Research issues in testing business components

Padmal Vitharana*, Hemant Jain

School of Business Administration, University of Wisconsin-Milwaukee, P.O. Box 742, Milwaukee, WI 53201, USA

Received 1 October 1998; accepted 27 July 1999

Abstract

The advantages of migrating from traditional monolithic business applications to reusable object-based business components (self-contained software that carries out a certain business task) are well documented. A business system assembled from reusable components is argued to be highly reliable since these components have been tested and used in many other business applications. However, all possible uses of components are not known at design and construction stage. Additionally, integration testing is needed as components are assembled to make business application systems. Component-based software development requires that testing issues be addressed adequately. In this paper, we explore testing related issues in business components and in particular, business application systems that are made by integrating these components. An integration test strategy for business component application systems is proposed. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Business components; Object-oriented; Software testing; Reuse; Component assembly

1. Introduction

The concept of building business applications from reusable components is becoming common in the industry. A component is a self-contained software package that carries out a certain business task. According to Dodd [4], component-based software development (CBSD) is grounded in component fabrication and component assembly. Component fabrication is the process of building reusable components, while component assembly is the process of integrating a set of components to build a component application system.

Business components typically emulate a specific business entity, such as a customer, order, or an account. When a set of coherent components is linked together, they can inter-operate to achieve an even higher level business function. Orfali et al. [20] argue that traditional monolithic mainframe-based business applications are behind us, and the trend is to build IS from reusable business components that operate in a distributed computing environment. With the increasing availability of enabling technologies, CBSD has recently gained considerable attention in the industry e.g. [2].

The idea of building software from reusable parts is not new. McIlroy [18] introduced the concept of formal software reuse by identifying source code as a reusable asset. Frakes and Terry [7] define software reuse as “the use of existing software artifacts or knowledge to create new software.” Research has shown that reuse improves both software quality and developer productivity [1, 8, 19]. Reusable artifacts include requirement documents, design structures, source code, or any other deliverable that results from the software development process.

The CBSD approach resembles Object-Oriented (OO) software development. Objects that are the
building blocks of the OO paradigm encapsulate data and methods. An OO program consists of a set of cohesive objects that communicate. However, objects live only within a single program. In contrast, components are packaged as independent pieces of software that can live anywhere on a network and inter-operate through method invocations. In addition, component objects persist outside the current application while OO objects do not remain beyond the application. In spite of these differences, there exist a number of similarities between component-based and OO-based development. For example, a component consists of one or more objects with each encapsulating its own data and methods similar to that of an object in the classical OO paradigm. Also, the inter-object communication via method invocations is comparable. Fig. 1 shows an object class, a component, and a component application system.

Although the importance of testing in component-based systems has been recognized [25], testing issues in this paradigm are not addressed adequately. In this paper, we explore issues related to testing component-based systems. Since the CBSD shares some similarities with OO development, current research on OO testing can be extended to develop methods for testing component-based systems.

2. Testing OO programs

An OO system consists of a number of objects that contain attributes and methods. Methods are used to manipulate the object’s attributes and act as an interface, so that other objects can communicate with it. Encapsulation, inheritance, and polymorphism are the key features of the OO paradigm. However, these powerful features introduce a new set of testing problems. Encapsulation allows for information hiding, so that other objects can communicate only through a well-defined interface. Inheritance is a relationship that allows a more specialized subclass to inherit both attributes and methods of a more general superclass. Polymorphism allows objects of different classes to interpret the same message differently. Research on OO testing includes [3,5,10,13,14,15,16,23].

Kung et al. [15] summarize OO testing problems as: (1) understanding; (2) complex interdependency; (3) object state behavior testing and (4) tool support. Understanding problem is introduced by the encapsulation and information hiding features where several member functions from several object classes are invoked. Dependency problem arises as a result of complex relationships that exist in OO programs including inheritance and polymorphism. State behavior-testing problem arises due to objects having multiple states and the need to test their state-dependent behaviors. Since a change to a single class may effect all its subclasses, and manually identifying the impact of change is both difficult and time consuming, tool support is crucial in testing.

