Java in the Box: Implementing the BoxScript Component Language

Yi Liu
Dept. of EE and Computer Science
South Dakota State University
Brookings, SD 57007
1-605-688-5280
yi.liu@sdstate.edu

H. Conrad Cunningham
Dept. of Computer and Information Science
University of Mississippi
University, MS 38677
1-662-915-5280
cunningham@cs.olemiss.edu

ABSTRACT
BoxScript is a Java-based language that supports the component-oriented programming paradigm. BoxScript introduces a composition strategy and type structure to support two main properties of component-oriented programming, compositionality and flexibility. This paper briefly introduces the fundamental concepts of BoxScript and then describes how BoxScript components (i.e., boxes) are realized as clusters of interrelated Java interfaces, classes, and packages.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *frameworks, modules, packages.*

General Terms
Design, Languages.

Keywords
Component-oriented language, BoxScript, interface, runtime structure, Java implementation.

1. INTRODUCTION
Component-oriented programming (COP) is a programming paradigm that seeks to build software systems quickly and reliably by assembling groups of independently developed software components. The most used definition of component is given by Szyperski [7], who 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.

As shown in Figure 1, the internal design and implementation of a software component are hidden behind its *provided* and *required interfaces*. A component implements its provided interfaces and makes them available for use by other components. In implementing its functionality, a component may use other components only through its required interfaces. During assembly, a required interface of a component must be connected to a matching provided interface of another component. Thus components depend upon the specifications of interfaces, not upon the specifications of other whole components.

ACMSE'07, March 23-24, 2007, Winston-Salem, North Carolina, USA.

Components are strongly encapsulated and compositional. This leads to two key properties of component-oriented programming– *compositionality* and *flexibility*.

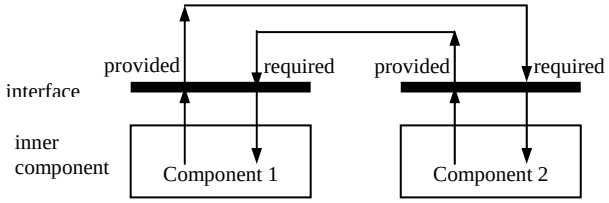

Figure 1. Components and their interconnections

Compositionality is the most distinctive property of component-oriented programming. An application is built by assembling components, each of which is sufficiently independent from the others to allow it to be developed individually without knowledge of the others' implementations. Composing such components into a system requires the selection of suitable components and the right strategies for assembly. When a subsystem is assembled from components, its behavior should be predictable based on the behavioral specifications of the components.

The flexibility property of a component-oriented system means that adapting the system to changing requirements is not difficult. It can be accomplished by replacing, adding, or removing a few components with minimal impact on the clients of the system.

By providing these two properties, a COP language can deliver a reasonable means for developing large-grained software systems. However, currently popular approaches to component-oriented programming have little language support.

BoxScript is a new Java-based language that responds to this need [3,4]. It introduces a composition strategy and a novel component type system to support the desired levels of compositionality and flexibility, respectively. Java code provides the computational notation and the BoxScript concepts enable the computational units to be composed flexibly into systems.

This paper focuses on how the BoxScript components are realized in Java. Section 2 introduces the BoxScript language. Section 3 presents the implementation of components, focusing on their runtime structure and their Java implementations. Section 4

discusses this work and addresses some possible future research. Section 5 concludes the paper.

2. BOXSCRIPT

The component concepts in the BoxScript language are based on Figure 1. A component is called a *box*. A box is a blackbox entity; that is, it strongly encapsulates its internal details, only exposing its interfaces to the outside. A box has the functionality needed to satisfy some requirement and can be used either individually or as a part of a larger box that contains it. A group of boxes can be composed to form a larger box that provides some higher level functionality.

