Persistence in BETA − Reference Manual

Mjølner Informatics Report
MIA 91−20
February 2002

Table of Contents

Table of Contents

List of Programs

1 Introduction

In the Mjølner System, objects generated by a BETA program execution may be saved on secondary storage and restored in another BETA program execution. Usually this property of a programming language or system is referred to as object persistence. Persistence in the Mjølner System is based on a *reachability model*, meaning that default behaviour when saving an object on secondary storage is to save everything reachable from the object in question.

Mjølner supports a single−user persistent store model as well as a multi−user shared persistent store. The multi−user persistent store makes it possible to build an object−oriented database (OODB) using a clienter/server model. Persistence in Mjølner is based on a number of frameworks:

- **Basic persistent object management.** A program may saves its objects in a persistent store and these may later be re−fetched from another or the same program. This framework supports single−user persistence.
- **Concurrency control.** If two or more programs (clients) need to acces the same persistent store at the same time, it is possible to synchronize acces to the persistent store through a server. The server implements a concurrency control mechanism that mediates the access to the persistent store. This framework supports multi−user persistent store.
- **Transaction management.** Transaction management deals with the problem of keeping a persistent store in a consistent state when two or more clients concurrently access the store or when failures happen during a program execution. This framework adds functionality to a multi−user
- **Notification control.** When two or more clients use the same persistent store, a notification mechanism is available for communication between the clients. Notifications may be sent to cleints when e.g. objects are written to the store by another client. This framework adds functionality to a multi−user persistent store.

In addition to the above main features, the following features are also supported:

- The basic object management supports reference between objects in different persistent stores.
- Recognizing that saving the full transitive object closure may at times be too coarse grained, the Mjølner System allows the programmer to control what part of the reachable object graph is actually saved along. This allows pointers from persistent objects to non−persistent objects despite the reachability based persistence model.

The persistence frameworks have gone through a number of iterations:

- The first ideas for persistence in Mjølner were presented in [Agesen 89] and a first implementation of single−user persistence was made by Claus Nørgaard.
- The first product−quality implementation of single−user persistence was made by Søren Brandt. Design considerations, experience and recommendations for a re−design are documented in the PhD Thesis of Søren Brandt [Brandt94]. This version of basic object management has been part of the Mjølner System for many years and is referred to as the *old basic object management*.
- A multi−user shared persistence store with concurrency control, transaction management and notifications has recently been developed by Johanna Wiederhold Tind [Tind97].
- Recently a new and more efficient implementation of the basic object management has been implemented by Stephan Erbs Korsholm [Korsholm99]. This work a.o. benefits from the work of Søren Brandt. This version of the basic object management is referred to as the *new basic object management*.

The new basic object management will eventually replace the old basic object management. In release 5.0 of the Mjølner System the old− as well as the new basic object management are included. We recommed to use the new basic object management since it is much more efficient than the old one. However, as this is the first release of the new basic object management, there may still be some errors or problems that have not yet been discovered.

The old and the new basic object management both work with the shared persistent store.

The directory

```
~beta/persiststore
```

contains the new basic object management as well as the shared persistent store.

The directory

```
~beta/persiststore/OLD
```

contains the old persistent store.

The interface to the old and the new persistent store is the same. That is, it is possible to alternate between the old and the new store by including either

```
~beta/persiststore/persistentstore
```

or

```
~beta/persiststore/OLD/persistentstore
```

2 Basic Definitions

2.1 Persistent Object

A *persistent object* is an object that is saved on secondary storage during a program execution and thus survives the program execution in which it was created. A persistent object may be read by another program execution. Any BETA object can in principle be persistent. In the current implementation, the execution state (i.e. *component stacks*) is not saved. Furthermore, for certain types of objects it may not be meaningful to make them persistent. This is e.g. the case with user interface objects generated by libraries such as xtenv, bifrost and macenv. Xtenv objects may e.g. have partial state information about windows, widgets, etc., but this information will not be sufficient to restore the screen.

2.2 Transitive Closure

By default, when an object is made persistent, all objects that can be reached through references are also made persistent. This includes statically enclosing objects [1]. The set of objects that can be reached from an object in this way is called the *transitive closure* of the object.

2.3 Persistent Store

Persistent objects are saved in a *persistent store*, which is a collection of persistent objects. A persistent store has a name. In the current implementation, the name of a persistent store is the name of a file system directory containing the files making up the persistent store. Several persistent stores may exist and references between objects in different persistent stores are supported. A persistent store is itself a BETA object with a number of attributes.

2.4 Persistent Root

An object may be pointed out to become a *persistent root* by means of the put operation on the persistent store. A persistent root must be given a logical name in the form of a text string. On checkpoint time, all objects reachable from persistent roots are saved in the persistent store.

[1] Objects that are instances of a nested pattern P.PP depends on, i.e. have a reference to, some instance of the enclosing pattern P

2.5 Objects and Patterns

An object generated as an instance of a pattern is only meaningful as an instance of that pattern. Consider the following example:

```
P: (# ... #);
R: ^P;

&P[] -> R[]; .... (* Save R *)
```

The object referenced by R is an instance of P. When this object is later read by a program, it must be interpreted by that program execution as an instance of P.

It is not enough that the program reading the object has a declaration of a pattern P which has the same structure as the pattern P which was used to generate the object. It must be the very same pattern.

In order for this to work, it is necessary to give a new interpretation to the notion of betaenv. This new interpretation is described in the next section. Before doing this we will shortly discuss what it means for objects to be instances of the same pattern.

Consider the following object descriptor:

```
(# T: (#  c: @ integer;
          A: (# b: @integer do c−>b #);
          X: @ A;
          Y: @ A
      #);
   V: @ T;
   W: @ T
#)
```

The outermost object has two internal objects V and W which are instances of the pattern T. Each of V and W has internal attributes c, A, X and Y. The attributes of V are different from the attributes of W. This should be obvious for c, X and Y, since they occupy different storage locations in V and W respectively. The pattern attribute A of V is also different from the pattern attribute A of W. The reason is that an instance of V.A is enclosed by V and may therefore refer to attributes of V −− V.A is said to have origin in V. The pattern W.A is an attribute of W and may refer to attributes of W −− W.A has origin in W. An instance of V.A is therefore NOT an instance of W.A.

