SPT Object Database System Manual Version 0.1.1

Prepared by
Rakesh Vidyadharan

Prepared for
Interested Developers/Deployers

Dated: July 12, 2008

Contents

List of Tables

List of Figures

1 Introduction

SPT Object Database is a simple Object Database developed around the Prevayler transactional object persistence framework. SPT Object Database attempts to address some of the most glaring deficiencies in Prevayler while making full use of its great features. See [1] for a presentation of object databases and Prevayler.

1.1 Object Database

An object database is a persistent storage engine of objects[1] in much the same way that a relational database is a storage engine for raw data. The following are the basic attributes that we feel are required in an object database.

1. Persistent store for objects and object graphs.
   1.1. Persistence by reachability.
   1.2. Transparent handling of references to other persisted objects. Honour references when de-serialising a serialised persisted object.
2. Transactional semantics for object storage and management.
3. Query mechanism to retrieve objects.
4. Handling of indices and constrains declared on the objects being persisted.
5. Managed relationships - both uni-directional and bi-directional relationships between the persistent objects.
6. Fault tolerance.
7. Simple backup and recovery processes.
8. Simple to use without having to write a lot of code.

1.2 Prevayler

Prevayler is an open-source transactional object serialisation framework. Implementation of Prevayler are available for other technologies as well. As covered in [1], Prevayler is not a database. On the other hand, it makes users create something akin to an object database. The best part of Prevayler is its high degree of fault tolerance. In numerous tests, Sans Pareil Technologies, Inc. has been unable to cause any data corruption in the data persisted using Prevayler.

[1] In our case only Java objects.

1.3 Source Repository

Full source code for the database engine and associated unit test suite are available in the java.net subversion repository. The following instructions describe the process of checking out the source code and building the engine.

```
svn co https://sptodb.dev.java.net/svn/sptodb/trunk \
  sptodb --username <username>
cd sptodb
cp ant.properties.sample ant.properties
vi ant.properties # Edit as necessary to suit environment
ant # Builds the jar file and cleans class files
ant test # Runs the unit test suite
```

The `trunk` area of java.net subversion repository will contain the latest snapshot of code for the engine. The `tags` area will contain tagged versions with numbers reflecting the progress of the database engine.

2 Project Goals

The goal of this project is to create a simple object database for persisting and managing Java objects without requiring the user to have to write their own database engine. The following are the features that are planned for the database engine.

1. Transparently persist complex object graphs. Prevayler requires the developer to de-couple direct references to other persistent objects due to the inherent nature of Java object serialisation. This is obviously entirely contrary to all Object-Oriented principles and is an unjustifiable requirement for application development. This project will transparently handle the de-coupling and reconstitution of the object references thus preserving the purity of the object model.
2. Genericised transaction and query objects. Prevayler requires the developer to create distinct transaction and query objects for every type of operation the user wishes to perform against the store. This is a tedious and unnecessary process that may also be avoided by created generic transaction and query objects that may be used under a wide range of operating conditions.
3. Automatically index persistent objects by annotating the appropriate fields in the class. Annotations attempt to stay similar to JDO annotations.
4. Automatically index fields annotated as searchable using Lucene to support full-text search capability.
5. Enforce constraints declared through annotations on the persistent class while managing (adding/updating/deleting) persistent objects.

6. Managed relationships. Persistent objects maintain a relationship with other persistent objects when they maintain references to them either directly or in collections. The database engine will ensure that proper actions are taken when a persistent object that is in use by other persistent objects is to be removed from the store.

3 Database System

Source code for the SPT Object Database system is organised under the `com.sptci.prevayler` root package. Table 1 shows the packages used and their purpose. Note that the root `com.sptci` has been omitted from the package descriptions for succinctness.

Table 1: Source file packages

Pacakge	Description
`prevayler`[a]	The root package under which the core classes for the database engine reside.
`prevayler.annotations`[b]	The root package under which the annotations that are used to configure rules for the persisted objects reside.
`prevayler.query`[c]	The root package under which the general purpose queries implemented for the system reside.
`prevayler.transaction`[d]	The root package under which the standard transactions implemented for the system reside.

