Perseus:
The persistence framework

–

Specification

AUTHORS:

S Chassande-Barrioz (France Telecom R&D)

P Dechamboux (France Telecom R&D)

L Garcia-Banuelos

Released: January 27, 2004

Status: Draft

Version: 1.3

TABLE OF CONTENTS

TABLE OF FIGURES

1 INTRODUCTION

1.1 Overview

1.2 Scope

1.3 Rationale

1.4 Goals

1.5 Document Convention

A Times Roman font is used for the default text.

```
A courier font is used for code fragments.
```

2 ARCHITECTURE

2.1 Overview

This document is the specification of the Perseus framework, originally initiated by Luciano Garcia-Banuelos. Perseus is a persistence framework which manages several aspects like caching, concurrency control, pool, logging. The architecture of the framework and the API are presented in this document.

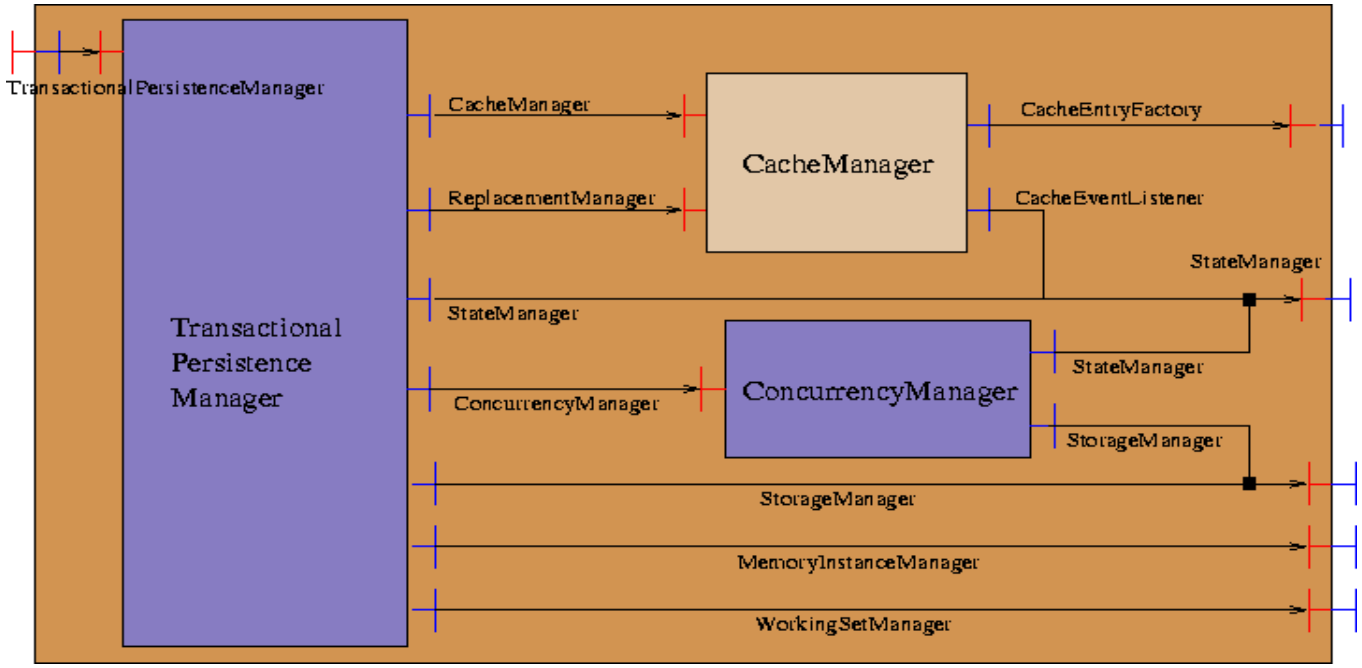


Figure 1: Perseus architecture overview

This figure presents the architucture of the perseus framework. Three inner components are defined. The next chapters presents the inner components, the interactions and the interfaces of the required external components.

3 PERSISTENT OBJECT

The perseus framework makes few hypothesis about the persistent object itself. The figure belove presents the organization arount a persistent object

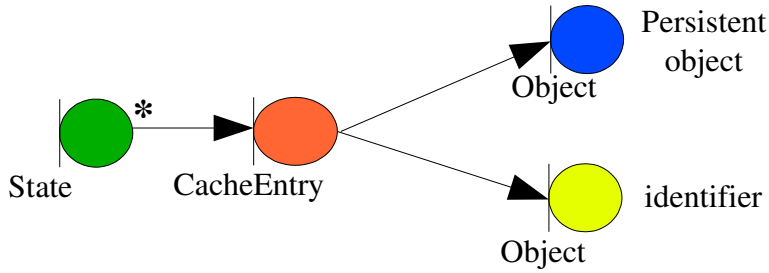


Figure 2: Arround a persistent object

- persistent object: The persistent object is simple java object (java.lang.Object). The framework does not take care of the 'equals' and 'hashcode()' methods.
- identifier: Any persistent object must have an identifier. Like the persistent object, the identifier must be a java object (java.lang.Object). In opposite to the persistent object, the 'equals' and 'hashcode()' methods are used to compare identifiers and then persistent objects.
- CacheEntry: A cache entry is a java object implementing the CacheEntry interface. The role of the cache entry is to bind the persistent object to its identifier. Indeed perseus does not want to impose to have a java reference between the persistent object to its identifier. This is the role of the cache entry.
- State: A State is a java object implementing the State interface, and containing the values of persistent fields associated to the persistent object. Several state instances can be associated to a persistent object. Indeed the use of separated states permits to implement some concurrency management policies (optimistic for example).

```
package org.objectweb.perseus.persistence.api;
public interface State {
     Object getCacheEntry();
}
```

The state, the cache entry and the persistent object can be composed in several manners. The following figure presents a nice and efficient compositon permiting the implementation of different concurrency management. In this composition the persistent object is also the cache entry:

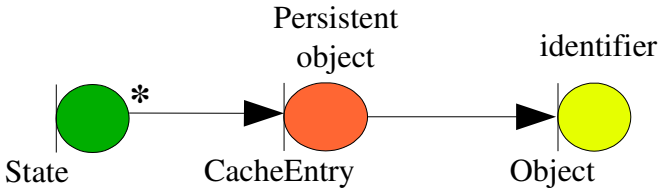


Figure 3: Nice composition arround a persistent object

4 StateManager

A StateManager is used by the PersistenceManager and the ConcurrencyManager in order to manage the life cycle of cache entries and their states. This management concerns the java life cycle of the state instance (create, destroy) and the status (dirty, new, deleted, ...). This interface is implemented by the Personality using the persistence framework of Perseus.

```
package org.objectweb.perseus.persistence.api;
public interface StateManager extends CacheEventListener {
    State createState(CacheEntry ce);
```

This method creates a new state for the cache entry. The returned Sate (never null) is linked to the cache entry.

```
    State createState(State s);
```

This method creates a new state whith the same values than an existing state.

```
    State getReferenceState(CacheEntry ce);
    void setReferenceState(CacheEntry ce, State state);
```

These methods permits to assign/fetch the reference state of a cache entry. Depending of the concurrency policy a particular state is named 'reference state'. This state represents the current value of the persistent object out side of any working set/transaction.

```
    void destroyState(State state);
```

This method is a sort of listener permiting to the StateManager to be informed about the end of the state use.

```
    void makeUnexported(State state);
    boolean isUnexported(State state);
```

These methods manage the unexported status of a state. When a persistent object (through its state) is marked as unexported, it will be removed after the working set commit step at least.

```
    void makeExported(State state);
    boolean isExported(State state);
```

These methods manage the exported status of a state. When a persistent object (through its state) is marked as exported, it will be created after the working set commit step at least.

```
    void makeDirty(State state);
    boolean isDirty(State state);
    void makeClean(State state);
```

These methods manage the dirty status of a state. A state is marked as dirty when one or several persistent fields are modified. At the commit time of a working set, dirty states are flushed into the data support and their status must be marked as clean.

```
void makeFlushed(State state);
boolean isFlushed(State state);
```

These methods manage the flushed status of a state. A dirty state can be flushed before the end of the working set for some reasons, but the entry stays dirty. The interest of the flushed falg is to avoid another I/O if the state is no more modified.

```
void makeUnbound(CacheEntry ce);
boolean isBound(CacheEntry ce);
```

These methods manage the persistent status of a cache entry. An object is persistent when it is bound to an identifier.