Since objects are usually constructed as classes, testing an OO program involves unit testing these classes individually and then integration testing the set of classes used in the program. The developer
usually carries out unit testing [12] by building test beds that incorporate a collection of test cases for testing various attributes and methods of a class. Generally two types of testing strategies are utilized: black box (or functional) testing and white box (or structural) testing [9]. In black box testing, test cases are derived from the preliminary design specification. Black box testing is done without any internal knowledge of a given class. In contrast, test cases in white box testing are derived from detailed design specification and use the knowledge of how a class is designed internally. The purpose of white box testing is to verify that the internal structure of a class is correct, whereas the purpose of black box testing is to verify that the external interface of a class is correct. Since they complement each other, typically both white box testing and black box testing are used.

A number of techniques for unit testing object classes has been proposed. For example, Fiedler [6] discusses unit testing of C++ classes. Doong and Frankl [5] describe an approach called ASTOOT that can be used to unit test OO programs and also present a set of tools based on this approach. Kung et al. [15] discuss how object relation diagram (ORD) and branch block diagram (BBD) can be used to derive test cases for unit testing a class. Turner and Robson [23] describe a technique called state-based testing that can be used to validate an OO program with emphasis on the interaction between the features (methods) and the object’s state (attribute values). Smith and Robson [21] present a framework for testing OO programs that helps in determining methods to be tested, the sequence in which to test these methods, and their parameter values. Similarly, Kirani and Tsai [14] propose a technique called method sequence specification to represent the causal relationship between methods of a class that can be used for generating test cases. Harrold et al. [10] describe how a class graph where each node represents a member function in the class or a primitive data member and each edge represents a message can be used to guide intra-class testing.

Integration testing is usually done by a special testing team and is carried out in the target environment. Integration testing techniques in traditional procedural-based systems vary from non-incremental methods such as big bang testing to incremental methods such as top-down, bottom-up, and sandwich (a hybrid of top-down and bottom-up) testing. In contrast, integration testing of subclasses in OO programs can occur soon after all their superclasses are unit tested. Kung et al. [16] suggest that ORD and test order can be used to provide a detailed road map for conducting integration testing of classes. Jorgensen and Erickson [13] propose a five-level integration testing approach that stresses behavior instead of structure in testing OO programs. Harrold et al. [10] describe how a class graph can be used to guide inter-class testing. The state-based testing technique as described by Turner and Robson [23] can be used for testing interactions between features (methods) of different classes.

Although there are some differences between the CBSD and classical OO-based software development, the similarities between these two software development paradigms may be used as a foundation for developing methods for testing components and component application systems.

3. Testing issues in component-based systems

3.1. Testing business components

Like a traditional program, a business component should be subject to verification and validation testing. Verification is the process of determining whether the product is being built correctly while validation is the process of determining whether the correct product is being built [9]. Verification testing is carried out throughout the software development process, and involves determining whether the software product at the end of a phase satisfies the conditions set forth at the beginning. More formal methods such as software inspections to less formal methods such as technical

---

1 An ORD represents the inheritance, aggregation, association, instantiation, uses and nested relationships among the object classes.

2 An BBD represents the control structure of a member function and its interface to other member functions so that a tester will know which data is used and/or updated and which other functions are invoked by the member function.

3 According to Kung et al. [16], the test order is computed according to the degree of dependencies between classes.
reviews and walkthroughs are used for verification testing. Validation testing is generally carried out upon the completion of the software product to determine whether it satisfies the users’ requirements. Unlike traditional software developers, component developers in most cases do not have a knowledge about all the possible uses of the component. The reason for this is that component developers are typically independent software vendors (ISV) who may not necessarily know the ‘future’ uses of their components. The requirement analysis in this case attempts to identify business components that are suitable for reuse and the requirement specification describes the functionality of the component. This requirement specification can be used to determine whether the component’s desired behavior matches with the component’s actual behavior.