The units of code needed to build a box are a *box description*, *interfaces* and their *implementations*, *configuration information*, and *box manager code*. A box description declares the features of the box (in a file with extension `.box`). The configuration information file gives some additional assembly information. The box manager is a Java class, with the same name as the box, that is generated by the BoxScript compiler [3, 4].

The example used in the following is a simple system called `CalPrice` that calculates the prices of items while applying a discounting policy. The system is componentized into four main components—`Pricing`, `Discounting`, `Storage`, and `CalPrice`. Component `Pricing` calculates the price of an item based on its original price and the discounting policy. Component `Storage` delivers the original price for an item and other necessary information from the data storage. Component `Discounting` applies the discounting policy to calculate the applicable discount for an item. The component `CalPrice` is assembled from `Pricing`, `Discounting`, and `Storage` to deliver the function of calculating the discounted prices of items.

2.1 Interfaces

BoxScript defines two types of interfaces. 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. BoxScript uses the term *interface type* to refer to a general interface with method declarations and uses the term *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. BoxScript uses Java classes to implement the interfaces. BoxScript introduces two relations between box interfaces: *extension* and *satisfaction.*

A box interface `x` *extends* box interface `y` (syntactically) if and only if type(`x`) = type(`y`) or type(`x`) extends type(`y`) in the Java type system. That is, all the operation signatures in `y` also appear in `x`, but `x` may have additional operations. Type extension does not allow covariant or contravariant changes [5] to operations.

Box interface `x` *satisfies* interface `y` when `x` provides at least the operations required by `y` and the operations of `x` have an equivalent meaning to the matching operations in `y`. To add more precision, let `I(x)` be an invariant for `x` and `pre(x,m)` and `post(x,m)` refer to the precondition and postcondition, respectively, for operation `m` on interface `x`. Then, box interface `x` *satisfies* box interface `y` if and only if `x` extends `y` [1]:

- `I(x) & C(x,y) ⇒ I(y)`
- `(∀m : m ∈ y :`
  `(pre(y,m) & I(y) & C(x,y)⇒ pre(x,m))&`
  `(pre(x,m)& post(x.m)& I(x) & C(x,y)`
  `⇒ post(y,m)))`

Assertion `C(x,y)` is a coupling invariant that relates the equivalent aspects of the information models for `x` and `y`.

The above definition of *satisfaction* is motivated by Meyer's treatment of inheritance in the design by contract approach (and the Eiffel language) [5] and the concept of a coupling invariant in program and data refinement [6].

Figure 2 shows interface types `Price`, `Discount`, and `PriceRetrieve` for the example.

```
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`

```
public interface PriceRetrieve
{  double itemPrice (int item);
   int total(int item);
   boolean clientExist (int client);
   boolean itemExist (int item);
}
```

Figure 2.c. Interface PriceRetrieve

2.2 Boxes

There are two kinds of boxes in BoxScript: *abstract* and *concrete.* An abstract box serves as an abstract type for which no implementation is provided. A concrete box provides implementation that delivers concrete functions.

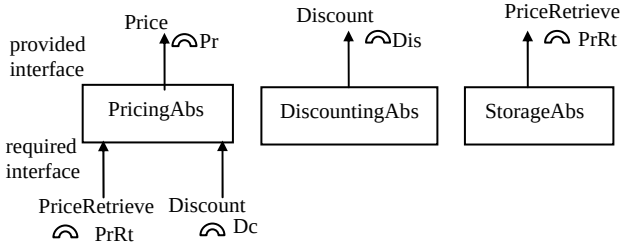

Figure 3. Abstract boxes

An abstract box is a box that describes the provided and required interfaces but does not implement the provided interfaces. 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 3 shows the structure of abstract box `PricingAbs`, `StorageAbs`, and `DiscountingAbs`. Figure 4 gives the box description for `PricingAbs`. `PricingAbs` requests the original price of an item from its required interface `PriceRetrieve` and the applicable discount from its required interface `Discount`. It provides interface `Price` for calculating the price after discounting.

A concrete box is either atomic or compound. 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. In the example, atomic boxes `Pricing`, `Storage`, and `Discounting` implement the abstract boxes `PricingAbs`, `DiscountingAbs`, and `StorageAbs`, respectively. Figure 5.a shows the atomic box `Pricing`. An atomic box must supply an implementation for each provided interface, i.e., a Java class that implements the interface type. Figure 5.b shows partial code for the implementation of its provided interface `Price`.

