BoxScript: A Component-Oriented Language for Teaching

Yi Liu
Computer and Information Science
University of Mississippi
University, MS 38677
1-662-915-7602
liuyi@cs.olemiss.edu

H. Conrad Cunningham
Computer and Information Science
University of Mississippi
University, MS 38677
1-662-915-5358
cunningham@cs.olemiss.edu

ABSTRACT
As component-oriented approaches become increasingly pervasive in the development of complex software systems, it becomes increasingly important to introduce computing science students to appropriate programming concepts, languages and techniques. This paper describes the design of the component-oriented language BoxScript, which seeks to address the needs of teachers and students for a clean, simple language. This paper first enumerates the principles applied in the language design and then presents the key concepts and features of BoxScript. The paper illustrates the language features by using an example and by comparing it with several other component-oriented programming languages.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications – *classes and objects, modules, packages.*

General Terms
Languages.

Keywords
Component-oriented language, BoxScript, education.

1. INTRODUCTION
Component-oriented programming (COP) is a programming paradigm in which a software system can be built quickly and reliably by assembling a group of separately developed software components to form the system. The system should be flexible enough to be readily adapted to changing requirements by replacing, adding, or removing components.

Szyperski [19] defines a *software component* as follows:

> A software component is a unit of composition with a contractually specified interface and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

A component can be represented as shown in Figure 1. A component's internal design and implementation are strongly encapsulated and it exclusively communicates with other components through its interfaces. A *required interface* of a component can connect to a *provided interface* of another component when the provided interface conforms to the required interface. That leads to inter-component dependencies being restricted to individual interfaces rather than encompassing the whole component specification.

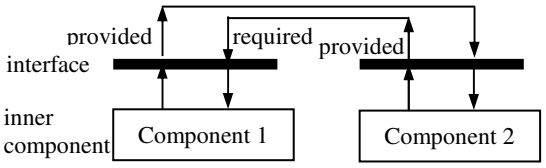

Figure 1. Components and Their Interconnections

Component-oriented programming has become a popular and important approach to software development. The concepts and technologies that support this approach should thus be taught to students in computing science and software engineering programs. Toward this end, the authors conducted an advanced software engineering class focused on component software [4]. The class presented a concept of component consistent with that of Szyperski and used a method for design similar to the "UML Components" approach [3]. From the experiences in the class, these were good choices. The component ideas could be presented in a way that was understandable to the students. However, the choice of the Enterprise JavaBeans (EJB) technologies [18] for practical exercises proved less satisfactory. The complexity of this technology tended to obscure the key component concepts and required a significant investment in time to master. The choice of an appropriate programming language is thus crucial to the task of introducing novices to the component concepts and design methodologies.

To introduce students to these component concepts and development methods effectively, the programming language should satisfy the following goals:

i. The language should be easy for students to learn.

ii. The language should embody the desired component concepts in a clean way. In particular, components should be strongly encapsulated modules, that is, syntactic structures that support information hiding approaches [14] to design and constrain interactions among the components to well-defined abstract interfaces [2] as shown in Figure 1.

iii. The language should be compositional. It should be possible to combine separately developed components to form a larger assembly such that the overall behavior can be accurately predicted from the known behaviors of its components.

43rd ACM Southeast Conference, March 18-20, 2005, Kennesaw, GA, USA. Copyright 2004 ACM 1-58113-000-0/00/0004…$5.00.

iv. The language should support the development of flexible programs that can be easily adapted to changing requirements with minimal impact upon the system. For example, it should be possible to substitute easily one component implementation for another that is equivalent in terms of the desired behaviors. It should be possible to define software families [15] where one group of components represents features common to all family members and other groups represent features that vary among family members.

Many component-based systems today are designed and built using object-oriented programming (OOP) languages and technologies. However, OOP does not fully satisfy the needs for development of complex component-oriented applications [20]. For example, implementation inheritance in OOP languages breaks encapsulation if used across component boundaries and also introduces safety issues, such as the *fragile base class* problem [12]; programmers must discipline themselves to avoid such usage when developing components. In addition, OOP languages do not provide efficient component composition strategies; programmers must write extra code to compose two or more components explicitly. Thus, OOP languages do not facilitate component development.

