

Programmer's Manual

© Tommi Johtela 1998

Copyright © 1998 by Tommi Johtela. All rights reserved.

A permission to redistribute this document is granted as long as no fee is charged and the document's contents is not modified in any way.

E-mail: tjohtela@cs.utu.fi WWW: <http://www.cs.utu.fi/tjohtela>

Contents

Contents.....	2
Introduction.....	3
Class Hierarchy and Applicability	4
PCL Framework Classes.....	4
<i>Class TItem.....</i>	<i>5</i>
<i>Class TContainer.....</i>	<i>5</i>
<i>Class TSequence</i>	<i>6</i>
<i>Class TSet.....</i>	<i>7</i>
<i>Class TDictionary.....</i>	<i>8</i>
Using Persistence	9
<i>Implementing the Read and Write Methods</i>	<i>9</i>
Writing the State of an Item to a Stream	10
Reading the State of an Item from a Stream.....	10
<i>Registering Persistent Classes</i>	<i>11</i>
<i>Saving Items to a Stream or File</i>	<i>11</i>
<i>Loading Items from a Stream or File.....</i>	<i>11</i>
Design Philosophy behind the PCL Framework.....	12
<i>Model–View–Controller Approach.....</i>	<i>12</i>
PCL Utility Classes.....	13
<i>Class THashTable.....</i>	<i>13</i>
<i>Class TStringTable.....</i>	<i>16</i>
<i>Class TBitSet</i>	<i>16</i>

Introduction

Perhaps the biggest shortcoming of the Delphi's Visual Component Library (VCL) is the lack of generic container classes. The only general data structure that Delphi offers is the *TList* class, which is a simple array-like container for storing untyped pointers. Compared to other object oriented languages, like C++ or Java, this kind of container supply is simply inadequate. For example, C++ includes the Standard Template Library (STL), which is a very powerful generic data structure library. Java has also recently added a set of container classes to its standard libraries (JDK 1.2), whereas Delphi has stuck with the *TList* class ever since the first version. To address this shortcoming I wrote a small class library that presents three elementary abstract data types (ADTs):

- The *TSequence* class represents an array-like, sequential data structure. The class uses Delphi's *TList* class for its implementation.
- The *TSet* class represents an unordered set structure, which can insert and remove items efficiently. The class is implemented with a hash table.
- The *TDictionary* class is a random-access data structure, which maps string keys to the items of a dictionary. Like the *TSet* class, a dictionary can also insert and remove items quickly.

All of these container classes inherit from the abstract base class *TContainer*, which defines the common functionality of the containers.

The most important property of these containers is *persistence*. This means that the objects stored in a container can be written to a data stream, like a file or network connection. The state of these objects can be restored later by reading them back from the stream. Each item in a container is responsible for saving and restoring its own state. For this reason, the item classes must be derived from the class *TItem* instead of *TObject*. This somewhat limits the usability of the containers but, on the other hand, the class design is much simpler when the items share a common superclass.

In addition to these abstract data types, the library includes a set of utility data structures similar to the VCL classes *TList* and *TStringList*. In fact, the interfaces of these classes are very similar to those found in the VCL, so they can be used, in some cases, interchangeably with the VCL classes. These utility classes are:

- *THashTable* is a hash table data structure, which can store and retrieve data quickly using integer keys (see for example *Introduction to Algorithms* by Cormen *et al.* for a description of hash tables).
- *TStringTable* is a descendant of *THashTable* that uses strings as its keys.
- *TBitSet* is a set data structure that is implemented as a bit string. It can be used to store positive integers in a compact form. Common set operations, such as union or intersection, can also be performed between *TBitSet* objects. Since these operations are coded mainly in assembly language, they are very fast.

Together all these classes (plus few more) form the Persistent Container Library (PCL). The design goals of this library are:

- **Simplicity.** Using the containers should be easy. This shortens the learning curve of the library.
- **Compactness.** Classes should be lightweight so that their inclusion does not increase the size of the executable code excessively.
- **Extensibility.** It should be easy to add new data structures to the library. There should be an abstract superclass that defines the obligatory services of the containers.
- **Consistency with the VCL.** The interfaces of the classes should be similar to those found in the VCL. The method names and parameters should be the same

than in the VCL, and the design philosophy of the Delphi standard libraries should be followed.

The library requires (at least) version 4 of the Delphi.

Class Hierarchy and Applicability

Figure 1 shows the class hierarchy of the Persistent Container Library.

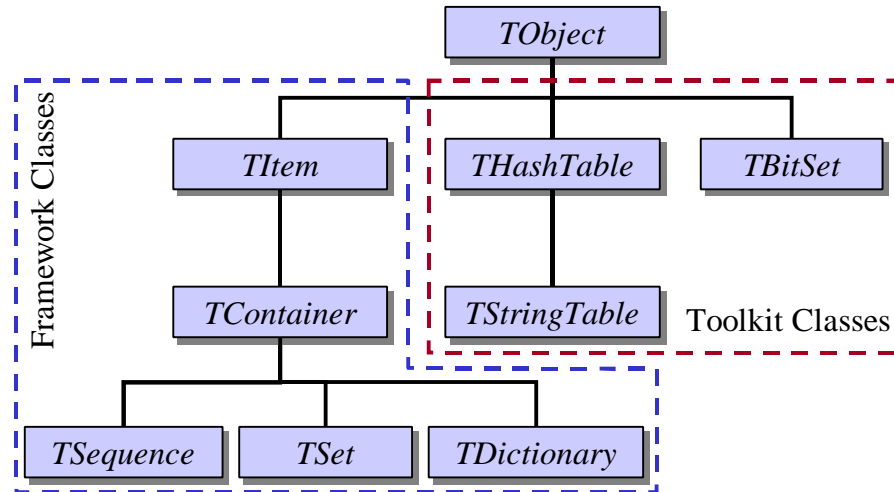


Figure 1: The class hierarchy of the library

The classes in the library can be divided into two categories:

1. Framework classes that are used to build applications with persistent states. To use the services of the library a client class has to inherit from the *TItem* class and implement a few methods (e.g., *Read* and *Write* to be able to use the persistence mechanism).
2. Utility classes that provide a common data structure toolkit. These classes are similar to the VCL classes like *TList* or *TStringList*. They are usually used by composition instead of inheritance, which makes them applicable practically anywhere.

It is important to make the distinction between toolkit classes that provide general services and framework classes that somewhat dictate the structure of an application using them. This gives the user a clear conception about the usability of a class library in a client application. Frameworks are used to construct certain types of applications, whereas toolkits contain helper classes that can be used in any application. Framework classes also impose some responsibility to a client class that has to implement some functionality to be able to use the framework. These approaches can also be complementary, like in this case. The framework classes encapsulate the toolkit classes in well-defined interfaces adding some functionality in the process. For instance, *TDictionary* uses the *TStringTable* class for its implementation.

PCL Framework Classes

This section introduces the framework part of the Persistent Container Library. The PCL framework provides the following services:

- **Memory management.** Every container class has a flag, which tells if the container *owns* the items that it contains. If it does, the items are disposed automatically before the container is disposed. This mechanism is similar to what used with the VCL components, so it should be already familiar to the majority of the Delphi developers. The persistence mechanism uses this information also to determine whether to write an item or only its reference to a stream.

- **Notification of attachment/detachment.** An item class can be notified when it is inserted to or removed from a container. For example, if an item class maintains a reference to the container it resides in, it can automatically update this reference when its parent container changes.
- **Persistence.** The container and its items can be saved to or restored from a stream. The framework provides reader and writer classes using which the item classes can save and restore their state. The container saves and restores its items automatically preserving their state and references to other objects.

Note: There are some limitations concerning this mechanism that are explained in the section “Using Persistence” at page 9.

Every class belonging to the PCL framework is described separately below.

Class TItem

The *TItem* class is the abstract superclass inherited by every item class stored in the containers and also by the container classes themselves. *TItem* defines four empty method bodies, which may be overridden by a subclass. The declaration of the class is shown in Listing 1.

Listing 1: *TItem* class

```
TItem = class
protected
  constructor Read(Reader: TItemReader); virtual;
  procedure Write(Writer: TItemWriter); virtual;
public
  procedure Associate(AObject: TObject); virtual;
  procedure Dissociate(AObject: TObject); virtual;
end;
```

Procedures *Associate* and *Dissociate* are used for notification of attachment and detachment. The *Associate* procedure is called by a container when an item object is inserted to it and *Dissociate* is called upon removal of an item. The container's reference is passed to both of these procedures as an argument. The item object can thus react to these changes, for example, by updating its parent references. The *Dissociate* method is also called when the container is disposed.

This mechanism can also be used in a more general way when implementing associations between classes. Any object, which establishes a reference to an object derived from the *TItem* class, can notify the item about the new association by calling its *Associate* method. Conversely, when the referring object no longer points to the item object or the referring object disposed, it calls the item's *Dissociate* method. This way the item object knows, which objects refer to it and can maintain opposite pointers back to them.

The *Read* and *Write* methods are used to store and restore the item's state to/from a stream. A reader or writer object corresponding to the input/output stream is given as an argument to these procedures. Each item object is responsible for serializing its own state. *TItemReader* and *TItemWriter* classes (derived from the Delphi classes *TReader* and *TWriter* correspondingly) provide with the means for performing this task. The serialization mechanism is discussed in more detail at page 8 in the section “Using Persistence”.

Class TContainer

TContainer is the abstract base class of all the containers. It inherits from the *TItem* class and overrides the *Read* and *Write* methods to make the containers persistent. The declaration of the class is shown in Listing 2.

Listing 2: *TContainer* class

```
TContainer = class(TItem)
private
```

```

    FHost: TItem;
    FOwnsItems: Boolean;
protected
    constructor Read(Reader: TItemReader); override;
    procedure Write(Writer: TItemWriter); override;
    function GetCount: Integer; virtual; abstract;
public
    constructor Create(AHost: TItem; AOwnsItems: Boolean =
        false); virtual;
    destructor Destroy; override;
    procedure Assign(Source: TContainer); virtual; abstract;
    procedure Clear; virtual; abstract;
    function First: TItem; virtual; abstract;
    function Next: TItem; virtual; abstract;
    property Count: Integer read GetCount;
    property Host: TItem read FHost;
    property OwnsItems: Boolean read FOwnsItems write
        FOwnsItems;
end;

```

TContainer defines a virtual constructor that must be used to create a container. It takes two arguments first of which tells the host object, that is, the object in which the container resides. This reference is stored to the *Host* property and can be later used, for example in the *Associate* and *Dissociate* methods of the item classes to get the reference to the parent object of an item. The second argument is optional and tells if the container owns its items. If it does, the items are disposed automatically when the container is disposed. The default value of this argument is *FALSE*. This value can be changed afterwards by modifying the *OwnsItems* property.