```
abstract box PricingAbs
{  provided interface Price Pr;
   //Pr is the handle for interface Price
   required interface Discount Dc,
               PriceRetrieve PrRt;
 //Dc and PrRt are handles for Discount and PriceRetrieve, respectively
}
```

Figure 4. PricingAbs.box

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

Figure 5.a. Atomic box description Pricing

As a concrete box, an atomic box has a box manager that is automatically generated during box compilation. The box manager code includes a constructor for the atomic box object and code that instantiates the interface handle objects. The box manager is a Java file with the same name as the box.

```
package boxes.PricingAbs.Pricing;
import interfaces.Price;
import interfaces.Discount;
import interfaces.PriceRetrieve;
import datatypes.systems.*;
public class PrImp implements Price
{ private BoxTop _box;
  Discount dc;         // required interface
  PriceRetrieve prRt; //required interface
  public PrImp(BoxTop myBox)
  { _box = myBox;
    InterfaceName name= new InterfaceName("Dc");
    dc = (DisCount)_box.getRequiredItf(name);
    name= new InterfaceName("PrRt");
    prRt=(PriceRetrieve)_box.getRequiredItf(name);
  }
  public double getPrice(int client,
       int item, int quantity)
  {            …
    double price = prRt.itemPrice (item);
    double disc= dc.getDiscount(client, item,
                      quantity);
    return price * (1 - disc* 0.01) * quantity;
  }
}
```

Figure 5.b. PrImp.java – Implementation of Interface Price

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, 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 it 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.

The valid relationship between a concrete box `B` and an abstract box `A` that it implements is that box `B` conforms to box `A`. Clearly, if abstract box `A` specifies the presence of a provided interface `p`, then concrete box `B` *must* have a provided interface that satisfies `p`. If concrete box `B` has a required interface `r`, then abstract box `A` *must* specify a required interface that satisfies `r`. In terms of operations, the provided interfaces of `B` should supply at least the operations of `A`, and the required operations of `B` should be at most those of `A`. A similar situation occurs if one considers an abstract box extending another abstract box [1].

More formally, box `B` *conforms* to box `A` if and only if:

```
I(B) & C(A,B) ⇒ I(A)
(∀p: p ∈ prov(A): (∃q: q ∈ prov(B):
   handle(q) = handle(p) & q satisfies p))
(∀r: r ∈ req(B):(∃s: s ∈ req(A):
   handle(r) = handle(s) & s satisfies r))
```

Above, `req(A)` refers to the required interfaces of box `A` and `C(A,B)` denotes a coupling invariant for the refinement of the information model when replacing `A` by `B`. In particular, `C(A,B)` serves as the coupling invariant for showing that the interfaces of `B` have the needed satisfaction relationship with the corresponding interfaces of `A`. The notation `handle(p)` refers to the interface handle of interface `p`.

The compound box is BoxScript's composition mechanism. The compound box is the only type of box that needs composition. The composition operation follows the composition strategy.

2.3 Composition Strategy

Compositionality requires that the composition be safe and the functionality of the composed component to be predictable. The following defines the composition of boxes in BoxScript [1, 3, 4]:

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.
ii. All the provided interfaces of the constituent boxes are hidden unless 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.

Figure 6 illustrates the *composition* process for compound box `CalPrice`. `CalPrice` is composed from three other boxes. The example uses abstract boxes to participate in composition to enable flexibility (discussed in section 2.4). `PricingAbs` has required interfaces `PriceRetrieve` and `Discount`, which are satisfied by the provided interface `PriceRetrieve` of component `StorageAbs` and the provided interface `Discount` of component `DiscountingAbs`, respectively. The example wires together boxes `PricingAbs` and `DiscountingAbs` and boxes `PricingAbs` and `StorageAbs`. It exposes the provided interface of `PricingAbs` to the outside. No required interface needs to be exposed since all of them are satisfied. Figure 7 shows the box description for `CalPrice`.

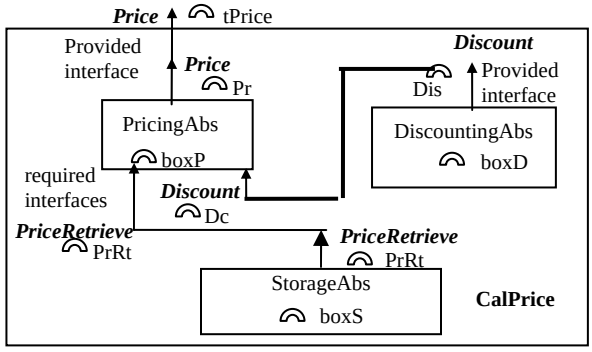

Figure 6. Composition process