Few programming languages are designed to support COP. Most that do support COP are for commercial use and thus deal with some issues that are important in commercial systems but which may make the language difficult for novices to learn. For example, the authors used the EJB component technology for student programming projects in the component software class. EJB is a reasonable solution for commercial client/server systems. However, the complexity of the EJB technology and the lack of composition strategy meant it was not ideal for use in an academic course. The technology got in the way of teaching the students how to "think in components" cleanly.

Other existing COP languages do not seem to meet the goals stated above. Some languages, such as C# [6] and Component Pascal [13], are built on component concepts other than what we assume here. Jazzi [10] and ComponentJ [16] are two component-oriented programming languages built upon the same component model as shown in Figure 1. However, Jiazzi's scripting language provides explicit names for the composite components in a connection; programs tend to be inflexible [9]. ComponentJ gives provided and required interface specifications. It, however, gives the implementations of the provided interface methods in the component definition itself. That tends to make the component definition crowded and exposes too much of the implementation detail to client programmers, breaking the concept of information hiding.

There is a definite need for a component-oriented programming language that meets the requirements for teaching component-oriented programming to novices. This research aims to design a simple and clean language that meets the above goals. This programming language is called BoxScript. Section 2 enumerates the principles chosen for the design and Section 3 describes how those principles are realized in BoxScript. Section 4 outlines an example program and Section 5 concludes the paper.

2. PRINCIPLES

In order to achieve the goals that are enumerated in Introduction, we adopt several principles for the design of BoxScript.

Principle 1: The language builds upon an existing programming language with which students are likely to be familiar.

Nowadays, advanced OOP languages such as Java and C++ are the most commonly used languages in the computing science classes. To build the language on top of such a language helps students to pick up the new language quickly and easily.

Principle 2: Components are encapsulated modules that only expose information made available via operations on explicitly defined interfaces and that can be combined by a well-defined composition strategy such that the behavior of the composition can be determined from the combined behaviors of the components.

The language adopts the component model shown in Figure 1. The syntax of the language should present each component as an encapsulated entity with its implementation details hidden inside and a set of provided interfaces and required interfaces exposed outside. The syntax of the language should also be able to build the connections when two components communicate. The syntax of composition should be simple.

Principle 3: The syntactic structure of a component matches its run-time structure.

There is a one-to-one mapping between component declarations appearing in the system description and component instances created at runtime. This is a principle that results in a static structure that runs counter to the tendency toward dynamic structures in most OOP languages. Although a bit "retro" in concept, this restriction should help COP novices visualize a component-oriented system and readily comprehend the composition of components.

Principle 4: The language supports the construction of large-grained software families.

Parnas defines a program family as a set of programs "whose common properties are so extensive that it is advantageous to study the common properties before analyzing the individual members" [15]. In recent years, software families are typically called software product lines [1]. Given its properties, a component-oriented language should provide a good basis for the design and implementation of software product lines. In particular, it should support the design of software frameworks [17]. In this context, a software framework is a reusable design captured as a set of inter-related component specifications.

Principle 5: The behavior of the component and composition features of programs can be specified formally using design-by-contract techniques.

Although the syntax of a language is important, it is the semantics of the language that describes what a program really does. The syntax should steer programmers into the construction of programs that are semantically correct. However, it is desirable for designers and programmers to be able to state the specification for a component or composition of components in a manner that supports formal reasoning and runtime monitoring. The design-by-contract technique is a frequently used approach [8,11] that can be adapted to work with the COP languages [5].

3. BOXSCRIPT DESIGN

BoxScript is a Java-based, component-oriented programming language whose design seeks to address the needs of teachers and

students for a clean, simple language. The design of BoxScript follows the principles that are listed above.

3.1 Design for Principle 1

Java has become the primary teaching language in computing science education in recent years and, hence, most students know it. To satisfy the first principle, we choose Java as the base language and build BoxScript on top of it.

3.2 Design for Principle 2

The component model for BoxScript is shown in Figure 1. In BoxScript, a component is called a *box*. A box is a strongly encapsulated module that hides its internal details while only exposing its interfaces. Each box has a corresponding *box description* (.box) file that gives the needed declarations. A box might have a *configuration information* file for specifying information in addition to what is described in its box description.