The following table defines the transitions between the different status of a state:

	writeField	***readField***	***bind***	***unbind***	***export***	***unexport***	***flush***
unbound	unbound	unbound	clean	unbound	exported	error	error
clean	dirty	clean	clean	unbound	error	unexported	clean
exported	exported	exported	error	error	error	unexported	ex_flushed
unexported	error	error	error	error	dirty	unexported	un_flushed
flushed	dirty	flushed	error	error	error	unexported	flushed
ex_flushed	ex_flushed	ex_flushed	error	error	error	unexported	ex_flushed
un_flushed	error	error	error	error	exported	error	un_flushed
dirty	dirty	dirty	error	error	error	unexported	flushed

Figure 4: Transition between status of a state

5 WorkingSet, TransactionalWorkingSet & WorkingSetManager

5.1 WorkingSet

A working set is a set of State used together in the same context (eg, a transaction, a thread, ..). A working set has a status and can be used to represent elements involved by a transaction. Only one state matches an identifier into a working set.

A transaction is a working set of which the modifications can be rolled back.

Here is the defintion of the WorkingSet interface:

```
package org.objectweb.perseus.persistence.api;
public interface WorkingSet {
     byte CTX_ACTIVE = 1;
     byte CTX_CLOSED = 32;
     byte getStatus();
     void setStatus(byte status);
```

These methods manage the status of the working set.

```
     State lookup(Object oid);
     void bind(State state, Object oid);
     boolean unbind(Object oid);
     void clear();
     Set entries();
     Set oids();
```

These methods manage the enlistement of persistent object (identifier + state) into the working set.

```
     Object getUserObject();
```

It retrieves the user object which can be associated to the working set at the creation time.

```
     ConnectionHolder getConnectionHolder();
```

It retrieves the ConnectionHolder associated to the working set at the creation time.

```
     boolean getWSRetainValues();
     void setWSRetainValues(boolean val);
```

The retainValues boolean property indicates if after the validation of a working set the state must be kept in the cache.

```
     boolean getWSRestoreValues();
     void setWSRestoreValues(boolean val);
```

The restoreValues boolean property indicates if after the abort of a working set the dirty persistent object must be reload immediatly or not.

```
}
```

5.2 TransactionalWorkingSet

The framework defines an extension of the WorkingSet definition in order to specify the possible status corresponding to the transaction life cycle.

```
public interface TransactionalWorkingSet extends WorkingSet{
     byte CTX_ACTIVE_TRANSACTIONAL = 2;
     byte CTX_PREPARED = 4;
     byte CTX_PREPARED_OK = 5;
     byte CTX_PREPARED_FAIL = 6;
     byte CTX_COMMITTED = 8;
     byte CTX_ABORTED = 16;
}
```

5.3 WorkingSetManager

The WorkingSetManager is in charge of the working set creation/initialization.

```
public interface WorkingSetManager {
     WorkingSet createWS(Object userObject)
          throws PersistenceException;
     WorkingSet createWS(Object userObject, Object workingSetType)
          throws PersistenceException;
```

The createWS methods create new instances of a working set whith a userObject. The workingSetType optional parameter could permit to choose the working type or to initialize it.

```
     void closeWS(WorkingSet ws);
```

The closeWS method permits to release a working set instance.

```
}
```

5.4 Working set life cycle

The following diagram shows the life cycle of a working set:

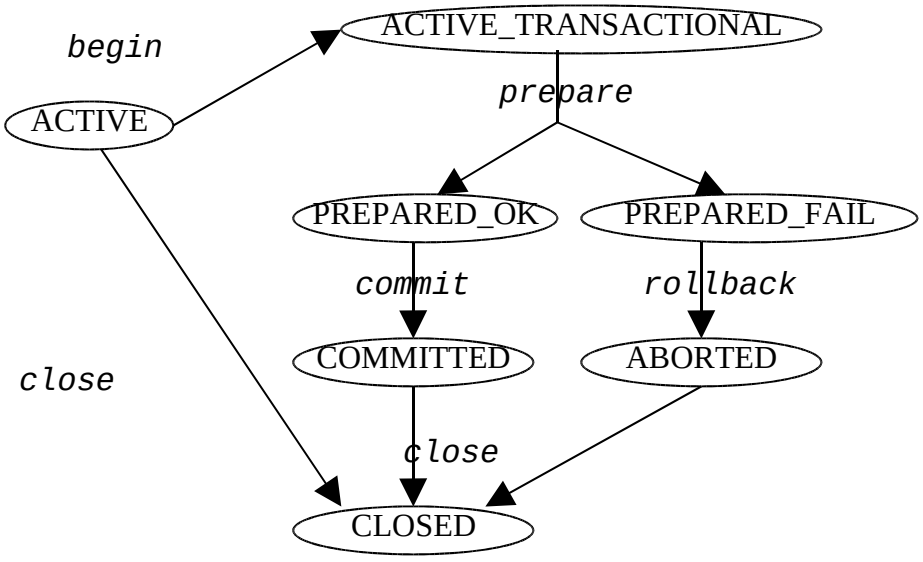


Figure 5: Working Set life cycle

6 DEPENDENCY GRAPH

In order to avoid dead lock during resource allocations, Perseus provides a dependency manager . A vertex represents a dependency between two tasks. A particular kind of task can be a working set.

Here is the definition of the DependencyGraph interface:

```
public interface DependencyGraph {
     boolean addVertex (Object src, Object dst);
```

The addVertex method creates a vertex (dependency) between two tasks if the new vertex does not create a cycle. The method returns a boolean indicating if the vertex has been create (ie if it does not create a cycle).

```
     void removeVertex (Object src, Object dst);
```

The removeVertex method removes a vertex between two tasks.

```
}
```

7 CONNECTIONHOLDER & CONNECTIONHOLDERFACTORY

The aim of a ConnectionHolder is to provide a way to find a connection to the data support. For example a previous used connection can be encapsulted in a ConnectionHolder in order to be reused it. The second interest of a connection holder is to abstract the transaction management. Then the perseus framework does not depend on the connection type and then it does not depend on the data source type too. Finally the transaction management can be delegate to a transaction manager (XA transaction) or implemented directly over a physical connection.

The definition of the ConnectionHolder interface is the following:

```
package org.objectweb.perseus.persistence.api;
public interface ConnectionHolder {

    Object getCHConnectionForRead() throws PersistenceException;
```

Retrieves a connection to access data on the support for a write action.

```
    Object getCHConnectionForWrite() throws PersistenceException;
```

Retrieves a connection to access data on the support for a read action.

```
    void begin() throws PersistenceException;
```

Demarcates the begin of a transaction.

```
    void commitCH() throws PersistenceException;
```

Committes the transaction.

```
    void rollbackCH() throws PersistenceException;
```

Rolles back the transaction.

```
    void releaseCHConnection() throws PersistenceException;
```

Indicates that the use of the connection is finished for instance.

```
    void closeCHConnection() throws PersistenceException;
```

Indicates that the connection is no more used definitively.

```
    void bindWorkingSet(WorkingSet ws);
    WorkingSet getWorkingSet();
```

Manages the association between the ConnectionHolder and a working set.

```
}
```

Perseus defines also a factory of ConnectionHolder instance:

```
public interface ConnectionHolderFactory {
   ConnectionHolder createConnectionHolder()
     throws PersistenceException;
}
```

