The DataManager Kernel: A Guide

Benjamin Keil

December 12, 1999

Contents

Chapter 1

Introduction

This documents explains at a moderate level some of the relationships of the classes within the kernel. A higher-level description can be found on the Data Manager API Design page, and should probably be read first, as many of the concepts it lays out are assumed in this document. A lower-level understanding of the kernel can be obtained through browsing the source code and the javadoc files.

Throughout this document, I have followed a convention of putting concepts in "NOUN STYLE" and actual classes in "`typewriter style`". Notice that this distinguishes between ENTITYs and objects of the `Entity` class (of which there are, of course, none — `Entity` is an abstract class).

Chapter 2

The Kernel package

In the kernel package are the core classes that make everything that takes place in the DataManager possible. The classes in this package deal with the creation and management of THREADs and ENTITYs. The kernel also contains code for the process of starting the DataManager up and shutting it down.

2.1 The ENTITY and its collaborators

ENTITYs have a dual nature. From a user's point of view, ENTITYs are the fundamental bricks of the data-manager, they store the data and build larger structures with attributes, etc. From the programmer's point of view, ENTITYs are responsible for persistence and event generation. Putting all of this functionality in a single class would be both a programming headache and a maintenance nightmare. For this reason, ENTITYs are support by a cast of several collaborating classes.

The main collaborating classes are:

- the `EntityProxy` class (see Section 2.1.1), which controls access to entity functions
- the `EntityManager` class (see Section 2.1.2), which provides an interface to track relations between `EntityProxy`s and the entities for which they are proxying, as well as any possible relations between the entities in memory and their stored representations
- the `EventGeneratorAssistant` (see Section 2.1.3) class, which takes care of the subscription and event framework
- the `EntityValue` interface (see Section 2.1.4), which provides a mechanism to store an ENTITY's data

The `Entity` class itself is basically empty. It is an abstract class, and could be an interface but for one thing: it holds the factory method, `Entity.create(String name)`.

2.1.1 The EntityProxy

Any reference of type `Entity` that escapes the inner workings of the Kernel is actually a reference of type `EntityProxy`. The key motivation behind the EntityProxy is that Entitys should be protected from malicious and/or thread-unsafe use. The indirection to an EntityProxy gives Hydrogen the (as of yet unused) ability to check permissions for the access to the Entity. This also allows Simpletons and other actors to keep entity Identifiers, with out forcing the (possibly memory intense) back end entity to actually be in memory.

The `EntityProxy` class is nearly as weightless in design as the `Entity` class. It contains an `Identifier`, which uniquely specifies the entity (whatever class is suitable for the current back end) for which it is proxying. It uses this Identifier to communicate with the EntityManager, which gives a reference to the appropriate back end entity. The proxy then uses this reference to pass on the method request to its intended target. This indirection is used so that the call can be wrapped with code checking permissions, etc.

A diagram showing how what a Simpleton or other actor does to the EntityProxy it receives from Kernel actually gets mapped to an action on a back end entity can be seen in Figure 2.1.

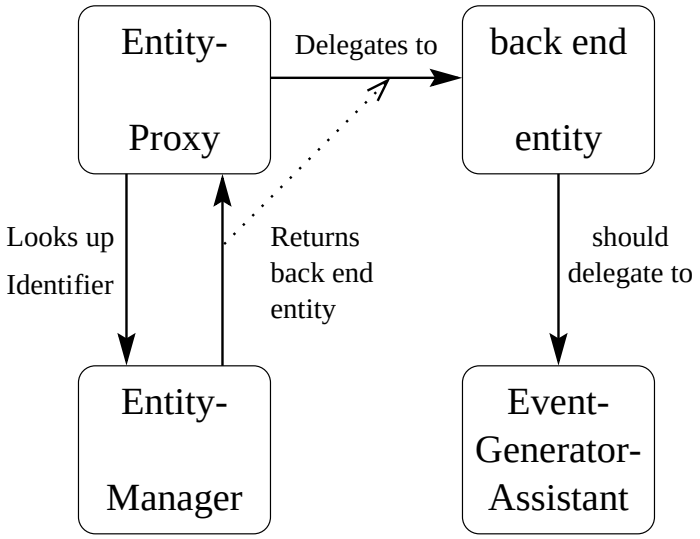

Figure 2.1: Proxies mediate between Entities and their back end counterparts

2.1.2 The EntityManager

The job of the EntityManager is to manage the relationships between in-memory entities and their on-disk counterparts. These relationships can be arbitrarily complex. The job of the `EntityManager` class is to create and initialize an instance of the class that implements the actual management of these relationships. Currently there is only one such implementation:

2.1.2.1 The `VerySimpleEntityManager` and `VerySimpleEntity` classes.

The `VerySimpleEntityManager` is just that, a very simple entity manager. It gains persistence through serialization. This is one of the weaker spots in the kernel implementation currently, as it relies on a block of static code in the `EntityManager` class to initialize an `ObjectInputStream` and load the Entitys into memory, and the deprecated `java.lang.System.runFinalizersOnExit` to call a finalizer method that writes them out to disk.

It keeps all the `VerySimpleEntitys` in memory at all times, relying on the Java Virtual Machine and operating system to provide a reasonable caching scheme. It contains `HashTables` relating `EntityProxy` to `Identifier` (which at the moment is not necessary — `EntityProxys` use their `Identifiers` as hash codes — but is included for flexibilities sake) and `Identifier` to `VerySimpleEntity`. These `HashTables` facilitate the lookup methods which the Proxys use to get a reference to the proper back end entity and any given time.

