Fifth International Symposium on Assessment of Software Tools and Technologies, Pittsburg, PA, USA, pp. 95-97, 1997.

[6] Kontio, J., Chen, S., Limperos, K., Tesoriero, R., Caldiera, G., Deutsch, M., A COTS Selection Method and Experiences of Its Use., presented at the Twentieth Annual Software Engineering Workshop, Greenbelt, MD, 1995.

[7] Kontio, J., A Case Study in Applying a Systematic Method for COTS Selection, in proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, 1996.

[8] Kunda, D., Brooks, L., Applying Social-Technical Approach for COTS Selection, in proceedings of the 4th UKAIS Conference, University of York, UK, 1999.

[9] Alves, C., Castro, J. CRE: A Systematic Method for COTS Selection, in proceedings of the 15th annual Brazilian Symposium on Software Engineering, Rio de Janeiro, Brazil, 2001.

[10] T.L. Satty, Analytic Hierarchy Process, New York: McGraw-Hill, 1990.

[11] Kunda, D., Brooks, L., Case study: Identifying factors that support COTS component selection, Workshop for Ensuring Successful COTS Development in conjunction with ICSE '99, Los Angeles, CA, 1999.