Java Components in BoxScript

Yi Liu and H. Conrad Cunningham
Department of Computer and Information Science, University of Mississippi, University, MS, 38677
{liuyi, cunningham}@cs.olemiss.edu

Abstract

Component-oriented software development has become an important approach to building complex software systems. This paper describes the component-oriented language BoxScript whose design seeks to support the concepts of components in a clean, simple, Java-based language. This paper presents the key concepts and syntax of BoxScript and how it supports compositionality and flexibility and gives an example to illustrate its usage.

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. The two most important properties of component-oriented programming are flexibility and compositionality. Flexibility allows programmers to adapt a component-oriented system to changing requirements by replacing, adding, or removing a few components. A component-oriented application is built by assembling components. This requires the selection of suitable components and effective strategies for assembly. The components should be compositional. That is, when components are selected and assembled into a system, the behavior of the system should be predictable based on the behavioral specifications of the components. Strong encapsulation of the internal details of components is needed to provide the desired flexibility and compositionality.

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

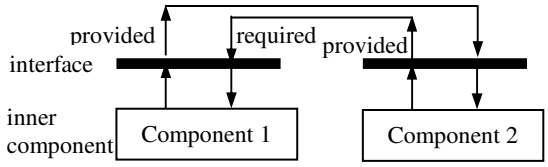

Figure 1. Components and Their Interconnections

Component-oriented software development has become an important approach to building complex software systems. Object-oriented programming (OOP) languages have been the predominant approach to the design of component-oriented programming applications. However, practice shows that OOP technology does not fully provide the above properties and leaves unsatisfied some of the needs for development of complex component-oriented applications [14]. The main difficulties lie in the implementation inheritance feature and composition strategy of OOP languages. Implementation inheritance is *whitebox* reuse [5]. A derived class is very tightly coupled to its base classes and thus breaks the base class encapsulation. This may also bring safety problems, e.g., the *fragile base class problem* [9]. In component-oriented applications, if implementation inheritance is applied across different components, then the strong encapsulation among components is broken and any changes made to the base class may cause unpredictable effects. The resulting component is not compositional. The composition strategy in OOP is awkward and inefficient because programmers may need to build extra classes to accomplish composition.

There is a need for programming languages that can provide safer composition strategies and cleaner component concepts. This research is to design a component-oriented programming language that should be easy for novices to learn and should provide convenient syntactic support for component concepts.

The initial motivation for this work arose from the authors' experience in a college class on component programming [2]. For design, the class used methods similar to the "UML Components" approach [1]. For the programming projects, the class used the Enterprise JavaBeans (EJB) component technology [13]. Although EJB is a reasonable solution for commercial client/server systems, the complexity of the EJB technology and its lack of simple composition strategies means it is 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. The prototype of BoxScript is intended primarily for educational and research purposes. However, a full implementation of these ideas should also be useful for building real applications.

This paper is organized as follows. Section 2 presents the key concepts and syntax of BoxScript. Section 3 describes the processing stages for building a box. Section 4 gives an example to demonstrate the usage of BoxScript. Section 5 compares BoxScript with several existing component-oriented programming languages and discusses its properties. Section 6 suggests the possible future work and Section 7 concludes the paper.

2. BoxScript

BoxScript is a Java-based, component-oriented programming language whose design seeks to address component concepts in 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 blackbox 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 constituent boxes. Every box has the ability to be composed with other boxes to form a larger box.

The necessary units of code for building a box are a *box description*, *interfaces* and their *implementations*, *configuration information,* and *box manager code*. A box description (a file with an extension of `.box`) gives declarations for the features of the box. The configuration information file gives the additional information that is not included in the box description. The box manager code is a Java class with the same name as the box that is generated by the BoxScript compiler. The box manager code is for concrete boxes only.

2.1.1 Interfaces

There are two types of interfaces in BoxScript. One is a *provided interface*, which describes the operations that a box implements 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. A box has one or more provided interfaces and zero or more required interfaces.

```
public interface DayCal
{
  public void setDay(int y, int m, int d);
  public int getWeekday();
  // 0 for Sunday, 1 for Monday and so on
}
```

Figure 2. Interface `Daycal`

In BoxScript, we use *interface type* to refer to a general interface with method declarations, in particular, a Java interface. Figure 2 shows an interface named `DayCal`. 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. Interface handles are specified in the box description for the corresponding box. The provided interfaces in a box are specified using the keyword `provided interface` followed by interface type and interface handle pairs. Similarly, required interfaces are specified using the keyword `required interface` followed by interface type and handle pairs.

