SOFTWARE REUSE STRATEGIES AND COMPONENT MARKETS

BY T. RAVICHANDRAN AND MARCUS A. Rothenberger

Black-box reuse with component markets could be the silver bullet solution that makes software reuse a reality, and advances software development to a robust industrial process—but only if market makers address the growing pains plaguing this immature industry.

Software reuse, the use of software artifacts across multiple projects, is an important strategy for improving software development efficiency and increasing the quality of software systems [1, 4, 7]. When reuse is attempted, developers usually have access to the implementation code that can be modified to match a new project's requirements. This approach, in which code modifications are permitted, is referred to as white-box reuse. Frequently discussed inhibitors of white-box reuse include the large up-front investment necessary to populate a repository with reusable components [7], the problems associated with the classification and retrieval of reusable components, potential lack of management support, and the need to change the organizational structure and processes [5, 8]. Each of these factors poses a challenge that must be addressed successfully by organizations that engage in developing software using white-box reuse.

In recent years, a reuse approach known as black-box reuse has become popular. In contrast to white-box reuse, black-box reuse entails using software components “as is,” with no code modification. Developers need only know the functionality of a software component and how it interfaces with its environment, and need not alter the component’s internal logic and code.

Although black-box reuse is as old a concept as white-box reuse, it is becoming popular now in part because of component-based development (CBD), a software development paradigm. CBD involves building software systems using prepackaged components, which are essentially executable units of code that provide a set of services through a specified interface. CBD also involves reusing application frameworks, which provide the architecture for assembling components into a software system. Components and frameworks may be either developed in-house or procured from the marketplace.

In the past, the lack of standardized interfaces prevented organizations from searching for reusable components beyond the confines of their own development groups. The recent standardization of reuse architectures, such as the Common Object Request Broker Architecture (CORBA), Sun’s JavaBeans, and Microsoft’s Component Object Model (COM) enable reuse of software artifacts across development groups. Moreover, it allows components to be freely exchanged between and across application domains and development contexts—to such an extent that it
is now feasible to recognize the existence of software component markets.

The trade press has reported the significant advantages of CBD over traditional development approaches including white-box reuse. Industry analysts have also argued that component-based development approaches are an important building block for Web services, and that the keen interest in Web services is likely to make CBD a mainstream technology soon [11]. This article positions CBD within reuse strategies that organizations can pursue, and discusses its advantages and risks. These comparisons highlight the importance of understanding the trade-off involved in using the various reuse strategies and judiciously choosing the one best suited for a specific development context.

**Software Reuse Strategies**

A review of the literature indicates three broad reuse strategies: white-box reuse, black-box reuse with in-house component development, and black-box reuse with components procured from the marketplace. Table 1 summarizes the features of these three reuse strategies and compares them on key dimensions.

White-box reuse gives developers the freedom to modify the code to suit their needs. This provides a flexible way to harvest software assets in development projects by fitting existing components to new requirements. While this maximizes reuse opportunities, code modification is also a key source of problems encountered during reuse. Code modification requires a high degree of familiarity with the implementation details. Such familiarity is achievable when the code is developed in-house or the code is accompanied with excellent documentation. The latter is often not the case; thus, using in-house developed code can be the only viable option for white-box reuse. Moreover, when software artifacts are reused by those not involved in their initial development, the availability of personnel familiar with the original code becomes necessary to reuse software artifacts in a cost-effective manner.

![Software reuse strategies.](image)

Search cost is inherent in software reuse. When reuse is attempted, the repository must be searched for the availability of suitable software artifacts that can be modified. The objective of searching is to find artifacts close enough in functionality to the desired specifications. Thus, instead of attempting a focused search, a larger set of artifacts is typically retrieved from which the most appropriate is selected [9]. Retrieval can become problematic since the potential for the repository to grow exponentially exists. Component modifications create newer versions that must be catalogued for future reuse. The effort required for future retrieval increases, since developers must identify the component with the best fit among a number of similar artifacts.