The *Assign* method can be used to replace the contents of a container with the items of another *TContainer* object. This operation works as follows: first all the items in a container are removed by calling the *Clear* method. Then the items of the container specified by the *Source* parameter are copied to this container. The source container is not affected by the operation. The *Assign* method is declared abstract in *TContainer*, so it is up to the subclasses to actually implement this operation.

Note: Only the references of the items in a source container are copied by the *Assign* method not the items themselves. Thus, after copying there exists at least two containers, which contain the same *TItem* objects. This may cause problems, for example, if both containers own the items they contain. An item can naturally have only one owner at a time.

The *Clear* method removes all the items in a container. If the *OwnsItems* property is set then the item objects are also freed. The implementation of this method is deferred to subclasses by declaring it as abstract.

Methods *First* and *Next* provide with a way to iterate through the items in a container. The *First* function returns the first item in a container or *NIL*, if the container is empty. The *Next* function returns the next item in the container or *NIL*, if there are no more items left. Both of these methods are declared as abstract in the *TContainer* class, so they must be implemented by the inheriting classes. Note that the order in which the items in a container are returned is not restricted in any way. A container may iterate through its items in an arbitrary order, even when the container keeps the items in some predefined order.

The *Count* property tells the number of items in a container. The property gets its value from the abstract method *GetCount*. Unlike in the *TList* class the value of this property may not be changed. This property is read-only.

Class TSequence

TSequence is a container that keeps its items in a sequential, random-access structure. The class delegates most of its operations to the Delphi's *TList* class wrapping it to the *TContainer* interface. The declaration of the class is shown in Listing 3.

Listing 3: *TSequence* class

```

TSequence = class(TContainer)
private
    FList: TList;
    FIndex: Integer;
protected
    constructor Read(Reader: TItemReader); override;
    procedure Write(Writer: TItemWriter); override;
    function GetCount: Integer; override;
    function GetItems(Index: Integer): TItem;
public
    constructor Create(AHost: TItem; AOwnsItems: Boolean =
        false); override;
    destructor Destroy; override;
    procedure Assign(Source: TContainer); override;
    function Add(Item: TItem): Integer;
    procedure Clear; override;
    procedure Delete(Index: Integer);
    procedure Exchange(Index1, Index2: Integer);
    function First: TItem; override;
    function Next: TItem; override;
    function IndexOf(Item: TItem): Integer;
    procedure Insert(Index: Integer; Item: TItem);
    procedure Move(CurIndex, NewIndex: Integer);
    function Remove(Item: TItem): Integer;
    property Items[Index: Integer]: TItem read GetItems;
        default;
end;

```

Since *TSequence* is an array-like, random access data structure, a client can refer to its items directly using an integer index. The *Items* array property can be used to read the *Index*'th item of the sequence. However, items cannot be inserted to a *TSequence* object using the *Items* property (i.e., the property is read-only). Its is also the default property of the class.

The class provides two methods for both insertion and removal operations. To insert an item to a *TSequence* object, a client can use either the *Add* or *Insert* method. *Add* inserts the item to the end of the sequence, and *Insert* to the position specified by the *Index* parameter. In the same manner, the *Remove* method removes the item which reference is given as an argument, whereas *Delete* removes the item specified by its position (index). These methods work in the same way as the ones in the *TList* class, so their usage should be familiar to an experienced Delphi programmer.

Finally, the *IndexOf* method returns the position of the given item or -1 , if the item is not in the sequence.

Class TSet

TSet is a container class that represents the mathematical set structure. It is an unordered structure that contains *unique* items, that is to say, no duplicates. The insertion and removal operations are fast in the *TSet* class, whereas iteration through the items in the set is not as efficient as, for example, in the *TSequence* class. This is due to the internal representation of the set, which is done using a hash table (the *THashTable* class). Listing 4 shows the declaration of the *TSet* class.

Listing 4: *TSet* class

```

TSet = class(TContainer)
private
    FTable: THashTable;
protected
    constructor Read(Reader: TItemReader); override;
    procedure Write(Writer: TItemWriter); override;
    function GetCount: Integer; override;
public
    constructor Create(AHost: TItem; AOwnsItems: Boolean =
        false); override;

```

```

destructor Destroy; override;
procedure Assign(Source: TContainer); override;
procedure Clear; override;
function First: TItem; override;
function Next: TItem; override;
function Belongs(Item: TItem): Boolean;
procedure Insert(Item: TItem);
procedure Intersection(Other: TSet);
function Remove(Item: TItem): Boolean;
procedure Subtract(Other: TSet);
procedure Union(Other: TSet);
end;

```

The function *Belongs* can be used to test if a given item is in the set. The function returns TRUE if the item belongs to the set, and FALSE otherwise.