[a] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/package-frame.html
[b] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/annotations/package-frame.html
[c] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/query/package-frame.html
[d] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/transaction/package-frame.html

3.1 PrevalentObject

SPT Object Database is designed to only persist instances of PrevalentObject[2]. PrevalentObject is an abstract class and as such cannot be used directly. The following reasons lead to Sans Pareil Technologies, Inc. deciding to make PrevalentObject an abstract class instead of a more flexible Interface.

- The PrevalentSystem needs to set the `objectId` for the prevalent object when not using application specified object id. This can be guaranteed only through the use of a super-class since interfaces cannot mandate the use of fields.
- The PrevalentSystem needs to maintain the metadata associated with the prevalent object that is maintained. This again requires the presence of a private field in the object.

[2] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/PrevalentObject.html

- Object id must be immutable. This is not possible to enforce by using an interface, since the implementation can chose to make the object id mutable via a publicly accessible mutator method.
- Interfaces have no implementation. An abstract base class allows the system to supply default implementations of critical methods such as `equals`[3] and `hashCode`[4] that sub-classes are discouraged from over-riding. Note that these methods have not been declared `final`, although that could change in future.

3.2 Prevalent System

The prevalent system is the primary class that is managed by an instance of Prevayler[5]. The system is broken up into an inheritance hierarchy based upon the facets managed by each class in the chain. The ultimate class in the chain is named PrevalentSystem and is used to initialise the Prevayler framework. Figure 1 shows the class diagram for the classes that comprise the prevalent system.

3.2.1 StorageSystem

The root of the inheritance hierarchy as depicted in figure 1. This class[6] is responsible for managing the various storage containers used to store and index the prevalent objects managed by the prevalent system. This class does not implement any rules for managing the persisted objects. All rules required for managing the persisted objects are implemented in the sub-classes as appropriate. Storage containers are usually Map[7] instances for efficient retrieval.

3.2.2 IndexSystem

The class[8] responsible for indexing the prevalent objects being persisted to the prevalent system. Indexing is performed by analysing the annotations defined on the prevalent class and populating the appropriate storage containers made available by StorageSystem (see section 3.2.1). This class is also responsible for enforcing any unique constraints defined on the prevalent objects.

[3] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/PrevalentObject.html#equals(java.lang.Object)
[4] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/PrevalentObject.html#hashCode()
[5] http://docs.rakeshv.org/java/prevayler/org/prevayler/Prevayler.html
[6] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/StorageSystem.html
[7] http://docs.rakeshv.org/java/j2sdk1.5/docs/api/java/util/Map.html
[8] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/IndexSystem.html