2.1.2 Abstract box

An *abstract box* is a box that only contains the descriptions of provided interfaces and required interfaces. 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.

All kinds of boxes, both abstract and concrete, must have *box descriptions*. In BoxScript, an abstract box should have a box description that declares the box as `abstract`, gives the box name, and specifies its provided interface and required interface descriptions. Figure 3 shows an abstract box named `DateAbs`, which has one provided interface `DayCal`. Declaration of interface `DayCal` is given in Figure 2. `Dc` is the interface handle that is used in box `DateAbs` for type `DayCal`.

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

Figure 3. DateAbs.box

2.1.3 Atomic box

An *atomic box* is a basic element in BoxScript. An atomic box does not contain other boxes. It must provide implementations for its provided interfaces. 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

An atomic box must provide the implementations of its provided interfaces. By default, an implementation of a provided interface is a Java class file that implements the interface type and whose filename is formed by suffixing `Imp` to the interface handle name. However, the

implementation of the interface type could be a Java class with a file name other than what is described above. If an implementation does not use a default file name, the implementation file name needs to be specified in the configuration information as a pair consisting of the interface handle and the Java class file name that implements the interface type of the handle. For example, Figure 5 gives an implementation for the provided interface `DayCal` of box `Date` shown in Figure 4. This implementation takes a default name combined interface handle `Dc` with suffix `Imp`.

```
import java.util.*;
import java.io.*;
public class DcImp implements DayCal
{
  public DcImp(BoxTop myBox)
  {    _box = myBox;    }
  public int getWeekday(int y,int m,int d)
      throws IllegalArgumentException
  {    year = y; month = m; day = d;
       if (!isValid())
         throw new
           IllegalArgumentException();
       return (toJulian() + 1) % 7;
  }
  private boolean isValid() {…}
  private int toJulian() {…}
  private BoxTop _box;
  private int year, month, day;
}
```

Figure 5. DcImp.java

An atomic box has a box manager that is generated by the BoxCompiler. The box manager code includes a constructor for the atomic box object and code that instantiates the interface handle objects.

2.1.4 Compound box

A *compound box* is composed from atomic boxes or other compound boxes. A compound box does not need to develop implementations for its provided interfaces because they exist in the boxes from which it is composed. The box description for a compound box not only supplies the information given in the atomic box, but also specifies three additional pieces of information: the box names from which this compound box is composed, the interface sources from which the provided and required interfaces of this compound box come, and the connection information that describes how the boxes connect interfaces together to compose this compound box. Detailed information on the strategy for composition and the description of the compound box are presented in Section 2.2.

In addition to a box descriptor and any necessary configuration information file, a compound box has a box manager that is generated during compilation. It 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.

2.1.5 Application

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 Composition in BoxScript

The *composition* process in BoxScript is illustrated in Figure 6. Suppose we wish to compose `Box1` and `Box2` into a compound box `Box1_2`. (In the following, `Box1` really means an instance of `Box1` and `Box2` means an 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 6.b, required interfaces `R11` and `R12` from `Box1` 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`.

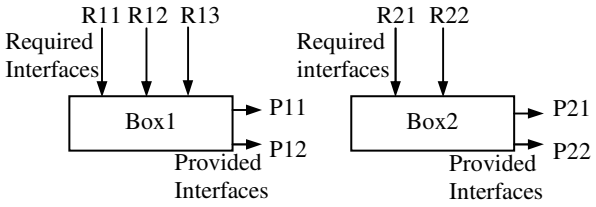

Figure 6.a Box1 and Box2

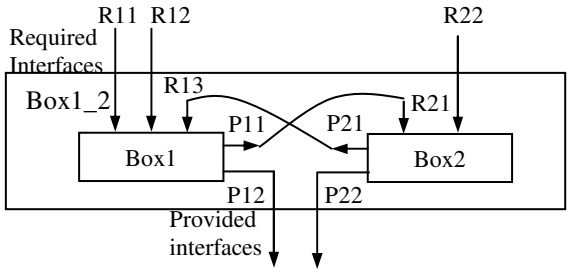

Figure 6.b Composition of Box1 and Box2

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.