3.2.1 Interfaces

There are two types of interfaces in BoxScript. A *provided interface* describes the operations that a box implements and that other boxes may use. A *required interface* describes the operations that the box requires and that must be implemented by another box. A box has one or more provided interfaces and zero or more required interfaces. In BoxScript, we use *interface type* to refer to a general interface with method declarations and we use *interface handle* to refer to each unique occurrence of an interface in a box. An interface handle has an interface type. To be compatible with Java, a BoxScript interface type is represented syntactically by a Java interface, that is, by a set of related operation signatures, and we use Java classes to implement the interfaces. Figure 2 shows interfaces Price and Discount.

```
public interface Price
{  double getPrice(int client, int item, int quantity);  }
```

Figure 2.a Interface Price

```
public interface Discount
{  double getDiscount(int client, int item, int quantity); }
```

Figure 2.b Interface Discount

In a box description, the declaration for a provided interface begins with the keyword provided interface and specifies the interface's type and handle. The declaration for a required interface has the same syntax except that it begins with the keyword required interface.

3.2.2 Composition Strategy

In BoxScript, separately developed boxes can be composed into *compound boxes* according to the following rules. (The following subsections define the concept of compound box more fully.)

i. A required interface of a box may connect to a provided interface of another box as long as the provided interface *satisfies* the required interface. We say interface x *satisfies* interface y if and only if x syntactically *extends* y in Java and the operations match semantically. (That is, x is a behavioral subtype of y [5].)
ii. All the provided interfaces of the constituent boxes are hidden unless explicitly exposed by the compound box.
iii. A compound box must expose every required interface of its constituent boxes that is not connected to a provided interface of another constituent box.

The composition process in BoxScript is illustrated in Figure 3. Suppose we wish to compose Box1 and Box2 into a compound box Box1_2. (In the following, Box1 really means the instance of Box1 and Box2 means the instance of Box2.) Assume that P11 satisfies R21, so we can connect P11 and R21. Similarly, we can connect P21 and R13. In this way, Box1 and Box2 are "wired" together. We choose for Box1_2 to expose provided interface P12 from Box1 and hide provided interface P11 from Box1 and P21 and P22 from Box2. Similarly, we choose to expose required interfaces R11 and R12 from Box1 and R22 from Box2 as required interfaces of Box1_2.

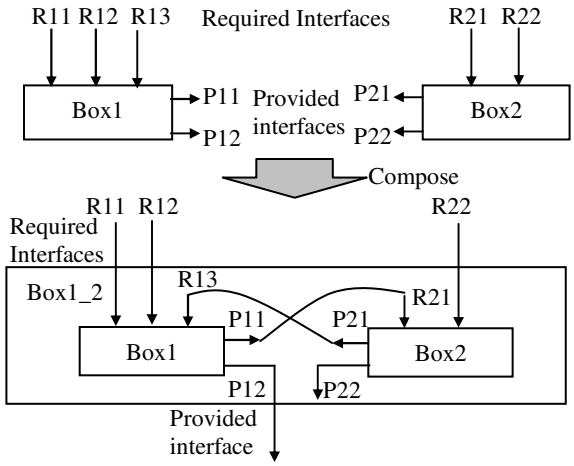

Figure 3. Composition Process

A box is strongly encapsulated with only its interfaces exposed. The behavior of a box remains unchanged unless its state is changed. The state of a box can only be changed when an action is performed on a provided interface method. A provided interface satisfying a required interface guarantees that the results of calls to the methods of a required interface are expected by the box with no side effects. Thus, the behavior changes of a box are predictable.

3.2.3 Boxes

BoxScript has three kinds of boxes: *abstract, atomic,* and *compound*.