The *Insert* and *Remove* methods are used to insert an item to a set or remove an item from it. If the user inserts a same item to the set twice, only one reference is actually stored to the set. The return value of *Remove* tells if the given item was actually in the set. No error occurs, if the user tries to remove a nonexistent item from a set.

TSet presents also common binary set operations: union, intersection, and subtraction. The result of these operations is stored to the current *TSet* object, that is, the object on which the method was called. The *Intersection* method performs an intersection operation between the current set and another set given as an argument. Intersection leaves only those items in the resulting set that belong to both sets. The *Subtract* procedure removes all the items from the current set that are present in both sets. The *Union* procedure combines two sets so, that items belonging to either of the source sets are in the resulting set.

Class TDictionary

The *TDictionary* class represents a random-access map structure that uses strings as keys to its items. This container is also implemented internally as a hash table (using the *TStringTable* utility class).

Listing 5: *TDictionary* class

```

TDictionary = class(TContainer)
private
    FTable: TStringTable;
protected
    constructor Read(Reader: TItemReader); override;
    procedure Write(Writer: TItemWriter); override;
    function GetCount: Integer; override;
    function GetItems(const Key: string): TItem;
public
    constructor Create(AHost: TItem; AOwnsItems: Boolean =
        false); override;
    destructor Destroy; override;
    procedure Assign(Source: TDictionary); reintroduce;
    procedure Clear; override;
    function First: TItem; override;
    function Next: TItem; override;
    procedure Insert(const Key: string; Item: TItem);
    function Remove(const Key: string): TItem;
    property Items[const Key: string]: TItem read GetItems;
        default;
end;

```

TDictionary (Listing 5) adds only two methods and one property to the *TContainer* interface. The *Insert* method is used to insert an item with a key given as the first argument to a dictionary. Similarly, the *Remove* function removes an item with the given key from a dictionary. To access the items in a dictionary, the class provides the *Items* array property. The key of the accessed item is used as an index argument.

It is not possible to add new key–item pairs or modify the old ones using the *Items* property (i.e., the property is read-only).

Using Persistence

There are some restrictions concerning the objects that can be serialized using the persistence mechanism. First of all, only the descendants of the *TItem* class can be made persistent. This means that no VCL or other legacy classes can be serialized using the framework.

Secondly, the objects that are to be written to a stream by the framework must form a closed system, which has a single root object that owns all the other serialized objects. The state of the application is saved by writing this root object to a stream. All the subobjects are then recursively written to the same stream by their owners. This might seem as a big restriction at first, but it is actually a good thing since it encourages the user to create hierarchical object structures, and therefore enforces good class designs. The persistence mechanism used to save Delphi components uses exactly the same kind of technique, so it has found to be a good solution to the persistence problem. The reason that I am not using Delphi's own persistence mechanism is that it is designed specifically for the needs of the component classes and doesn't suit very well for more generic purposes.

To use the persistence mechanism provided by the framework, the user has to follow the five steps described below:

1. Derive all the classes that need to be serialized from the *TItem* class.
2. Implement the virtual *Read* and *Write* methods of the *TItem* class.
3. Call either *RegisterItemClass* or *RegisterItemClasses* procedure in the initialization block of the application to register the persistent classes.
4. If some persistent objects reside in a container that handles the serialization, make sure that these objects are owned by exactly one container. The owner container is responsible for serializing its items. Other containers (that are not owners) may refer to persistent objects, but they do not store them—instead they write only the references of the contained objects to a stream.
5. To write the objects to a stream, call either the *SaveItemToStream* or the *SaveItemToFile* procedure giving the root object in the object hierarchy (the one that owns all the other objects) as an argument. Conversely, to read the objects from a stream, call either the *LoadItemFromStream* or *LoadItemFromFile* function. They return the root object of the hierarchy along with all the other saved objects.

We will now walk through the steps in the preceding list in more detail to give a clearer picture of how the persistence mechanism is used in the framework.

Implementing the Read and Write Methods

The policy of the framework is such that each persistent object is responsible for serializing its own state. This is done by implementing the virtual *Read* and *Write* methods of the *TItem* class. These methods are declared empty in *TItem* so they do not have to be implemented in every inheriting class. However, if the inheriting class intends to use the persistence services, this task is obligatory. The actual reading and writing of an object's state is delegated to the *TItemReader* and *TItemWriter* classes, which are derived from the VCL classes *TReader* and *TWriter* respectively. These reader and writer classes always represent some input/output stream and provide with means to read and write common data types from/to a stream. *TItemReader* and *TItemWriter* extend these classes so that also *TItem* descendants can be read from or written to a stream.

Writing the State of an Item to a Stream

The writing of an item's state is done using the *TItemWriter* class, which extends the VCL class *TWriter*. All the elementary data types, including integers, real numbers, strings, etc. can be serialized using the methods of the *TWriter* class. Refer to the VCL documentation for further information about the usage of the reader and writer classes.

Listing 6: *TItemWriter* class

```
TItemWriter = class(TWriter)
public
  constructor Create(Stream: TStream);
  procedure WriteClassTable;
  procedure WriteItem(AItem: TItem);
  procedure WriteReference(AResource: TItem);
end;
```

The stream that the writer object corresponds to is given as an argument to the constructor of the *TItemWriter* class (Listing 6). The class adds two procedures to the *TWriter* class that can be called in the *Write* method of an item class. The *WriteItem* method writes an item object to the stream. This method effectively calls the *Write* method of the item given as an argument, which recursively writes all the objects owned by it to the same stream. *WriteItem* can be called only for those items that are owned by the serialized item. If the serialized item refers to another item that it does not own, a reference to this item must be written to the stream using the *WriteReference* method.