The compound box is the only type of box that needs composition. Composition is specified in the box description for the compound. Each constituent box of the compound box has a *box handle*, which is a way of describing the constituent box as a specific feature of composition. As discussed in section 2.1.4, a compound box should specify three additional pieces of information about the participants in the composition: names for the constituent boxes, sources of its provided interfaces and required interfaces, and connection information.

Consider the component box `BuildCalendar` shown in Figure 7, `BuildCalendar` is composed from boxes `Date` and `Calendar`, which extend abstract boxes `DateAbs` and `CalendarAbs`, respectively. In this example, we use abstract boxes `DateAbs` and `CalendarAbs` instead of concrete boxes `Date` and `Calendar` in describing the composition that forms `BuildCalendar`. A concrete compound box gives the box names from which it is composed in the declaration portion `composed from`. 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 abstract boxes `DateAbs` and `CalendarAbs`, respectively. Figure 7.a shows the box description for abstract box `BuildCalendarAbs`. Figure 7.b shows the box description for concrete compound box `BuildCalendar` that implements `BuildCalendarAbs`.

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

Figure 7.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 7.b BuildCalendar.box

Each interface of the concrete compound box is from one of the boxes that compose it. In `provided interface` and `required interface` declarations, 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 Figure 8. The full example is described in Section 3.

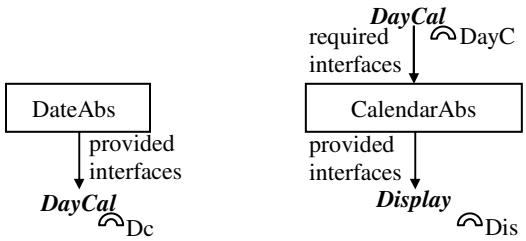

Figure 8.a DateAbs and CalendarAbs

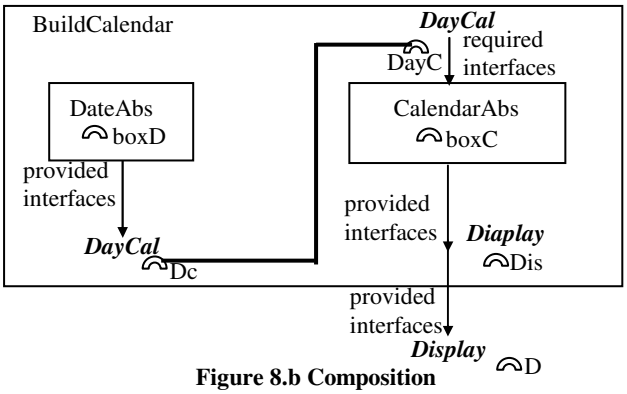

Figure 8.b Composition

A box is strongly encapsulated with only its interfaces exposed. The behavior of a box remains unchanged unless the state of the box is changed. The state of a box can only be changed when an action on a provided interface method is performed. The match of a provided interface to 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. That means, the behavior changes of a box are predictable.

2.3 Flexibility in BoxScript

BoxScript introduces the concept of *variant* to support flexibility, especially to enable substitution of one alternative implementation for another. 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* syntactically to the original one.

- 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) and
  type (x) extends type(y) in Java.

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`. In addition to conforming syntactically, the boxes must conform semantically, which is discussed in [3].

To support flexibility, in the composition of a compound box, the constituent boxes that are composed to form it are shown as abstract box names in the box description. The abstract box name allows the ability to plug in different variants of the constituent boxes. Consider the example in Figures 3, 4 and 7. 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`, no box source files other than the configuration file needs to be changed: pair `boxD` and `Date` should be changed into pair `boxD` and `DateNew`.

3. Box Processing Stages

There are five stages for building a box: editing, locating, compiling, shipping and executing.

After being edited, a box should be placed into a directory structure called a *warehouse* as shown in Figure 9. This is the *locating* stage. The warehouse directory has subdirectories `boxes`, `interfaces` and `datatypes`. 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. Each box has a directory whose name is the same as its box name. An abstract box's directory stores its box description. A variant of an abstract box is in a subdirectory of the abstract box directory. The directory of an atomic box stores its box description, the implementation of the provided interfaces and the configuration file if it has one. The directory of a compound box stores the box description and configuration file. The directory `interfaces` 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.