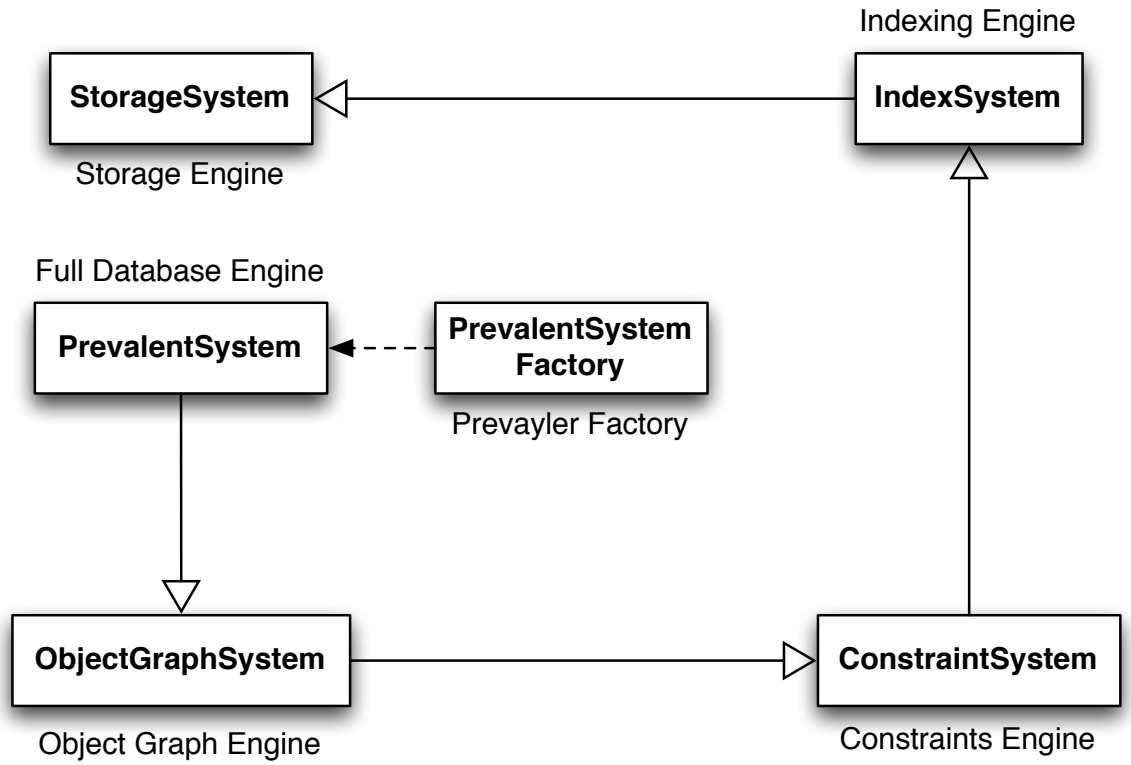

Figure 1: Storage Model Class Diagram

3.2.3 ConstraintSystem

The class[9] responsible for processing foreign key relations as annotated in the prevalent class. Foreign keys are implicitly indexed and hence fields annotated as foreign keys do not need to be additionally indexed. The operations defined in this class are complimentary to the operations defined in IndexSystem (see section 3.2.2). Similar to IndexSystem this class also enforces unique constraints defined for the foreign key fields.

3.2.4 ObjectGraphSystem

The class[10] responsible for decomposing and reconstituting the direct references from a prevalent object to other prevalent objects in the prevalent system.

[9] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/ConstraintSystem.html
[10] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/ObjectGraphSystem.html

3.2.4.1 Decomposition Decomposition[11] of a prevalent object is performed by applying the following rules:

1. Operations on direct references to other prevalent objects.
   1.1. Clone the prevalent object that is being persisted. This ensures that the objects persisted in the prevalent system are encapsulated from client code. Note that it is highly recommended that the Object#clone[12] method be implemented in all the prevalent classes in your system to ensure proper encapsulation of persisted objects.
   1.2. Persist the referenced object if it is a prevalent object and not already persisted.
   1.3. Add the reference key (the object id for the referenced object) along with the type information to a container managed specifically for instances of the object being persisted. This step is recursive ensuring that the entire object graph is decomposed.
   1.4. Set the field in the cloned prevalent object instance to `null`.
2. Operations on references to other prevalent objects stored as collections in the prevalent object being persisted.
   2.1. Clone the prevalent object that is being persisted. This ensures that the objects persisted in the prevalent system are encapsulated from client code.
   2.2. Clone the collection field as default cloning does not clone the objects contained in the object that was cloned.
   2.3. Persist the referenced objects in the collection if they are not already persisted.
   2.4. Add the reference keys (the object id for the referenced objects) along with the type information to a container managed specifically for instances of the object being persisted.
   2.5. Remove the processed prevalent object from the collection.

3.2.4.2 Recomposition Recomposition[13] of a prevalent object is performed when clients request instances of the object through a query. Recomposition is performed by applying the following rules:

1. Operations on direct references to other prevalent objects.

[11] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/ObjectGraphSystem.html#decompose(com.sptci.prevayler.PrevalentObject)
[12] http://docs.rakeshv.org/java/j2sdk1.5/docs/api/java/lang/Object.html#clone()
[13] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/ObjectGraphSystem.html#compose(com.sptci.prevayler.PrevalentObject)

1.1. Clone the prevalent object that was persisted. This ensures that the objects returned are encapuslated against unwitting modifications through client code. Note that it is highly recommended that the Object#clone[14] method be implemented in all the prevalent classes in your system to ensure proper encapsulation of persisted objects.

1.2. Reconstitute the prevalent object for the referencing field by fetching the recomposed instance based upon its object id as stored in the relevant container. This step is recursive and ensures that the entire object graph is recomposed.