8 MemoryInstanceManager

The memory instance manager is able to create memory instances from an identifier.

```
public interface MemoryInstanceManager {

   Object newInstance(Object oid, ConnectionHolder context)
     throws PersistenceException;
}
```

9 StorageManager

A StorageManager is able to make action on the data support in order to allocate identifier, to load or to store data. The main methods of the interface StorageManager are:

```
package org.objectweb.perseus.persistence.api;
public interface StorageManager {
   Object export(ConnectionHolder context, Object obj)
        throws PersistenceException;
   Object export(ConnectionHolder context, Object obj, Object hints)
        throws PersistenceException;
```

The export methods make persistent an object and to build a new identifier for this object. The optional hints parameter can help the identifier creation.

```
   void unexport(ConnectionHolder context, Object oid)
        throws PersistenceException;
   void unexport(ConnectionHolder context, Object oid, Object hints)
        throws PersistenceException;
```

The unexport methods mark as removed a persistent object.

```
   void read(ConnectionHolder context, Object oid, State state)
        throws PersistenceException;
   void read(WorkingSet context, Object oid, State state)
        throws PersistenceException;
```

The read methods load a persistent object from the data support into a memory instance(State).

```
   void write(ConnectionHolder context, Object oid, State state)
        throws PersistenceException;
```

The write methods flush into the data support, the persistent fields of a memory instance (State) identified by an oid.

```
}
```

Writing a persistent object can mean a creation of the persistent image, an update of the values or the deletion of the persistent image on the data support.

The use of this interface matches the use of JORM framework (http://jorm.objectweb.org). Therefore it is easy to implement a StorageManager based on JORM:

- The object identifier is a PName
- The state implements the Paccessor interface
- The state must permit to reach the PBinding. For example the PBinding can be the persistent object, the cache entry or another instance.

10 PConcurrencyManager

This section describes the extension of a generic concurrency manager in order to manage accesses to persistent objects managed with a PersistenceManager. The figure below shows the Concurrency Manager component used for the persistent object management:

Figure 6: A ConcurrencyManager for peristent objects

As shown on the figure the concurrency manager has two additional requirements:

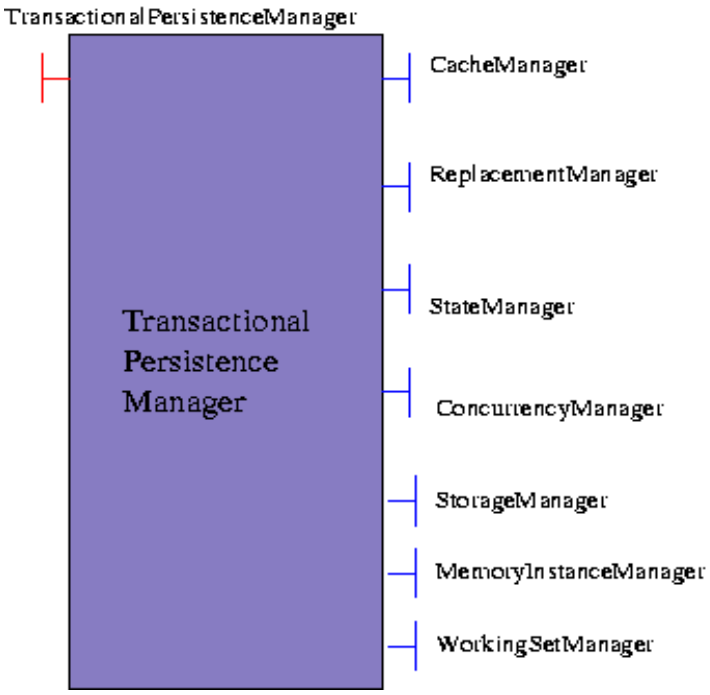
- The StateManager permits to manage the State of persistent objects (life cycle and status),
- The StorageManager permits to load data from the support into State intances.

In this particular use, we can advice the following implementation rules:

- a context is a a working set,
- the resource identifier is a the identifier of a cache entry (cacheEntry.getCeIdentifier()),
- the resource states match State instances
- in order to permit to the ConcurrencyManager to allocate State for a persistent object, the CacheEntry can be passed as hints of the readIntention and writeIntention methods.

11 PersistenceManager & TransactionalPersistenceManager

The persistence manager is in charge of managing the life cycle of persistent objects. It permits to create, remove and inform about intentional operations on persistent objects. The persistence manager component has 7 required interfaces as shown by the figure below:



- CacheManager: It permits to lookup an entry, to bind a new entry or to un/fix an entry.
- ReplacementManager: It permits to unbind entries from the cache
- StateManager: It permits to manage the state
- ConcurrencyManager: It permits to obtain right access to a persistent object and to fetch the state to use into a working set.
- StorageManager: It permits to manage identifier (export/unexport) and to load or to store state into the data support.
- MemoryInstanceManager: It permits to create new instance of persistent objects
- WorkingSetManager: It permits to manage the working set life cycle.

This chapter presents the API concerning a persistence manager and object interaction diagrams in order to better understand the role of important methods.

11.1 API overview

11.1.1 The PersistenceManager interface

```
package org.objectweb.perseus.persistence.api;
public interface PersistenceManager {

  State export(WorkingSet context, Object obj)
       throws PersistenceException;
  State export(WorkingSet context, Object obj,
       Object hints) throws PersistenceException;
```

The export methods permit to make persistent an object. The additional hints parameter can help the storage manager to allocate the identifier of the persistent object. For example if the identifier is based on a field of the persistent class, the hint could be the value of the field. The State of the new persistent object and bound to the working set is returned.

```
  void unexport(WorkingSet context, Object oid)
       throws PersistenceException;
  void unexport(WorkingSet context, CacheEntry ce)
       throws PersistenceException;
```

The unexport methods markes as removed a persistent object.

```
  CacheEntry readIntention(WorkingSet context, Object oid)
       throws PersistenceException;
  CacheEntry readIntention(WorkingSet context, CacheEntry ce)
       throws PersistenceException;
```

The readIntention methods are used in order to ask the read access to a persistent object. The use of these methods can occur the data loading if the persistent object with a valid state is not present in the cache.

```
  CacheEntry writeIntention(WorkingSet context, Object oid)
       throws PersistenceException;
  CacheEntry writeIntention(WorkingSet context, CacheEntry ce)
       throws PersistenceException;
```

The writeIntention methods are used in order to ask the write access to a persistent object. The use of these methods can occur the data loading if the persistent object with a valid state is not present in the cache.

```
  CacheEntry accessCompletion(WorkingSet context, State s)
       throws PersistenceException;
```

This method informes the persistent manager that a persistent object is no more used in a working set. The use of this method is optional.

```
void flush(WorkingSet context, StateFilter statefilter)
     throws PersistenceException;
```

The flush method flushes on the data support (through the storage manager) the dirty states. The statefilter parameter permits to the perseus user the filtering of the dirty states to treat.

```
boolean evict(WorkingSet context, Object oid, boolean force)
     throws PersistenceException;
boolean evict(WorkingSet context, CacheEntry ce, boolean force)
     throws PersistenceException;
int evictAll(WorkingSet context, boolean force)
     throws PersistenceException;
```

The evict methods try to evict persistent object from cache and to dissociate it from the working set. To be sure that no value stays in memory the reference state is removed. However if the persistent object is use by several working sets, then the persistent object cannot be evicted. A dirty object used inside a transaction cannot be evicted too. In this last error case a PersistenceExcpetion is thrown. However the implementation may support the eviction of a dirty object used out of a transaction.

```
void unbind(WorkingSet ws, Object oid)
     throws PersistenceException;
void unbind(WorkingSet ws, CacheEntry ce)
     throws PersistenceException;
```

The unbind methods unbind a persistent object from the persistent image into the data support. In other word the java instances is no more managed by perseus, and persistent data are not modified. Then the persistent object will be not availlable in the cache, and no identifier will be bound to it. A dirty object used inside a transaction cannot be unbound. In this last error case a PersistenceExcpetion is thrown. However the implementation may support a dirty object used out of a transaction.

```
WorkingSet createWS(Object userObject)
     throws PersistenceException;
WorkingSet createWS(Object userObject, Object workingSetType)
     throws PersistenceException;
```