Black box and white box testing techniques discussed earlier can be adopted for verification testing of business components. These techniques are explained here with the help of an example business component PayrollDeptBudget (see Fig. 2) illustrated in Interface Definition Language (IDL) format. The PayrollDept-Budget component can be used to keep track of employee payroll and department budgets in a company. This component contains two object classes, PayrollBudget and DeptBudget. For each employee, an instance of PayrollBudget object class is created. Each of these instances consist of attributes that hold

![Module PayrollDeptBudget](image)

```cpp
Module PayrollDeptBudget
{
    interface PayrollBudget
    {
        attribute char[] Social-Security-Number;
        attribute int Pay-Category;
        attribute float YTD-Earnings;
        attribute float YTD-Taxes;
        attribute float Max-Deductions-Allowed;
        attribute float Before-Tax-Deductions;

        // OUT methods
        void Update-SSN ( in char[] ssn );
        void Update-Pay-Category ( in int category );
        void Update-Max-Deductions-Allowed ( in float deductions );
        void Report-SSN ( out char[] ssn );
        void Report-Pay-Category ( out int category );
        void Report-Max-Deductions-Allowed ( out float deductions );
        void Report-YTD-Earnings ( out float earnings );
        void Report-YTD-Taxes ( out float taxes );
        void Report-Before-Tax-Deductions ( out float deductions );
        void Issue-Paycheck ( out float pay );
        void Add-To-Before-Tax-Deductions ( in float deductions );

        // IN methods
        void Get-Name ( in char[] ssn, out char[] name );
        void Get-Number-of-Exemptions ( in char[] ssn, out int exemptions );
    }

    interface DeptBudget
    {
        attribute char[] Dept-Name;
        attribute float Current-Dept-Budget;

        // OUT methods
        void Update-Dept-Name ( in char[] name );
        void Update-Current-Dept-Budget ( in float budget );
        void Report-Dept-Name ( out char[] name );
        void Report-Current-Dept-Budget ( out float budget );
    }
}
```

Fig. 2. Class structure of the PayrollDeptBudget component.
earnings data about the employee (e.g. \textit{YTD-Earnings}) and methods that manipulate its attributes and act as an interface (e.g. \textit{Report-YTD-Earnings}). Similarly, for each department, an instance of \texttt{DeptBudget} object class is created.

Testing \texttt{PayrollDeptBudget} component is analogous to testing a classical OO program that involves unit testing base classes, integration testing derived and adjacent classes, and finally system testing the program as a whole. Unit testing an object class such as \texttt{PayrollBudget} involves deriving test cases in both black box and white box testing methods.

3.1.1. Black box testing

In black box testing, test cases are derived from both class specification and method specification. Each class has a specification that describes the overall functionality of the class including its attributes and methods. Each method in a given class has a specification that describes the method in detail including its interface, valid (expected) parameter values, pre and post-conditions, and exception handling. A black box test case that invokes \texttt{Add-To-Before-Tax-Deductions} method in the \texttt{PayrollBudget} class for example should ascertain that an amount less than the maximum allowed is deducted. A comprehensive test bed for black box testing should contain test cases for each method under various scenarios (e.g. different parameter values).

3.1.2. White box testing

In white box testing, test cases are generated by examining the internal structure of the class. For example, a white box test case that invokes \texttt{Issue-Paycheck} method in \texttt{PayrollBudget} class should check whether the attribute \textit{YTD-Earnings} is correctly adjusted after a paycheck is issued. A comprehensive test bed for white box testing should contain test cases for each method under various scenarios (e.g. different parameter values).

Building a comprehensive test bed for unit testing and integration testing object classes involves four important tasks, namely, test case generation, test driver generation, test execution, and test checking. Out of these four tasks, test case generation is the more difficult and time consuming. Recently, attempts have been made to automate some phases of test case generation as well as other tasks listed above (e.g. [5]). Since it is not possible to generate test cases that test every aspect of an object class, test cases must be chosen such that a minimum number of test cases satisfies a maximum number of testing criterion. Black box test cases can be generated using techniques such as equivalence partitioning, boundary value analysis, error-guessing, and cause-effect graphs while white box test cases can be generated by considering various coverage levels for the internal logic of the class.

