Packages White Paper

VERSION 2018

jade

Contents

Packages

This document discusses the use of packages in a JADE system. Since the introduction of subschemas, JADE has allowed a class, along with the properties and methods of a class, to be used through the single inheritance of schemas. Packages are an adjunct to the existing schema structure that allows similar access but in a more controlled manner and without the need for the schema defining the class to be a superschema of the schema using it.

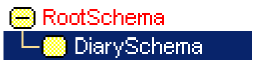
The developer of a package decides what functionality needs to be implemented in an exporting schema and then decides which classes, properties, and methods should be exported in one or more packages. Usually only a small subset of the classes, properties, and methods are exported in the package; only those necessary to expose the functionality required.

Users of the package can then load the exporting schema into their system, usually into an unrelated schema branch, and then import the package into their schema. They then have access to the functionality provided by the package via the classes, properties, and methods exported by the package.

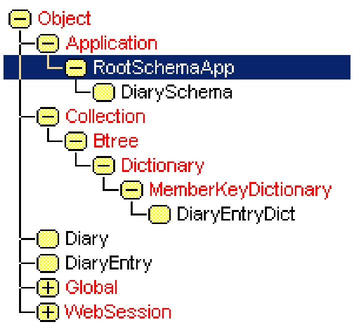
This document covers some of the issues you should consider when both developing and using packages in your JADE applications. For more details, see the following subsections.

A Simple Diary Package

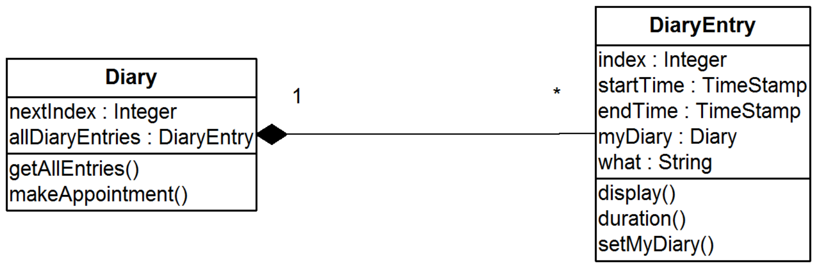
In order to explain how a package is written and used, let us develop a small example package and work our way through the issues involved in design, implementation, and use. The running example will be a simple diary system that allows the storage of appointments, their retrieval, and display.



We begin by creating a schema that defines the basic classes with their corresponding properties and methods. Note that there is nothing special about a schema that exports a package and it can be tested as a standalone schema before we convert it to a package. The schema we define will be called **DiarySchema** and will be a subschema of the **RootSchema**.



The two major classes will be a **DiaryEntry** class that will represent an appointment and a **Diary** class that will hold a collection of these. Any system can have a number of instances of **Diary**, one for each user for example. The simple system is shown in the following image.



The **DiaryEntry** class has attributes **startTime** and **endTime** (both of type **TimeStamp**) for the appointment, along with an attribute **what** (of type **String**) that holds the description of the appointment. The attribute **index** (of type **Integer**) provides a unique number for each appointment within the diary. The manual reference **myDiary** (of type **Diary**) has an inverse **allDiaryEntries** of type **DiaryEntryDict**, a member key dictionary with keys **startTime**, **endTime**, and **index**.

The **duration** method returns a **String** (for example, **"30mins"**) showing the timespan of an appointment that is used by the **display** method, which provides further details such as the index, day, time, duration, and description as in:

```
0 Thu 24 Jul 2003 12:30 30mins Dentist
```

The **Diary** class has a single attribute **nextIndex** (of type **Integer**) that tracks the next-highest unique index for a **DiaryEntry** and two methods. The **getAllEntries** method returns a **String** containing all appointments formatted one for each line in the above format.

The **makeAppointment** method has the following signature.

```
makeAppointment(startTime         : TimeStamp;
                durationInMinutes : Integer;
                what              : String) updating;
```

This method creates a new **DiaryEntry** object, sets all of the attributes of the object, and finally sets the **myDiary** reference to **self** to add the new entry to the diary's **allDiaryEntries** collection.

Writing **JadeScript** methods to create a new diary, make new appointments, and write them all out could then test this simple schema.

Having tested our schema thoroughly, we now decide to make it available to other users. Without packages, users would need either to load the classes directly into their schema or include **DiarySchema** somewhere in their schema hierarchy. If they included the complete schema, they would then be able to use these classes in any subschemas below the **DiarySchema** schema.

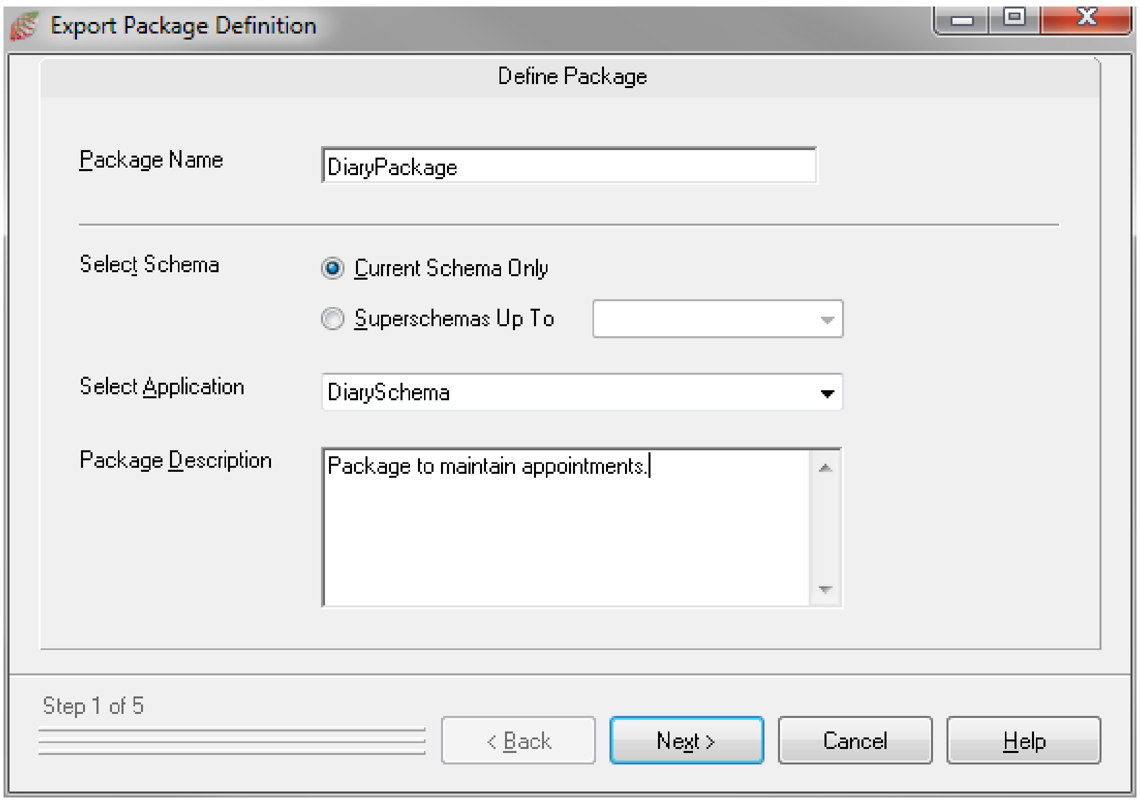
However, inserting a schema into an existing schema hierarchy, especially one not designed with this in mind, can be problematical. Firstly, it is not trivial to insert a new schema, and the class names used in the schema may already exist in the schema hierarchy below the level at which you want to insert it. This would preclude the schema from being inserted, as class names must be unique in a schema branch.

A better solution is to create and export a package from **DiarySchema** and then import it into the schema or schemas where we want to use it. The first step is to decide which classes, properties, and methods we want to export and which should remain hidden. In general, it is good practice to export only those parts of the system that are essential for the package to be useful and to hide all non-essential details.

In our example, we have decided to hide the manner in which the collection of **DiaryEntry** objects are stored. The user of the package does not need this information, so we do not need to export the **DiaryEntryDict** class, the **allDiaryEntries** collection property, or its inverse **myDiary**. An advantage of not exporting this information is that if we decide at a later date to change the manner in which these are stored, we can do so without any changes to any code that imports the package. Such a change may require a reorganization of any persistent instances of these classes that the user has created. Another property that we do not need to export is the **nextIndex** property on **Diary**. In our case, we decide to export all of the methods shown in the previous image but we would not export any helper methods that these might use.

Having decided on what classes, properties, and methods to export, we can now use the package export wizard to define the package. This wizard is available from the **Browse** menu via the **Packages** and **Export Browser** menu items. This brings up the DiarySchema Export Packages Browser form, which enables you to select **Add** from the Packages menu to display a series of wizard forms, shown in the following images. The wizard proceeds through the following steps.