Black-box reuse aims to avoid these pitfalls by not allowing the modification of retrieved components. While this approach can address the problem of increasing search costs and the difficulties in component modification inherent to white-box reuse, it also has its shortcomings. In a traditional reuse setting, only a small share of components are reused as-is without modification. Limiting reuse to those instances dramatically reduces the reuse rate, raising doubts about the feasibility of this approach. Hence, the inability to modify a component must be counterbalanced by a feature that increases flexibility. Black-box reuse based on CBD principles allows the component user to customize the artifacts using predefined parameters and switches. Although component customizability can partially compensate for the flexibility loss that comes with disallowing modifications, it is likely the project reuse rate will be lower with black-box reuse than with white-box reuse.

Procuring components from the marketplace can increase the reuse rate. This approach allows developers to search from a potentially larger set of components, increasing their chances of identifying ones that fit their needs. Thus, the inability to modify components in black-box reuse could be offset by the opportunity to choose a better-suited component in the first place. Table 1 indicates the initial develop-
In choosing among the reuse strategies, organizations are confronted with a trade-off between component acquisition cost and component customization cost. Each of these strategies. In addition, the framework indicates that when customization cost and acquisition cost are high, reuse itself may not be feasible—and the best alternative may be to develop the component from scratch. Here, we identify the key parameters that determine customization and acquisition costs, and discuss an approach to assess these costs.

**Customization cost.** Reuse is essentially an attempt at fitting a software artifact that is already in existence to new requirements. In doing so, developers must traverse a notional distance, as they transform an artifact from one form into another that fits

<table>
<thead>
<tr>
<th>Features</th>
<th>White-Box Reuse</th>
<th>Black-Box Reuse (CBD)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition</strong></td>
<td>Use of existing software artifacts found in in-house repository to build new software. Modification of code is allowed for customization.</td>
<td>Use of existing software artifacts found in in-house repository to build new software. Modification of code is not allowed; only configuration through defined parameters is possible. Use of existing software artifacts purchased from component markets to build new software. Modification of code is not allowed; only configuration through defined parameters is possible.</td>
</tr>
<tr>
<td><strong>Reuse rate</strong></td>
<td>High reuse rate because of the capability to modify existing components to suit new requirements.</td>
<td>Potentially a lower reuse rate is likely, because of the necessity to exactly match components to needs. Moderate reuse rates possible, because of the likelihood of finding a component in the marketplace that fits the requirements well.</td>
</tr>
<tr>
<td><strong>Flexibility in Component Use</strong></td>
<td>Higher flexibility in reuse as developers have the option to select the best component from a larger set of potentially reusable ones.</td>
<td>Limited flexibility as customization opportunities are largely dictated by how well the component interfaces have been parameterized. Limited flexibility as customization opportunities are largely dictated by how well the component interfaces have been parameterized.</td>
</tr>
<tr>
<td><strong>Scope of Reuse</strong></td>
<td>Reuse is local and typically restricted to development groups in the same location. Inter-firm sharing of software artifacts is uncommon.</td>
<td>Reuse is local but need not be restricted to development groups in the same location. Inter-firm sharing of software artifacts is easier than white-box reuse. Components can be purchased from vendors. Components can be bought and sold in markets.</td>
</tr>
<tr>
<td><strong>Component Development Cost</strong></td>
<td>Lower up-front development cost. Higher component development cost due to parameterization requirements. Higher component development cost due to parameterization requirements.</td>
<td>Higher search cost, but low customization cost. Higher search cost, but low customization cost.</td>
</tr>
<tr>
<td><strong>Component Usage Cost</strong></td>
<td>Moderate search cost, but high customization cost. Low search cost and low customization cost.</td>
<td>Higher search cost, but low customization cost.</td>
</tr>
<tr>
<td><strong>Quality</strong></td>
<td>Component quality is known to the organization. Easier to control component quality. Component quality is known. Easier to control component quality. Component quality could be an unknown. Risks associated with using poor quality software increases.</td>
<td></td>
</tr>
<tr>
<td><strong>Payback Time</strong></td>
<td>Moderate reuse rates required to recover investments in reusable components. High reuse rates required to recover investments in reusable components.</td>
<td>No up-front investment in software repository required.</td>
</tr>
<tr>
<td><strong>Repository Management Issues</strong></td>
<td>Exponential growth of repository size, version control issues become very important. Linear growth in repository size. Version control issues are important, but more easily addressed, since multiple versions of the same component need not exist. Limited need for in-house repository since the markets assume responsibility for cataloging and maintaining a repository of software artifacts.</td>
<td></td>
</tr>
</tbody>
</table>

