Component-oriented software development has become an important approach to building complex software systems. Computer Science students should learn the concepts and skills needed for component-oriented programming. This paper describes a component-oriented language BoxScript whose design seeks to address the needs of teachers and students for a clean, simple language. This paper presents the key concepts and syntax of BoxScript and gives an example to illustrate its usage.

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 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. A component can be represented as shown in the diagram in Figure 1 [4]. 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 two interfaces match each other. That leads to inter-component dependencies being restricted to individual interfaces rather than encompassing the whole component specification.

The two most important properties of component-oriented programming are flexibility and compositionality. Flexibility allows the possibility of adapting a system easily to changing requirements by replacing, adding, or removing a few components. An application is built by assembling components. This requires the selection of suitable components and the right strategies for assembly. Compositionality makes the components able to be effectively combined into a larger component.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. 

Copyright 2004 ACM 1-58113-000-0/000/0004–$5.00.

Component-oriented programming has become a trend in software development. The concepts and languages that support this approach should be taught to computing science students. However, few programming languages support the current component-oriented approaches and even fewer languages have been designed to express clean component concepts and provide simple ways to teach how to “think in components”.

In 2002 the second author taught a software engineering class focused on Component Software [2]. The class used an approach similar to the “UML Components” approach [1]. For the programming projects, the class used the Enterprise JavaBeans (EJB) component technology. EJB is a component model for building server-side, enterprise-class applications [9]. It is a reasonable solution for commercial client/server systems. However, the complexity of the EJB technology 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. As a result of the experience in the class, the first author undertook the design of a simple, component-oriented language with features that support its use in teaching. The language is named BoxScript.

2. BOXSCRIPT
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 concept of component is based on Figure 1 and the assumed method for component specification is similar to the “UML Components” approach [1].

2.1 Key Concepts
A component is called a box in BoxScript. A box is a black-box entity; that is, it strongly encapsulates its internal details while only exposing its interfaces. The user of a box does not need to know its implementation details. A box, no matter how small or large, has the functionality needed to satisfy some requirements and can be used either individually or as a part of a larger box. A box can be either small enough so that it contains no other boxes, or composed from several smaller boxes. Every box has the ability to work with other boxes to form a larger box.

There are two types of interfaces that are considered in BoxScript. One is a provided interface, which describes the operations that a box can implement and that other boxes may use. The other is a required interface, which describes the operations that the box requires and that must be implemented by another box.

An abstract box is a box that only contains the descriptions of provided interfaces and required interfaces. It has one or more provided interfaces and zero or more required interfaces. However, it does not include the implementations of the provided interfaces. An abstract box should be implemented by concrete boxes, i.e., atomic boxes or compound boxes.

An atomic box is the basic element in BoxScript. An atomic box does not contain other boxes. Each atomic box has one or more provided interfaces and zero or more required interfaces. It must provide implementations for its provided interfaces.
A compound box is composed from atomic boxes or other compound boxes. The composition process is illustrated in Figure 2. 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.) A provided interface of a box might connect to a matching required interface of the other box, such as P11 of Box1 connects to R21 of Box2 and P21 of Box1 connects to R13 of Box2. In such a way, the boxes are “wired” together. The required interfaces of both Box1 and Box2 would be required interfaces of Box1_2 except for those satisfied by the other box. Similarly, the provided interfaces of both Box1 and Box2 would be the candidate provided interfaces of Box1_2. In Figure 2.b, required interfaces R11 and R12 from Box1 and R21 and R22 from Box2 are exposed to be required interfaces of Box1_2; provided interfaces P11 from Box1 and P22 from Box2 are exposed to be provided interfaces of Box1_2. Box1_2 does not need to implement its provided interfaces since the implementations are available from Box1 and Box2.

Similarly to the atomic box, a compound box has one or more provided interfaces and zero or more required interfaces. It does not implement the provided interfaces. When components are composed, the composition must follow the rules below:

i. All their provided interfaces are hidden unless explicitly exposed by the compound box.

ii. A component box must expose every required interface that is not wired to a provided interface of another box.

An atomic or compound 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; if the box is replaced by a newer version, the new version must continue to use the same name. An abstract box can extend another abstract box and the former one is considered to be a variant of the latter one.

When one box is replaced by another box, the new one should conform to the original one.

- Box B conforms to box A if and only if \( \forall p : p \in \text{provided interfaces}(A) \Rightarrow (\exists q : q \in \text{provided interfaces}(B) \land q \text{ extends } p) \)

- Box interface x extends box interface y if and only if interface handle \( (x) = \text{interface handle}(y) \) and type \((x) \text{ extends } (y)\) in Java. (The definition of interface handle is given below.)