In addition to unit testing and integration testing, system testing is crucial to developing quality business components. System testing must verify that the software product operates as documented, interfaces correctly with other systems, performs as required and satisfies the user’s needs [17]. Test cases for system testing are typically generated from the requirement specification.

In CBSD, each prospective customer of the component has access to the interface specification that describes the external functional interface of the component. In documenting this interface specification, developers consider the preliminary design specification used in building the component. To help potential customers in component selection, the interface specification needs to also include possible uses of the component. However, as noted earlier, one of the greatest challenges facing component developers is the identification of all possible uses of a given component. Additionally, the conditions under which the components are used may impact quality. Therefore, developers may first identify a restrictive set of uses for the component and test the component under those settings. Any other use of the component will require further testing [25].

3.1.3. Testing business component application systems

Once individual components are unit tested, they can be integrated together to build a business component application system. Using off-the-shelf components,
component integrators will be able to assemble application systems in record time. Since these components are already unit tested and reused many times in other applications, they are argued to be of high quality. However, if these components are to be assembled to create a component application system, then they have to be integration tested.

Both procedural-based and OO-based software development practices are rooted in the traditional software development life cycle (SDLC) that comprises analysis, design, implementation (coding), and testing. Recall that the CBSD is grounded in component fabrication and component assembly. Component fabrication process closely follows the SDLC. Issues in testing these components were discussed earlier. Interestingly, the component assembly process illustrated in Fig. 3 also resembles the SDLC as described below.

1. Requirement analysis: Identify and document user needs in the system requirement specification.
2. Preliminary design: Specify software architectures including required components and their interfaces, and document them in the preliminary design specification.
3. Detailed design: Search and identify components that satisfy the functionality as specified in the preliminary design specification.
4. Implementation: Develop an integration strategy to assemble the components identified above in a business component application system.
5. Testing:
   (a) Unit testing: No unit testing is done since the component is assumed to be unit tested by the vendor and the code is unavailable to the component integrator.
   (b) Integration testing: Component integrators use the preliminary design specification of the proposed application and vendor-supplied interface specification of each component in integration testing the set of components to create a component application system. In the next section, we discuss these integration testing issues in detail.
   (c) System testing: The system requirements specification is used for system testing the business component application system (note that system testing as well as acceptance testing using user requirements specification is beyond the scope of this paper).

Note that in the CBSD, each phase takes more of an iterative form since it involves finding the best fit

---

Fig. 3. Graphical view of the component assembly process (modified 'V' diagram adapted from DOD-STD-2167A [24]).

---

\[1\] Specify required components.
\[2\] Search and identify components.
\[3\] Develop an integration test strategy and assemble components.
\[4\] No unit testing is required.
\[5\] Integration test the set of components to create a component application system.

---

\[4\] Jorgensen and Erickson [13] observe that documents created as a result of requirement, preliminary design, and detailed design phases of the SDLC correspond with, and therefore, used in system, integration, and unit testing, respectively.
between the requirements and the available components. For example, if a precise component required according to the preliminary design specification cannot be found during the detailed design phase, then requirements may have to be adjusted or a component be built in-house. Furthermore, if integration testing reveals a defect in one of the components, then the vendor is requested to supply a replacement component.

Integration testing a component application system involves determining whether the selected components will work together. Note that implementation (i.e. component assembly) and integration testing are distinct, yet interacting tasks. As noted earlier, since the component assembly process closely resembles the class integration process in the classical OO paradigm, integration-testing needs of component application systems can be fulfilled with existing research in class integration testing.

In the previous section, we presented a description of the **PayrollDeptBudget** component. Figs. 4 and 5 show the specification (in the IDL form without the
attributes) for two additional business components, \textit{Employee} and \textit{EmployeePension}. The \textit{Employee} component can be used to record demographic information of an employee. The \textit{EmployeePension} component can be used to satisfy the functionality of an employee pension system. Let’s consider a scenario in which a company purchases \textit{Employee}, \textit{PayrollDeptBudget}, and \textit{EmployeePension} components with the intention of building a business component application system that integrates the functionality of these components.