1. The package is named **DiaryPackage** and an application from the schema is selected.



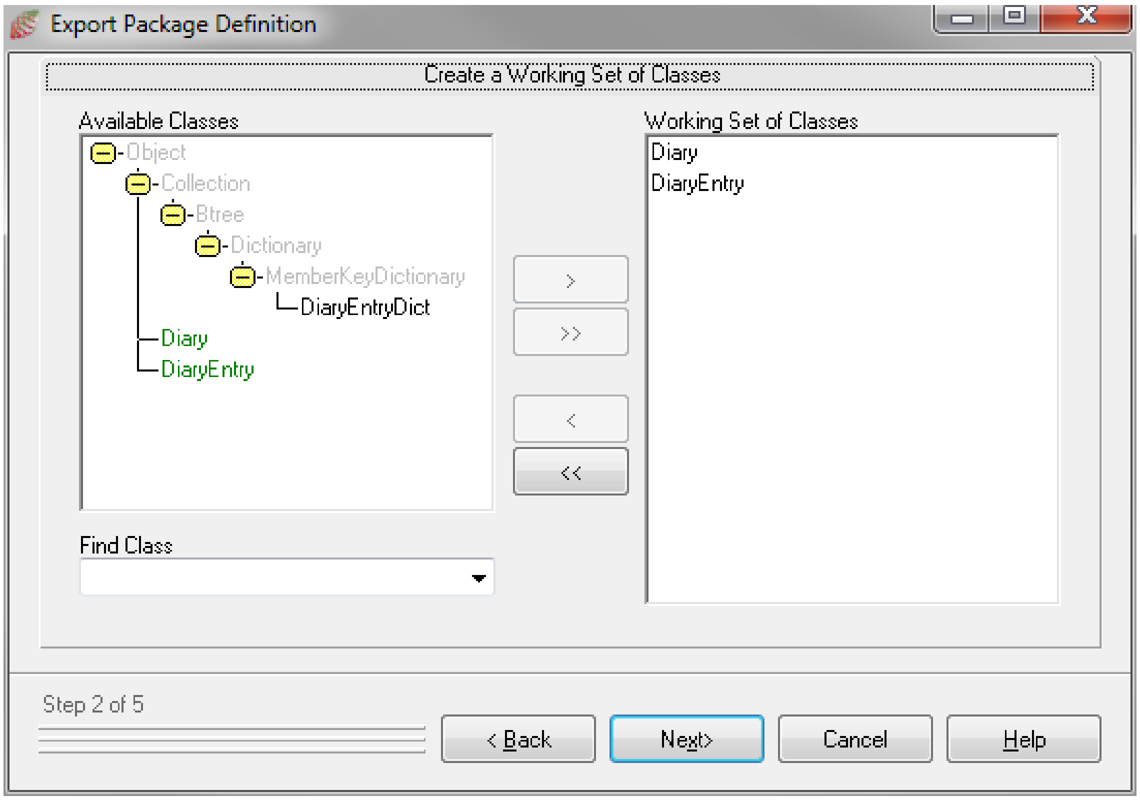
Package Name: DiaryPackage

Select Schema: [✓ Current Schema Only] Superschemas Up To

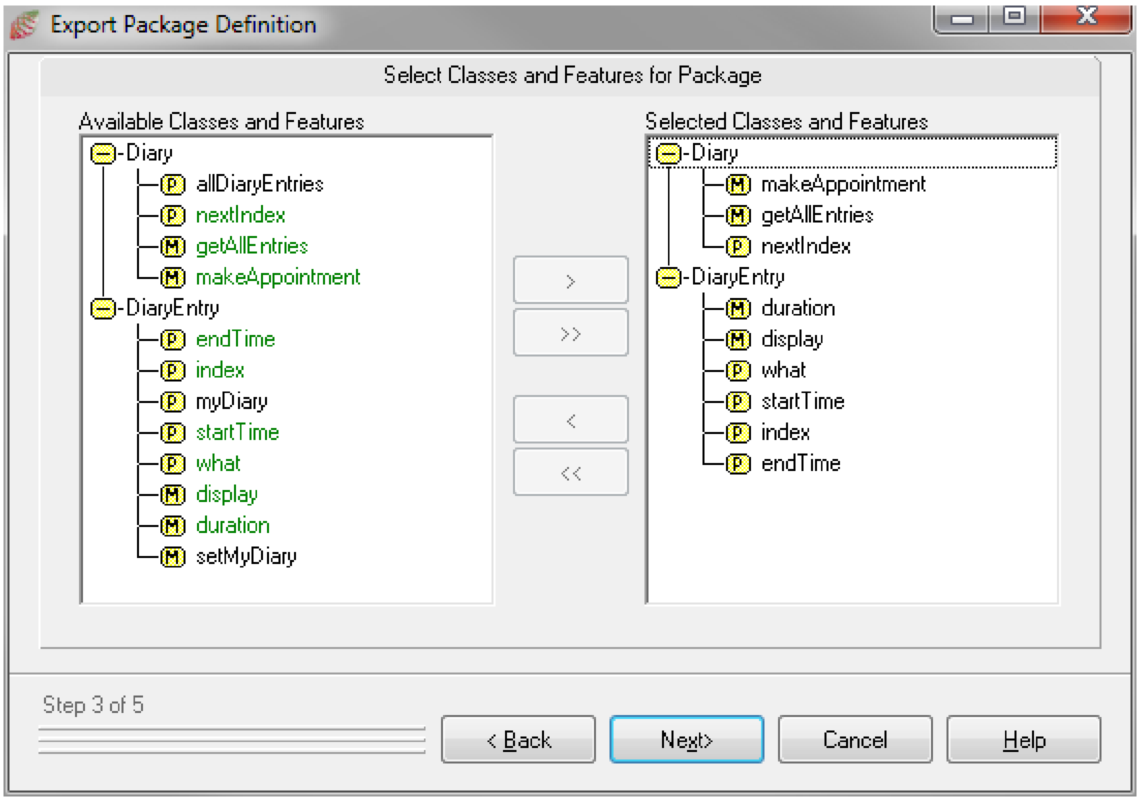
Select Application: DiarySchema

Package Description: Package to maintain appointments.

2. The classes to be exported, **Diary** and **DiaryEntry**, are selected. Note that we have not selected **DiaryEntryDict**, and the system classes (for example, **Collection**), are greyed out, indicating they cannot be exported. Classes that have been selected to be exported are shown in green.



3. The properties and methods to be exported are selected. Note that the protected members (for example, properties **nextIndex**, **allDiaryEntries**, and **myDiary**) are not available for export. In addition, we have chosen not to export the method **setMyDiary**, which we want to be purely internal to **DiarySchema**. (The method **setMyDiary** could not be made protected, as it must be visible to the method **makeAppointment** on **Diary**.)


Export Package Definition

Select Classes and Features for Package

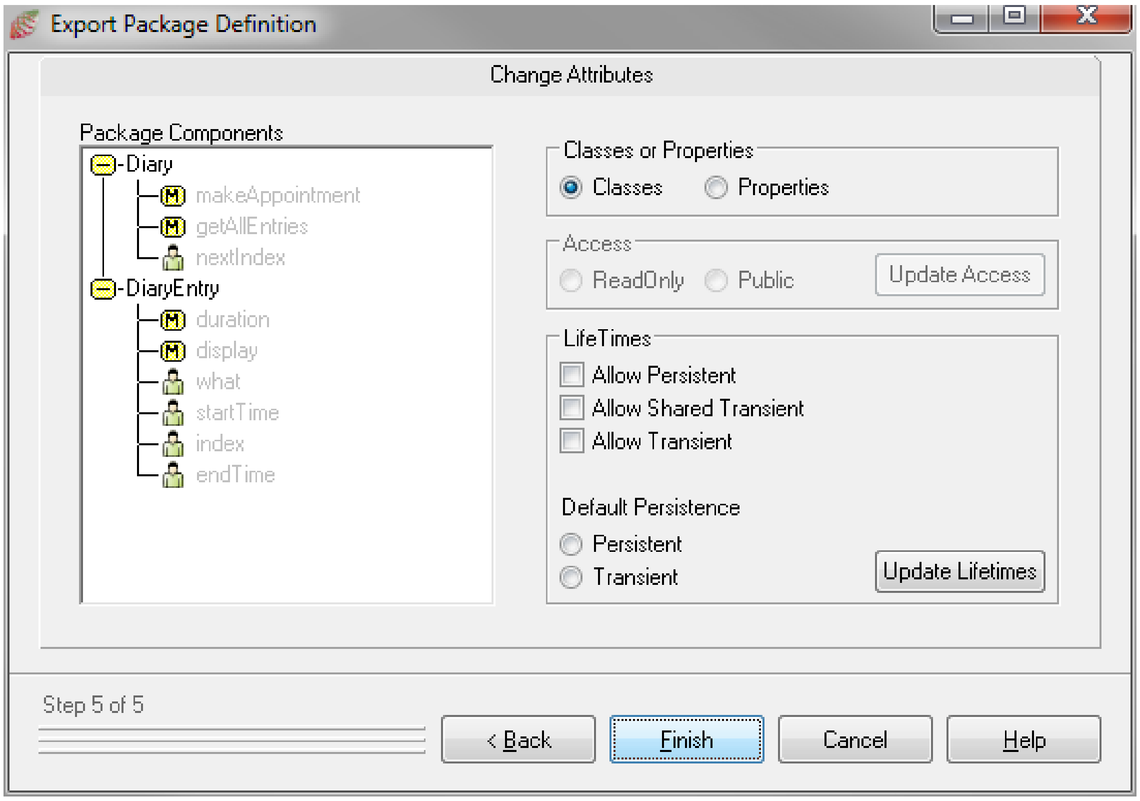
Available Classes and Features
- Diary
  - allDiaryEntries
  - nextIndex
  - getAllEntries
  - makeAppointment
- DiaryEntry
  - endTime
  - index
  - myDiary
  - startTime
  - what
  - display
  - duration
  - setMyDiary