An *abstract box* is a box that describes the provided and required interfaces but does not implement the provided interfaces. An abstract box should be implemented by concrete boxes, i.e., atomic or compound boxes. In the box description, an abstract box is identified by a keyword abstract followed by its box name, and is specified with its provided interface and required interface descriptions. Figure 4 shows an abstract box PricingAbs.

```
abstract box PricingAbs
{  provided interface Price Pr;
        // Pr is the handle for Interface Price
   required interface Discount Dc;
        // Dc is the handle for Interface Discount
}
```

Figure 4. PricingAbs.box

An *atomic box* is a basic element in BoxScript. It does not contain any other boxes. The description of an atomic box gives the box's name and the name of any abstract box that it implements. It also gives its provided and required interfaces by listing their interface types and handles. Figure 5 shows the box description for atomic

box Pricing that implements the abstract box PricingAbs. An atomic box must supply an implementation for each provided interface, that is, a Java class that implements the interface type. By default, the file name of the Java class is formed by suffixing Imp to the provided interface handle name. However, programmers may choose different file names other than the default ones; in this case, the file name needs to be specified in the *configuration information* file as a pair consisting of the interface handle and the Java class file name that implements the interface type of the handle.

```
box Pricing implements PricingAbs
{  provided interface Price Pr;
   required interface Discount Dc;
}
```

Figure 5. Pricing.box

A *compound box* is composed from atomic boxes or other compound boxes. It does not implement its provided interfaces, but uses the implementations provided by its constituent boxes. In the box description for a compound box, each constituent box is given an identifier, called its *box handle*, to enable it to be uniquely identified as a participant within the composition. The box description for a compound box not only supplies the information given in the atomic box, but also specifies (1) the boxes from which the box is composed from, (2) the interface sources from which the provided and required interfaces come, and (3) the information about how provided and required interfaces connect to one another.

```
abstract box CalPriceAbs
{  provided interface Price tPrice;
}
```

Figure 6.a CalPriceAbs.box

```
box CalPrice implements CalPriceAbs
{  composed from PricingAbs boxP, DiscountingAbs boxD;
     // boxP is the box handle for PricingAbs
     // boxD is the box handle for DiscountingAbs
   provided interface Price tPrice from boxP.Pr;
   connect boxP.Dc to boxD.Dis;
}
```

Figure 6.b CalPrice.box

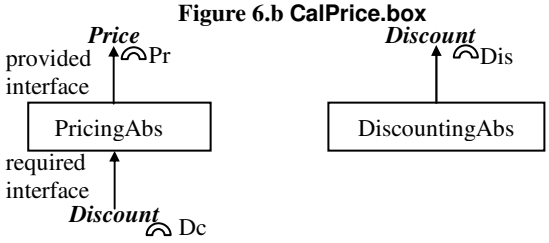

Figure 7.a PricingAbs and DisCountingAbs

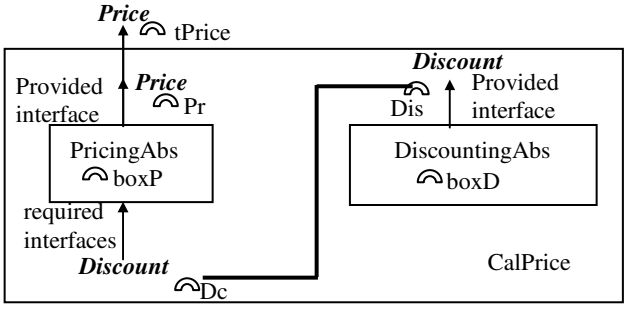

Figure 7.b Composition

Figure 6 shows a compound box CalPrice that implements an abstract box CalPriceAbs. CalPrice is composed from boxes Pricing and Discounting. To enable flexibility in the program, we use abstract boxes Pricing and Discounting in defining the composition. This is discussed further in Section 3.4. BoxScript uses the box handles to expose and connect the interfaces of the constituent boxes. The composed from declaration in CalPrice assigns boxD and boxP as the box handles for DisCountingAbs and PricingAbs, respectively. The provided interface of CalPrice comes from interface Price (handle Pr) of PricingAbs (handle boxP). The provided interface declaration gives the information on the interface type and the interface handle and its source (a box handle and interface handle associated with a constituent box). The connect statement connects a required interface of a box to a provided interface of another box. In this example, the connect statement shows that box PricingAbs's (handle boxP) required interface Discount (handle Dc) to box DiscountingAbs's (handle boxD) provided interface Discount (handle Dis). Figure 7 shows the inner composition of CalPrice.