```
box CalPrice implements CalPriceAbs
{  composed from PricingAbs boxP,
        DiscountingAbs boxD,StorageAbs boxS;
 // boxP, boxD and boxS are the box handles for PricingAbs,
 // DiscountingAbs, and StorageAbs, respectively
   provided interface Price tPrice from boxP.Pr;
   connect boxP.Dc to boxD.Dis,
           boxP.PrRt to boxS.PrRt;
}
```

Figure 7. CalPrice.box

2.4 Flexibility Strategy

The concepts of abstract boxes and *box 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 abstract box; one variant of a box can be substituted for another in any context that calls for the abstract box. 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.

To support flexibility, a compound box is normally defined to be composed from abstract boxes instead of concrete boxes. The abstract box name allows the ability to plug in different variants of the constituent 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. If a programmer wishes to change to this new variant, he or she only needs to change the configuration file for that abstract box to give the new variant's name.

```
(boxP, "\warehouse\boxes\PricingAbs",
          "Pricing\Pricing");
(boxD, "\warehouse\boxes\DiscountingAbs",
           "Discounting\Discounting");
(boxS, "\warehouse\boxes\StorageAbs",
           "Storage\Storage");
```

Figure 8. CalPrice.conf -Configuration file for CalPrice

At runtime, concrete boxes must replace the abstract boxes. The configuration file for a compound box must indicate which concrete box is to be used for a constituent abstract box. The configuration information contains: the box handle of the abstract box to be configured, the physical location for the abstract box, and the physical location for the concrete box. Figure 8 shows an example configuration file for compound box `CalPrice` that binds `Pricing` to `PricingAbs` (handle `boxP`), `Discounting` to `DiscountingAbs` (handle `boxD`), and `Storage` to `StorageAbs` (handle `boxS`).

3. BOXSCRIPT IMPLEMENTATION

3.1 Box Source Code and Compilation

There are five kinds of code units necessary for building a box:

a) a Java interface definition for each provided and required interface,
b) a BoxScript box description, a file named with the same base name as the box plus extension `.box`,
c) a Java class to implement each provided interface,
d) configuration information, a file named with the same base name as the box plus extension `.conf`,
e) the box manager class for the box, a Java file whose base file name is the same as the box name.

Both abstract and concrete boxes have interface files and box description files. Only atomic boxes provide implementations for its provided interfaces. A compound box should have a configuration information file to specify the location of its constituent boxes. Each concrete box has a box manager, while an abstract box does not. All the files associated with a particular box, abstract or concrete, are separated into a Java package.

BoxScript is built on top of Java. Programmers express the box interfaces and their implementing classes directly in Java. Box descriptions, including the composition and flexibility strategies, must be translated from the BoxScript notation into Java code. A *box manager* is the bridge that connects a BoxScript box to the Java implementations of the box's interfaces. The box manager is a Java class whose function is to organize the interface references and box references at runtime. Each concrete box has a corresponding box manager class.

The BoxScript compiler, called BoxCompiler, takes the box description and other necessary files as input, checks the syntax and interface conformity, translates the BoxScript code into Java, and then delegates the translation of the generated Java code to the Java compiler. Once compiled by the Java compiler, a BoxScript program can run on the Java Virtual Machine (JVM). The BoxCompiler itself is written in BoxScript, having been bootstrapped from an initial version written in Java [4].

3.2 Box Runtime Structure

To the Java compiler, a BoxScript box is a package holding a cluster of related Java classes and interfaces. At runtime, the box is represented by a cluster of instances of these classes executing on the JVM. The runtime system manages the instantiations of these classes and the linking of the box's required interfaces with provided interfaces of other boxes. This section examines the runtime structure of BoxScript concrete boxes. Abstract boxes are compile-time structures in the current implementation.

An atomic box has two kinds of Java class files—a class for its box manager and a class for the implementation of each provided interface. When a BoxScript atomic box is instantiated, what really occurs in the Java-based runtime environment is that an instance of the box manager class is instantiated. The box manager object creates instances of the interface implementation class for each provided interface. As needed, the box manager for an atomic box obtains references to the objects in other boxes that implement the required interfaces.