When purchasing a component from an outside vendor, the company will be furnished with an interface specification that describes the overall functionality of the component, and more importantly, its external interface. Orfali et al. [20] suggest that these specifications be written in a neutral Interface Definition Language (IDL) that defines a component’s boundaries and acts as a contractual interface with potential clients. Moreover, components should interoperate with each other across programming languages, operating systems, and computer networks. The underlying architecture provides an environment for these components to interact with each other by passing messages through a method interface. This supporting architecture typically called middleware provides the necessary system services for the components transparent of the operating system. Currently there are two competing middleware architectures. Common Object Request Broker Architecture (CORBA) is the result of a consortium of organizations called Object Management Group (OMG) that separates the specification of services from the implementation. Object-Linking and Embedding/Common Object Model (OLE/COM) is the middleware architecture of Microsoft and its products. Both of these architectures will certainly impact the CBSD in the future. However, recently Sun’s JavaBeans have also gained acceptance.

Over the years, several CORBA-compliant implementations have emerged [22]. Examples include Orbix (by IONA), System Object Model (by IBM), and Visibroker (by Visigenic). Orbix for instance is available on numerous platforms and supports C++ and Smalltalk language bindings. Microsoft components on the other hand are based on COM. Distributed COM (or DCOM) was introduced as a means to extend COM services transparently. Sun developed the JavaBeans architecture for constructing reusable components written in platform independent Java language. Several firms (e.g. IBM) currently offer JavaBeans based products.

The challenge for the component integrator is to assemble components, such that they work together correctly. As stated previously, common integration testing techniques vary from non-incremental big bang to incremental bottom-up. Kung et al. [16] recognize that a bottom-up strategy is preferred for OO programs. Jorgensen and Erickson [13] proposed and outlined a five level testing approach that advocates bottom-up integration. Bottom-up testing starts with lowest level modules, whereas top-down testing starts with highest level modules. Since each business component is a complete and fully implemented unit, bottom-up strategy becomes the obvious choice in integrating these components to create a component application system. In incremental bottom-up strategy, the number (granularity) and type (e.g. less complex versus more complex) of components to integrate in each iteration and sequence of integration become important issues.

As noted earlier, each component is accompanied by an interface specification that describes its functionality, method names, and parameter list. The component assembly process involves mapping method’s names between components. For example, lets consider the case where \textit{EmployeePension} component needs to set a dollar amount to be deducted from an employee’s paycheck through a call to \textit{PayrollDeptBudget: Add-To-Before-Tax-Deductions} method. The assembly process will be responsible for building this mapping. Fig. 6 shows the method interactions among the three components. To keep the example simple and tractable, only four interactions between the components have been identified.

Two major problems may arise during the mapping process. First, the number of parameters of the caller component method may differ from that of the called component method (Param-Number-Mismatch). Second, the type of parameters of the caller component method may differ from that of the called component method (Param-Type-Mismatch). These represent some of the difficulties in the CBSD that are not experienced in traditional built-from-scratch software ideology. A component assembly and testing strategy is proposed here to address some of the above issues.
3.1.4. Component assembly and testing strategy

The proposed component assembly and integration testing strategy consists of the following steps.

**Step 1:** Select a set of components that has the **maximum** number of calls to other components within the set, but has **minimum** number of calls to components outside of the current set.

This strategy increases the probability of finding any interface conflicts early, while keeping the level of complexity to a minimum by considering only a limited set of components initially. Test cases in Step 1 should emphasize method interactions within the selected components as well as persistence characteristics of their component objects. We call this partially assembled set of components, an application sub-system.

**Step 2:** Of the remaining components, select a component (or a set of components) that has the **maximum** number of interactions with methods in the current application sub-system. Continue this step until all the remaining components are assembled and integration tested.