Reading the State of an Item from a Stream

A *TItemReader* object, which is used to read an item's state, is given as an argument to the *Read* method of the *TItem* class. This method is in fact a constructor, which has to allocate first all the dynamic resources that the object requires. Then, the fields of an item are read using the methods the *TItemReader* class. The declaration of this class is shown in Listing 7.

Listing 7: *TItemReader* class

```
TItemReader = class(TReader)
private
  FClasses: THashTable;
  FItems: THashTable;
  FReferences: TList;
  FCallBackRefs: TList;
public
  constructor Create(Stream: TStream);
  destructor Destroy; override;
  procedure ReadClassTable;
  function ReadItem: TItem;
  procedure ReadReference(var AReference: TItem);
  procedure ReadRefCallBack(CallBack: TResolveCallBack);
  procedure ResolveReferences;
end;
```

Again, the stream corresponding to the reader object is given as an argument to the constructor. The *ReadItem* method reads an item owned by the current item from a stream, and *ReadReference* reads a reference to another item object.

Note: A field variable, which points to another item object, is given to the *ReadReference* procedure by reference. This is because the item, to which the field points to, is not necessarily read from the stream at the time when the *ReadReference* procedure is called. In fact, all the references are resolved in separate pass after all of the items are read from the stream. This is done automatically by the global *LoadItemFromStream* procedure, so that the user needs not to worry about this chore. The user should note, however, that a reference field is not initialized right after the *ReadReference* method returns, but after *all* the stored items have been read.

Registering Persistent Classes

All the persistent item classes have to be registered with the framework so that the class table, which tells the types of the saved items, can be stored to a stream along with the objects. This registration is done usually in the initialization block of the unit in which the classes are declared. As an example, Listing 8 shows the code that registers the container classes provided by the framework. This code resides in the initialization block of the unit “Containers.pas”.

Listing 8: Registration of the container classes

```
initialization
  RegisterItemClasses([TSequence, TSet, TDictionary]);
```

Either the *RegisterItemClass* or *RegisterItemClasses* procedure (Listing 9 shows the declarations of these procedures) can be used to register the persistent classes. The only difference between these procedures is that *RegisterItemClasses* accepts a variable-length array of classes, whereas *RegisterItemClass* takes only one class as an argument.

Listing 9: Registration procedures

```
procedure RegisterItemClass(AClass: TItemClass);
procedure RegisterItemClasses(AClasses: array of TItemClass);
```

Saving Items to a Stream or File

Saving persistent items to a stream is easy. Just call the global *SaveItemToStream* procedure giving the root object as an argument (Listing 10). The root object is the item that either directly or indirectly owns all other items. Usually the root object represents some fundamental object of the application such as a document in a word processor or a picture in a drawing application. The second argument of the *SaveItemToStream* procedure specifies the stream to which the items are stored. This stream must be write enabled.

Listing 10: Procedures used to save items

```
procedure SaveItemToStream(Item: TItem; Stream: TStream);
procedure SaveItemToFile(Item: TItem; const FileName: string);
```

Alternatively, if items are saved to a disk (as usually is the case), the procedure *SaveItemToFile* can be used. This procedure takes a file name as its argument instead of a stream. Listing 11 demonstrates how to create an appropriate stream for the *SaveItemToStream* procedure by showing the implementation of the *SaveItemToFile* procedure. Note, how the stream is created on the fly using the try—finally statement. This ensures that the file is closed and all other resources freed in case of failure or some other interruption in the saving process.

Listing 11: *SaveItemToFile* procedure

```
procedure SaveItemToFile(Item: TItem; const FileName: string);
var Stream: TFileStream;
begin
  Stream := TFileStream.Create(FileName, fmCreate);
  try
    SaveItemToStream(Item, Stream);
  finally
    Stream.Free;
  end;
end;
```

Loading Items from a Stream or File

Loading an entire object graph from a stream is done by reading the root item of the system using the *LoadItemFromStream* function (Listing 12). This function returns the root object along with all the subobjects saved with it. The stream from which the

item is read is given as an argument. Similarly, the *LoadItemFromFile* function restores the saved object graph from a file, which name is given as an argument.

Listing 12: Functions used to load items

```
function LoadItemFromStream(Stream: TStream): TItem;  
function LoadItemFromFile(const FileName: string): TItem;
```

These functions work in the same manner as procedures used to save items to a stream or file. Usually the *LoadItemFromStream* function is called inside a try—finally block as shown in Listing 13, which shows the implementation of the *LoadItemFromFile* function.