A compound box manager instantiates the box manager objects for its constituent boxes (at this time, the concrete boxes replace the corresponding abstract boxes), connects their required interfaces to the provided interfaces, passes required interface references to its constituent boxes, and gets the provided interface references from its constituent boxes.

The prototype BoxScript system uses a lazy strategy for interface object instantiation. The implementation class for a provided interface will not be instantiated until the first request for this interface. If an atomic box manager gets a request to provide the reference of one of its provided interfaces, it checks whether the interface object reference is null or not. If the object reference is not null, it returns the reference; otherwise, it instantiates the interface implementation class and returns the reference. If a compound box gets a call on one of its provided interface, it checks the interface object reference. If the object reference is not null, it returns the reference; otherwise, it will pass the request for getting the interface object reference to the constituent box that provides this interface. If no object reference is found, this request passing continues until it meets an atomic box, which instantiates the interface implementation class and returns the reference. This reference is passed back though the path the request through which it came, and the reference is set to the provided interface of each box on the path.

When a required interface reference for a compound box is set, the box passes the interface reference to the constituent box that requires the interface.

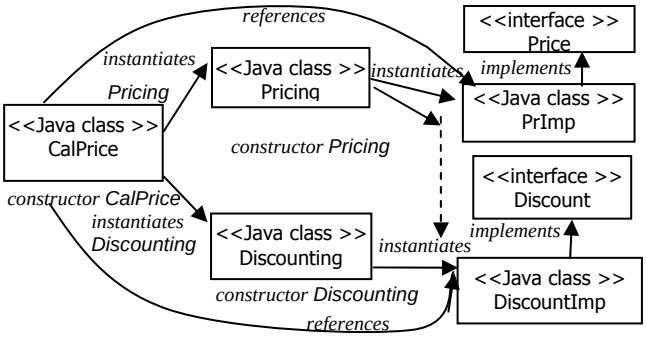

Figure 9. Box Manager and Interface Object Instantiation

We use the `CalPrice` system to illustrate the box manager object and provided interface object instantiation. `CalPrice`'s box manager object is created when the system is invoked. `CalPrice`'s box manager instantiates the box manager objects for its constituent boxes, `Pricing` and `Discounting`, as shown in Figure 9. `Pricing`'s box manager object instantiates `PrImp`, the implementation class of its interface `Price`, when the `Pricing` object is requested to provide the reference for `PrImp`. Similarly, the `Discounting` box manager object instantiates `DiscountImp` when the `Disounting` object is requested to provide the reference for `DiscountImp`. `CalPrice` gets the reference to `DisImp` in `Discounting` and sets it to the reference of `Discount` in `Pricing` when it builds the connection.

3.3 Box Manager Implementation

The box manager class for a box is created by the BoxCompiler based on the box description file and the associated configuration file. For an atomic box, the information for its provided and required interfaces is given in its box description. Some optional implementation information may be given in its configuration file. For a compound box, the box description gives information not only on the interfaces but also on the composition of the constituent boxes. Its configuration file gives the information on the concrete boxes that implement the abstract boxes participating in the composition. Given box descriptions and configuration files, the box manager class can be generated.

To create the needed runtime structure, the box manager class `B.java` for a concrete box `B` should have four parts as follows.

- a `package` *declaration* to declare the filesystem subdirectory under which the `B.java` file is located.
- `import` *statements* to include interface declarations for the box's provided and required interfaces and data types and for any system classes that are needed by the current box.
- *variable declarations* to define the instance variables for the class. For a compound box, its constituent box manager objects are instantiated in this section of code.
- a `class` definition for the class `B`. An instance of this class sets the interface instance references, box instance references, and connections among the interfaces for the runtime system.

As noted above, the provided interface objects for box `B` are lazily instantiated, that is, a provided interface object will not be instantiated until the first call. Similarly, a box's reference to a required interface (another box's provided interface) will not be set until its first use. The `BoxTop` and `InterfaceDsc` classes are designed for this purpose. `InterfaceDsc` defines a table structure for the interface handles, which are typed as `InterfaceName`, and their references. BoxTop uses one interface reference table for its required interfaces and another for its provided interfaces. The former is called a *required interface reference table* and the latter a *provided interface reference table*. Both are defined to have type `InterfaceDsc`. Figure 10 shows the `BoxTop` and Figure 11 shows the `InterfaceDsc`.