We believe that this strategy will uncover any method interface conflicts early in the component assembly process. If there is a large number of components to be assembled, then instead of selecting one component at a time, a set of components that has an equal number of method interactions with the current application sub-system could be selected.

The following example illustrates the proposed component assembly and testing strategy.

3.1.5. Example

Let’s consider six components and their method interactions as shown in Fig. 7. Assume that the direction of an arrow symbolizes caller and called relationship of a method invocation between two components, and the number listed on each arrow...
symbolizes the number of invocations from the caller to the called method.

**Step 1:** Since the component set AB has one method invocation between each other, and makes no calls to any method of other components, select components A and B for assembly and integration testing first.

**Step 2**

**Iteration 1:** Determine the method interactions between the current application sub-system AB and the rest of the components:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
</tr>
</thead>
<tbody>
<tr>
<td>AB</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

Since component F has the most method interactions with the current application sub-system AB, assemble and integration test component F.

**Iteration 2:** Determine the method interactions between the current application sub-system ABF and the rest of the components:

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABF</td>
<td>6</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Since component C has the most method interactions with the current application sub-system ABF, assemble and integration test component C.

**Iteration 3:** Determine the method interactions between the current application sub-system ABFC and the rest of the components:

<table>
<thead>
<tr>
<th></th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABFC</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Since component D has more method interactions with the current application sub-system ABFC, assemble and integration test component D.

**Iteration 4:** Finally, assemble the only remaining component E and integration test it.

Now consider the example discussed earlier as illustrated in Fig. 6, and observe the following behavior in method interactions among the three components. **Employee** and **PayrollDeptBudget** have two method invocations between each other, but have no calls to methods in **EmployeePension**. **Employee** and **EmployeePension** have one method invocation between each other and one call to a method in **PayrollDeptBudget**. **PayrollDeptBudget** and **EmployeePension** have one method invocation between each other and three calls to methods in **Employee**. Therefore, **Employee** and **PayrollDeptBudget** should be assembled first and integration tested during Step 1. During Step 2, select the remaining component **EmployeePension** for assembly and integration test with the current application sub-system.

Two problems related to component integration were identified as Param-Number-Mismatch and Param-Type-Mismatch. Here, we propose possible solutions to overcome these difficulties. Param-Number-Mismatch issue may be resolved by using techniques such as ignoring unwanted parameters or introducing dummy parameters. Parameter-Type-Mismatch issue may be resolved by using techniques such as compromising on the level of precision, truncating parts of a character string, and padding characters to the end of a string. Following section discusses Param-Number-Mismatch and Param-Type-Mismatch issues in assembling the **Employee**, **PayrollDeptBudget**, and **EmployeePension** components.

3.1.6. Example: Param-number-mismatch

Note that **PayrollDeptBudget** component (caller) needs to obtain the name of an employee through a call to **Employee:Report-Name** (called) in order to issue his/her paycheck. It is possible that the caller’s method has only one parameter that expects the full name in a single text string while the called method has two parameters that return the name in two text strings-first name and last name. If there is a Param-Number-Mismatch as illustrated here, then the caller-called
methods can be mapped such that the caller’s method may accept only the last name from the caller and ignore the first name or merge first and last names from the called method upon return.

3.1.7. Example: Param-type-mismatch

Consider the case where EmployeePension component (caller) needs to set a dollar amount to be deducted from an employee’s paycheck through a call to PayrollDeptBudget:Add-To-Before-Tax-Deductions method. If the caller’s parameter value has a lower level of precision than that of called, then the caller-called methods can be mapped such that upon return, some precision may be lost (i.e. due to rounding-off).

Problems could also occur due to different interpretations of parameter values between components. For example, let’s take the case where PayrollDeptBudget component needs to obtain number of exemptions for an employee through a call to Employee:Report-Number-Of-Exemptions in order to calculate his/her taxes. When considering exemptions, do these component methods include the employee himself/herself as an exemption? If this difference is not uncovered early, it may result in incorrect taxes being calculated. Again, these types of discrepancies may be resolved by mapping methods of the two components involved during the assembly process.