Listing 13: *LoadItemFromFile* function

```
function LoadItemFromFile(const FileName: string): TItem;  
var Stream: TFileStream;  
begin  
  Stream := TFileStream.Create(FileName, fmOpenRead or  
    fmShareDenyWrite);  
  try  
    Result := LoadItemFromStream(Stream);  
  finally  
    Stream.Free;  
  end;  
end;
```

Design Philosophy behind the PCL Framework

As stated in the section “Class Hierarchy and Applicability” at page 4, a framework always dictates somewhat the structure of applications using the framework. For example, a Delphi programmer who uses the Visual Component Library has to derive all the windows his/her program uses from the *TForm* class. Thus, the VCL can be seen as a framework for building graphical Windows programs. Programs, which use the VCL must follow certain rules and conventions to be able to use the services provided by the library. As another example, consider the Delphi's component architecture. To make a custom component with a window handle, the programmer has to derive his/her component from the *TCustomControl* class and override the *Paint* method. All this results that the programs build on the VCL framework contain similar structural parts, which gives uniformity to the design process and makes it easier to get started with the programming of a new application.

The VCL framework deals mainly with the user interface of an application and provides services for making new windows and controls easily. Other kinds of services, such as providing persistent data structures, are left to the application programmer. This is where the PCL framework comes to the picture. The application classes representing the structure and logic of a modeled problem can use the PCL framework to get the same kind of services that is provided by the VCL components. The VCL and PCL libraries are thus complementary to each other and, consequently, form a combined design framework.

Model–View–Controller Approach

The PCL framework itself follows the Model–View–Controller approach as described in the book *Design Patterns* by Gamma *et al.* This approach separates the data that a program is modeling from its representation. The idea is to isolate the classes modeling the problem environment (sometimes called document classes) in their own unit and make them independent of the user interface classes such as windows and controls that visualize and manipulate the model (view and controller classes). As an example, consider a spreadsheet application, which essentially manages cells filled with various types of data arranged into rows and columns. The data is stored internally in some data structure, for example, in a two-dimensional array. This structure can be represented graphically in many ways (e.g., as a table or chart). User can manipulate

the data using the user interface; for example, switch between views or enter new values to cells. To summarize, the model is an independent unit that does not know about the view or the controller. Similarly, the view shows the model visually and is not concerned about the ways to modify the model. Lastly, the controller knows how to manipulate the model without any knowledge about the view.

The Model–View–Controller approach is used in many object-oriented libraries that provide with graphical user interfaces. Most notably, the Microsoft Foundation Classes (MFC) shipped with the Microsoft's Visual C++ development environment follows this approach quite faithfully. The only difference is that in MFC terminology a model corresponds to a document.

The original idea of the Model–View–Controller approach was to split the user interface into two logical units, one for providing a views to the modeled data on a screen, and other for controlling the model with, for example, menus or buttons. Since the VCL combines these elements rather tightly with its event handling mechanism, it is usually quite difficult to separate the controllers and views completely from each other. Nevertheless, by doing so the program structure becomes more flexible and extensible.

Note: The action objects (derived from the *TAction* class) in the version 4 of Delphi take a step towards separating the controller classes from the view classes. An action is basically a user command that is triggered by some user interface element, such as menu or keyboard. The command represented by this action can then be executed without any knowledge of the component that triggered it. See the VCL documentation for further information about the action classes.

The PCL framework is primarily used in the model part of an application, that is, in the classes that represent the handled data. The persistent containers can be used as building blocks for modeling the data. Continuing the previous example, the object model of a spreadsheet application could have a root class that represents a single table. Inside the table class could be classes that represent the rows of the table, which in turn contain the cell classes modeling an individual cell. Since the cells of a table are indexed, it is natural to store them in a *TSequence* object, which is an array-like, random-access data structure. By deriving all the classes representing different elements of the spreadsheet table from the *TItem* class and implementing the *Read* and *Write* methods, the table can be made persistent with little effort.

PCL Utility Classes

This section describes the usage of the utility classes of the Persistent Container Library. These utility classes form a toolkit of primitive data structures that can be used for many purposes. The interfaces of these classes resemble the VCL classes like *TList* or *TStringList* and their usage is similar to these classes in many respects. The data structures have been made as customizable as possible, which sometimes exposes the internal representation of the classes, but is nevertheless necessary for providing sufficient freedom to the user. So, instead of hiding the implementation behind an abstraction, these classes expose their representation for the sake of flexibility and efficiency. In practice this means that there is no memory management or persistence support for these classes. Also, the user has to understand the main principles of the data structures that these classes represent, in order to make best use of them.

Each utility class belonging to the Persistent Container Library is described separately in its own section below.

Class THashTable

A hash table is an associative data structure that uses keys to refer to its elements. The elements of a hash table reside in a normal array. A special function called the *hash function* is used to calculate the position of each key–item pair. The hash function tries to distribute the keys evenly across the array, so that there are as few *collisions* as possible. A collision happens, when the hash function maps two different keys to

the same position. Collisions are handled using some resolution method, like chaining or open addressing. The benefits for using a hash table are that its insertion and removal operations are on average very fast. The hash table as a data structure combines the direct access used in vectors with the flexibility of linked lists. For these reasons, the hash table has become very popular data structure for many purposes. Some languages, like Perl or Java, use it very extensively.

The class *THashTable* is a hash table implementation that uses integer keys. The item type of the class is untyped pointer, so virtually any kind of data can be stored in it. The class resides in the unit “Hashtables.pas” and its declaration is shown in Listing 14.