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 the BoxB conforms to BoxA.

An application, or say, a system, is an executable box that meets some particular requirements to fit a problem domain. An application should expose no required interfaces.

### 2.2 Box Elements

The necessary units of code for building a box are box description, interfaces and their implementations, configuration information and box manager code.

In BoxScript, we use interface type to refer to a general interface with method declarations, in particular, a Java interface. The term interface handle refers to the interface as a way of describing a specific feature of a box. An interface handle has an interface type. The term box handle refers to the box as a way of describing a specific feature of composition.

All kinds of boxes, both abstract and concrete, must have a box description. A box description gives information such as the type, provided and required interface descriptions, and connection information (if a compound box is considered). It is a file with an extension of .box. For an abstract box, the box description declares the box as abstract and gives the box name, at least one provided interface type, zero or more required interface types, and the interface handles corresponding to these types. Figure 3 shows an example box named DateAbs, which has one provided interface.

```
abstract box DateAbs
{  provided interface DayCal Dc;
  //Dc is handle of interface DayCal
}
```

**Figure 3. DateAbs.box**

For an atomic box, the box description gives the box name and, if appropriate, the name of the abstract box it implements. If it implements an abstract box, it is said to be a variant of the abstract box. Also, the atomic box describes its provided and required (if there are any) interface types with the interface handles corresponding to these types. Figure 4 shows the box description for atomic box Date that implements the abstract box DateAbs.

```
box Date implements DateAbs
{  provided interface DayCal Dc; }
```

**Figure 4. Date.box**

For a concrete compound box, the box description not only supplies the information given in the atomic box, but also specifies three additional pieces of information: the box names from which the box is composed, the interface sources from which the provided and required interfaces come, and the connection information that describes how the box interfaces are wired together. If the boxes that compose this compound box implement abstract boxes, the abstract box names are used.
A concrete compound box gives the box names from which it is composed in the declaration portion. We assign each abstract box a handle to specify the composition information. Figure 5.a shows the box description of abstract compound box BuildCalendarAbs. Figure 5.b shows the box description of concrete compound box BuildCalendar that implements BuildCalendarAbs. The full example is described in section 3.

```
abstract box BuildCalendarAbs
{  provided interface Display D; }

Figure 5.a BuildCalendarAbs.box

box BuildCalendar implements BuildCalendarAbs
{  composed from DateAbs boxD,
    CalendarAbs boxC;
    // boxD is box handle for DateAbs
    // boxC is box handle for CalendarAbs
    provided interface
        Display D from boxC.Dis;
        connect boxC.DayC to boxD.Dc;
    }

Figure 5.b BuildCalendar.box
```

A concrete compound box gives the box names from which it is composed in the declaration portion. We assign a box handle for each box and the box handles help describe other composition information. In BuildCalendar, boxD and boxC are the box handles for DateAbs and CalendarAbs, respectively. Each interface of the concrete compound box is from one of the boxes that compose it. In the declaration portions, provided interface and required interface, after each interface type and interface handle is given, the source of the interface handle should be given as the description of the abstract box name and the interface handle in that abstract box. In BuildCalendar, interface handle D of interface Display is from interface handle Dis in box CalendarAbs (its handle is boxC). The connect statement gives information on how the boxes are wired. The syntax is to connect a required interface handle of a box to a provided interface handle of another box. In buildCalendar, the required interface handle DayC of box handle boxC (i.e., boxC.DayC) is connected to the provided interface handle Dc of box handle boxD (i.e., boxD.Dc). The composition process is illustrated in Figures 6.a and 6.b. For all kinds of boxes, the interface types that are used in the boxes should exist before the boxes are compiled. These interface types are Java interfaces in BoxScript.

An abstract box does not have implementations for its provided interfaces. A compound box does not need to develop implementations for its provided interfaces because they exist in the boxes that it is composed from.

An atomic box should provide implementations for its provided interfaces. An interface handle corresponds to each provided interface type. By default, an implementation of a provided interface is a Java class that implements the interface type, whose name is formed by appending Imp to the interface handle name. However, the implementation could be a Java class with a different name. In this case, the name must be specified in the configuration information.