The createWS methods permit to create new working sets. In fact the real creation is delegated to the WorkingSetManager. The userObject is an object which can be associated to the new working set. The optional workingSetType parameter might help the WorkingSetManager to create or initialize the working set.

```
void close(WorkingSet context) throws PersistenceException;
```

The close method permits to close a working set. The dirty object are flushed and the working set is released.

```
}
```

11.1.2 The TransactionalPersistenceManager interface

The TransactionalPersistenceManager is a PersistenceManager taking in account transactional aspects.

```
package org.objectweb.perseus.persistence.api;
public interface TransactionalPersistenceManager
   extends PersistenceManager {

   void begin(TransactionalWorkingSet context)
        throws PersistenceException;
```

The begin method demarcates the begining of a transaction. The working set becomes a transaction.

```
   boolean prepare(TransactionalWorkingSet ws)
        throws PersistenceException;
```

The prepare method demarcates the end of a transaction. The returned boolean value indicates if the transactional persistence manager allows the commit of the transaction. During this prepare step, the dirty object are flushed.

```
   void commit(TransactionalWorkingSet ws)
        throws PersistenceException;
```

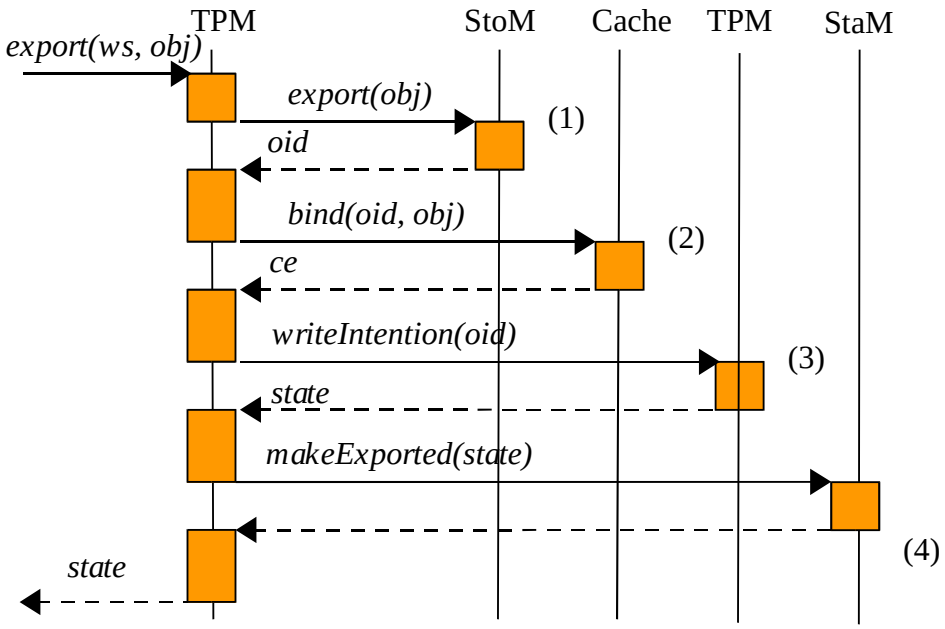
The commit method validates a transaction. This method can be call after the prepare method or directly for a one pahse commit.

```
   void rollback(TransactionalWorkingSet ws)
        throws PersistenceException;
```

The rollback method aborts a transaction. This method can be call after the prepare method or directly at any time in the transaction.

```
}
```

11.2 Export



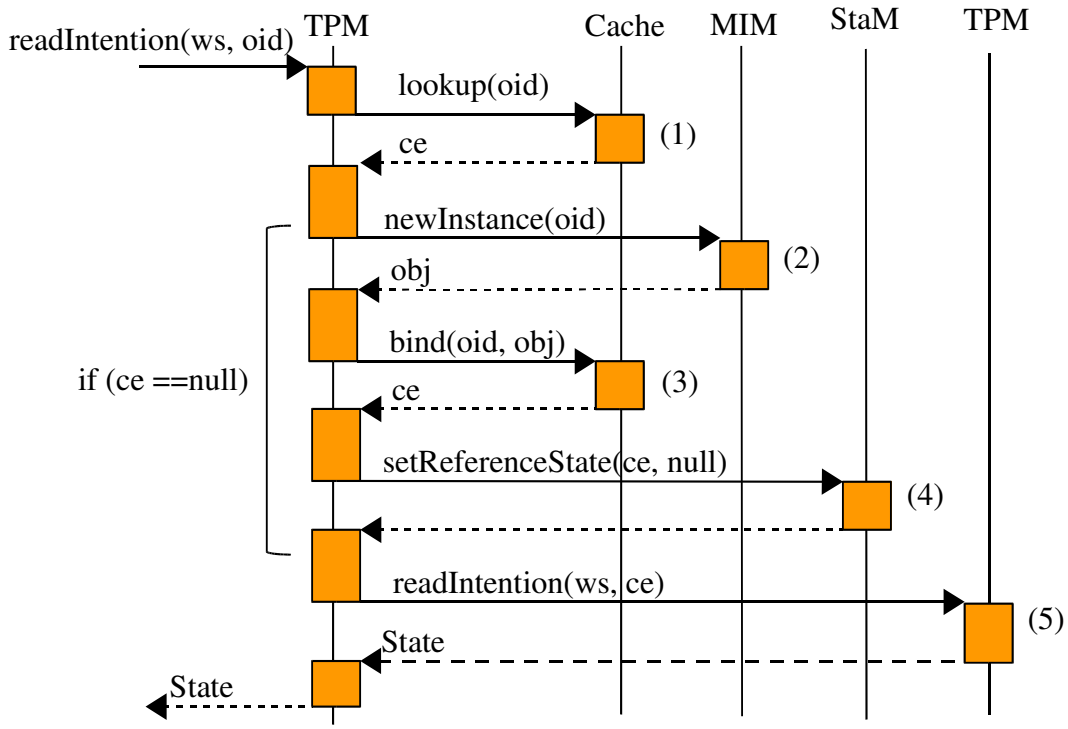
Get an identifier from the storage manager,
Put the object in the cache and fetch a CacheEntry,
Call the writeIntention,
Mark the object as exported with the StateManager.

Figure 7: PersistenceManager.export(WorkingSet, Object)

11.3 ReadIntention

The PersistenceManager interface provides two readIntention methods permiting to obtain read access to a persistent object.. The first has an Object parameter which represents the identifier of the persistent object to read whereas the second has the cache entry parameter of the persistent object to read.

Here is the object interaction diagram of the readIntention(WorkingSet,Object) method:



(1) Lookup in the cache,
if (ce == null) {
(2) Instanciate a new persistent object,
(3) Add the persistent object in the cache,
(4) Remove the reference state in order to force the loading,
}
(5) Call the readIntention method on the TPM

TPM: TransactionPersistenceManager
StaM: StateManager
MIM: Memory Instance Manager

Figure 8: PersistenceManager.readIntention(WorkingSet, Object)

Here is the object interaction diagram of the readIntention(WorkingSet, CacheEntry) method:

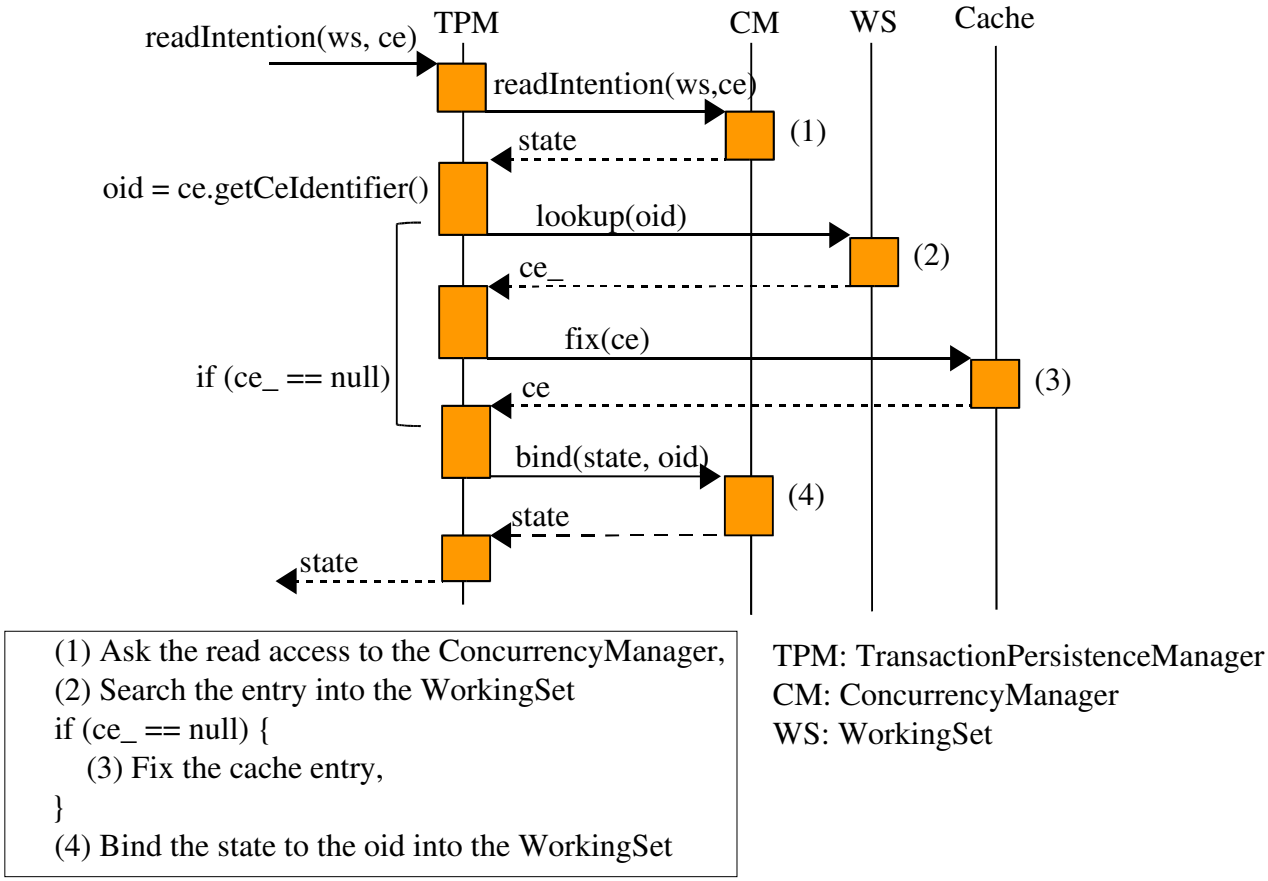


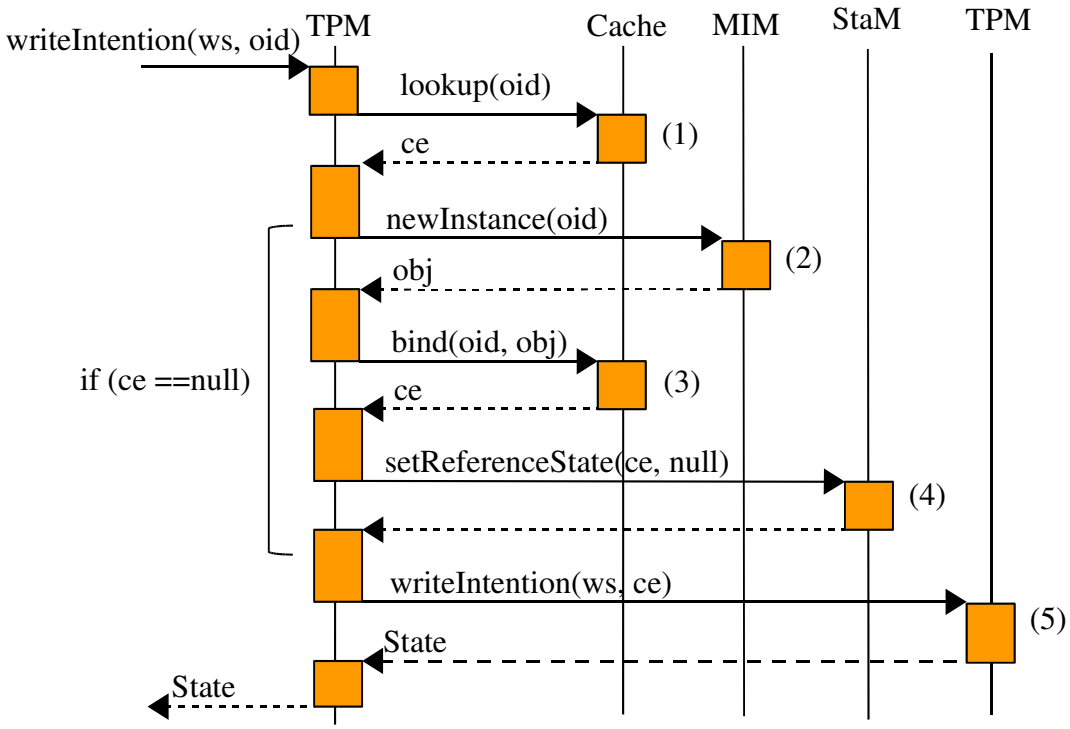
Figure 9: PersistenceManager.readIntention(WorkingSet, CacheEntry)

Note: If the ConcurrencyManager throws an exception and if there is an active transaction, the Perseus user must roll back this transaction by calling the rollback method. In addition, the exception means that the current user (identified by the working set) did not modify the resource because the read intention has been refused. Therefore, to avoid synchronization, it is advised to *unbind* the entry from the working set and then to *unfix* the element from the cache. This optimization makes the hypothesis that, when a resource is acquired in a read mode one time, a next request in read mode is always accepted. This is a constraint on the ConcurrencyManager implementation.

11.4 WriteIntention

The PersistenceManager interface provides two writeIntention methods permiting to obtain write access to a persistent object.. The first has an Object parameter which represents the identifier of the persistent object to modify whereas the second has the cache entry parameter of the persistent object to modify.

Here is the object interaction diagram of the writeIntention(WorkingSet,Object) method:



(1) Lookup in the cache,
if (ce == null) {
(2) Instanciate a new persistent object,
(3) Add the persistent object in the cache,
(4) Remove the reference state in order to force the loading,
}
(5) Call the writeIntention method on the TPM

TPM: TransactionPersistenceManager
StaM: StateManager
MIM: Memory Instance Manager

Figure 10: PersistenceManager.writeIntention(WorkingSet, Object)

Here is the object interaction diagram of the writeIntention(WorkingSet, CacheEntry) method:

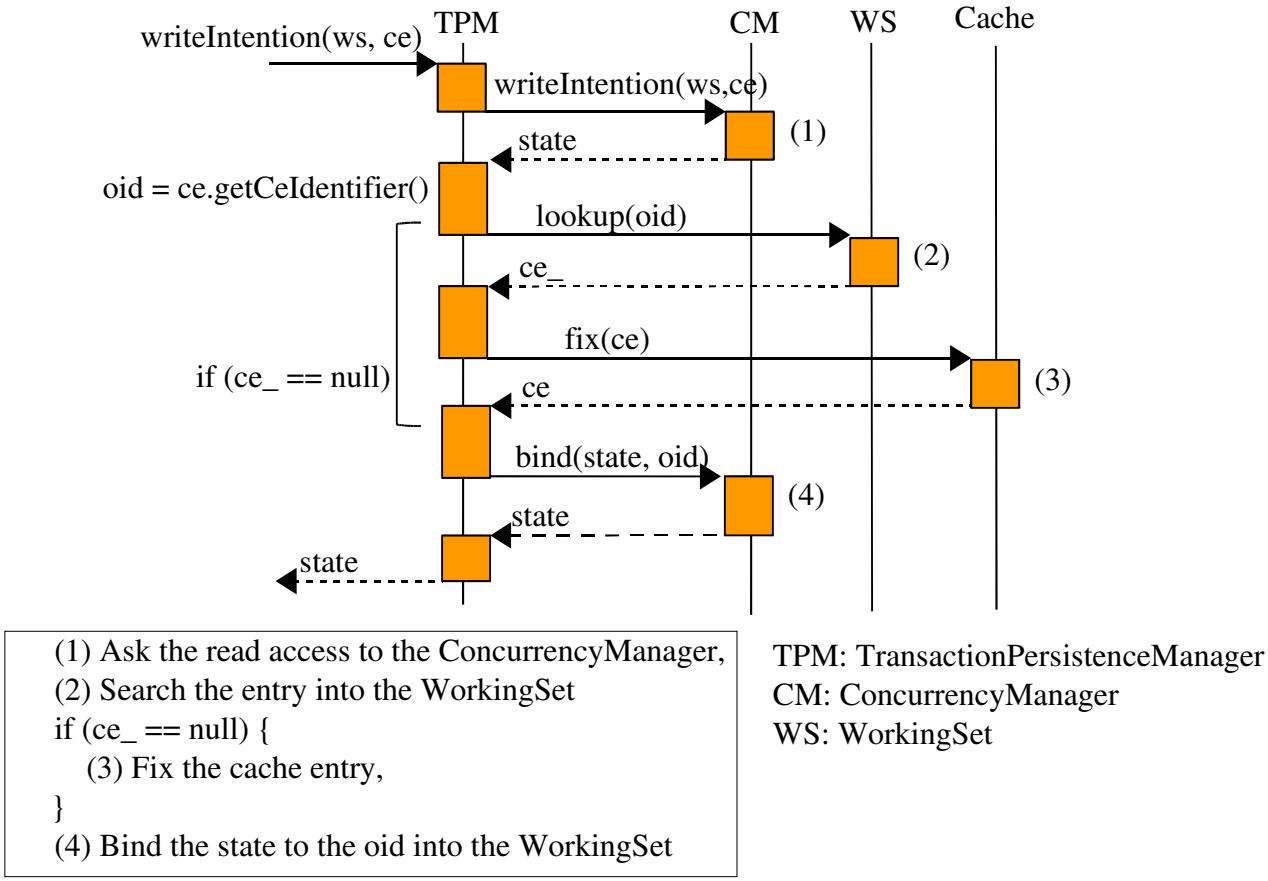


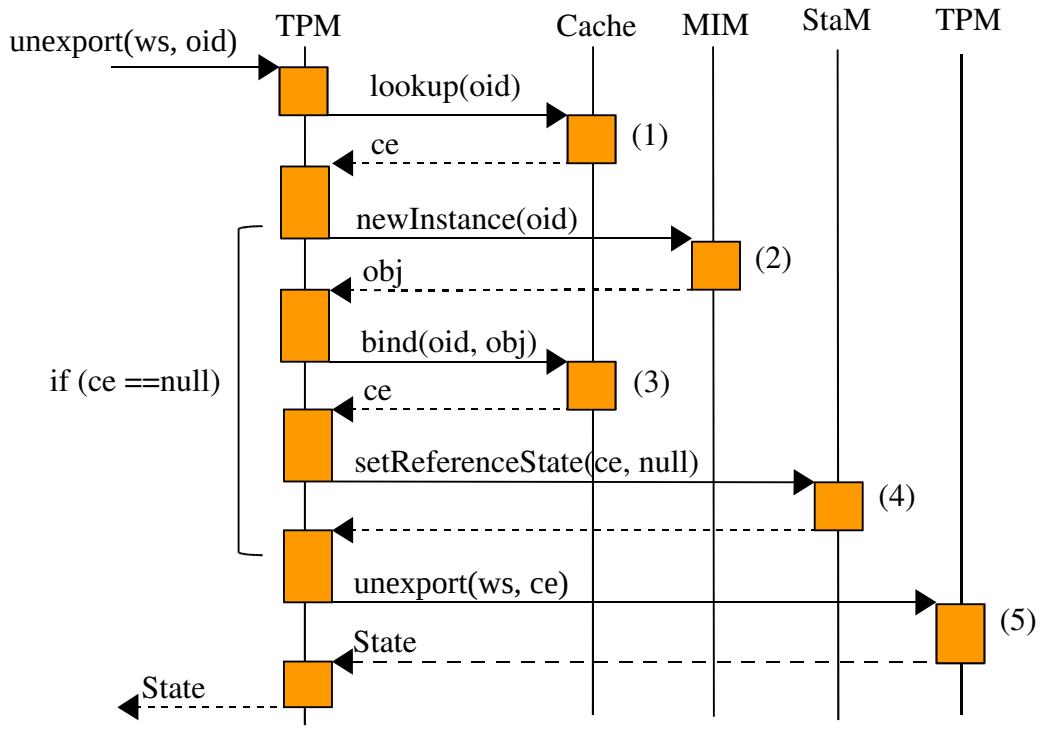
Figure 11: PersistenceManager.writeIntention(WorkingSet, CacheEntry)

Note: If the ConcurrencyManager throws an exception and if there is an active transaction, the Perseus user must rollback this transaction by calling the rollback method. In addition, the exception means that the current user (identified by the working set) did not modify the resource because the write intention has been refused. Therefore, to avoid synchronization, it is advised to *unbind* the entry from the working set and then to *unfix* the element from the cache. This optimization makes the hypothesis that, when a resource is acquired in a write mode one time, a next request in read mode is always accepted. This is a constraint on the ConcurrencyManager implementation.

11.5 Unexport

The aim of the unexport methods is to mark an instance as removed. The image in the data support can be removed immediatly or at commit time.

Here is the object interaction diagram of the unexport(WorkingSet, Object) method:



(1) Lookup in the cache,
if (ce == null) {
(2) Instanciate a new persistent object,
(3) Add the persistent object in the cache,
(4) Remove the reference state in order to force the loading,
}
(5) Call the unexport method on the TPM

TPM: TransactionPersistenceManager
StaM: StateManager
MIM: Memory Instance Manager

Figure 12: PersistenceManager.unexport(WorkingSet, Object)

Here is the object interaction diagram of the unexport(WorkingSet, CacheEntry) method:

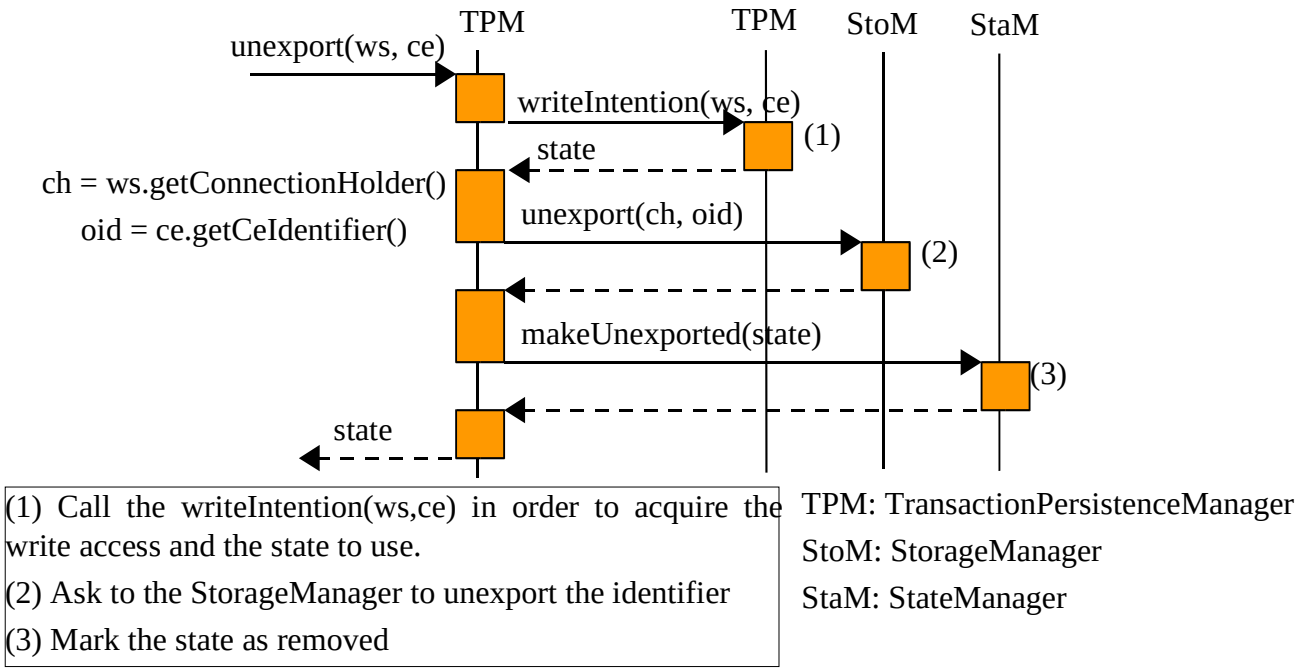
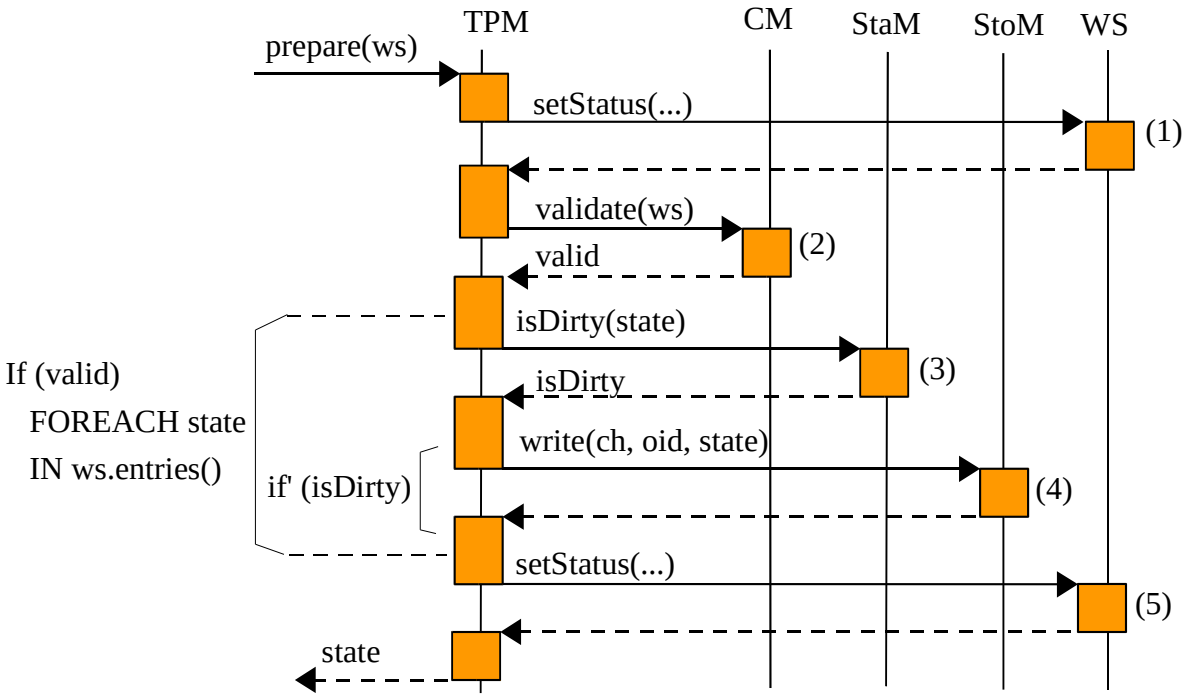


Figure 13: PersistenceManager.unexport(WorkingSet, CacheEntry)

11.6 Prepare



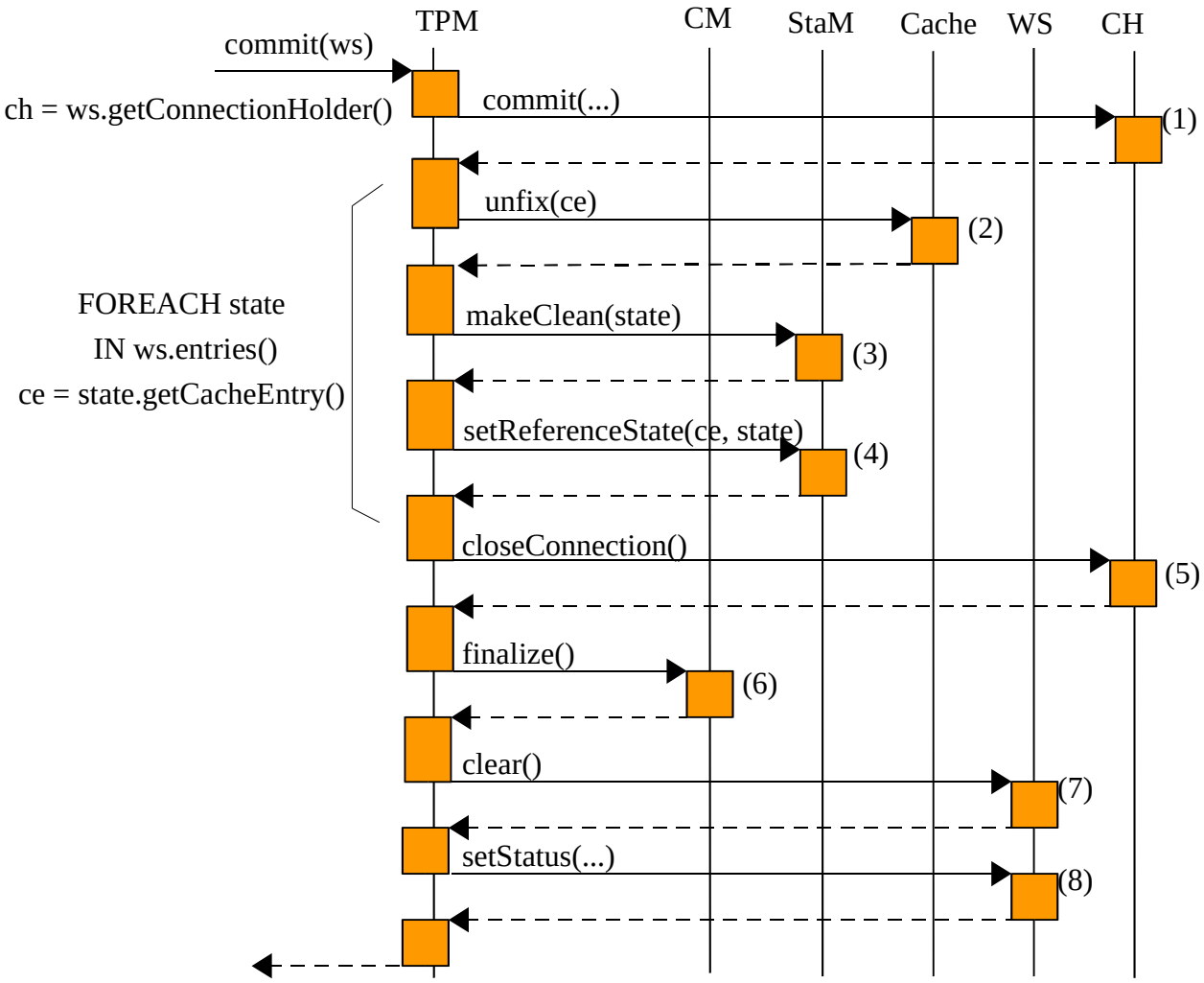
(1) Assign the new status to working set (PREPARE)
(2) Ask to the CM if the transaction can be validated
if (valid) {
  foreach state in ws.entries {
    if (staM.isdirty(state)) {(3)
      (4) Ask to the StorageManager to write the state
    }
  }
}
(5) Assign the new status to working set (PREPARE_OK)

TPM: TransactionPersistenceManager
CM: ConcurrencyManager
StoM: StorageManager
StaM: StateManager
WS: WorkingSet

Figure 14: PersistenceManager.prepare(WorkingSet)

11.7 Commit

The commit method permits to valid a set of actions grouped in a working set.