Listing 14: *THashTable* class

```
THashTable = class(TObject)
private
    Alpha: Double;
    FTable: PPointerList;
    FCount: Integer;
    FCapacity: Integer;
    FMaximumFillRatio: Double;
    FPosition: Integer;
    FCollisions: Integer;
    FInsertions: Integer;
    function GetAverageCollision: Real;
    procedure SetMaximumFillRatio(Value: Double);
protected
    procedure Error(const msg: string);
    function Get(Key: Integer): Pointer; virtual;
    function GetIndex(Key: Integer): Integer;
    procedure Grow; virtual;
    function Hash(Key: Integer): Integer; virtual;
    procedure Put(Key: Integer; Item: Pointer); virtual;
    procedure Rehash(OldTable: PPointerList; OldCount:
        Integer);
    procedure SetCapacity(NewCapacity: Integer);
public
    constructor Create;
    destructor Destroy; override;
    procedure Clear; virtual;
    function Current: Pointer; virtual;
    function DeleteCurrent: Pointer;
    function First: Pointer; virtual;
    function Insert(Key: Integer; Item: Pointer): Pointer;
        virtual;
    function Next: Pointer; virtual;
    function NextSame(Key: Integer): Pointer;
    function Remove(Key: Integer): Pointer; virtual;
    procedure Pack;
    property Capacity: Integer read FCapacity write
        SetCapacity;
    property Count: Integer read FCount;
    property MaximumFillRatio: Double read FMaximumFillRatio
        write SetMaximumFillRatio;
    property Items[Key: Integer]: Pointer read Get write Put;
        default;
    property AverageCollisions: Real read GetAverageCollision;
end;
```

Only the public interface of the *THashTable* class is documented here. Those who wish to extend the class by inheritance should refer to the source code for details of the implementation.

The *THashTable* object is created normally using the *Create* constructor, which takes no parameters. Default values are assigned to the properties when a hash table is created. The properties of the class are:

- **Capacity.** Corresponds to the number of items that can be stored to the hash table without reallocating new memory. The value of this property can be changed to speed up the insertions when the upper limit of the number of items stored to a hash table is known beforehand. The default value is 256.

Note: Actually, the value of the *Capacity* property should be set to the number of elements inserted in the hash table divided by the value of the *MaximumFillRatio* property because it is not desirable to fill the hash table completely up to the capacity limit. See the description of the *MaximumFillRatio* property below for further information.

- **Count.** The number of items currently stored to a hash table. This property is read-only. Initially a hash table is empty, so the default value is 0.
- **MaximumFillRatio.** This property tells, how full a hash table may become before allocating more memory for it. As a hash table fills up, the number of collisions also increases. To prevent the deterioration of performance, new space is allocated before the table becomes completely full. The ratio between the number of items (*Count*) and the capacity of a hash table (*Capacity*) is bounded by the value of the *MaximumFillRatio* property. The default value is 0.8. The user should seldom alter this value, since it is set initially to a good trade-off between the number of collisions and the utilization of the memory.
- **AverageCollisions.** This is a read-only property, which tells the average number of collisions of all the insertions thus far. Sometimes, when the number of items stored in a hash table is large and unknown, it may be useful to adjust the maximum fill ratio at run time to better utilize the memory. The value of the *AverageCollisions* property can be used to determine the direction and magnitude of this adjustment. The need for adjustments should, nevertheless, occur only on special situations. Generally, the optimal value of the maximum fill ratio is so difficult to determine that little is gained in the long run by modifying the value of the *MaximumFillRatio* property constantly.
- **Items.** This array property provides access to the items in a hash table. The key of the item is given as an index argument. Items can be both fetched and inserted using this property. If an item with the given key already exists in the hash table, the value of the key is changed. Otherwise a normal insertion occurs.

The public methods of the *THashTable* class are described in the following list:

- **Clear.** Removes all the items in a hash table.
- **First and Next.** The user can iterate through the items of a hash table using these methods. The *First* function returns the first item in a table or NIL, if the table is empty. Similarly, the *Next* function returns the next item in a table or NIL, if there are no more items left. Both these functions change the current item of a table to the one they return.
- **NextSame.** Returns the next item with the same key as the current item or NIL, if the table does not contain such an item.
- **Current.** This function returns the current item in a hash table.
- **DeleteCurrent.** Deletes the current item in a table.
- **Insert and Remove.** These functions are used to insert an item to a hash table and remove an item from it. The key of the inserted item must be a positive integer and the item itself is an untyped pointer. The key–item pair is given as an argument to the *Insert* function. The pointer, which is returned, tells the address of the item pointer. If an item with the same key already exists in the table, a duplicate key is created. Conversely, the *Remove* function removes the first item with the given key and returns it. If an item with the given key is not found in the hash table, NIL is returned.
- **Pack.** Packs a hash table to reduce its memory usage. This is done by setting the capacity to the smallest possible value taking into account the number of items

and the maximum fill ratio. Since the packing is quite time consuming operation, this procedure should be called only after no more items are inserted to the table.

Class TStringTable

The *TStringTable* class (Listing 15) is derived from the *THashTable* class and it has basically the same public interface. Only differences are that the *TStringTable* class uses string keys and contains items of the *TObject* type. The item type is changed to *TObject* to make the interface similar to the Delphi's *TStringList* class.