The objects V.X and V.Y are thus instances of the same pattern V.A. Similarly the objects W.X and W.Y are instances of the same pattern W.A.

2.6 Betaenv

Betaenv is a fragment defining a pattern that currently encloses all BETA code being compiled and executed by the Mjølner System. This means that each program execution creates a new betaenv object. Patterns described in different programs will thus never be identical, since they will always directly or indirectly be attributes of the betaenv instance created by the program execution. Cf. the above discussion.

To overcome this problem, the persistentstore treats all betaenv instances as if they were actually the same object, although in practice a new instance is created in each program execution. For example, consider the following library fragment:

mylib.bet

```
ORIGIN 'betaenv'
−−−LIB:attributes−−−
  A: (# ... #);
  B: (# ... #);
```

If mylib.bet is included by two or more different programs, then the pattern A will logically be the same pattern in both programs, since it is an attribute of the same betaenv object in all the corresponding program executions. The same is of course true for B and any other pattern declared in mylib.bet

Patterns used for generating persistent objects should normally be defined in the lib:attributes library slot as in mylib above.

However, by using the support for special objects as described in a later section, it is possible to obtain the same treatment of other patterns as just described for betaenv, i.e. treating instances of the pattern in different program executions as logically the same object. This also allows patterns used for generating persistent objects to be declared in attribute slots different from lib:attributes.

2.7 Fragments and Compiled Code

A BETA pattern is declared in some fragment of the BETA fragment system. The fragment in turn is part of a fragment group, corresponding to a BETA source file. For a description of the fragment system, see [MIA 90−2]. To identify the pattern from which a persistent object was created, the object has a reference to the fragment in which the pattern is declared. The fragment is unique in the sense that it is the version used for generating the code that instantiated the object. In order to load an object into the memory of some program execution, the program loading the object must be compiled from and linked with the same version of the fragment from which the object was originally created. Note that this does not prevent exchange of objects between different platforms, since it is the fragments that must correspond, not the compiled code.

The current implementation does NOT check if the fragments used for creating a persistent object has been changed since the object was created. It is currently the responsibility of the user to keep track of this.

It should be obvious that changes to a fragment may cause inconsistencies with previously generated persistent objects. Neither is it allowed to change other fragments within the same fragment group.

The fragment used for generating a persistent object is currently identified using the name without path of the fragment group in which the pattern is declared. This means that a BETA program using persistence cannot contain two source files with the same name. If this restriction is violated, the program will stop with an error message as soon as the first persistent store is opened.

Multiple equally named BETA source files are not allowed when using persistence and distribution libraries.

The above problems related to the unique identification of patterns will be avoided in future versions of the persistent store.

3 The Persistentstore Pattern

The Mjølner System includes a library defining a persistent store, which keeps track of a directory of references to persistent objects. Some of these objects, the persistent roots, may have a logical name. A BETA program using persistent objects must include the persistent store library which is contained in the file

```
~beta/persistentstore/persistentstore
```

3.1 Basic Operations

The basic operations of the persistent store are as follows:

- create, openRead, openWrite are used for creating and opening a persistent store in order to access its contents. The name parameter is interpreted as a pathname relative to the current working directory of the process. When opening a persistent store for reading, it is not possible to update the contents of the persistent store, although the objects fetched may be changed in memory.
- put points out an object to become a persistent root. This does not affect the contents of the persistent store, but registers the object is be saved in future checkpoint operations. At the same time the object is given a textual name to be used in get operations.
- get retrieves an object identified by its textual name from secondary storage. If the object is already in memory, a pointer to the in−memory version is returned without changing the state of the object.
- checkpoint saves the state of persistent objects on secondary storage. The transitive closures of all persistent roots in process memory are traversed, and all the objects saved. Checkpoint has no effect on stores opened by openRead.
- close closes the persistent store. By default close performs a checkpoint operation before closing the files making up the persistent store. In most uses of the persistent store, it is therefore not necessary to call checkpoint explicitly.

Other operations on persistent stores are supported. These operations are described in section 4 on advanced features. In addition to the persistentstore operations, the deletePersistentStore pattern is available for deleting a persistent store.

A simple example using the persistentstore pattern is shown in section 3.3.

3.2 Restrictions

In the single−user case at most one program at a time should open a given persistent store in order to avoid problems with concurrent access.

If multiple programs are going to use the same persistent store, the multi−user version should be used.

As already mentioned, application programs exploiting object persistence should not include multiple source files with the same name.

Component objects, i.e. objects with their own execution stack, are not allowed in the transitive closure of persistent roots. If components are met during a checkpoint operation, the program will terminate with an error message.

3.3 Example

The fragment TextHashTable.bet defines a pattern TextHashTable whose instances are to be made persistent.

Program 1: TextHashTable.bet

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '~beta/containers/hashTable';
INCLUDE '~beta/basiclib/texthash';

---lib: attributes---
TextHashTable: hashTable
  (# honey: @honeyman;
     init::< (# do honey.init #);
     element::< Text;
     rangeInitial:: (# do 300  -> value #);
     hashfunction::< (# do e[]->honey.hash -> value #);
  #);
```

The fragment fooprod.bet describes a program that creates a new persistent store and saves some persistent objects:

Program 2: fooprod.bet

```
ORIGIN '~beta/basiclib/betaenv';
INCLUDE '~beta/persistentstore/persistentstore';
INCLUDE 'TextHashTable';

--PROGRAM:descriptor--
(# PS: @persistentstore;
   H: ^TextHashTable;
do (* Create the persistent store *)
   PS.init;
   'myStore' -> PS.create;

   (* Create a table of objects. *)
   &TextHashTable[] -> H[];
   H.init;
   'first' -> H.insert;
   'second' -> H.insert;
   'third' -> H.insert;

   (* Make the table a persistent root. *)
   (H[],'TextTable') -> PS.put;