1.3. Set the field in the cloned prevalent object instance to the recomposed prevalent object.

2. Operations on references to other prevalent objects stored as collections in the prevalent object being persisted.

2.1. Clone the prevalent object that was persisted. This ensures that the objects returned are encapsulated against unwitting modifications through client code.

2.2. Instantiate a new collection and populate it with the recomposed references based upon the object id values stored in the relevant container.

2.3. Set the field in the cloned prevalent object instance to the initialised collection.

3.2.5 PrevalentSystem

This is the last class[15] in the inheritance hierarchy that constitutes a prevalent system that may be managed by a Prevayler[16] instance. This class defines the methods that are necessary to support object manipulation and query methods.

3.2.5.1 save This is the method that supports saving prevalent object instances to the prevalent system. Constraint checking as configured for the prevalent class through annotations are performed prior to persisting the object to the prevalent store. The prevalent object is decomposed (see section 3.2.4.1) prior to persisting. Note that this method may be used to add or update prevalent instances in the system.

3.2.5.2 fetch A few over-loaded `fetch` methods are provided for fetching prevalent instances from the prevalent system. Standard methods for fetching by `objectId`, by indexed fields, objects within a range . . . are provided by default.

[14] http://docs.rakeshv.org/java/j2sdk1.5/docs/api/java/lang/Object.html#clone()
[15] http://www.sptci.com/products/odb/docs/com/sptci/prevayler/PrevalentSystem.html
[16] http://docs.rakeshv.org/java/prevayler/org/prevayler/Prevayler.html

3.2.5.3 delete This is the method that supports deleting prevalent object instances from the prevalent system. Constraint checking is performed on objects that hold references (direct or via collections) to the object being deleted to determine if the object may be deleted.

3.3 Transactions

Genericised Transaction[17] objects are provided for invoking the add (see section 3.2.5.1) and delete (see section 3.2.5.3).

3.4 Queries

Genericised Query[18] objects are provided for the various standard queries supported by the PrevalentSystem.

3.5 PrevalentManager

This is a façade around the PrevalentSystem that presents a more natural higher level API than that provided by Prevayler. The interface is designed to be similar to JDO PersistenceManager[19] and abstracts operations on the PrevalentSystem through a simple direct API. Using the prevalent system through this façade is recommended over using the system through the PrevalentSystem and Prevayler.

Note:

The PrevalentManager does not allow for customisation of the storage for the persited objects. If you wish more control over the number of PrevalentSystem instances used to manager your object graph then you will need to use PrevalentSystem directly. Other aspects such as controlling the location of the persistent storage may be controlled as usual through JVM properties[20].

3.6 Configuration

The prevalent system may be configured through JVM system properties. Table 2 shows the system properties that are supported and their purpose.

[17] http://docs.rakeshv.org/java/prevayler/org/prevayler/TransactionWithQuery.html
[18] http://docs.rakeshv.org/java/prevayler/org/prevayler/Query.html
[19] http://docs.rakeshv.org/java/jdo2/javax/jdo/PersistenceManager.html
[20] See section 3.6 for details.

Table 2: JVM System Properties

Property	Default Value	Description
`sptodb.data.dir`	`/var/data/sptodb`	The root directory under which the database snapshot and journal files are stored. Lucene search indices are also stored under this directory.
`sptodb.snapshot.interval`	`86400`	The interval in seconds at which snapshots of the prevalent system are to be taken. Note that snapshots reduce the start-up time of the prevalent system.
`sptodb.serialiser.format`	`java`	The format to use for taking snapshots of the prevalent system and creating transaction journals. The supported options are: **java** - Indicates that regular Java object serialisation be used to take the snapshot and write journals. **xml** - Indicates that the journals should be written and snapshot taken using XStream. XML serialisation is slower (both serialisation and deserialisation), however gives you more options for processing the prevalent system for other purposes. Also useful if you need to restore the system after heavy modifications (refactoring) to the object model.

References

[1] *Object Databases*[21] - *CJUG*[22] *Presentation.*

[21] http://www.sptci.com/products/articles/ObjectDatabases.pdf
[22] http://www.cjug.org/Wiki.jsp?page=2008.5.20.downtown