```
public class BoxTop
{ InterfaceDsc pItfDsc, rItfDsc;
  public BoxTop()
  {  pItfDsc = new InterfaceDsc();
     rItfDsc = new InterfaceDsc();
  }
  public void setProvInterfaceDsc
       (InterfaceDsc pItfDsc)
  {  this.pItfDsc = pItfDsc;  }
  public void setRequInterfaceDsc
              (InterfaceDsc rItfDsc)
  {  this.rItfDsc = rItfDsc;  }
  public Object getProvidedItf(InterfaceName name)
  {  return pItfDsc.getInterfaceRef(name); }
  public Object getRequiredItf(InterfaceName name)
  {  return rItfDsc.getInterfaceRef(name); }
  public void setRequiredItf
       (InterfaceName iname, Object objRef)
  { rItfDsc.setInterfaceRef(iname, objRef); }
}
```

Figure 10. BoxTop.java

Each box manager class extends base class `BoxTop`. The constructor of box manager class `B` should do the following.

- Add the handles of the exposed required interfaces to the box's interface reference table, with the reference `null` for each.
- If `B` is compound, the constructor must set up the connections from the required interfaces to the provided interfaces among its constituent boxes. Each connection is built by setting a provided interface instance reference to the reference of a

required interface in the box's required interface reference table. `BoxTop`'s accessor methods `getProvidedItf()` and `setRequiredItf()` provide this functionality.

Other than the constructor, the class body of `B` must define the following methods:

- `getProvidedItf`: `B.java` overrides this method to instantiate each of `B`'s provided interface objects at their first use.
- `setRequiredItf`: If `B` is a compound box and it has exposed required interfaces, `B` overrides this method to set the references to the exposed required interfaces of `B`.