For a compound box, boxes that are composed to form it are shown as abstract box names in the box description. This supports the ability to plug in different variants of the constituent boxes. Consider the example in Figures 3, 4 and 5. When the compound box BuildCalendar is executed, the concrete boxes Date and Calendar that implement the abstract boxes (DateAbs and CalendarAbs) should be accessed. So, we introduce a configuration file to indicate which concrete box is used for each abstract box in the compound box specification. In the configuration file, there is a pair for matching each box handle with the concrete box that is used. In this example, boxD (the box handle of DateAbs) and Date would be a pair, the former is the box handle that denotes the abstract box and the latter is the concrete box that the compound box accesses when executing. When a new version of Date, say DateNew, is substituted for Date, pair boxD and Date should be changed into pair boxD and DateNew in the configuration file.

```
DateAbs

DayCal
Dc

BuildCalendar

CalendarAbs

DateAbs

BoxD

DayCal
Dc

CalendarAbs

Date

BoxC

DayCal
Dc

Display

Diaplay

Figure 6.a DateAbs and CalendarAbs

Figure 6.b Composition
```

If the implementation of an interface handle does not use a default file name, the configuration file needs to describe the information as a pair consisting of the interface handle and the Java class file name that implements the interface type of the handle.

Each concrete box has box manager code that can be used to invoke the box. The box manager code is a Java class with the same name as the box and is generated by the BoxScript compiler. For an atomic box, this code has a constructor for the atomic box object and code that instantiates the interface handle objects. For a compound box, this code has a constructor for the compound box object and code that instantiates the boxes that comprise it. It passes the interface handle references from the constituent boxes to the compound box.

2.3 Box Processing Stages

After being edited, a box needs to be processed in four stages. The first stage is locating. After being edited and before being compiled, the files of a box should be placed into a directory.
structure called a warehouse. As shown in Figure 7, there are three subdirectories: boxes, interfaces and datatypes.

Figure 7. Directory Structure
The directory boxes is a multi-layer structure. The top layer holds the abstract box packages and box packages that do not extend any abstract boxes. A variant of an abstract box is located in a subdirectory below the directory that stores the abstract box. The directory of an abstract box stores its box description and has the same name as the abstract box. The directory of an atomic box stores its box description, the implementation of the provided interfaces and configuration file if it has one. The directory of a compound box stores the box description and configuration file.

The directories interfaces and holds the interface types and directory datatypes holds the utility data types that are used in the boxes. Data types refer to the types that are not standard Java data types but which are defined by the users for certain purposes. In particular, these may be user-defined types for parameters passed to or values returned from operations on an interface.

The second stage is compiling. Every box must be processed by the BoxScript compiler, called BoxCompiler. Figure 8 shows the relationship between the BoxCompiler and the Java compiler. BoxCompiler takes the box description and other necessary files as input, checks the syntax and interface conformity, and generates the box manager code for the concrete boxes.

Figure 8. BoxCompiler and Java Compiler

The third stage is shipping. When a BoxScript application is ready to be executed, it needs to be copied into a directory that the user appoints. The tool for shipping is called BoxShipper. On shipping, BoxShipper detects changes in the box’s directory. If any changes are detected, the box is re-compiled. BoxShipper bundles the necessary files for the application into a Java archive (jar) file and stores it into the appointed directory. The separation of the storage and the execution locations of an application ensures that the modification of the source code does not effect the execution of a previously compiled version.

After the application is shipped into the appointed directory, the boxes of that application reach their last processing stage – executing. The application cannot be modified while it is running. If any changes occur, such as the substitution of a new version of a box for the old one, the application should be ceased first and then replaced by the new version.

3. EXAMPLE
This section uses a simple example of displaying a calendar to illustrate BoxScript. To display a calendar for a certain month or months in a given range, we need to do two things: calculate the day of the week for each day in a month and display each month in a group of days with the weekdays illustrated. We can decompose the system into two boxes, one for calculating the day of the week for a given date, the other for displaying the calendar for a given interval. We design these as atomic boxes. To allow variants of boxes, we assign each an abstract box, calling the first DateAbs and the second CalendarAbs.

Box DateAbs has one provided interface DayCal as shown in Figure 9. DayCal has two methods: setDay() for setting a particular date and getWeekday() for getting the weekday for the day that was set by the setDay() method. The box description of DateAbs is shown in Figure 3. The concrete box that implements DateAbs is called Date and is shown in Figure 4. Since Date is an atomic box, it needs to provide the implementation of its provided interfaces. The implementation uses a default file name DcImp, which is combined with Dc, the interface handle of DayCal in Date and a suffix Imp.