   (* Checkpoint and close the store. *)
   PS.close
#)
```

The fragment foocons.bet describes a program that makes use of some persistent objects:

Program 3: foocons.bet

```
ORIGIN '~beta/basiclib/betaenv';
```

```
INCLUDE '~beta/persistentstore/persistentstore';
INCLUDE 'TextHashTable';
--PROGRAM:descriptor--
(# PS: @persistentstore;
   H: ^TextHashTable;
   T: ^Text;
   do PS.init;
      'myStore'-> PS.openWrite;
      ('TextTable', TextHashTable## ) -> PS.get -> H[];
      'fourth' -> H.insert;
      H.scan  (# do current[] -> putLine #);
      PS.close
#)
```

Other example usages of the persistent store may be found in the directory BETALIB/demo/persistentstore. These demo programs, part of the Mjølner System, are listed below:

- build.bet and oobench.bet implements a simple benchmark of the persistense framework. Run build to build the test store, then run oobench to time vaious operations.
- index.bet is a file indexer using a tree−like datastructure. The search function finds all words beginning with a given character.
- crossStoreDemo.bet demonstrates using multiple persistent stores from one program.

Example of usages of the OLD persistent store may be found in the directory BETALIB/demo/persistentstore/OLD. These demo programs, part of the Mjølner System, are listed below:

- largeWrite.bet, largeRead.bet work as described above.
- showregister.bet is an example of how to save a simple register in a persistent store. The register is built in a simple interaction with the user and finally saved. Later runs of showregister may read the saved register and perform simple queries. Finally the persistent store containing the register may be deleted.
- hashdemo.bet builds a simple hashtable of text strings. For each run of the program, an extra element is inserted into the table. If the persistent store does not already exist, it is created, and a new hashtable instance is made a persistent root. If the store already exists, the table is read, an extra element inserted, and the table scanned before the persistentstore is closed, implicitly implying a checkpoint operation.
- structdemo.bet is similar to hashdemo.bet, but illustrates the possibility of saving pattern variables in a persistent store. Pattern variables cannot become persistent roots, but as demonstrated by structdemo.bet, they are allowed in the transitive closure of a persistent root.
- special.bet is an example of how to limit the part of the transitive closure of persistents roots saved along during checkpoint operations. By registering the program pattern as a special object, even objects with origin [2] in the program object can be made persistent roots. Furthermore, by registering the IntegerObject pattern as a runtime type, references to all instances of IntegerObject are saved as NONE references. Runtime types and special objects are described in detail in the next section.
- crossstore.bet illustrates the handling of references between objects in different persistent stores. The same element is put into two different hashtables that in turn are saved in two different persistent stores. When one table is then fetched from its persistent store, it becomes necessary to open the persistentstore containing the shared element. The example shows how this must be taken care of by the programmer using the persistent store. The shared element is modified through the second table. On second run of the crossstore executable this modification is made visible through a scan of the first table. Details on references between different persistent stores are described in the following

section.

[2] Instances of patterns nested in the program pattern

4 Multi−User Shared Persistent Store

The multi−user shared persistent store makes it possible for two or more programs (clients) to acces the same persistent store. Access to the store has to be mediated through a server. The shared persistent store consists of a number of frameworks that may be used to build clients and servers. A client/server may contain part of the multi−user functionality or all of it.

The following functionalty is supported

- **Concurrency control.** The Multi−User Shared Persistent Store implements a concurrency control mechanism enabling several applications to simultaneously access the same persistent store. During execution, these applications may reget persistent objects to gain access to the changes any of the other applications might have made to these objects.
- **Transactions.** If two applications access the same persistent store simultaneously, race conditions may occur. To cope with this, the two processes have to be synchronized.

  This is supported by a locking mechanism, and with a transaction mechanism.

- **Notifications.** In order to enable coordination between cooperative applications, the applications can be notified of events occurring in persistent stores, the they have opened.

4.1 Multi−User Shared Persistent Store Fragments

The multi−user persistent store is implemented in the form of a number of fragments, augmenting the single−user persistent store with facilities for concurrency control, transactions, and notifications.

These fragments can be split up into several groups:

1. The `persistentstore` library. See the section 'The persistent store library' below for more details.
2. Libraries important to the application programmer when implementing the client programs. These libraries are listed in the section called 'Configuration of the Client'.
3. Libraries important to the application programmer when implementing/configuring the server program. These libraries are listed in the section called 'Configuration of the Server'.
4. Libraries used by the latter two groups of libraries. These libraries are not important to application programmers. They are listed in the section called 'Utilities'.
5. Example implementations for both client and server programs can be found in the directory called 'examples'. One example is described in detail in appendix~\ref{bygexample}. Before running the examples, make sure that an ensemble deamon is running on the host(s) involved.

4.1.1 The Persistent Store Library

The multi−user persistentstore library is augmented with the `reget−` and `commit−`operations.

4.1.2 Configuration of the Client

- `clientinterface.bet`: This file describes the interface to a client using the shared persistent store (named `PSclient`). This interface file has to be included by all client programs. The `PSClient` pattern is subclassed in the `notification.bet` file in order to yield the `NPSClient` pattern. This pattern has to be used instead of the `PSClient` pattern, when the client wants to subscribe to notifications.

- `serverinterface.bet`: This file contains the interface to the server, the `PersistentStoreManager`. This interface file has to be included in all client and server programs.
- `sharedpersistentstore.bet`: This file contains the `sharedpersistentstore` library. It provides concurrency control for a persistent store. This library has to be included by the client program, if a persistent store is being accessed in a multi−user environment.
- `transaction.bet`: This file contains the `transaction` library for the shared persistent store. It has to be included by the clients if they want to use transactions.
- `notification.bet`: This file is a library containing the declarations used when using notifications. This file has to be included by the clients, if they want to subscribe to notifications.

4.2 Configuration of the Server

- `serverinterface.bet`: This file contains the interface to the server, the `PersistentStoreManager`. This interface file has to be included in all client and server programs.
- `notificationmanager.bet`: This file contains the interface to a notification manager serving notification control. A server wishing to provide notification control has to include this file in order to run an instance of the `notificationmanager`.
- `lockmanager.bet`: This file contains the interface to a lock manager serving concurrency control. This file is included by the file `serverinterface.bet`, because every server starts a `lockmanager`.

4.2.1 Utilities

- `samefile.bet`: This file contains the method `samefile`, which returns true, if the two entered files are the same and false otherwise. This library is used instead of `file1name[]−>file2name.equal` because the same file can have different names depending on the machine, where a process starts. F.eks. `/a/home/scandium9/jojo/speciale/pstore1` and `~jojo/speciale/pstore1` are the same files, but would result in false, if using the `equal`−operation.
- `perslib.bet`: This fragment is a library containing the pattern declarations used to save the lock information on disk. This is done in order to facilitate recovery from a server crash.
- `sharedPSnotifications.bet`: This file is included by the `sharedpersistentstore` library in order to send notifications to the server. Notifications are only sent if a `notificationmanger` is started by the server. If a `notificationmanager` is started, notifications are sent automatically by the shared persistent store and not explicitly by the user.

4.2.2 Example Programs

- `PSManager.bet`: This file contains an example implemenation of the server, the `PersistentStoreManager`. It provides only a lock manager, no extra functionality. This is the smallest possible server implementation.
- `NPSManager.bet`: This file contains an example implemenation of the server, the `PersistentStoreManager`. It provides not only a lock manager, but also a notification manager serving notfication control.
- `recordlib.bet`: This fragment is a library containing the declarations of `record`, `person` and `register`. It is used in order to store registers containing persons in a persistentstore. The client example implementations use this library.
- `simpleclient.bet`: This file contains an example implementation of the client. The client opens a shared persistent store. The client does not make use of transactions or

notifications. The user can determine the used lock mode and lock policy.

- `notclient.bet`: This file contains an example implementation of the client. It opens a shared persistent store and subscribes to UPDATE and WAITFORLOCK notifications for it. When being notified, the client displays the notification on the screen. The access to the persistent store is done using transactions. The user can determine the used lock mode and lock policy.
- `regetclient.bet`: This file contains an example implementation of the client. The client opens a shared persistent store. The client subscribes to UPDATE events. When receiving a notification about an UPDATE, the client regets the in−memory objects of the updated persistent store. Note: This is only done, when the client is accessing the persistent store at the same time, where the persistent store is updated (only possible if the lock policy is MRSW).

5 An example − A Multi−User person register system

Assume an application, where multiple users access one or more person registers concurrently. The registers are stored in one or more persistent stores. This makes a concurrency control mechanism necessary. Moreover, the users could be interested in being informed about important actions performed by the other users. This makes a notification control mechanism necessary.

We will in the following show example implementations of the server and the clients for this application.

5.1 The Server

The server has to provide a lock manager and a notification manager in order to provide concurrency control and notification control. The server has to further bind the `PersistentStoreManager`, which provides concurrency control, in order to provide also notification control. The implementation of the server, called `NPSManager` is shown in the following.

5.1.1 NPSManger.bet