Table 1. Comparison of software reuse strategies.
the specific requirements at hand. In general, the greater this notional distance to be traversed, the greater the cost involved in reusing a software artifact. A more sophisticated analysis of customization cost involves examining the different dimensions on which the specifications gap between a reusable component and a new component should be bridged. We have abstracted two dimensions likely to govern customization cost: contextual distance and domain distance.

Contextual distance is the extent to which the target implementation environment for a software artifact differs from the existing artifact. It is common to port artifacts developed for one implementation environment or platform to another and such porting affects customization cost. Domain distance is the extent to which the target application domain for a software artifact differs from the existing software artifact. This dimension captures the effort involved in tailoring the functionality of a software component to fit an application domain.

Since a major cost in software development is the development effort, we operationalize the two distances in terms of that effort. Domain distance can be measured by the effort required to modify the functionality of an existing component to exactly meet the requirements at hand. Contextual distance can be measured by the effort required to convert an existing component to the desired implementation environment in terms of the platform and programming language.

Estimation by analogy can be used to determine the effort required for component modifications. This approach, used in software cost estimation, assesses similarity between a known software project and a new one, to determine the new project’s cost. Similarity is determined by measuring the Euclidean distance in the n-dimensional space that describes a set of project characteristics [10]. The approach itself does not prescribe the variables that determine the distance between two projects. Instead, the users choose the variables that characterize the projects and construct the estimation model that best fits the development context.

This approach can be adapted to assess the domain distance and contextual distance. The domain distance between an existing component and the specifications for the desired component can be determined by selecting variables that depict the subset of the desired functionality for a component. An n-dimensional space, with each dimension representing a desired functionality, could be constructed with these variables. Each existing component can be positioned in this space based on the subset of their desired functionalities. Calculation of the Euclidean distance in this n-dimensional space between each existing component and the desired functionality set can yield a measure of the domain distance. Components with the least domain distance can be expected to require lower customization effort.

In a similar manner, contextual distance can be computed using variables that characterize the language, architecture, and platform of the existing component, and the requirements at hand. The use of language-independent architectures, such as
CORBA, and a platform-independent language, such as Java, makes a zero contextual distance theoretically possible. However, either technology alone does not meet this goal; for example CORBA's underlying components are often platform-dependent. Java components, on the other hand, are generally language-specific. Only the combination of the two, a platform-independent language wrapped within a language-independent middleware would result in a component with a zero contextual distance to any specifications desired [3]. Nevertheless, when components that meet the specifications are available for the desired platform and language, the contextual distance can be close to zero.

**Acquisition cost.** Two key costs involved in component acquisition are search cost and component price. When reusing a software artifact, substantial effort must be invested in the retrieval of artifacts. Possible sources for reusable components include both in-house repositories and the component markets. Locating a component from either source requires developers to traverse through classification schemes, conduct keyword searches, and even manually select suitable components and inspect their functionalities. The magnitude of such a retrieval effort is related to the number of available artifacts in the repository. Thus, a search that is limited to the in-house repository will be less expensive than a search of the component markets.

While a component available in-house can usually be reused without payment of additional royalties or license fees, component markets sell software artifacts for profit. Markets theoretically can offer components at lower costs than in-house development efforts can yield, because of scale economies. However, the immaturity of the component markets at the present time does not allow such efficiencies to be realized. Thus, it is likely that the acquisition cost is higher when a component is purchased from the market. Nevertheless, we will show that obtaining software artifacts from component markets with lower notional distances and, as a consequence, lower customization cost, may offset the higher acquisition costs.

**A Component-Level Reuse Decision Model**

How does a development group go about choosing a reuse strategy? To address this issue, we propose a simple decision tree at the component level (see Figure 2) that incorporates component acquisition and component customization costs. The model depicts the various reuse strategies and the questions developers must answer in choosing each strategy. The model also depicts the cost equations that help developers answer these questions. It illustrates that component price is an important factor in determining the extent to which the component markets are likely to be used.