public interface DayCal
brace public int getWeekday(int y, int m, int d);
brace // 0 for Sunday, 1 for Monday and so on
brace
brace
brace Figure 9. DayCal.java
brace import java.util.*;
brace import java.io.*;
brace public class DcImp implements DayCal
brace (public DcImp(Date myBox)
brace { _box = myBox; }
brace public int getWeekday(int y, int m, int d)
brace throws IllegalArgumentException
brace { year = y; month = m; day = d;
brace if (!isValid())
brace throw new IllegalArgumentException();
brace return (toJulian() + 1) % 7; }
brace private boolean isValid() { _box = myBox; }
brace private int toJulian() { return (toJulian() + 1) % 7; }
brace private Date _box;
brace private int year, month, day;
brace
brace Figure 10. DcImp.java

Box CalendarAbs, which is shown in Figure 12.a, has one provided interface Display and one required interface DayCal. The interface Display is shown in Figure 11. Display takes the time range from the user and displays the calendar. Display uses DayCal to calculate the day of week for each date. The atomic box that implements CalendarAbs is called Calendar, which is illustrated in Figure 12.b. Figure 13 illustrates only a partial implementation of the provided interface Display, provided by box Calendar, due to the limitation on the paper length. The implementation is named DisImp, which is a default name that combines Dis, the interface handle of Display in Calendar, and suffix Imp.In DisImp, class InterfaceName that is used by method generateSeq is a public class provided by BoxScript. Method getRequiredItf enables a program to get the interface reference by its interface handle name.

The calendar system BuildCalendar is composed from DateAbs and CalendarAbs. The composition is illustrated in Figure 6. The box description of BuildCalendar and its abstract box BuildCalendarAbs are shown in Figure 5.

4. DISCUSSION
The concepts of component-oriented programming are not simple. Most of the commercial languages for building component software introduce complicated environments, which make the component-oriented programming even more complicated.
Component Pascal [8] is a dialect of Oberon 2 [7]. It combines object orientation with modules in a language whose syntax is similar to Pascal. Again, the concept of component in Component Pascal is different from our component model. It does not have provided and required interfaces and has no particular composition techniques. Nowadays, most computer science students have Java or C++ as their first programming language. It is not convenient for students to be required to learn component programming using an unfamiliar language.

Jiazzi [6] and ComponentJ [10] are two extensions to Java. Both use the component concept described in Figure 1. Jiazzi works on top of the Java Virtual Machine (JVM) to support deployable, type-based components. However, Jiazzi’s scripting language provides explicit names for the composite components in a connection; there is no support for flexibility [5]. ComponentJ is also built on top of Java to support component composition. Components in ComponentJ are first class values and import and export methods but not types [6]. It gives provided and required interface specifications and gives the method implementations of the provided interfaces in the component definition. However, the method implementations make the component definition crowded, especially when the provided interface has several methods and each method definition is quite long. Moreover, the user of the component does not necessarily know the detailed implementations of the provided interface.

BoxScript is built upon a clear concept of component and provides simple language syntax. It supports the flexibility and compositionality of component-oriented programming. Boxes in BoxScript are separated by boundaries. The possible changeable aspects (secrets) should be designed to be inside a box. Under such a design, when a change occurs, only the boxes that hold the affected secrets would need to be changed. Any box that is modified should keep its interface unchanged. Also, boxes that allow variants are described by abstract boxes. The configuration files give matching pairs of abstract boxes and concrete boxes that are really being used. When a new version of a box comes, only the configuration file needs to be modified by replacing the old box package by this new package. The implementation code for the box should not need to change. Thus, no changes should be needed in the other boxes in that application. In BoxScript, three kinds of information participate in composition: interfaces, box descriptions of the boxes to be composed, and configuration file. No additional work is needed to compose boxes. Meanwhile, the rules guarantee the composition to be safe.

The three main programming units of BoxScript are box descriptions, interfaces and implementations of the interfaces. BoxScript separates an interface from its implementations. Both interface and its implementation are in the Java style familiar to students and the syntax of box description is simple enough for students to master. BoxScript avoids the issues that are irrelevant to the component view and platforms that would make the language unnecessarily complicated. BoxScript also delegates some coding, which is needed for boxes but probably too complicated for students to write, to the BoxCompiler. So, as far as the usage of the language is concerned, BoxCompiler provides a simple and friendly environment for building components.

C# [3] is probably the first commercial component-oriented programming language. However, the component concept in C# is different from the one shown in Figure 1, which we assume here.

Component Pascal [8] is a dialect of Oberon 2 [7]. It combines object orientation with modules in a language whose syntax is similar to Pascal. Again, the concept of component in Component Pascal is different from our component model. It does not have provided and required interfaces and has no particular composition techniques. Nowadays, most computer science students have Java or C++ as their first programming language. It
6. ACKNOWLEDGEMENTS
Removed for anonymity.

7. REFERENCES