Selected Classes and Features
- Diary
  - makeAppointment
  - getAllEntries
  - nextIndex
- DiaryEntry
  - duration
  - display
  - what
  - startTime
  - index
  - endTime

4. When you have selected the features for inclusion in the exported package working set and clicked the **Next >** button, the **Select Interfaces for Package** sheet of the Export Package Definition Wizard is then displayed.

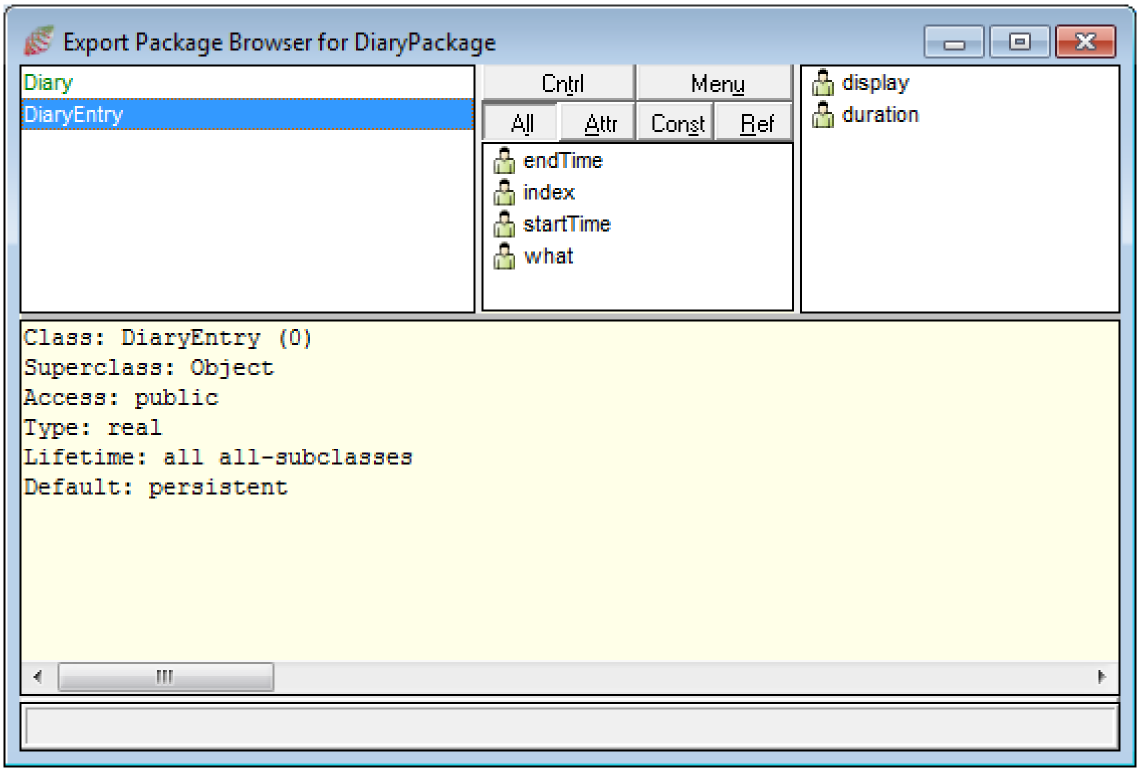
   This fourth sheet enables you to select the interfaces that you want to include in the exported package, if required.

5. Choose lifetimes and default persistence for the exported classes and access modes for their properties. By default, these are the same as those declared in the exporting schema and can be made only "more restrictive" than their declaration.



An alternative to using the export wizard when the DiarySchema Export Packages Browser is displayed and a package has been defined is to use the Export Package Class Browser. This can be opened by double-clicking on the package in the DiarySchema Export Packages Browser or by selecting **Browse** from the Packages menu (when the DiarySchema Export Packages Browser is displayed).

The Export Package Class Browser, shown in the following image, is then displayed.



```
Class: DiaryEntry (0)
Superclass: Object
Access: public
Type: real
Lifetime: all all-subclasses
Default: persistent
```

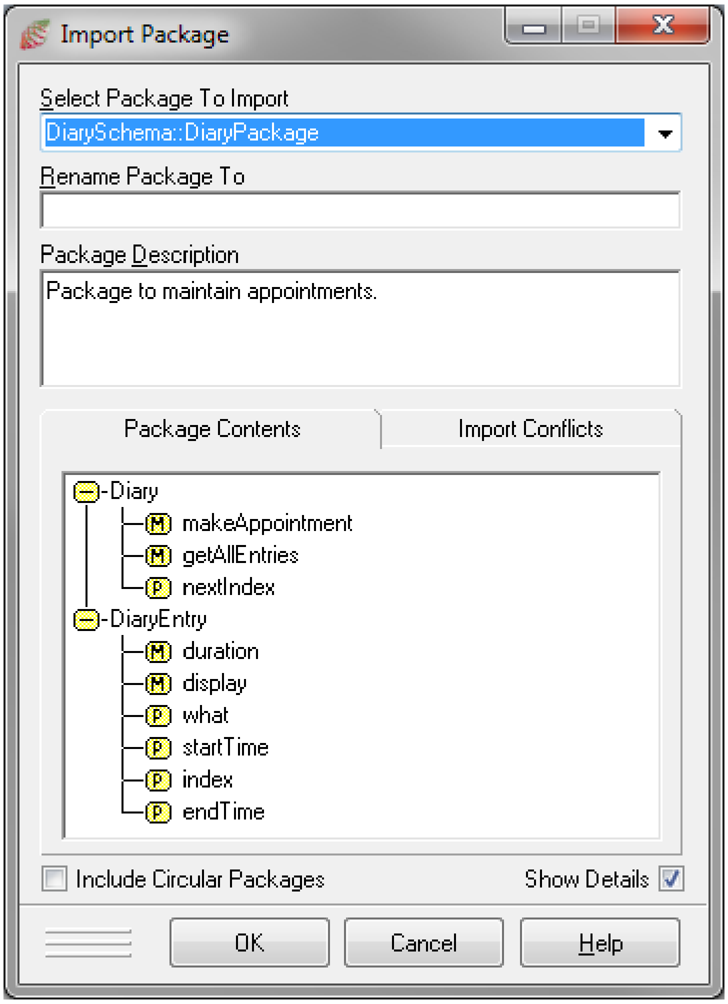
This export packages view is similar to a standard Class Browser but limited to the classes, properties, and methods exported in the package. Properties and methods can be added to the package by dragging and dropping them between the normal schema class and package class browsers.

Having defined our exporting package, we can now import it into another schema and use it to create and manipulate diaries. The importing schema need not be a subschema of the exporting schema and would normally have only the **RootSchema** as a common ancestor in our example schema **DiaryTester**. We can now import the **DiaryPackage** into this schema using the **Browse** menu from the **Packages** and **Import Browser** menu items. This displays the DiaryTester Import Packages Browser.

Select the **Add** menu item from the **Packages** menu, and then select which exported packages to import.

The **Rename Package To** text box in the Import Package form enables you to rename the package if the name of the package conflicts with another package that has already been imported.

As we will see, it is not a problem if the names of any of the imported classes conflict with classes already defined in the schema or imported from other packages.



Select Package To Import: DiarySchema::DiaryPackage

Rename Package To:

Package Description: Package to maintain appointments.