The first question shall be whether components with acceptable notional distances are available in-house, since the reuse of such components does not require payment of royalties, license fees, or procurement cost. The threshold of an acceptable notional distance can differ for black-box and white-box reuse. The domain distance between a black-box component and the requirements on hand is deemed acceptable if the component can be customized to specifications by setting the parameters provided by the component. The contextual distance must be close to zero for black-box reuse. For notional distance to be acceptable in white-box reuse, the effort required to modify the component to meet specifications must be lower than the effort required to develop the desired component from scratch. As white-box reuse allows for both domain and context modifications, the acceptable threshold for contextual distance is much higher than for black-box reuse.

If a suitable black-box component is available in the repository, use of this component is preferable to other alternatives, as the configuration cost of a black-box component is lower than the customization cost of a white-box component or its development from scratch (see node 1 of Figure 2). If a black-box component is not available, the developers must decide between procuring a component from the market or the in-house alternative, which could be either white-box reuse or development from scratch, depending on the availability of an acceptable white-box component (see node 2 of Figure 2). In both cases, the decision depends on the price of the component in the market (see nodes 3 and 4 of Figure 2).

The decision model reflects that markets can reduce the need for customization by allowing a better match of developer needs with available components. Such decreased domain and contextual distance may tilt the balance toward black-box reuse if the component price is attractive. In the long run when component markets mature, search costs are not likely to be significant. Having access to intermediaries that aggregate components from numerous sources, standardized templates for cataloging components, and searchable indexes, can facilitate component searches and drastically reduce search costs. Thus, CBD with component markets may be the natural choice over white-box reuse as long as compo-
components with low notional distances to the desired specifications are available, and their market price allows the developers to retain cost savings over white-box reuse.

**Conclusion**

Meyer [6] defines a component as “a software element that must be usable by developers who are not personally known to the component’s author to build a project that was not foreseen by the component’s author.” This definition could apply to both white-box and black-box components. In fact, the first criteria is easier to facilitate in black-box components since CBD forces component developers to clearly define the interfaces and therefore requires no additional communication with component users. However, the second requirement is easier to meet if modifications of the component are possible. A well-written black-box component with a sufficient number of customization options can also meet this requirement adequately.

We argued that component markets could play an important role in making well-written components available to developers. In the last few years, a few component markets have been established (such as www.flashline.com and www.componentsource.com) that sell a variety of components ranging from low granularity graphical user interface objects to high granularity components that offer business functionality. Moreover, some domain-specific initiatives have been undertaken to develop and sell components supporting core processes in industries, such as financial services and insurance. Despite these developments, the market is relatively immature and highly supply-constrained; developers face difficulties in finding components that meet their requirements in part due to ineffective market mediation.

With industrial products, supply chains perform the dual role of market mediation and distribution. Market mediation involves bridging the gap between what the market wants and what suppliers produce. Distribution involves the physical movement of goods and services across the supply chain. Component markets currently perform only the role of distributing components, and fail to be effective market mediators.

While it is too early to predict how they will evolve, one can speculate that for component markets to flourish, several issues must be addressed. Licensing and ownership becomes tricky when components are sold in a market and could potentially be used many times by buyers. Issues such as ongoing support for and quality assurance of components will become important if component markets are to gain traction among developers [4]. These issues take on added importance when components are reused across national borders and different legislatures. When purchasing components, organizations must be able to assess the quality of the software artifacts in order to develop the confidence to use them in their applications. Component quality can be certified in license agreements [4], or through independent quality assessments [2, 5]. Similarly, component users will look for assurances from intermediaries about ongoing support and component upgrades. Unless the market makers gear up to address these issues, component markets could go the way of many other innovations in software development—a lot of hype and no discernable improvement in software development practices. Such an outcome would be unfortunate, because black-box reuse with component markets could be the silver bullet solution that makes software reuse a reality and as a consequence moves software development from a craft to a robust industrial process.

**References**