3.2.4 Inheritance

To preserve the encapsulation of boxes, BoxScript disallows implementation inheritance across box boundaries. It does not necessarily encourage the use of implementation inheritance inside the implementation of a box, but BoxScript does not forbid its use. If this kind of usage can simplify the code and not bring serious side effects, implementation inheritance might be used.

3.3 Design for Principle 3

Each concrete box has *box manager code* that initializes the box. The box manager code consists of a Java class with the same name as the box; it is generated by the BoxScript compiler. For an atomic box, the manager code consists of a constructor for the atomic box object and code that instantiates the interface handle objects. For a compound box, the manager code consists of a constructor for the compound box object and code that instantiates the boxes that comprise the compound. Each box handle results in creation of a separate constituent box instance. The compound box's manager object retrieves its interface handle references from the constituent boxes that implement them. Thus there is exactly one box manager object for each box handle that occurs in the BoxScript program.

3.4 Design for Principle 4

The concepts of abstract boxes and variants bring flexibility into BoxScript programming.

A concrete box can be either an implementation of an abstract box or a standalone box that has no related abstract box. For the former case, all the implementations of an abstract box are considered to be *variants* of the box; one variant of a box can be substituted for another. For the latter case, the atomic or compound box is considered to have no variant. An abstract box can extend another abstract box and the former one is considered to be a variant of the latter one.

A variant B of abstract box A must conform to abstract box A.

- Box B *conforms* to box A if and only if
  (∀p: p ∈ provided interfaces(A):
  (∃q : q∈ provided interfaces (B) : q extends p)) and
  (∀r: r ∈ required interfaces(B):
  (∃s : s∈ required interfaces (A) : s extends r)).

- Box interface x *extends* box interface y if and only if interface handle (x) = interface handle(y), type (x) extends type(y) in Java, and x is a behavioral subtype of y [5].

That is, the provided interfaces of B should provide at least the operations of A, and the required interfaces of B should be at most as much as that of A. For example, BoxA has provided interfaces Pa, Pb and Pc and required interfaces Ra and Rb; BoxB has provided interfaces Pa, Pb, Pc and Pd and required interface Ra. We say that BoxB conforms to BoxA.

A compound box is normally defined to be composed from abstract boxes instead of concrete boxes. If a concrete variant of an abstract box preserves the semantics of the abstract box's operations, it may be substituted for an occurrence of the abstract box when the system executes. This substitution is given in the configuration information file as a pair (abstract box handle, concrete box name).

Consider the example in Section 3.2. Suppose that abstract box PricingAbs is an abstraction that represents the family of components for assigning prices to items for a business. Concrete box Pricing, which implements PricingAbs, is one particular pricing policy that has been implemented. In the configuration file for compound box CalPrice, we specify pair (boxP, Pricing) to indicate that Pricing is used for PricingAbs (which has box handle boxP) when CalPrice executes. Suppose that the pricing policy changes and a new variant Pricing1 of abstract box PricingAbs is developed. If we wish to change to this new variant, we only need to change the configuration file for CalPrice to give the new variant's name.

Note that compound box CalPrice is composed from two abstract boxes, PricingAbs and DiscountingAbs. The latter is an abstraction that represents the family of components for assigning price discounts. Thus CalPrice itself represents a software family. The interfaces of the three boxes and the defined connection between the two constituent boxes represent the common aspects (i.e., frozen spots [17]) of the family. The family has two primary points of variability (i.e., hotspots [17]) represented by the two constituent boxes. Furthermore, CalPrice itself is a variant of abstract box CalPriceAbs, providing another layer of variability in a system that includes CalPriceAbs.

Of course, a compound box can be defined to have concrete boxes as its constituents. These might be concrete variants of some abstract box, or they might be standalone concrete boxes (i.e., boxes that do not extend an abstract box). This reduces the flexibility of the system because the constituent boxes have no variants. If a box is replaced by a newer version, the new version must have the same name as the original box.

3.5 Design for Principle 5

The BoxScript language is based on strongly encapsulated modular units with a well-defined composition strategy. It thus seems to be amenable to the application of formal specification and verification techniques. Some preliminary ideas on this topic are discussed in a separate paper [5].