Another problem could arise due to the lack of functionality of components. Although requirements in preliminary design specification are matched against interface specification of each component during the assembly process, a component integrator may realize the lack of functionality of a component that is already selected. For example, let’s consider PayrollBudget object class in PayrollDeptBudget component. As described in Fig. 2, an employee’s pay is based on his/her pay category. What if the company hires a new employee whose salary is to be based on sales commission? The current methods may not support the notion of commission in determining an employee’s salary. If this is the case, possible solutions include returning current PayrollDeptBudget component back to original vendor for a new one with the added functionality, searching for a new component, or building the required component in-house.

4. Implications and limitations

The CBSD presents opportunities as well as challenges to building quality information systems. Components attempt to provide the software industry what integrated circuits (ICs) already provide the hardware industry — the ability to build integrated products from reusable parts. In the CBSD paradigm, each component has to be unit tested, so that component integrators will have confidence in using these components to build quality business component application systems. In this paper, we first provided evidence to support the parallelism between the CBSD and classical OO-based software development. Second, we proposed a component assembly and integration test strategy that will help component integrators.

Our component assembly and integration test strategy is based on the number of method interactions between the components. However, a more comprehensive component assembly and integration test strategy could also incorporate the level of complexity of a component. Similarly, a subset of the components selected for assembly may have a logical relationship with one another, and therefore, should be accounted for in developing an integration test strategy. Although we have not discussed issues of complexity and logical inter-relationships between components, we believe that these issues could play an important role in the component assembly and integration testing process.

Earlier we recognized the need to search and identify components that satisfy the functionality specified in the preliminary design specification. However, we have not addressed these issues in great detail. There is some research on the search for reusable software artifacts. For example, Isakowitz and Kauffman [11] presented a tool to support developers search large repositories for reusable software objects. In building business component application systems by assembling a set of components, component integrators need efficient tools to search for necessary components.

In this paper, we have discussed testing related issues in developing component-based software. We believe that the traditional built-from-scratch software ideology is behind us, and the trend is in the CBSD involving component fabrication and component
assembly. Therefore, unit testing business components and integration testing business component application systems play a crucial role in building quality component-based software systems.

Throughout the paper, we have presented evidence to support the parallelism between the CBSD and classical OO-based software development. Consequently, we have argued that testing needs of components and component application systems can be drawn from current research on testing classical OO programs. We have explored testing issues in business components and, in particular, business application systems that are developed by integrating a set of components. We have outlined the component assembly process and proposed a strategy for integration testing. We hope that the issues presented in this paper will provide some valuable guidance to component integrators in building quality business component application systems.

References


Padmal Vitharana is a Ph.D. candidate in Management Information Systems at the University of Wisconsin-Milwaukee. Padmal’s research interests include reusable business components, software quality and process improvement, Internet and E-commerce, and group support systems. He received a B.S. in Computer Science and Mathematics and an M.B.A. from the University of Wisconsin-La Crosse. In addition, Padmal has worked in the software industry for over 4 years and has taught at the University of Wisconsin-Milwaukee.
Hemant Jain is Professor of Management Information System in the School of Business Administration at the University of Wisconsin-Milwaukee. Professor Jain is also a director of UWM MIS Consortium a partnership between the University and MIS community. Prof. Jain received his Ph.D. in Information System from Lehigh University in 1981, a M. Tech. in Industrial Engineering from I.I.T. Kharagpur (India) and B.S. in Mechanical Engineering from University of Indore (India). Prof. Jain’s interests are in the area of Electronic Commerce, System Development Using reusable components, Multi-Criteria Decision Making, Distributed and Cooperative Computing Systems, Architecture Design, Database Management and Data Warehousing, Data Mining and Visualization. He has published large number of articles in Information Systems Research, MIS Quarterly, IEEE Transactions on Software Engineering, Naval Research Quarterly, Decision Sciences, Decision Support Systems, Information & Management, Information Technology & Management and is book review editor for Journal of Information Technology Cases & Applications.