```
ORIGIN '~beta/distribution/shell';
INCLUDE '../serverinterface';     (* the interface to the server *)
INCLUDE '../notificationmanager'; (* to provide a notificationmanager *)

−−− program:descriptor −−−
shellEnv
(#
   shellType:: PersistentStoreManager
     (#
        thenotificationmanager:^notificationmanager;
     do
        registernotificationmanager:
          (# do
             &notificationmanager[]−>thenotificationmanager[];
             thenotificationmanager.init;
             (thenotificationmanager[],'Notificationmanager')
                 −>myEnsemble.ns.put(# overWrite:: (# do true −> value #)#);;
          #);
     #);
#)
```

5.2 The Clients

The clients want to make use of the concurrency control mechanism and the notification mechanism provided by the server. A client opens a shared persistent store and subscribes to UPDATE and WAITFORLOCK notifications for it. It uses the transaction mechanism when accessing a persistent store in order to ensure its consistency. The client can determine, if he wants to read or update the persistent store and he can determine the lock policy to be guaranteed. When being notified, a client displays the notification on the screen. One or more clients can interact concurrently with the server, i.e. several program executions of the client can be executed in parallel. The implementation of a client, called `notclient`, is shown in the following.

5.2.1 notclient.bet

```
ORIGIN '~beta/distribution/shell';

INCLUDE 'serverinterface'; (* to communicate with the server *)
INCLUDE 'clientinterface'; (* to bind shelltype to
                              NPSclient (because it uses notifications) *)
INCLUDE 'notification';    (* to subscribe to notifications *)
INCLUDE 'sharedpersistentstore'; (* to make use of the concurrency control mechanism *)
INCLUDE 'recordlib';       (* to store persons in a register *)
INCLUDE 'transaction';     (* to use transactions *)

−−− program: descriptor −−−
shellEnv
(#
   shellType:: NPSclient
     (# notifyImpl::(#
                    do theNotification.display;
                    #);

        ps: @sharedpersistentstore
          (# maxCountOnDanglerHit::<
               (# do 10−>value #);
          #);

        serverEnsembleName: ^Text;
        serverEnsemble: ^ensemble;
        thenotificationmanager:^notificationmanager;  PSname:^text;


        newreg:^register;
        s:^register;
        rootname:^text;
        regname:^text;

 do
        (* Get a reference to the ensemble on which the server is running *)
        (*****************************************************************)

        'Enter Persistens Server hostname: ' −> screen.putText;
        keyboard.getLine −> serverEnsembleName[];

        getHost: (ensemble##, serverEnsembleName[])
          −> theShell.myEnsemble.ns.get −> serverEnsemble[];


        (* open shared persistent store *)
        (*******************************)

        'Open PS ? '−> screen.puttext;
        keyboard.getline−>PSname[];

        open:
          (#  exists:@boolean;
          do true−>exists;
             (PSname[],theShell[],serverensemble[])−>PS.open
                     (# notfound::&lt;(# do false−>exists;
                                        'The store is created ...'−>puttext;
                                         PSname[]−>PS.create;
                                         leave open;
                                   #);
                      #);
              (if not exists then 'done'−>putline if);
          #);
```