```
public class InterfaceDsc
{ private MyMap m;
  public InterfaceDsc()
  { m = new MyMap();  }
  public void addInterface
          (InterfaceName itfName,Object itfRef)
  { m.put(itfName, itfRef);   }
  public Object getInterfaceRef
       (InterfaceName itfName)
  { return m.get(itfName);    }
  public void setInterfaceRef
       (InterfaceName itfName, Object itfRef)
  { m.put(itfName, itfRef);   }
}
```

Figure 11. InterfaceDsc.java

The algorithm for code generation is shown below in pseudocode:

```
generateCode (B'dscFile, mngFile)
 //mngFile is the filename for the box manager code
   read in dscFile
   genPackage();
   genImports();
   genVars();
   genConstructor();
   genGetProvidedItf();
  // generate overriding method getProvidedItf
   if B is compound
        genSetRequiredItf();
       // generate overriding method
       setRequiredItf
   writeIntoFile(mngFile);
```

4. DISCUSSION AND FUTURE WORK

BoxScript is a relatively simple component language built on top of Java. The most significant contribution of this language is the introduction of the concepts of abstract boxes, box variants, box conformity, and interface satisfaction to support the two key properties of component-oriented programming: compositionality and flexibility. The BoxScript concept of a box conforming syntactically and semantically to an abstract box seems to be a new concept for component-oriented programming languages that have explicitly defined provided and required interfaces. In conjunction with the composition facilities, it enables flexible but safe component reuse capabilities. The concept of interface satisfaction is the basis for the idea of box conformity since, when a box conforms to an abstract box, we examine their interface satisfactions. The use of interface satisfaction enables flexible but safe composition: `R` and `P` can be composed as long as the provided interface `x` of box `P` satisfies the required interface `y` of box `R` to which it is connected.

The current BoxScript prototype has a static runtime structure, that is, concrete boxes are bound during compilation time. The advantage of this strategy is that it keeps the language simple, understandable, and typesafe. However, the boxes would be more flexible if the binding of box variants can be safely deferred past assembly time, to runtime.

The box type structure and the concept of a box variant are novel and useful features of BoxScript. The box variant enables different implementations of an abstract box to be inserted into the program being developed at the time of compilation. But it could be more useful. The current BoxCompiler requires that a concrete variant be chosen for each abstract constituent of a compound box before the compound box can be compiled correctly. However, this seems unnecessarily restrictive. The compiler should be able to compile most of the compound box features based on the characteristics of the abstract constituents. Then, before the system is deployed, a tool that assembles subsystems, binds the variant and performs any additional interface conformity checks.

The advocates of the new concept of software factories [2] argue that components must exhibit *deferred encapsulation* if they are to be broadly reusable. That is, a component should not be a complete blackbox. It should be parameterized with a number of variable aspects that can be bound at the time of assembly of a subsystem. However, each of these aspects should be encapsulated pieces of well understood functionality that have been predefined. That is essentially what the box variant supplies.

5. CONCLUSION

BoxScript is a Java-based, component-oriented programming language that provides convenient syntactic support for component concepts. It supports compositionality and flexibility in component-oriented systems and encourages good practices for component-oriented development. This paper briefly introduces the fundamental concepts of BoxScript and then describes how BoxScript components (i.e., boxes) are realized as clusters of interrelated Java interfaces, classes, and packages.

6. REFERENCES

[1] 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.

[2] Greenfield, J. and Short, K. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools.* Wiley, 2004.

[3] Liu, Y. and Cunningham, H. C. BoxScript: A Component-oriented Language for Teaching. In *Proceedings of the ACM SouthEast Conference*, Vol. 1, pp. 349-354, March 2005.

[4] Liu, Y. *BoxScript: A Language for Teaching Component-Oriented Programming*, Ph.D. Dissertation, Department of Computer and Information Science, University of Mississippi, August 2005.

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

[6] Morgan, C. *Programming from Specifications*, Prentice Hall International, 1994.

[7] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*, Second Edition. Addison Wesley, 2000.