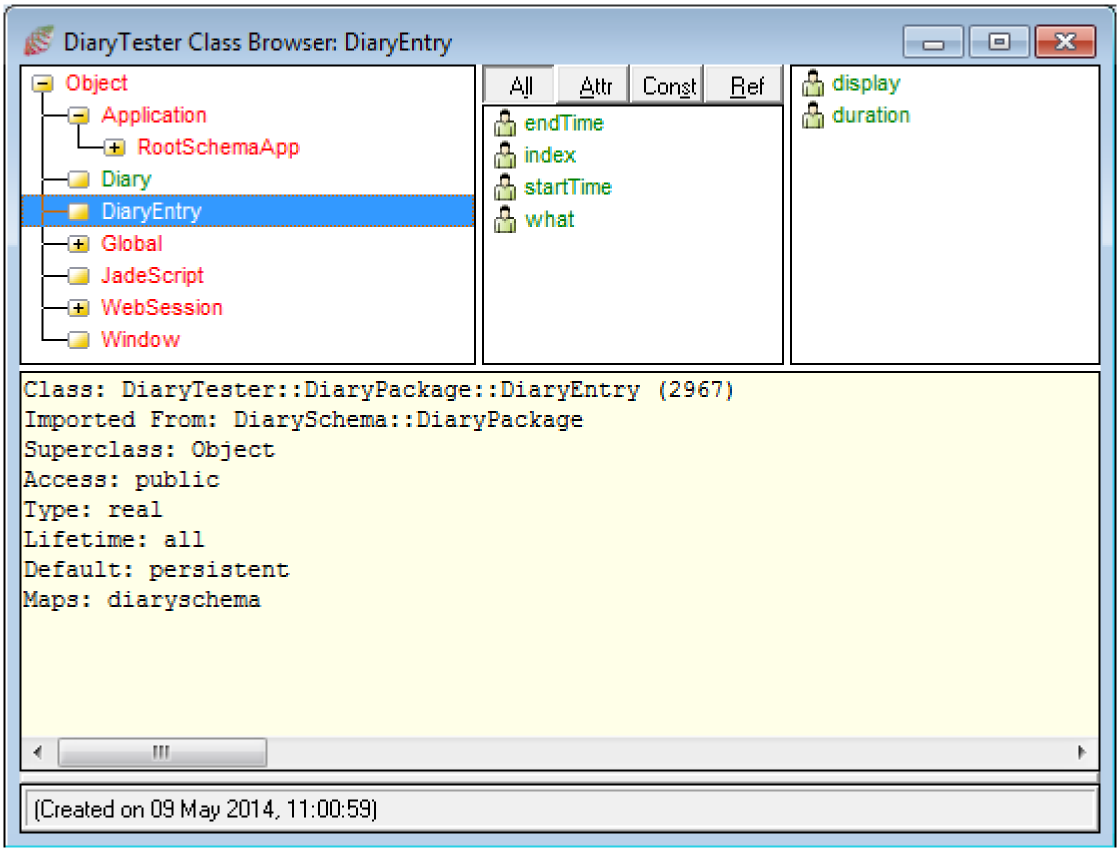
```
Diary
  makeAppointment
  getAllEntries
  nextIndex
DiaryEntry
  duration
  display
  what
  startTime
  index
  endTime
```

Include Circular Packages [ ]   Show Details [✓]

If you want to allow a circular dependency between packages in the schema hierarchy, check the **Include Circular Packages** check box. This permits the loading of an incomplete package (for example, **Schema1** exports **Package1** and imports **Package2**, while **Schema2** exports **Package2** and imports **Package1**). When you subsequently create a package that would result in circularities, you are prompted to confirm that you want to continue and allow a circular dependency between packages in the schema hierarchy.

Check the **Show Details** check box to display the package contents.

Having imported the **DiaryPackage**, opening a normal Class Browser for the schema **DiaryTester** displays the imported classes along with their properties and methods in green, as shown in the following image. These imported entities cannot be modified. If the package schema has been encrypted when it was extracted, the source of any exported methods would not be shown in the browser.



```
Class: DiaryTester::DiaryPackage::DiaryEntry (2967)
Imported From: DiarySchema::DiaryPackage
Superclass: Object
Access: public
Type: real
Lifetime: all
Default: persistent
Maps: diaryschema
```

(Created on 09 May 2014, 11:00:59)

The imported classes and methods can now be used as in the following **JadeScript** method example.

```
createDiary();
vars
    diary: DiaryPackage::Diary;
    today: TimeStamp;
begin
    beginTransaction;
    if app.diary = null then
        create app.diary;
    endif;
    today.setTime('12:30'.Time);
    app.diary.makeAppointment(today, 30, 'Dentist');
    today.setTime('18:00'.Time);
    app.diary.makeAppointment(today, 60, 'Squash');
    write app.diary.getAllEntries;
    commitTransaction;
end;
```

The reference to the imported class **Diary** uses the double colon (**::**) scope operator as in **DiaryPackage::Diary**. If there is no ambiguity as to which class **Diary** represents, as in our schema where there is no local **Diary** class and no other imported **Diary** class, **Diary** can be used without the prefix.

When the method call **app.diary.getAllEntries** is made in this script, a switch is made from the **DiaryTester** schema into the exporting **DiarySchema**. Along with this switch is a change in the environmental context of the process. In particular, there is a switch in the meaning of environmental variables such as **app**, **global**, and **currentSession** to those of the schema that exported the package. This enables the package developer to use references to these in their code. A common use of this is to access the properties and methods of **app** in the package, to save context information.

To illustrate this switch of context, suppose we add the following method to class **Diary** and export it in **DiaryPackage**.

```
printAppGlobal();
begin
    write 'App and Global in Diary::printAppGlobal';
    write '    app=' & app.getName & ' global=' & global.getName;
end;
```

We then run the following **JadeScript** method in the **DiaryTester** schema.

```
showSwitch();
vars
    diary: DiaryPackage::Diary;
begin
    diary := DiaryPackage::Diary.firstInstance;
    write 'App and Global in showSwitch before call';
    write '    app=' & app.getName & ' global=' & global.getName;
    diary.printAppGlobal;
    write 'App and Global in showSwitch after call';
    write '    app=' & app.getName & ' global=' & global.getName;
end;
```

The output of this **JadeScript** method is:

```
App and Global in showSwitch before call
    app=DiaryTester global=GDiaryTester
App and Global in Diary::printAppGlobal
    app=DiarySchema global=GDiarySchema
App and Global in showSwitch after call
    app=DiaryTester global=GdiaryTesterer
```

Notice that both **app** and **global** have switched from those that apply in the importing schema **DiaryTester** to those that apply in the exporting schema **DiarySchema** while executing the exported method **printAppGlobal**, and then have switched back.

Scope Rules for Method Calls

Having imported the class **DiaryEntry** along with its associated imported properties and methods, we are able to add local methods to the class. These methods can be called from the importing schema and any subschemas, just like normal methods without any schema switch. For example, we could add the following method to the class.

```
weekendEvent(): Boolean;
begin
    // write app.getName();
    if startTime.date.dayOfWeek = 6 or startTime.date.dayOfWeek = 7 then
        return true;
    else
        return false;
    endif;
end;
```

This **weekendEvent** method can be called from a **JadeScript** method, as follows.

```
    testWeekendEvent();
    vars
        de: DiaryEntry;
    begin
        createDiary;
        de := DiaryEntry.firstInstance;
        write de.weekendEvent;
    end;
```

If the commented out **write** statement in the **weekendEvent** method was uncommented, it would yield **DiaryTester**, showing that the **app** had not been switched from the schema in which the **JadeScript** was executed.

However, what if we add a local method to an imported class that has the same name as a method that already exists on the class in the exporting schema? Note that the importer of the package may not even know this has happened, as the method with the same name may not have been exported. Adding a local method with the same name is allowed, and the local method does not even have to have the same signature as the method in the exporting schema. For example, although a **display** method is exported on **DiaryEntry**, we could also add a local **display** method with a different implementation, as follows.