```
(* subscribe to UPDATE and WAITFORLOCK notifications *)
(****************************************************)

getnotificationmanager:(NotificationManager##, 'Notificationmanager')
     -> serverEnsemble.ns.get -> thenotificationmanager[];

(if (thenotificationmanager[] <> NONE) then
  subscribe: (# thesubscription: ^subscription;
               do
                  &subscription[]->thesubscription[];
                  (theShell[],'all',PSname[],Update)
                        ->subscribetoPSEvent->theSubscription[];
                  theSubscription[]
                        ->thenotificationmanager.subscribeNotification;
                  (theShell[],'all',PSname[],Waitforlock)
                        ->subscribetoPSEvent->theSubscription[];
                  theSubscription[]
                        ->thenotificationmanager.subscribeNotification;
               #);
if);

(* start transaction loop *)
(**************************)

transactionloop:
  (# lmode:@integer;
     lpolicy:@integer;
  do
     'Access the ps for reading [r] or writing [w]'->putline;
     (if getNonBlank = 'w' then write -> lmode;
     else read -> lmode;
     if);

     'The policy to be guaranteed: MRSW allowed [y,n]'->putline;
     (if getNonBlank = 'y' then shared -> lpolicy;
      else exclusive ->lpolicy;
     if);

     mytrans: ps.transaction
       (# LockMode::<(# do lmode->value#);
          LockPolicy::<(# do lpolicy->value #);
          abort::<(# do leave mytrans #);
          getregister:(# rootname:^text;
                      do
                         'Which register? '-> screen.puttext;
                          keyboard.getline->rootname[];
                         (rootname[],register##)->get->s[];
                      exit s[]
                      #);
       do
          commandLoop: cycle
            (#
               newreg:^register;
            do
               'Get a register (persistent root) [g]'->putline;
               'Display register [d]'->putline;
               'Find person in register [f]'->putline;
               'Add person to register [a]'->putline;
               'Delete person from register [k]'->putline;
               'Change name of person [c]'->putline;

               'Show registers in location [s]'->putline;
               'Put register [p]'->putline;
```

```
'Remove register [r]'->putline;
'Checkpoint/Update persistent store [u]'->putline;
'Reget in−memory objects [e]'->putline;

'Abort the transaction [b]'->putline;
'Commit the transaction [q]'->putline;

(if getNonBlank

 // 'g'  then
    getregister->s[];
    (if (not (s[] = NONE)) then
        'Done'->putline;
    if);

 // 'd' then
    getregister->s[];
    (if (not (s[] = NONE)) then
        screen[]->s.display;
    if);

 // 'f' then
    getregister->s[];
    (if (not (s[] = NONE)) then
        findWithKey (# do screen[] −> found.display #);
    if);

 // 'a' then

    getregister->s[];
    (if (not (s[] = NONE)) then
        addPerson;
    if);

 // 'k'  then
    getregister->s[];
    (if (not (s[] = NONE)) then
        delete
    if);

// 'c'  then
    getregister->s[];
    (if (not (s[] = NONE)) then
      change;
     if);

 // 's' then
    'The following registers are found in the PS:'
                                               ->putline;
    scanrootnames (# do current[]->putline; #);

 // 'p' then
    'Name of register to be put? '−> screen.puttext;
    keyboard.getline->rootname[];
    &register[]->newreg[];
    (newreg[],rootname.copy)->put;

 // 'r' then
    getregister->s[];
    (if (not (s[] = NONE)) then
        NONE->s[];
    if);

 // 'e' then
    reget;
```

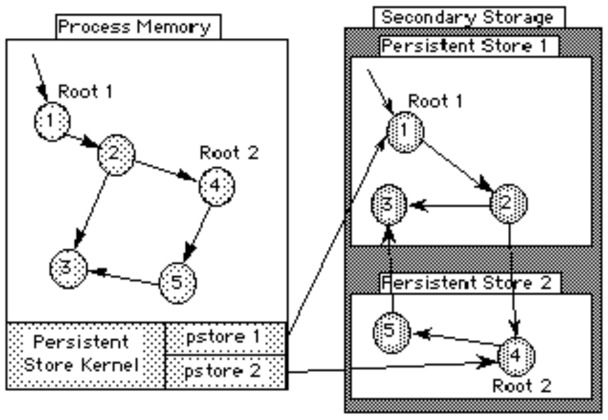
```
                        // 'u' then
                           checkpoint;


                        // 'b' then
                           abort;

                        // 'q' then
                           leave commandLoop;
                     if);
                    #);
               #);
             'Continue with new transaction [yn]?'−>screen.puttext;
             (if getnonblank = 'y' then
                 restart transactionloop
             if);

          #);

        (* close shared persistent store *)
        (**********************************)

        PS.close;
        theShell.kill;
     #);
#)
```

6 Advanced Features

The following sections describe some advanced features of using persistent stores.

6.1 References between different persistent stores

References between objects saved in different persistent stores are allowed, but beware that the support for such references is still somewhat experimental and requires special care in order to avoid problems. This section describes how to exploit cross store references. The crossstore.bet demo program is an example usage of cross store references. The drawing below illustrates references between objects saved in different persistent stores. In memory, there is no difference between cross store references and other object references. However, on secondary storage these references cross boundaries between persistent object stores.



6.2 Where are objects saved?

When performing a checkpoint on a persistent store, either explicitly by calling the checkpoint operation or implicitly by closing the store, the object graphs rooted in the persistent roots of the store are traversed and the objects met saved to the persistent store. However, if an object is met that has already been saved in another persistent store, only the identification of that object is saved, and the graph traversal is not continued in that particular direction. Thus, a checkpoint operation on a persistent store only saves objects already belonging to that store, or objects that do not yet belong to any persistent store at all. The result is that a persistent object is saved in the store that first sees the object during a checkpoint operation.

The description above of course demands that we are able to recognize objects that 'belong to' other persistent stores. However, when a persistent store is closed, objects that belonged to the store before the close operation will, after the close operation, semantically turn into copies of the objects belonging to the store. Thus, in order to maintain references between objects in different persistent stores, it is necessary to make explicit checkpoint operations and avoid the implicit checkpoint done by the close. The latter is done by further binding the doCheckpoint virtual:

```
PS.close (# doCheckpoint:: (# do false -> value #)#);
```

When exploiting cross store references, avoid the implicit checkpoint operation by furtherbinding the doCheckpoint virtual on close operations. Instead explicit checkpoint operations should be executed on all stores before closing any store.

6.3 Following references between persistent stores

If a persistent root is fetched whose transitive closure contains references to objects in other persistent stores, these persistent stores must be open in order to fetch the objects referenced. However, if the store referred is not already open, the persistent store containing the reference is not able to open the store automatically since it has no idea whether it should be opened for reading or writing. Instead it calls the persistentstore.openpstore virtual with the full pathname of the store needed, and expects the further binding to open and return the store.

6.4 Lazy object fetch

Note that the new persistense always runs using lazy fetch. This section is therefore only relevant to users of the old persistense

When fetching an object from a persistent store using the get operation, the default is to eagerly fetch all objects in the transitive closure of the persistent root specified. However, since this may involve a huge number of objects not really needed by the current program execution, the persistent store offers the possibility to fetch the transitive closure lazily as the program goes along following references from the persistent root.

By further binding the persistentstore.allowLazyFetch virtual to trueObject, the default fetch strategy is changed to lazy fetch. Alternatively the fetch strategy may be set on a per get basis by further binding the get.allowLazyFetch virtual to trueObject.

In short, lazy fetch works as follows. Using the persistent store get operation, the object graph reachable from the persistent root is always fetched in a breadth−first manner, whether or not lazy fetch is applied. In the case of lazy fetch, instead of fetching the full object graph, only a limited number of objects are fetched from secondary storage and instantiated in the current process. The objects fetched are the persistentstore.maxFetchOnDanglerHitfirst objects met during the breadth−first traversal. The default number of objects fetched may be changed by further binding the maxFetchOnDanglerHit virtual.

So, what about the objects not fetched? Since these objects are not instantiated, it is impossible to setup usual in−memory references. Instead socalled dangling references are used. Simply stated, a dangling reference is a negative number uniquely identifying a persistent object to the current process. If a dangling reference is ever followed [3], the same mechanism that checks for NONE references will trap to the persistent store kernel in order to transparently fetch the object needed from secondary storage. Also in this case the objects fetched are the maxFetchOnDanglerHit first objects met during a breadth−first traversal of the object graph rooted in the object needed. All dangling references in the process referencing newly fetched objects are replaced by genuine in−memory references. A more detailed description of the implementation of lazy object fetch may be found in [Brandt 94].

Note that the −−noCheckNone (or −s 14 0) compiler switch suppressing the generation of runtime checks for NONE references cannot be used in programs using lazy object fetch!

In addition to maxFetchOnDanglerHit and allowLazyFetch, the persistentstore.OnDanglerHit and persistentstore.AfterDanglerHit virtuals are used in conjunction with lazy object fetch. OnDanglerHit is called when a dangling reference has been hit, but before the object is actually fetched from secondary storage. AfterDanglerHit is called when the object has been fetched, giving the object as parameter. When AfterDanglerHit returns, the program continues whatever it was doing when the dangling reference was hit. The purpose of these virtuals is to offer informative callbacks that may be used for example in interactive programs where lazy object fetch may otherwise result in inexplicable delays.

It should be noted that there are no semantic differences whatsoever between lazy and eager object fetch. The practical difference lies in different efficiency/memory usage trade−offs.

The demo programs largeWrite.bet and largeRead.bet together illustrates the use of lazy fetch.

[3] Followed here means "accessing the state of the object referred". Usual reference assignment on dangling references are not different from ordinary reference assignments

6.5 Limiting reachability based persistence

6.5.1 Special objects

For the primary intended usage, a special object is an object that is thought of as a single logical object that is always present in program executions using some persistent store. Support for special objects may thus be thought of as generic support for pointing out patterns that to some extend have only a single instance shared between all program executions using the pattern.

The state of special objects is never saved persistently. However, references to these objects should be saved so that they may be setup correctly when saved objects referencing a special object are re−instantiated in another process. Typical examples or special objects are application framework objects that are known to be present in the program executions exchanging persistent objects, but should not be saved themselves. Examples of application framework objects are instances of guienv, XtEnv, systemenv and shellenv. These application framework objects, of which there should be at most one in each program execution, are not to be saved persistently, but instances of patterns nested inside the application frameworks should be allowed to persist. As already mentioned, betaenv is always treated as a special object.

Special objects are registered once in the lifetime of a persistent store by supplying name and type of the object to the persistentstore.registerSpecialObject method. The type is saved in the persistent store in order to be used for type checking when registering special object instances as described below.

In addition to the initial registration, an instance of the special object must be supplied by each process using the persistent store by calling the persistentstore.registerSpecialInstance method when the persistent store has been opened, but before any get operations are made. The instance given to the registerSpecialInstance method must be a subtype [4] of the type given to the registerSpecialObject operation.

The demo program special.bet contains an example usage of special objects.

6.5.2 Runtime types

Runtime types are patterns whose instances are used at runtime, but should not persist across program executions. As mentioned in section 2, an example of this is interface objects such as windows. Another example is objects used for caching purposes at runtime and referenced from persistent objects although the cache objects themselves should not be saved across program executions. By registering the pattern p as a runtime type, instances of p are not saved during checkpoint operations even though they are found in the transitive closure of a persistent root. Instead references to these objects are saved as NONE references.

Runtime types are registered by calling persistentstore.registerRuntimeType. As runtime types registered using registerRuntimeType are not saved persistently in the store, registerRuntimeType must be called for each runtime type in each session using the persistent store in question. If needed, it is of course possible to save a table of runtime types in a persistent store. The demo program structdemo.bet is an example of how a table of pattern variables may be saved in a persistent store.

The demo program special.bet contains an example usage of runtime types.

6.5.3 Combining runtime types and special objects

Since references to special object instances are treated differently than references to instances of runtime types, it is a contradiction to register the same pattern as a special object and as a runtime type in the same persistent store. Doing so will result in a runtime error.

Furthermore, since it is not allowed to save an object without the knowledge that all its origins will be available when the object is to be reinstantiated, instances of runtime types should not be origins of objects saved. If an instance of a runtime type is needed as origin for some other object to be saved, the runtime type instance is saved anyway, disregarding the fact that it is an instance of a runtime type.

Different persistent stores used in the same program execution may have different sets of special objects and runtime types registered.

---

[4] The subtype relation is reflexive, i.e. any pattern is a subtype of itself

6.6 Files used for storing objects

The name parameter to the create, openRead and openWrite operations in the persistentstore pattern is interpreted as a directory name relative to the current directory of the process. When creating a new persistent store, this directory is created along with the file db. (Note, the old persistent store creates the files locg, oinx and data.) For deleting the files making up a persistent store, the deletePersistentStore pattern is available.

7 Known Bugs and Inconveniences

7.1 Garbage collection and persistence

With respect to garbage collection and persistence, there are two separate issues to consider, namely the usual in−memory garbage collection and garbage collection of the persistent store on secondary storage. These are considered in turn below.

7.1.1 In−memory garbage collection

The persistent store kernel keeps track of persistent objects loaded into the current process by maintaining a table of references to these objects. This table is shared by all persistent stores in a program execution. As long as a persistent store is open, no objects from that store can thus become garbage, since they are at least referenced from the internal object table. Currently the only way to delete objects from the internal table is to close the store. Thus, to allow in−memory garbage collection of persistent objects, the persistent store in which these objects are saved must be closed.

A side−effect of deleting objects from the object table is is course that the persistence kernel no longer knows that these objects are persistent, and thus semantically these objects turn into in−memory copies of the real persistent objects, now only available on secondary storage.

7.1.2 Secondary storage garbage collection

Currently there is no built−in support for garbage collection of persistent stores. Thus, once saved in a store, an object stays there until the store is deleted, even though the object may no longer be reachable from any persistent root.

7.1.2.1 Secondary storage garbage collection using the OLD persistense implementation

Note, this technique does not work when using the new persistense implementation.

However, for small persistent stores whose objects fit into virtual memory of the computer at once, and that are not referenced from other persistent stores, it is possible to perform a simple garbage collection using the basic operations of the persistentstore pattern. This is illustrated by the PersistentGC.bet demo fragment. PersistentGC simply reads the transitive closures of all persistence roots into memory, deletes the store, and then saves the persistence roots in a new store with the same name as the old store.

Note, however, that the fragments used to generate the objects saved in the store must be linked with the executable performing the collection.

The demo program gc.bet illustrates how to first delete a number of elements from the persistent table generated by largeWrite.bet, and then perform a garbage collection on the store, using PersistentGC.

7.2 Persistent store identification and cross store references

The persistent store identifies objects using a two−part object id, each part being a 32 bit integer. The first part identifies the persistent store in which the object is saved, and the second part is a unique identification of the object within that store. Currently the persistent store id is simply the system time (in seconds) when the store was created. A persistent store containing references to other persistent store thus maintains a mapping from these creation times to the full pathname of the persistent stores, in order to be able to call the openpstore virtual with the correct pathname.

Unfortunately this identification is not entirely unique. The persistent store kernel ensures that no two stores created by the same process gets the same creationtime, but there is currently no way to ensure that different processes do not create persistent stores with the same creation time. A process simultaneously opening two persistent stores with the same creation time will therefore in the best case receive wrong alreadyOpen exceptions, and in the worst case wrong in−memory object graphs may be created.

In future versions of the persistent store, this problem will be solved by using an alternative identification scheme.

8 Bibliography

[Agesen 89] Ole Agesen, Svend Frølund, Michael Hoffmann Olsen: Persistent and Shared Objects in BETA, Computer Science Department, Aarhus University, DAIMI IR−89, April 1989.

[Brandt 94] Søren Brandt: Implementing Persistent and Shared Object in BETA. Progress report. Technical Report, Computer Science Department, Aarhus University, May 1994.

[Tind 97] Johanna Widerhold Tind: A Multi−User Persistence Framework: Building Customized Database Solutions Using the BETA Persistent Store, Master Thesis, Computer Science Department, Aarhus University, 1997.

[Korsholm 99] Stephan Erbs Korsholhm: Transparent, Scalable, Efficient OO−Persistence, Presented at the 1st ECOOP Workshop on Object−Oriented Databases, ECOOP'99, Lisbon, Portugal, 1999.

9.1 Persistentstore Interface