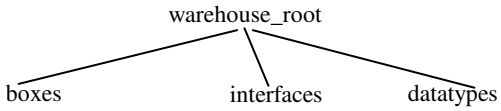

Figure 9. Directory Structure

The stage after locating is *compiling*. Every box must be processed by the BoxScript compiler, called BoxCompiler. Figure 10 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.

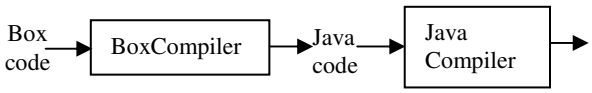

Figure 10. BoxCompiler and Java Compiler

When a BoxScript application is ready to be executed, it needs to be copied into a directory that the user chooses. We call this process *shipping*. 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.

4. Example

This section uses a simple example of displaying a calendar to illustrate the use of 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 (shown in Figure 3) has one provided interface DayCal (shown in Figure 2). 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 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 (shown in Figure 5) uses a default file name DcImp, which is combined with Dc, the interface handle of DayCal in Date and a suffix Imp.

Box CalendarAbs, which is shown in Figure 12, 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 13. Figure 14 gives the implementation of the provided interface Display, supplied by box Calendar. 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.

```
public interface Display
{ //display a single month calendar
  //according to input year and month
  public void displayCalendar(int year,
                              int month);
  //display months of a calendar
  // according to input years and months
  public void displayCalendar(int year1,
        int month1, int year2, int month2);
}
```

Figure 11. Display.java

```
abstract box CalendarAbs
{ provided interface Display Dis;
  required interface DayCal DayC;
}
```

Figure 12. CalendarAbs.box

```
box Calendar implements CalendarAbs
{ provided interface Display Dis;
  required interface DayCal DayC;
}
```

Figure 13. Calendar.box

The calendar system BuildCalendar is composed from DateAbs and CalendarAbs. The composition is illustrated in Figure 8. The box description of BuildCalendar and its abstract box BuildCalendarAbs are shown in Figure 7.