- Exported **display** method:

  ```
  display(): String;
  begin
      return index.String &
      startTime.date.format(" ddd d MMM yyy ") &
      startTime.time.format("hh:mm") &
      Tab & duration() &
      Tab & what & CrLf;
  end;
  ```

  This exported method produces the following output.

  ```
  0 Wed 6 Aug 2003 12:30    30mins    Dentist
  ```

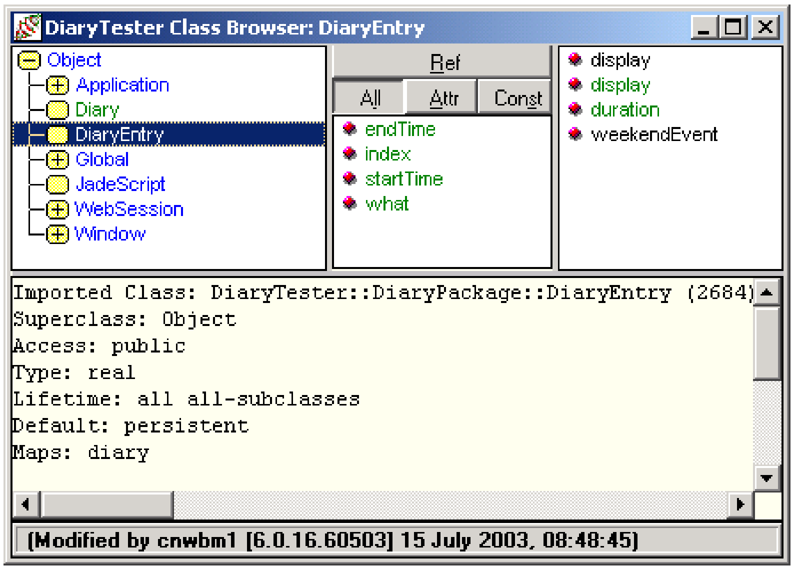
- Local **display** method:

  ```
  display(): String;
  begin
      return what &
      Tab & duration() & " at " & startTime.time.format("hh:mm") & CrLf;
  end;
  ```

  This local method produces the following output.

  ```
  Dentist    30mins at 12:30
  ```

After this, two **display** methods show in the method frame of the Class Browser for the **DiaryEntry** class: one green for the imported method and one black for the local method. So which **display** method gets called? More precisely, the question should be which **display** method gets invoked in which context?



```
Imported Class: DiaryTester::DiaryPackage::DiaryEntry (2684)
Superclass: Object
Access: public
Type: real
Lifetime: all all-subclasses
Default: persistent
Maps: diary
```

(Modified by cnwbm1 [6.0.16.60503] 15 July 2003, 08:48:45)

The resolution of this is that the most-local definition of the method is called. In particular, this means that when the method is called from within the **DiaryTester** schema (from a **JadeScript** method, for example), the local version is invoked. However, when execution has switched to the exporting schema (for example, as a result of invoking the exported method **getAllEntries** on class **Diary**, which invokes **display** to produce the **String** of all appointments), the **display** method defined within and exported from **DiarySchema** is used.

Note It is useful to understand why the scope rules are defined in this manner.

The rule that the local definition (which may or not be exported) is used within the exporting package schema protects the developer of the package. The developer knows that the functionality of the package will not be changed and potentially compromised by any user of the package adding local methods to the exported class.

The rule that the local definition is used within the importing schema allows the developer of the importing schema to control how that imported class behaves. For example, it may be that within their schema, all classes must have a method **display** that behaves in a specific way so that code such as that shown in the following method will behave correctly.

```
displayAllObjectsInSet(oSet: ObjectSet);
vars
    o: Object;
begin
    foreach o in oSet do
        write o.display;
    endforeach;
end;
```

Suppose we now removed the local **display** method on the imported **DiaryEntry** call and executed the call **de.display** on a **DiaryEntry** object **de**. What output would be expected? It might be surprising that the output would be something like the following.

```
 ---Diary/2683.7---
```

This is because the **display** method defined on class **Object** in the **DiaryTester** schema has been called. This is regarded as the "most-local definition", because it is defined on a superclass of **DiaryEntry**. Only if no definition was found in the local schema or inherited from a superschema would the imported **display** method be called.

However, what if we really wanted to call the imported **display** method rather than the method on **Object**? How can this be done? We can achieve this by defining a local **display** method on **DiaryEntry**, as follows.

```
callImportedDisplayMethod(): String;
begin
    return importMethod display();
end;
```

The keyword **importMethod** tells the compiler that we want the imported **display** method of the current class to be invoked, rather than any local **display** method.

Switching app on Exported Method Call

As noted in a previous section, when a method in another package is called, a switch is made from the current schema into the package's exporting schema along with a change in the environmental context of the process. In this section, we will see how the resulting switch of **app** enables us to store context information.

In our running **Diary** example, we made a decision to hide the manner in which **DiaryEntry** objects were stored. Although a method **getAllEntries** was supplied to return all appointments as a formatted list, this would be inconvenient for users of the package. To extract appointments for a specific day or the details of a specified appointment, the user of the package would need to parse this string.

We decide we want to provide methods on **Diary** that will allow the user of the package to specify which appointments he or she wants to see, and then return them one at a time as **DiaryEntry** objects. To do this, we will add two properties to the **DiaryPackage** class that is the type of the **Application** object exported as part of the package. These properties are **savedDiaryEntries** of type **DiaryEntryArray** (a subclass of **Array** with membership **DiaryEntry**) and **iterator** of type **Iterator**. Neither of these properties or the **DiaryEntryArray** class will be exported, but are available within the package code and can be accessed via **app**.

We can now export methods on **Diary** like those in the following examples.

```
getDayEntriesBegin(day: Date);
// initialize iterator over all diary entries occurring on day
vars
    de: DiaryEntry;
begin
    app.savedDiaryEntries.clear;
    foreach de in allDiaryEntries where de.spansDay(day) do
        app.savedDiaryEntries.add(de);
    endforeach;
    app.iterator.reset;
end;

getNextDiaryEntry(de: DiaryEntry output): Boolean;
// Return the next DiaryEntry object, if any,  from the iterator
begin
    if app.iterator.next(de) then
        return true;
    else
        return false;
    endif;
end;
```

These two properties on **app** will need to be initialized before any importer of the package calls them. After a package is imported into a schema (for example, the **DiaryPackage** into the **DiaryTester** schema), when any **app** is run from the importing schema, all packages are initialized. This initialization consists of calling the initialize method for the application, which by default is the method **initialize**. Similarly, the finalize method of the application, which by default is the method **finalize**, is called when the application finishes execution. This gives the package designer the opportunity to perform any initialization and finalization required for the package to function correctly. Note that the application might have different initialization and finalization methods defined for when it is run in its own schema. For our package, we will add the following methods to the **DiaryPackage** class and use the Define Application form shown in the following image to set them as the initial and final methods.

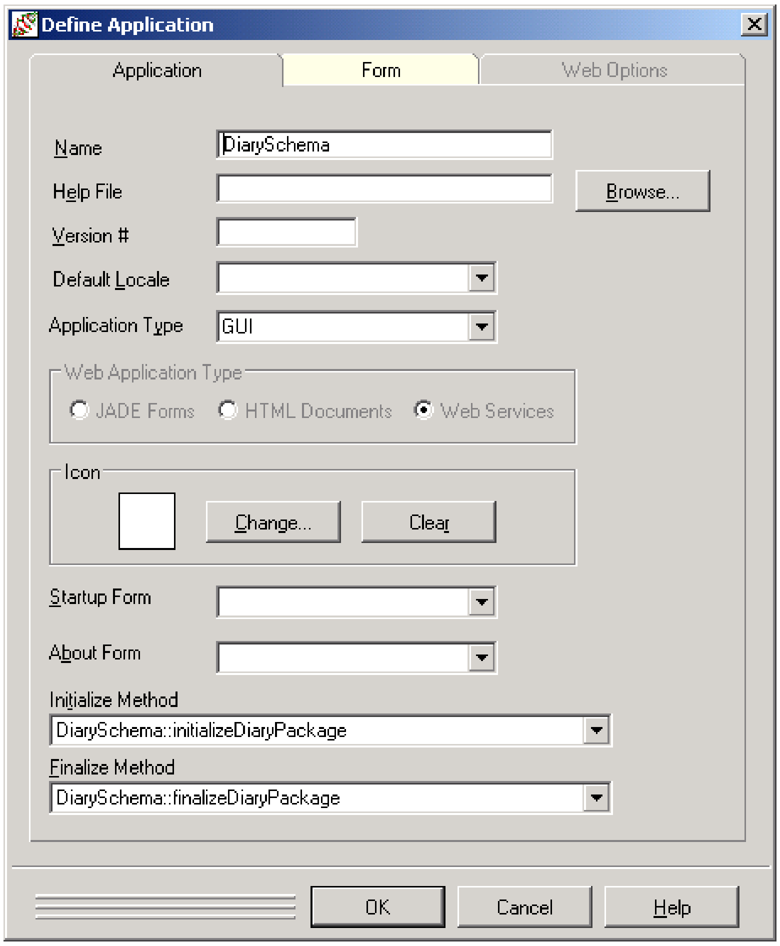
```
initializeDiaryPackage() updating;
// Initialize the Diary package properties on app
begin
    savedDiaryEntries.clear;
    iterator := savedDiaryEntries.createIterator;
end;

finalizeDiaryPackage() updating;
// Finalize the Diary package properties on app
begin
    delete iterator;
end;
```

These can be used in the importing schema such as in the following **JadeScript** testing method.

```
printAllForDay();
vars
    de: DiaryEntry;
    today: Date;
begin
    process.initializePackages;
    app.initialize;
    if app.diary <> null then
        app.diary.getDayEntriesBegin(today);
        while app.diary.getNextDiaryEntry(de) do
            write de.display;
        endwhile;
    endif;
    process.finalizePackages;
end;
```

The initialization and finalization of packages does not occur for **JadeScript** or Workspace methods, just as the default **app** is not initialized. This ensures that these are lightweight operations and also because the default **app** may not be the appropriate application to initialize. The calls to **process.initializePackages** and **app.initialize** at the start, and **process.finalizePackages** at the end, of the above **JadeScript** method force this initialization and finalization to occur. JADE does not initialize any packages for **RootSchema** applications running in user schemas, including unit tests run using the JADE Unit Test framework and the default RPS **Datapump** application in the **RootSchema**. If you require this initialization, you must initialize packages in your application code (using the **Process** class **initializePackages** and **finalizePackages** methods).