`VerySimpleEntitys` are also very simple. They reference each other (i.e., as attributes and bases) by `Identifier`, as that is guaranteed to be the same between sessions. Otherwise they are inert, as the Java serialization does all of the persistence work.

2.1.2.2 What other EntityManagers could do.

1. Implement a caching scheme
2. Use a database as a back end. If the pool (see Section 2.2) could use this too, we would have a better system for doing searches
3. Support transactions for updates to several Entitys simultaneously

2.1.3 The `EventGeneratorAssistant`

The demands of the `EventGenerator` interface can add a lot of complexity to a class. To avoid this unnecessary complexity, an EventGenerator should delegate the responsibility. This is a kernel implementation to that end.

2.1.4 The EntityValue

EntityValues come in two flavors: Passive (see Chapter 3) and Active (see Chapter 4). These two flavors are fundamentally different, as described in the design document and the API. Each receives a formal treatment in its individual chapter.

The `EntityValue` interface serves as a marker for all EntityValues and extends the `java.io.Serializable` interface to facilitate easy dumping of their data to disk. For hackers writing new EntityValues, this means that it is important to think about using the `transient` keyword wisely in their designs.

2.2 The Pool class

The Pool is a special collection of Entitys, primarily special in that it can be searched. This search is done in terms of Constraints (see Chapter 5), which are a powerful filter language used in the data manager.

Currently the pool performs the search via a brute force mechanism, that is, every Entity that has been added to the pool is tested against the given Constraint. This is horrible inefficient, and could be much improved upon. If, for example, there were a kernel interface to a database engine, the constrain could be translated into the appropriate query and answered by the database.

2.3 Thread Stuff

to be added

2.4 The DataManager class and its loaders

to be added

2.5 Support for the Constraint package

One use of constraints is to listen to a filtered stream of events. Since the code that the constraints use to check Events or Entitiys is code that (normally) would trigger events, this could lead to a disastrous infinite loop, where a Constraint checking an Event could create an Event, that would be checked by a Constraint, generating an Event and so forth into infinity. This is, of course, not the desired behavior. For this reason there are in the kernel two classes to deal with this:

2.5.1 The `AbstractEntityConstraint`

Any CONSTRAINT that extends this CONSTRAINT, will implement a `describe` method. When anything using the CONSTRAINT calls the `accepts`, the `accepts` method in `AbstractEntityConstraint` will temporarily disable the current thread's ability to trigger events, check the `describes` method, reset the event-triggering state of the thread, and return the appropriate value. See figure 2.2 for a visual account of this process.

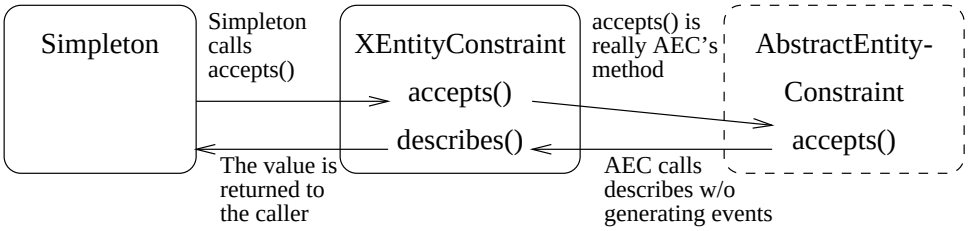

Figure 2.2: A flow diagram for AbstractEntityConstraint.

2.5.2 The `AbstractEventConstraint`

This CONSTRAINT works exactly as above, but with `EventConstraints`.

Chapter 3

The PassiveEntityValue package

PassiveEntityValues are wrappers around other data types, such as int, double, boolean, String, and Vector. There are a few key classes interfaces in this package:

3.1 The `NumericalEntityValue` interface

This interface serves as a marker for all numerical EntityValues. It provides common methods for arithmetic and comparison. It extends the comparable interface, so groups of numeric EntityValues can easily be sorted. Currently implemented `NumericalEntityValues` are:

- `IntegerEntityValue`
- `DoubleEntityValue`
- `NumberEntityValue` (an abstract class that holds code common to the two above)

3.2 The `HierarchicalEntityValue` interface

This interface serves as a marker for any kind of EntityValue that can be described as a hierarchy, or tree, with nodes that are parents and children of each other. A good example of this is the relationship between classes and subclasses. For that reason we have:

- `EventEntityValue`

3.3 The `String-` and `BooleanEntityValue` classes

These classes are wrappers for their java.lang counterparts. They implement the exact same API.

3.4 The `NullEntityValue` class

This class represents the null value as an EntityValue. All references to `NullEntityValues` are considered equal.

Chapter 4

The ActiveEntityValue package

To be added

Chapter 5

The Constraint Package

There are three types of CONSTRAINTS in the constraint package. There are constraints that are specifically for defining types of ENTITYs (main grouped by ENTITYVALUE) and there are constraints for defining certain kinds of EVENTS. Each of these constraints is a powerful tool in and of itself, but a third type of constraint gives the systems of CONSTRAINTS an even greater power: boolean logic CONSTRAINTS. The purpose of each constraint is well documented in the DataManager API.

It is important for Constraint writers to note, however, that even though the the public API method is `accepts()`, the appropriate method to implement in a new constraint is `describes()`. The mechanism for `accepts()` is implemented in the Kernel package. See Section 2.5 for a description of how this works and why we did it.

Chapter 6

The DataManagerEvent Package

to be added