4. BOXSCRIPT APPLICATION

Consider the pricing application introduced in section 3. Figure 8 shows the structure of the CalPrice software family. The design adapts the *strategy design pattern* [7] to this pricing application. Applying a discount is a hotspot in the price calculation since different strategies might be needed within the same pricing structure. Pricing delegates the work of determining the discount to abstract box DiscountingAbs that can be implemented by different concrete boxes to present different strategies. The CalPrice family members differ from each other by implementing abstract boxes PricingAbs and DisCountingAbs in different ways. A possible member of CalPrice is to have Pricing (Figure 5) implement PricingAbs (Figure 4) and to have Discounting (Figure 10) implement DiscountingAbs (Figure 9).

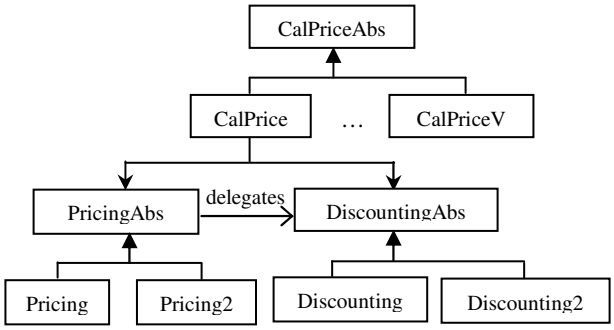

Figure 8. CalPrice family

```
abstract box DiscountingAbs
{  provided interface Discount Dis; }
```

Figure 9. DiscountingAbs.box

```
box Discounting implements DiscountingAbs
{  provided interface Discount Dis; }
```

Figure 10. Discounting.box

```
public class PrImp implements Price
{  private BoxTop _box;
   Discount dc;    // required interface
   public PrImp(BoxTop myBox)
   {   _box = myBox;
       InterfaceName name = new InterfaceName("Dc");
       dc = (DisCount)_box.getRequiredItf(name);
   }
   public double getPrice(int client,int item,int quantity)
   {  double disc = dc.getDiscount(client, item, quantity);
      return PriceList.p[item] * (1 - disc* 0.01) * quantity;
   }
}
```

Figure 11. PrImp.java

```
public class DisImp implements Discount
{  private BoxTop _box;
   public DisImp(BoxTop myBox)
   {   _box = myBox;          }
   public double getDiscount(int client, int item, int quantity)
   {   return discountPolicy(PriceList.p[item]);  }
   private double discountPolicy(double price)
   {   if (price < 100.00)
               return 0.0;  // price < 100, no discount
        return 10.00; // price >= 100, discount 10%
   }
}
```

Figure 12. DisImp.java

```
final class PriceList extends Object
{  public static double [] p = new double []
   { 120.00, 14.45, 16.99, 23.78, 130.89, 239.99 };
}
```

Figure 13. PriceList.java

The purpose of this example is to illustrate how an application (and software family) can be built in BoxScript. So this example ignores the details of the policies and the implementations of discounting and pricing and keeps the definitions and implementations of the interfaces simple. Figure 2 shows the two interface types that participate in the boxes. In the methods shown, integer variables client and item uniquely identify the customer and item being purchased, respectively, and quantity gives the number of items being purchased. In the implementations of both interfaces, we assume that the item argument is valid and ignore the possibility of applying different discounts to different customers. Atomic box Pricing supplies the implementation (shown in Figure 11) of interface type Price. This implementation uses the default filename, that is, Pr, the interface handle of Price, combined with suffix Imp. Figure 12 shows the implementation of the atomic box Discounting. Classes BoxTop and InterfaceName that appear in both implementations are abstract classes from BoxScript runtime system. Every box manager extends BoxTop. Figure 13 shows a helper class used within the implementation of the Price interface of the Pricing box.