Name: DiarySchema
Help File:
Version #:
Default Locale:
Application Type: GUI
Web Application Type: ( ) JADE Forms ( ) HTML Documents [✓ Web Services]
Startup Form:
About Form:
Initialize Method: DiarySchema::initializeDiaryPackage
Finalize Method: DiarySchema::finalizeDiaryPackage

An observant reader will have noted that our design is somewhat limiting, as it allows only a single **Diary** to be iterated at a time, and a schema importing our **DiaryPackage** may want to iterate multiple **Diary** objects simultaneously. This could be accommodated by having a collection of transient **DiaryContext** objects on **app**, each of which holds a **Diary** reference along with any context information such as the **iterator** and **savedDiaryEntries** collections. Such a collection could be keyed on a **name** or **id** property added to **Diary**.

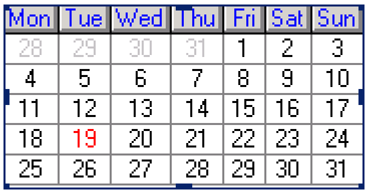
When a new **Diary** instance is created, the constructor method **create**, defined on the exported **Diary** class, is called. (Note that it is not allowed to add a local constructor or destructor method in the importing schema and all of the constructor and destructor methods for the class and superclasses are called in the exporting package schema when an exported class instance is created or deleted.) This would mean the package could create a new transient instance of the **DiaryContext** class when a new **Diary** object was constructed, initialize it, and then add it to the collection of those held on the **app**.

A similar creation and initialization would occur the first time a new **Diary** instance was seen inside the package; for example, instances that came from **Diary.firstInstance** calls.

Exporting GUI Subclasses

In the examples earlier in this document, the classes exported by our packages are all subclasses of **Object**. In this section, we will develop a **DateSelector** schema that defines a **DateSelectorPackage**, which exports a **DateTableSelector** control that can be added to forms to allow a date to be selected. Having done this, we will import it into our **DiaryTester** schema. This will expose a number of further issues relevant to the use of packages.

The **DateTable** class is a control that is a subclass of **Table**, and builds a control that has the form of a table without tabs.

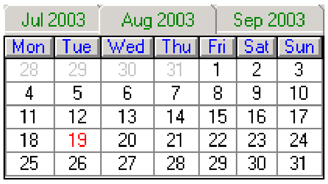


Mon	Tue	Wed	Thu	Fri	Sat	Sun
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

The control class has a protected property **startOfMonth** of type **Date**, which specifies which month is being displayed and is used to determine on which day of the week the month starts and the number of days in the month. The **windowCreated** method, which is called when the control is placed on a form or when a form containing the control is painted, sets the **startOfMonth** property based on the current date. It then calls **fillTable** to fill in the details of the table, as shown in the following example.

```
windowCreated(cntrl: Control input; persistCntrl: Control) updating,
clientExecution;
begin
    setStartOfMonth(app.actualTime.date);
    fillTable;
    inheritMethod(cntrl, persistCntrl);
end;
```

The **DateTableSelector** class is a subclass of the **DateTable** class, with its own **windowCreated** and **fillTable** methods.

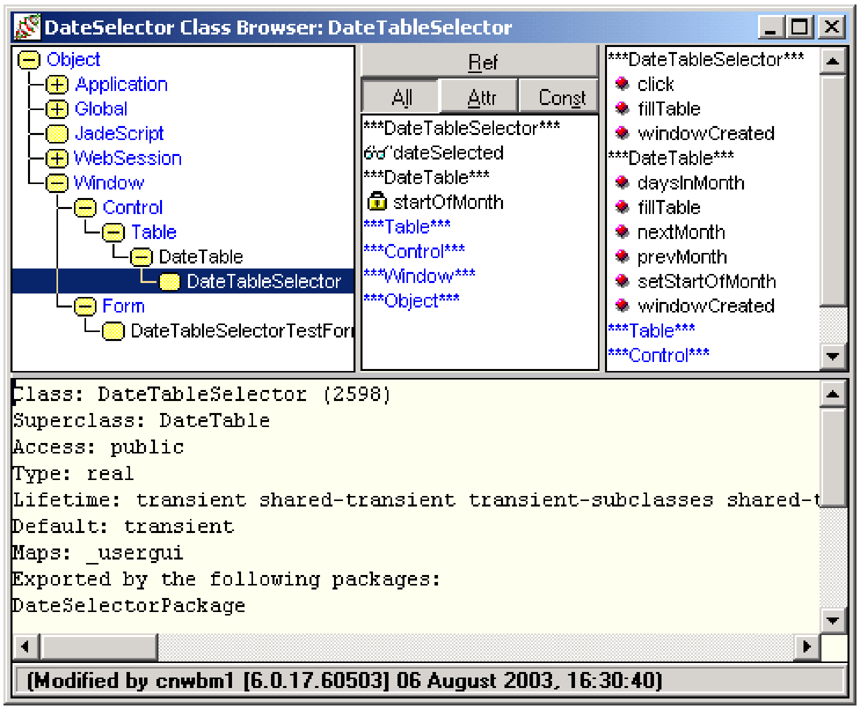


Jul 2003 | Aug 2003 | Sep 2003

Mon	Tue	Wed	Thu	Fri	Sat	Sun
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

The **windowCreated** method has the same code as the previous method example but the **fillTable** method has code to add the month tabs to the table. In addition, the class has a property **dateSelected** of type **Date**. A **click** method is also defined, which moves the table to show the next or previous month if the appropriate tab of the table is clicked, or sets the **dateSelected** property to the date selected if one of the days in the month is selected.

The Class Browser for the **DateSelector** schema is shown in the following image.



```
Class: DateTableSelector (2598)
Superclass: DateTable
Access: public
Type: real
Lifetime: transient shared-transient transient-subclasses shared-t
Default: transient
Maps: _usergui
Exported by the following packages:
DateSelectorPackage
```

(Modified by cnwbm1 [6.0.17.60503] 06 August 2003, 16:30:40)

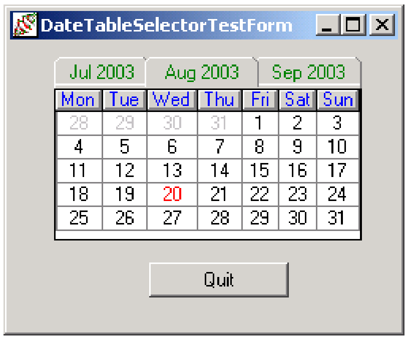
The following example is the generic **click** method for the **DateTableSelector** control.

```
click(table: DateTableSelector input) updating, clientExecution;
vars
    startDay, endDay: Integer;
begin
    dateSelected := null;
    if sheet=1 then
        self.setStartOfMonth(prevMonth(self.startOfMonth));
        fillTable;
        return;
    elseif sheet=3 then
        self.setStartOfMonth(nextMonth(self.startOfMonth));
        fillTable;
        return;
    elseif sheet=2 then
        accessCell(row, column);
        if accessedCell.foreColor = Gray or not accessedCell.selected then
            return;
        endif;
        dateSelected.setDate(accessedCell.text.Integer,
                                        self.startOfMonth.month, self.startOfMonth.year);
  endif;
  inheritMethod(table);
end;
```

We can test whether this control works correctly within the **DateSelector** schema by creating a form containing the control. We can then check that the table is painted correctly, that it moves to the correct month when the previous and next month tabs are clicked, and that the **dateSelected** property of the control is correctly set when a date is clicked.

We now want to export this control as part of a package **DateSelectorPackage**, so we must decide which classes and features to expose. Clearly, we want to export the **DateTableSelector** class, but there seems to be no compelling reason to export its **DateTable** superclass. As to what features to export, we decide that only the **dateSelected** property needs to be exported, to allow users of the control to determine which date of the control has been clicked.