```
ORIGIN '~beta/basiclib/betaenv' (*
 * COPYRIGHT
 *       Copyright Mjolner Informatics, 1992-98
 *       All rights reserved.
 *)
;
INCLUDE '~beta/basiclib/directory'
        '~beta/basiclib/external'
        '~beta/basiclib/directory'
        '~beta/sysutils/objinterface'
        '~beta/containers/list';
BODY 'private/psbody';
-- lib: Attributes --
(* PSEXCEPTION
 * ===========
 *
 * PSexception is used in several exceptional situations where the files
 * making up an persistent store are not accessable. The fullName parameter
 * is the full path of the directory expected to be a persistent store. *)
PSexception: exception (# fullName: ^Text;  enter fullName[] do INNER #);

PSroot:
  (# name: ^text; type: ##object; obj: ^Object
  enter (name[],type##,obj[])
  #);
PSrootsList: list (# element:: PSroot #);

(* PERSISTENTSTORE
 * ===============
 *)
persistentstore:
  (#
     <<SLOT PSlib:Attributes>>;
     PSroots: ^PSrootsList;
     PSstoreID: @integer;
     assertOpen:
       (#
          notOpen:< PSException
            (#
            ...
            #);

       ...
       #);
     assertInit:
       (#
          notInitialized:< PSException
            (#  ... #);