5. 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. C# [4] is probably the first commercial component-oriented programming language. Component Pascal [11] is a dialect of Oberon 2. It combines object orientation with modules in a language whose syntax is similar to Pascal. However, the component concepts in both C# and Component Pascal are different from the one shown in Figure 1, which we assume here. Jiazzi [8] and ComponentJ [12] are two extensions to Java that are built upon the component model shown in Figure 1. However, Jiazzi's scripting language provides explicit names for the composite components in a connection; there is thus no support for flexibility [7]. ComponentJ 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; 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 a simple language syntax. It supports the requirements for flexibility and compositionality of component-oriented programming. Boxes in BoxScript, being separated by boundaries, are strongly encapsulated. The possible changeable aspects (secrets) of a component should be designed to be inside a box. Implementation inheritance is not allowed across box boundaries. 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. Originally designed for an educational purpose, BoxScript adapts its interfaces and interface implementations to a Java style that is familiar to students.

```
public class DisImp implements Display
{ private BoxTop _box;
  DayCal d; //required interface
  public DisImp(BoxTop myBox)
  { _box = myBox;
    InterfaceName name =
                new InterfaceName("DayC");
    d =(DayCal)_box.getRequiredItf(name);
  }
  public void displayCalendar(int year, int month)
  { setDate(year,month, 0,0);
    generateSeq(year, month);
    display(year,month);
  }
  public void displayCalendar
   (int year1,int month1,int year2,int month2)
  { setDate(year1,month1,year2,month2);
    processCalendar(year1,month1,year2,month2);
  }
  private void generateSeq(int y, int m)
  { int i = 0, weekday;
    int totaldays = getDays(y,m);
    ClearDateArr();
    weekday = d.getWeekday(y,m,1);
    dateArr[i][weekday] = 1;
    for(int t = 2; t <= getDays(y,m); t++)
    {   dateArr[i][weekday] = t;
        if (weekday == 6)
        {   i++; weekday = 0;}
        else
            weekday ++;
    }
  }
  private void display(int y, int m)
  { System.out.println
      ("\n" + y + "    " + monthStr(m)+ "\n");
    for (int i = 0; i < WEEKS; i++)
    {   if (! ((i == WEEKS-1)
                && dateArr[i][0] == 0 ) )
          System.out.println("Sun" + "\t" +
              "Mon" + "\t" + "Tue" + "\t" +
               "Wed" +"\t" + "Thu" + "\t" +
              "Fri" + "\t" + "Sat" );
        for (int j = 0; j < WEEKDAYS; j++)
          if (dateArr[i][j] == 0)
             System.out.print("   \t");
          else
             System.out.print
                 (dateArr[i][j] + "\t");
          System.out.println();
     }
   }
   private void processCalendar
          (int y1, int m1, int y2, int m2)
   {    int startY = y1, endY = y2;
        int startM = m1, endM = m2;
        int yy, mm;
        if (y2 < y1)
        {  startY = y2; endY = y1;
           startM = m2; endM = m1;
        }
        if ((y2 == y1) && (m2 <m1))
        {  startM = m2; endM = m1; }
        // print the first year
        if (startY != endY)
           for (mm = startM; mm <= 12; mm++)
           {   generateSeq(startY, mm);
               display(startY, mm);
           }
        else
        {  for (mm = startM; mm <= endM; mm++)
           {   generateSeq(startY, mm);
               display(startY, mm);
           }
           return;
        }
        for (yy = startY+1; yy <= endY - 1; yy++)
        {   for (mm = 1; mm <= 12; mm ++)
            {   generateSeq(yy, mm);
                display(yy, mm);
            }
        }
        // print the last year
        for (mm = 1; mm <= endM; mm ++)
        {   generateSeq(endY, mm);
            display(endY, mm);
        }
    }
    private void setDate
        (int y1, int m1, int y2, int m2)
          throws IllegalArgumentException
    {   year1 = y1; month1 = m1;
        year2 = y2; month2 = m2;
        dateArr = new int [WEEKS][WEEKDAYS];
        if (!isValid())
           throw
             new IllegalArgumentException();
    }
    private boolean isValid()
    {   boolean mark = false;
        if (year2!=0)
           mark = month2 > 0 && month2 <= 12;
        else
           mark = month2 == 0;
        return year1 > 0 && month1 > 0
               && month1 <= 12 && mark;
    }
    private int getDays(int y, int m)
    {   switch (m)
        {   case 1: case  3: case  5: case 7:
            case 8: case 10: case 12:
                    return 31;
            case 4: case 6: case 9: case 11:
                    return 30;
        }
        if ((y % 4) == 0)
            return 29;
        return 28;
    }
    private String monthStr(int m)
    {   switch (m)
        {   case  1: return "January";
            case  2: return "February";
            case  3: return "March";
            case  4: return "April";
            case  5: return "May";
            case  6: return "June";
            case  7: return "July";
            case  8: return "August";
            case  9: return "September";
            case 10: return "October";
            case 11: return "November";
            case 12: return "December";
        }
        return null;
    }
  private void clearDateArr()
  {  for (int i = 0; i < WEEKS; i++)
        for (int j = 0; j < WEEKDAYS; j++)
            dateArr[i][j] = 0;
  }

  private static final int WEEKS = 6;
  private static final int WEEKDAYS = 7;
  private int year1, year2;
  private int month1, month2;
  private int day1, day2;
  private int dateArr [][];
}
```

Figure 14. DisImp.java

The syntax of box description is simple enough for novices to master. It avoids the issues that are irrelevant to the component view and platforms that would make the language unnecessarily complicated. BoxScript also delegates some code generation, which is needed for boxes but probably too complicated for most users 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.

6. Future Work

The prototype of BoxScript is under implementation and it is scheduled to be evaluated in a graduate course during the 2005 Spring semester. There are several areas for future work. First, to keep the language clean in thought and simple in use, the prototype of BoxScript 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) [6] to enable use of design-by-contract techniques.

7. Conclusion

BoxScript is a component-oriented programming language that is easy for novices to learn and provides convenient syntactic support for component concepts. It supports compositionality and flexibility in component-oriented systems and encourages good practices for component-oriented development. BoxScript gives a simple Java-based language syntax and a clean working environment. BoxScript should be a language that is suitable for both education of and use by novices for component-oriented programming. It can help them learn component concepts and skills thoroughly and apply them effectively.

Acknowledgments

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

References

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

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

[3] H. C. Cunningham, Y. Liu, and P. Tadepalli. "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.

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

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

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

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

[8] 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.

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

[10] H. Mössenböck and N. Wirth. "The Programming Language Oberon-2," *Structured Programming*, Vol. 12, pp. 179–195, 1993.

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

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

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

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