To test this package, we create a **DateSelectorTester** schema, import **DateSelectorPackage**, and then use the JADE Painter to create a **DateTableSeclectorTestForm** on which we paint an instance of the imported control.

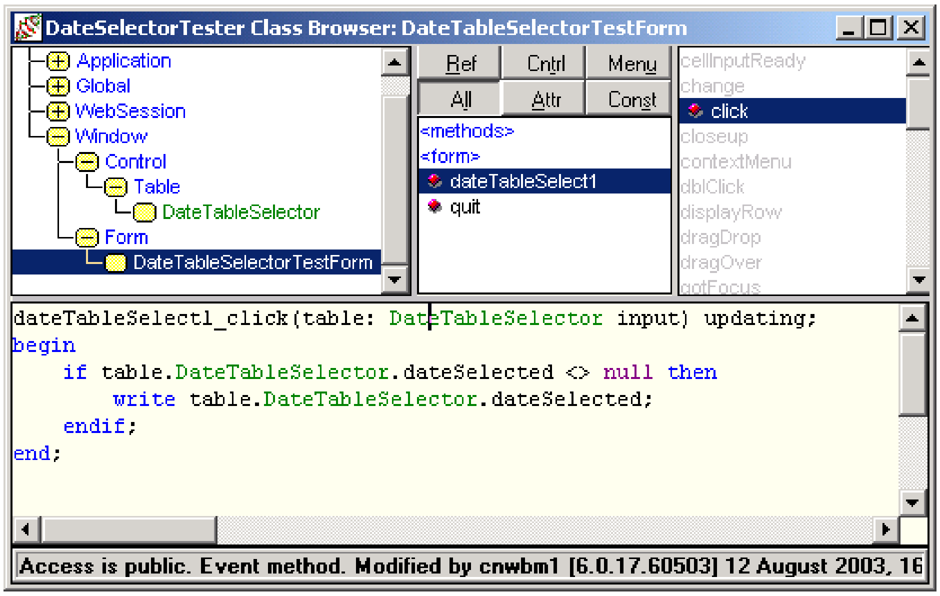


Mon	Tue	Wed	Thu	Fri	Sat	Sun
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

Opening a Class Browser for this schema shows that the imported **DateTableSelector** control class appears with a superclass of **Table**. The superclass of an imported class is defined to be the superclass of the class in the exporting schema if that superclass is also exported in the *same* package. If that superclass is *not* exported, as in our current case, the superclass of the imported class is the first system class in the superclass hierarchy; in this case, the **Table** system class. Note that if we had also exported the **DateTable** class in the **DateSelectorPackage**, it would have been the superclass of **DateTableSelector**. If instead we had exported the **DateTable** class in a completely different package and then imported that into our **DateSelectorTester** schema, both the **DateTable** and **DateTableSelector** classes would have the *same* superclass; that is, **Table**.

The consequence of these rules means that when a method is searched for locally on an object of the imported class **DateTableSelector**, it will be searched for locally on the chain of classes **DateTableSelector**, **Table**, **Control**, **Window**, **Object**. If no method of that name is found on that chain, a switch will be made into the exporting schema and the method will be searched for on the chain **DateTableSelector**, **DateTable**, **Table**, **Control**, **Window**, **Object**.

For example, if the statement **write dts.display;** is executed in the **DateSelectorTester** schema, where the variable **dts** is of type **DateTableSelector**, a **display** method would be found within the local hierarchy (by default, the one on class **Object**). The following image is an example of the **DateSelectorTester** schema Class Browser.



```
dateTableSelect1_click(table: DateTableSelector input) updating;
begin
    if table.DateTableSelector.dateSelected <> null then
        write table.DateTableSelector.dateSelected;
    endif;
end;
```

Access is public. Event method. Modified by cnwbm1 [6.0.17.60503] 12 August 2003, 16

However, there are a number of additional rules that need to be kept in mind when importing classes.

- Implementing locally any constructor (**create**), destructor (**delete**), or mapping method is not permitted. When the statement **create dts;** is therefore executed, the context is switched to the exporting schema, and all **create** methods on the chain **DateTableSelector**, **DateTable**, **Table**, **Control**, **Window**, **Object** are executed.

- Event messages cannot be implemented locally. This includes methods such as **click** and **paint** on GUI classes, and timer and notification handlers such as **timerEvent**, **userNotification**, and **sysNotification**.

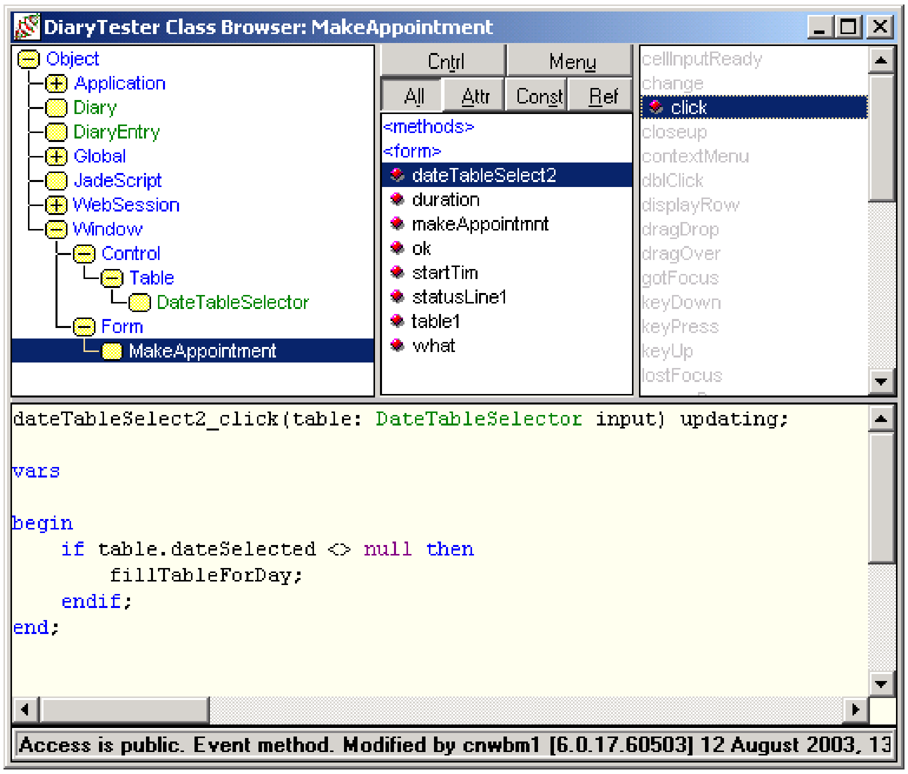
  One reason for this is that such methods would always be found locally, as they are defined on system classes, and the *wrong* method would invariably be found. For example, if the **timerEvent** method were searched for locally, a method would always be found locally as there is an abstract method by that name on the class **Object**. Another reason for looking only for the method in the exporting schema is to avoid compromising the behavior defined by the package implementation.

Although we cannot override the generic **click** method defined on the exported **DateTableSelector** class, we can define a **dateTableSelect1_click** method to the local instance of the control **dateTableSelect1** that appeared as a result of painting an instance of the control on the form. As can be seen from the definition of this method in the previous image, this method writes the date held in the property **dateSelected** if the user clicked on a date in the table. This method would be called when the generic **click** method calls **inheritMethod(table)** near the end of the method.

Combined Appointment Book Example

Having written both a **DateSelectorPackage** and a **DiaryPackage**, we can use both of these to implement a simple appointment book by extending our **DiaryTester** schema.

After importing both, the **DiaryTester** schema Class Browser shown in the following image displays all three imported classes.



DiaryTester Class Browser: MakeAppointment

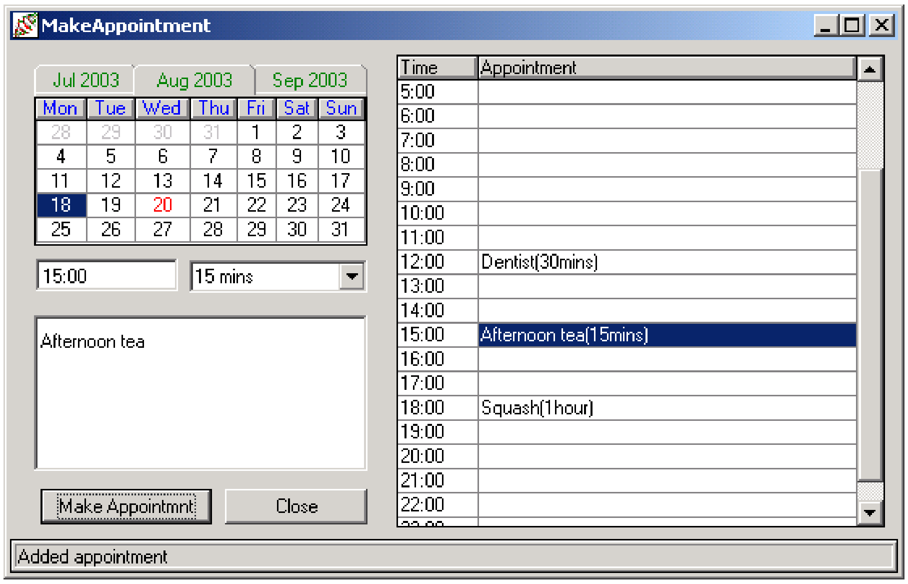
```
dateTableSelect2_click(table: DateTableSelector input) updating;

vars

begin
    if table.dateSelected <> null then
        fillTableForDay;
    endif;
end;
```

Access is public. Event method. Modified by cnwbm1 [6.0.17.60503] 12 August 2003, 13

We now add a MakeAppointment form, as shown in the following image.