(1) Commit the transaction into the data support
foreach state in ws.entries {
ce = state.getCacheEntry()
(2) unfix the cache entry
(3) initialize the status of the state
(4) replace the reference state by the current
}
(5) close the connection
(6) free the lock
(7) empty the working set
(8) change the status of the working set

TPM: TransactionPersistenceManager
CM: ConcurrencyManager
CH: ConnectionHolder
StaM: StateManager
WS: WorkingSet

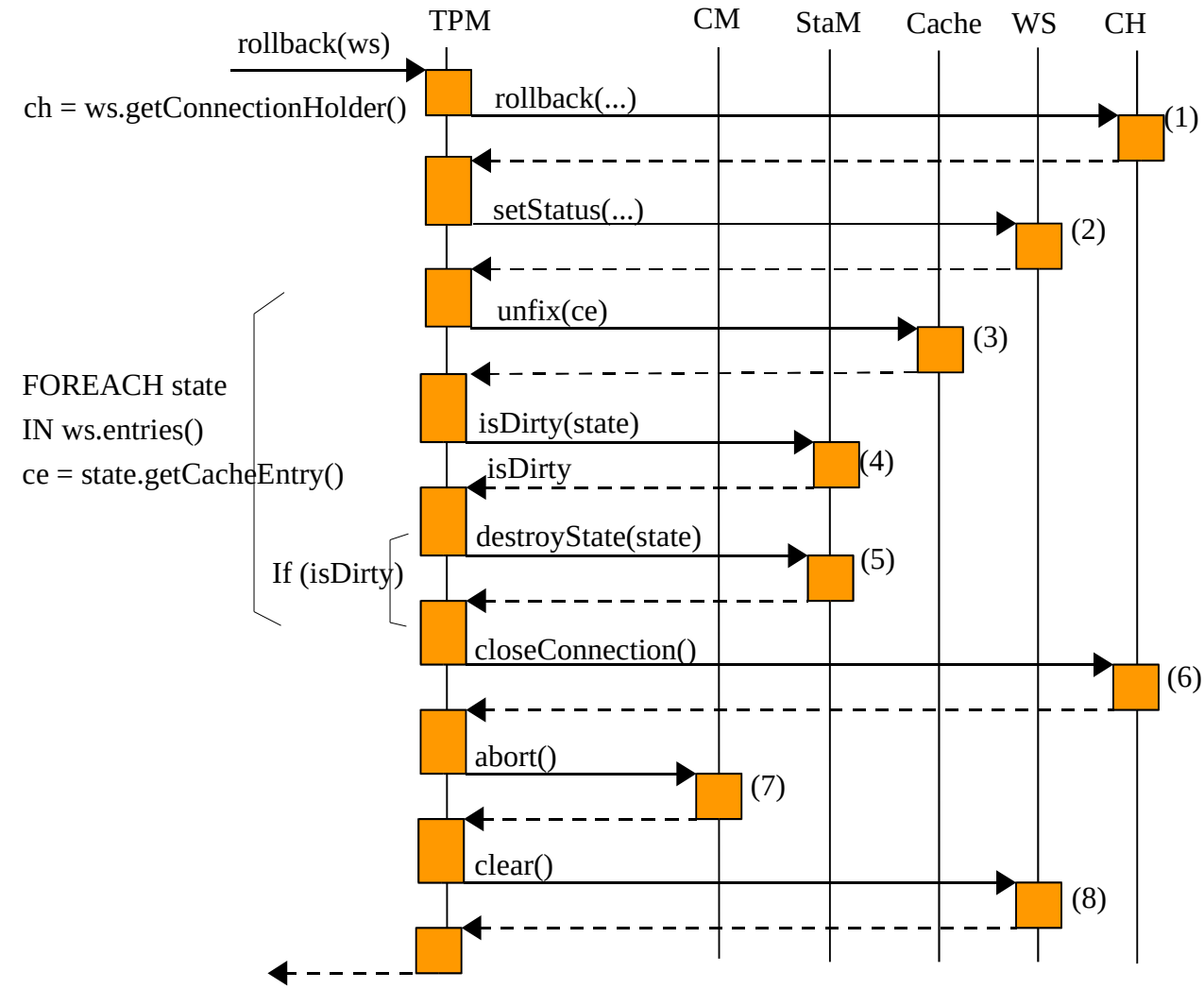
Figure 15: PersistenceManager.commit(WorkingSet)

In order to simplify the OID, the case of the persistent object deletion (unexport) is not taken in account. Then the (3) & (4) actions should be replaced by the following actions:

```
if (stateManager.isUnexported(state)) {
     // unbind the removed entry from the cache
     replacementManager.unbind((FixableCacheEntry) ce, true);
     // unbind the removed entry from the persistence system
     stateManager.makeUnbound(ce);
} else {
     if (tx.getWSRetainValues()) {
          stateManager.makeClean(state);
          stateManager.setReferenceState(ce, state);
     } else if (stateManager.getReferenceState(ce) == state) {
          //the current and reference state is removed
          stateManager.setReferenceState(ce, null);
     } else {
          // remove the current state
          stateManager.destroyState(state);
     }
}
```

11.8 Rollback

The rollback method permits to cancel a set of actions grouped in a transaction (particular working set).



(1) rollback the transaction into the data support
(2) change the status of the working set
foreach state in ws.entries {
  ce = state.getCacheEntry()
  (3) unfix the cache entry
  if (isDirty) {(4)
    (5) destroy the state
  }
}
(6) close the connection
(7) free the lock
(8) empty the working set

TPM: TransactionPersistenceManager
CM: ConcurrencyManager
CH: ConnectionHolder
StaM: StateManager
WS: WorkingSet

Figure 16: PersistenceManager.rollback(WorkingSet)

In order to simplify the OID, the case of the persistent object creation (export) is not taken in account. Then the (4) & (5) actions should be replaced by the following actions:

```
if (stateManager.isExported(state)) {
     //Created persistent object must be removed from the cache
     replacementManager.unbind((FixableCacheEntry) ce, true);
}
if (stateManager.isDirty(state)) {
     //The used state is dirty
     if (state == stateManager.getReferenceState(ce)) {
          //The used state is the reference state
          if (tx.getWSRestoreValues()) {
               //Restore value from the data support (loading)
               storage.read(tx.getConnectionHolder(),
                    ce.getCeIdentifier(), state);
          } else {
               //At the next use it will be loaded
               stateManager.setReferenceState(ce, null);
               stateManager.destroyState(state);
          }
     } else { //the used state is not the reference state
          //destroy the used state
          stateManager.destroyState(state);
     }
} else if (state != stateManager.getReferenceState(ce)) {
     //As the used state is not the reference state then destroy it
     stateManager.destroyState(state);
} //else reference state is already clean
```

There are several reasons to call this rollback method:

- The Perseus user wants to cancel the transaction,
- The validation of a transaction failed,
- The ConcurrencyManager decided to rollback a transaction, due to a dead lock problem for example.

11.9 Close

Closing a working set depends on the type of working set. If the working set is a transaction, the close operation does nothing that is of interest to Perseus. Otherwise it means that the modification on persistent objects has been done outside a transaction. Then it could be interesting to flush the modification into the data support like this following exemple:

```
if (prepare(ws)) {
     commit(ws);
} else {
     rollback(ws);
}
workingSetManager.close(ws);
```