The configuration file for a compound box must indicate which variant is to be used for a constituent abstract box. Figure 14 gives a configuration file for compound box CalPrice that binds Pricing to PricingAbs (handle boxP) and Discounting to DiscountingAbs (handle boxD).

```
(boxP, Pricing)
(boxD, Discounting)
```

Figure 14. CalPrice.conf

5. CONCLUSION

BoxScript is a language designed to make the concepts of component-oriented programming accessible to students. By following the principles listed, the BoxScript design supports the major component concepts and exhibits the primary aspects of component-based development. It provides a simple, Java-based language syntax and a clean working environment. The authors are developing a prototype and plan to use it to support the teaching of a module in a 2005 spring semester class.

There are several areas for future work. First, to keep the language clean in thought and simple in use, the BoxScript prototype does not support distributed components. However, future development will seek to extend the model to handle distributed computing issues. Second, future work will seek to integrate BoxScript with techniques and tools such as those associated with the Java Modeling Language (JML) [8] to enable use of design-by-contract techniques.

6. ACKNOWLEDGMENTS

This work was supported, in part, by a grant from Acxiom Corporation titled "The Acxiom Laboratory for Software Architecture and Component Engineering (ALSACE)."

7. REFERENCES

[1] Ardis, M., Daley, N., Hoffman, D., Siy, H. and Weiss, D. Software Product Lines: A Case Study. *Software– Practice and Experience*, Vol. 30, pp. 825-847, 2000.

[2] Britton, K. H. , Parker, R.A. and Parnas, D.L. A Procedure for Designing Abstract Interfaces for Device Interface Modules. In *Proceedings of the 5th International Conference on Software Engineering*, pp. 195-204, March, 1981.

[3] Cheesman, J. and Daniels, J. *UML Components: A Simple Process for Specifying Component-Based Software.* Addison Wesley, 2001.

[4] Cunningham, H. C., Liu, Y., Tadepalli, P. and Fu, M. Component Software: A New Software Engineering Course. *Journal of Computing Sciences in Colleges*, Vol. 18, No. 6, pp. 10-21, June 2003.

[5] Cunningham, H. C., Liu, Y. and Tadepalli, P. Toward Specification and Composition of BoxScript Components. In *Proceedings of the Workshop on Specification and Verification of Component-Based Systems (SAVCBS)*, pp. 114-117, November 2004.

[6] Deitel, H. M. and Deitel, P. J. *C#: How to Program*. Prentice Hall, 2003.

[7] Gamma, E., Helm, R. , Johnson, R. and Vlissides, J. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, 1995.

[8] Leavens, G. T. and Cheon, Y. Design by Contract with JML. Draft paper, Iowa State University, August 2004.

[9] Lüer, C. *Environments for Deployable Components*. Technical Report #02-15, Dept. of Information and Computer Science, University of California, Irvine, May 2002.

[10] S. McDirmid, M. Flatt, and W. C. Hsieh. "Jiazzi: New-age Components for Old-fashioned Java," In *Proceedings of the 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications,* pp. 211-222, 2001.

[11] Meyer, B. *Object-Oriented Software Construction*. Second Edition, Prentice Hall, 1997.

[12] L. Mikhajlov and E. Sekerinski. *The Fragile Base Class Problem and Its Solution*. Technical Report 117, Turku Centre for Computer Science, Finland, June 1997.

[13] Oberon Microsystems, Inc. *Component Pascal Language Report*. May 1997.

[14] Parnas, D. L. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, Vol. 15, No. 12, pp. 1053-1058, December 1972.

[15] Parnas, D. L. On the Design and Development of Program Families. *IEEE Transactions on Software Engineering*, Vol. SE-2, pp. 1-9, March 1976.

[16] Seco, J. C. ComponentJ in a NutShell. http://ctp.di.fct.unl.pt/~jcs/bibIndex/papers/ComponentJ.pdf. Last accessed: 24 Jan 2004.

[17] Schmid, H. A. Systematic Framework Design by Generalization. *Communications of the ACM*, Vol. 40, No. 10, pp. 48-51, October 1997.

[18] Singh, I., Stearns, B., Johnson, M. and the Enterprise Team. *Designing Enterprise Applications with the J2EE™ Platform*. Second Edition. Addison Wesley, 2002.

[19] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Second Edition. Addison Wesley, 2002.

[20]Udell, J. Componentware**.** *BYTE,* pp. 46-56, May 1994.