MakeAppointment

Jul 2003		Aug 2003		Sep 2003		
Mon	Tue	Wed	Thu	Fri	Sat	Sun
28	29	30	31	1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	31

15:00 | 15 mins

Afternoon tea

Time	Appointment
5:00	
6:00	
7:00	
8:00	
9:00	
10:00	
11:00	
12:00	Dentist(30mins)
13:00	
14:00	
15:00	Afternoon tea(15mins)
16:00	
17:00	
18:00	Squash(1hour)
19:00	
20:00	
21:00	
22:00	

Added appointment

The form includes a date selector control that has a **dateTableSelect2_click** method. This method is called when the user selects a day on the calendar. This calls a local method **fillTableForDay**, which uses imported methods **getDayEntriesBegin** and **getNextDiaryEntry** on **Diary** to obtain the **DiaryEntry** instances from the **Diary** and fill the table on the right of the form with the appointments. Appointments can be added to the **Diary** by filling in the details and then using the imported method **makeAppointment** on **Diary**.

Although this is a rather rudimentary appointment book, it would be straightforward to extend it to provide additional facilities such as multiple diaries, importing or exporting appointments, and the other features that are normally provided by appointment books.

By using packages to develop the basic facilities rather than building them all into a single schema, we can clearly separate the implementation of the basic facilities from their use, and they can easily be imported into other schemas that require diary or date-selecting facilities. Note that it would be easy to package up the final schema to produce an **AppointmentBookPackage**, which could also be imported into schemas to provide a complete appointment book facility.

How do Packages Call User Methods?

Finally, we address the question of how a package can cause actions to occur back in the schema that imported them. This topic, which covers issues that are more advanced than those covered earlier in this document, can be skipped on an initial reading.

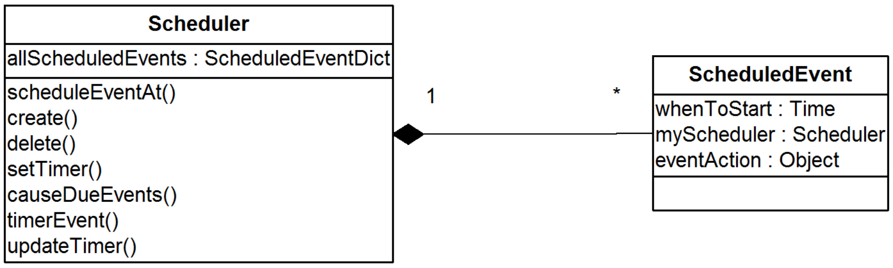
A feature common to many appointment book systems is the ability to associate an alarm with an appointment so that it is activated when the starting time is reached. This alarm may send an e-mail message, bring up an alert form, or perform some other action to alert the user of the impending appointment. How might this feature be added to our example system?

We will create a new **CronSchema** schema that exports a **CronPackage** with classes **Scheduler** and **ScheduledEvent**. The method **scheduleEventAt** on the root transient class **Scheduler** will be used to add new events.

```
scheduleEventAt(when: Time; action: Object) updating;
vars
    se: ScheduledEvent;
begin
    create se transient;
    se.whenToStart := when;
    se.eventAction := action;
    se.myScheduler := self;
    updateTimer;
end;
```

The **when** parameter specifies the time at which we want the event to occur and the **action** specifies what is to occur. The **scheduleEventAt** method creates a new **ScheduledEvent** transient object, initializes the **whenToStart** and **eventAction** attributes, and then sets the reference **myScheduler**. This adds it to the **allScheduledEvents** collection, (which is a queue sorted in ascending time order), before calling **updateTimer**.

The following image shows the **ScheduledEvent** and **Scheduler** classes defined in the **CronSchema** schema.



The **updateTimer** method shown in the following example begins by calling **causeDueEvents** to start any events that are due to run. It then checks if a timer is set to fire when the first event in the queue is scheduled and if there is none, calls **setTimer** to arm a timer by calling **beginTimer**.

```
updateTimer();
// Check that the timer will fire for the first event in the
// queue and if not, cancel existing timer and start another
vars
    se: ScheduledEvent;
    option: Integer;
    timeLeft:  Integer;
begin
    causeDueEvents;
    se := allScheduledEvents.first;
    if se = null then
        return;
    endif;
    if getTimerStatus(0, option, timeLeft) then
        // already a timer, check it corresponds to time of se
        if se.whenToStart - app.actualTime.time > timeLeft then
            return;
        endif;
    endif;
    // cancel current timer and start again
    endTimer(0);
    setTimer;
end;
```

When the timer fires, it calls the **timerEvent** method, which in turn calls **causeDueEvents** to start any due events and then **setTimer** to re-arm the timer.

How can the **causeDueEvents** method cause events to happen in the schema that called the package? It cannot call a method in that schema, as it will have no knowledge of such methods. It *does* have access to the **eventAction** object that was passed and saved when the event was initially scheduled, but does not know what type it is.

The approach we will take will be to use notifications. This approach will allow the actions to take place in the importing schema with the correct context and also allow **CronPackage** to be imported into multiple schemas, because it does not need to know anything about the type of the **eventAction** object. As the designer of the package, we must decide on a receiver for the notifications. In this design, we will choose the user **app** for this object. The notification, shown in the following example, will be registered in the constructor of the **Scheduler**.

```
create() updating;
begin
```

```
        process.getProcessApp.beginNotification(process, Cron_Event_Type, 0, 0);
    end;
```

This registers the notification on the object **process.getProcessApp**, which is the main process **Application** object, (that is, the **app** in the importing schema) with **process** being the notification target object on which the notification is invoked. The destructor terminates the notification, by calling **endNotification**. As can be seen in the following **causeDueEvents** method, the notification is caused by the call to **process.causeEvent**, which will cause the method **userNotification** on the importing schemas application class to be invoked, passing it the saved **eventAction** object.

```
    causeDueEvents();
    // Called when timer fires. Inspect all of the events at
    // the start of the queue and call all those due.
    vars
        se : ScheduledEvent;
    begin
        foreach se in allScheduledEvents do
            if se.whenToStart > app.actualTime.time then
                return;
            endif;
            // cause event passing eventAction as action to perform
            if se.eventAction <> null then
                process.causeEvent(Cron_Event_Type, true, se.eventAction);
                se.myScheduler := null;
                delete se;
            endif;
        endforeach;
    end;
```

The **CronPackage** will export only the **Scheduler** class and expose the **scheduleEventAt** method, hiding all of the complications of how events are queued, the setting of timers, and the way events are caused to occur.

A schema **CronTester** that imports **CronPackage** can use it by adding a **userNotification** method like that shown in the following example to the application **CronTester** class.

```
    userNotification(eventType: Integer; theObject: Object;
                     eventTag: Integer; userInfo: Any) updating;
    begin
        if not userInfo.isKindOf(Method) then
            write 'Error...method expected';
            return;
        endif;
        sendMsg(userInfo.Method.name);
    end;
```

It can then create a transient instance of **CronPackage::Scheduler** and schedule an event at a specific time by code like that shown in the following fragment.

```
    scheduler.scheduleEventAt(app.actualTime.time + 5000, CronTester::eventOne);
```

This will result in the **userNotification** method being called in five seconds, which will in turn call the method **eventOne** defined on the application class **CronTester**.

Summary

In this document, we have seen how packages extend the existing schema structure to allow access to the functionality provided, without the need for the schema to be among the superschemas. The package developer can decide which classes, and which features of those classes, should be exposed by exporting just those classes and features. This allows access to the functionality of the package in a controlled manner.

The user of the package can then import the package and need only be concerned with the exposed interfaces in a type-safe manner, without being exposed to irrelevant implementation detail.

This document gives examples of how packages can be developed and used, by developing a simple appointment book system. Some of the design decisions were made to expose new and pertinent features of packages, possibly at the expense of efficiency or simplicity. However, the developed system could be extended to form the basis of a realistic appointment book system.