Listing 15: *TStringTable* class

```
TStringTable = class(THashTable)
private
    function ConvertKey(const Key: string): Integer;
    function FindKey(const Key: string; var Node:
        PStringTableNode): Boolean;
protected
    function Get(const Key: string): TObject; reintroduce;
    procedure Put(const Key: string; Item: TObject);
        reintroduce;
public
    destructor Destroy; override;
    procedure Clear; override;
    function Current: TObject; reintroduce;
    function CurrentKey: string;
    function First: TObject; reintroduce;
    function Insert(const Key: string; Item: TObject): Pointer;
        reintroduce;
    function Next: TObject; reintroduce;
    function Remove(const Key: string): TObject; reintroduce;
    property Items[const Key: string]: TObject read Get write
        Put; default;
end;
```

The *TStringTable* class introduces only one new method namely *CurrentKey*. It returns the key of the current item in a hash table. The signatures of other methods are modified to correspond to the new key and item types using the *reintroduce* keyword provided by Delphi 4. This keyword is used to change the signature of a virtual method in a subclass.

Class TBitSet

The *TBitSet* class is a set data structure that can be used to store positive integers. In some sense, it is the dynamic version of the set type in Object Pascal, because it is implemented in the same way. The set is actually a bit string in the memory where one bit corresponds to every element of the set. If the bit corresponding to an element is set in the bit string then the element belongs to the *TBitSet* object otherwise it does not. The *TBitSet* class resides in the "BitSet.pas" unit and its declaration is shown in Listing 16.

Listing 16: The *TBitSet* class

```
TBitSet = class
private
    FBitString: Pointer;
    FCapacity: Integer;
    FComplemented: LongInt;
    FBlock: Pointer;
    FBlockFirst: Integer;
    FBlockLeft: Integer;
    FCurrentDWord: Longword;
    function GetCount: Integer;
protected
    procedure Error;
    procedure SetCapacity(NewCapacity: Integer);
    procedure SetBit(Index: Integer; Value: Boolean);
```



```

public
  destructor Destroy; override;
  procedure Insert(Element: Integer);
  procedure Remove(Element: Integer);
  function Belongs(Element: Integer): Boolean;
  procedure Clear;
  procedure Pack;
  procedure Complement;
  procedure Intersection(Other: TBitSet);
  procedure Union(Other: TBitSet);
  procedure Subtract(Other: TBitSet);
  function IsEqualTo(Other: TBitSet): Boolean;
  function IsSubsetOf(Other: TBitSet): Boolean;
  function IsProperSubsetOf(Other: TBitSet): Boolean;
  function First: Integer;
  function Next: Integer;
  property Count: Integer read GetCount;
end;

```

There is only one property in the *TBitSet* class namely *Count*, which tells the number of items stored in the set. This property is read-only. The class supports all the common set operations. These operations are described separately in the list below:

- **Insert and Remove.** These methods are used to insert and remove an element to/from a *TBitSet* object. The element is given as an argument and it must be a positive integer.
- **Belongs.** Tests if an element belongs to a set. The function returns TRUE, if the element is in the set, and FALSE otherwise.
- **Clear.** Removes all the elements from a set clearing it completely.
- **Pack.** Packs a set to reduce the memory usage. Calling this function is profitable only when we know that no more elements will be inserted to the set.
- **Complement.** This procedure complements a set. This means that every element that belongs to the set will not be in it after calling this method, and conversely, all the elements that do not belong to the set will be in it upon completion of this operation.
- **Intersection.** This method performs an intersection operation with another set that is specified by the *Other* parameter. The result is stored in the current *TBitSet* object, that is, the object on which the method was called. An intersection leaves only those elements to the current set that belong to both *TBitSet* objects.
- **Union.** This method performs a union operation with another set that is specified by the *Other* parameter. If an element belongs to either of the source sets, then it will be in the resulting set after executing this method.
- **Subtract.** This method subtracts another set from the current set. This means that if an element belongs to the current *TBitSet* object and also to the set specified by the *Other* parameter, then it will be removed from the current set.
- **IsEqualTo.** This method tests if the current set, that is, the set on which the method was called, is same than another set specified by the argument. If the sets are identical, TRUE is returned.
- **IsSubsetOf.** Tests if the set given as an argument is a subset of the current *TBitSet* object, that is, the object on which the method was called. A set is a subset of another set, if every element belonging to it is present also in the other set. If the set specified by the argument is indeed a subset of the current set, then TRUE is returned. Otherwise the function returns FALSE.
- **IsProperSubsetOf.** Same as above, only now we test also if the sets are equal. If the *TBitSet* object given as an argument is a subset of the current set and the sets are not equal, TRUE is returned. Otherwise the function returns FALSE.

- **First and Next.** These methods are used to iterate through the element of a *TBitSet* object. The *First* function returns the first element of the set or -1 , if the set is empty. Similarly, the *Next* function returns the next element in the set or -1 , if there are no more elements left.

The majority of these operations are implemented using the assembly language, so they are very optimized and fast. The aim of the *TBitSet* class was to make algorithmic problems easier and raising their abstraction level by providing with a toolkit for calculating with mathematical sets. *TBitSet* objects are usually used in connection with a list or array so that the elements of the set correspond to the indices of the list. Thus, the elements of the *TBitSet* object are associated with the items stored in the list. The only requirement is that the indices of the items in the list or array do not change while we are working the *TBitSet* objects, because this would destroy the mapping from the elements to the items.