       ...
       #);
     numberOfUpdates:
       (# result: @integer ... exit result #);
     lastupdate: @integer;
     hostName: (#  exit 'localhost' #);
     openCrossStoreItem: ^openpstore;
     UNKNOWNTAG: (#  exit 0 #);
     rebinderItem: ^rebindSpecialReference;
     rebindSpecialReference:<(#
          target: ^Object;
          toSpecialObject:<
            (#
               objectTag: @integer;
               NotHandled:< PSException
                 (#  ... #)
            enter objectTag
            do INNER
            #);
          toSpecialType:<
            (# typeTag: @integer
            enter typeTag
            do INNER
            #);
          objectTag,typeTag: @integer
       enter (objectTag,typeTag)
       ...
       exit target[]
       #);
     commit: assertInit
       (#
          doUpdateInMemoryObjects:< BooleanValue
            (#  do false−>value; INNER #);
          doCheckpoint:< BooleanValue (#  do true−>value; INNER #);

       do INNER
       #);
     reget: assertInit
       (#
          NotFound:< PSException (#  ... #);
          theObject: ^Object;

       enter theObject[]
       do INNER
       #);
     init: (#  ... #);
     create: assertInit
       (#
          alreadyOpen:< PSexception
            (#  ... #);
          exists:< PSexception
          (* The old store is deleted if exists returns.
           *) (#  ... #);
          creationError:< PSexception
            (#
            ...
            #);
          name: ^Text;

       enter name[]
       ...
       #);
     abstractOpenOperation: assertInit
       (#
          alreadyOpenMessage:<
            (# msg: ^text
            do INNER
            exit msg[]
            #);
          notFoundMessage:<
            (# msg: ^text
            do INNER
            exit msg[]
            #);
          accessErrorMessage:<
            (# msg: ^text
            do INNER
            exit msg[]
            #);
          alreadyOpen:< PSexception
            (#
            ...
            #);
          notFound:< PSexception (#  ... #);
          accessError:< PSexception
            (#  ... #);
          name: ^Text;
          openFunc:<
            (#
               host,path: [1] @Char;
               rcode: @integer
            enter (host,path)
            do INNER
            exit rcode
            #)
       enter name[]
       ...
       #);
     openRead: abstractOpenOperation
       (#
          alreadyOpenMessage::
            (#  ... #);
          notFoundMessage::
            (#
            ...
            #);
          accessErrorMessage::
            (#  ... #);
          openFunc::
            (#
            ...
            #);

       do INNER
       #);
     openWrite: abstractOpenOperation
       (#
          alreadyOpenMessage::
            (#  ... #);
          notFoundMessage::
            (#
            ...
            #);
          accessErrorMessage::
            (#  ... #);
          openFunc::
            (#
            ...
            #);

       do INNER
       #);
     checkpoint: assertInit (#  ... #);
     close: assertInit
       (#
          danglersExists:<
            (#
               toDo: @Integer;
               kill: (#  exit 0 #);
               (* Kill the process. Default action. *)
               fetch: (#  exit 1 #);
               (* Fetch the missing objects.        *)
               ignore: (#  exit 2 #);
               (* Ignore the dangling references.   *)
            do kill−>toDo; INNER
            exit toDo
            #);
          doCheckpoint:< BooleanValue (#  do false−>value; INNER #);

       ...
       #);
     get: assertInit
       (#
          quaError:< Exception
            (#  ... #);
          notFound:<
           Exception (#  ... #);
          name: ^Text;
          type: ##Object;
          theObject: ^Object;

       enter (name[],type##)
       ...
       exit theObject[]
       #);
     ALREADYTHEREERROR: (#  exit 6 #);
     HASOVERWRITTENERROR: (#  exit 7 #);
     put: assertInit
       (#
          AlreadyThere:< PSexception
            (#
            ...
            #);
          HasOverWritten:< PSexception
            (#  ... #);
          dooverwrite:< BooleanValue
            (#
            do true−>value; INNER
            #);
          obj: ^Object;
          name: ^text;

       enter (obj[],name[])
       ...
       #);
     scanRootNames: assertInit
       (# current: ^Text;
       do
          PSroots.scan
            (#  do current.name[]−>THIS(scanRootNames).current[] #);
          INNER
       #);
     registerSpecialObject: assertInit
       (#
          alreadyThere:< Exception
            (#  ... #);
          name: ^Text;
          type: ##Object;

       enter (name[],type##)
       ...
       #);
     registerRuntimeType: assertInit
       (# type: ##Object
       enter type##
       do
          'registerRuntimeType: This function is not implemented in the new'
            −>screen.putLine;
          'persistent store, and the call should be removed.'−>screen.putLine;
```

```
       #);
     allowLazyFetch:<
       BooleanValue;
     maxCountOnDanglerHit:< IntegerValue (#  do 100−>value; INNER #);
     OnDanglerHit:< Object;
     AfterDanglerHit:<
       (# theObject: ^Object;  enter theObject[] do INNER #);
     openpstore:<
       (# psname: ^Text; ps: ^persistentstore;
       enter psname[]
       do INNER
       exit ps[]
       #);
     deletePersistentStore: assertInit
       (#
          alreadyOpen:< PSexception
            (#  ... #);
          accessError:< PSexception
            (#
            ...
            #);
          notFound:< PSexception (#  ... #);
          name: ^Text;
          storeDir: @directory;

       enter name[]
       ...
       #);
     registerSpecialInstance: assertInit
       (#
          quaError:< Exception
            (#  ... #);
          notFound:< Exception
            (#
            ...
            #);
          o: ^Object;
          tag: @integer;
          (* The arguments to 'RegisterSpecialInstance' is now no longer
           * an object and a name, but an object and an 8 bit integer value.
           * The integer tag will be the id of the special object.
           *)

       enter (o[],tag)
       ...
       #);

  do INNER ;
  #)
```

Index

The entries in the alphabetic index consists of selected words and symbols from the body files of this manual − these are in **bold** font − as well as the identifiers defined in the public interfaces of the libraries − set in regular font.
In the manual, the entries, which can be found in the index are typeset like this. This can help localizing the identifier, when the link from the index if followed − especially in the case where the browser does not scroll the line to the top, e.g. because there is less than a page of text left.
In the small table of letters and symbols below, each entry links directly to the section of the index containing entries starting with the corresponding letter or symbol.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z