Component Design: MSE Project

November 22, 2010

Prepared by Doug Smith
Version 0.1

Table of Contents

Revision History

Version	Date	Changes
0.1	11/22/2010	First draft.

Introduction

This document captures the internal component design for the software associated with my MSE project. This document expands on the information presented in the Software Architecture document [1].

This document is organized by the package structure of the system, as well as the services offered by the system.

References

[1] Software Architecture: MSE Project, Doug Smith, http://people.cis.ksu.edu/~dougs/Site/Phase_2_Artifacts_files/sad.pdf

[2] Project source code repository: https://code.google.com/p/ds-ksu-mse/

[3] Neeraj Sangal, Ev Jordan, Vineet Sinha, and Daniel Jackson. 2005. Using dependency models to manage complex software architecture. *SIGPLAN Not.* 40, 10 (October 2005), 167-176. DOI=10.1145/1103845.1094824 http://doi.acm.org/10.1145/1103845.1094824

Package Structure

The following diagram shows the package structure of the project:

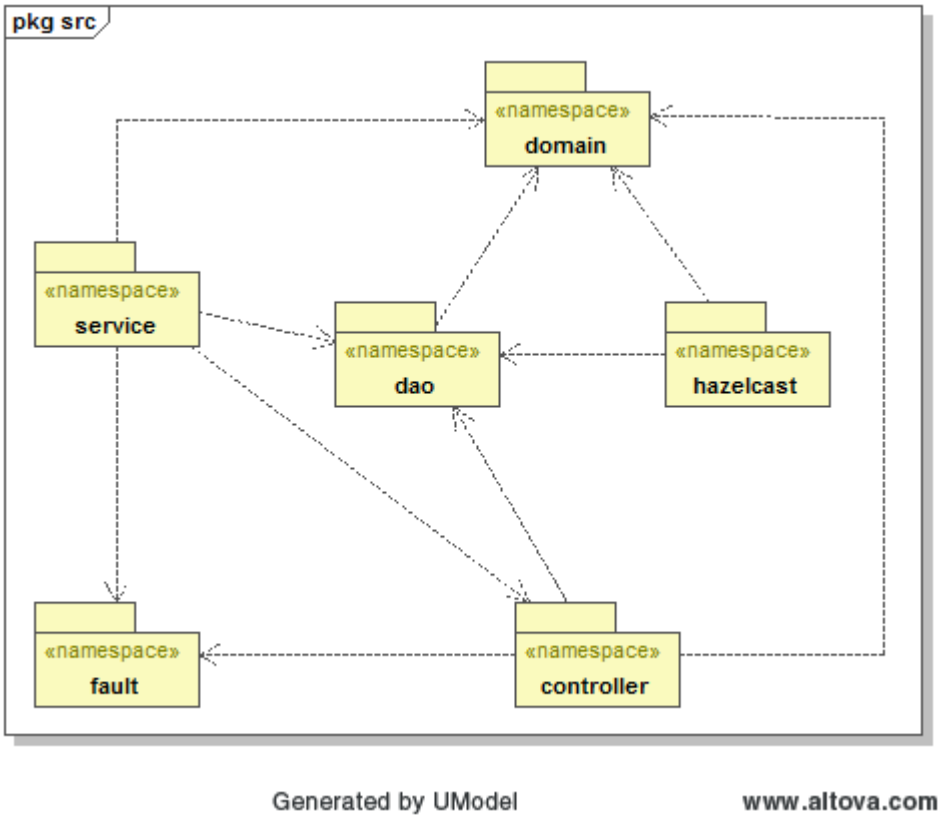


Figure 1 Package Structure

The packages structure the code as follows:

- Controller – classes used to orchestrate the collaboration of other classes to realize service transactions.
- DAO – data access objects used to encapsulate access to the database.
- Domain – classes representing entities in the problem domain. These directly correspond to the classes detailed in the formal specification.
- Fault – contains code related to service faults that are marshaled via the web service layer.
- Hazelcast – contains implementation of Hazelcase load and store interfaces, used to read data into the in-memory data grid (IMDG) and for write behind cache writes.
- Service – contains web service implementations.

From a dependency perspective, the following figure shows the Dependency State Matrix (DSM) [3] for the system:

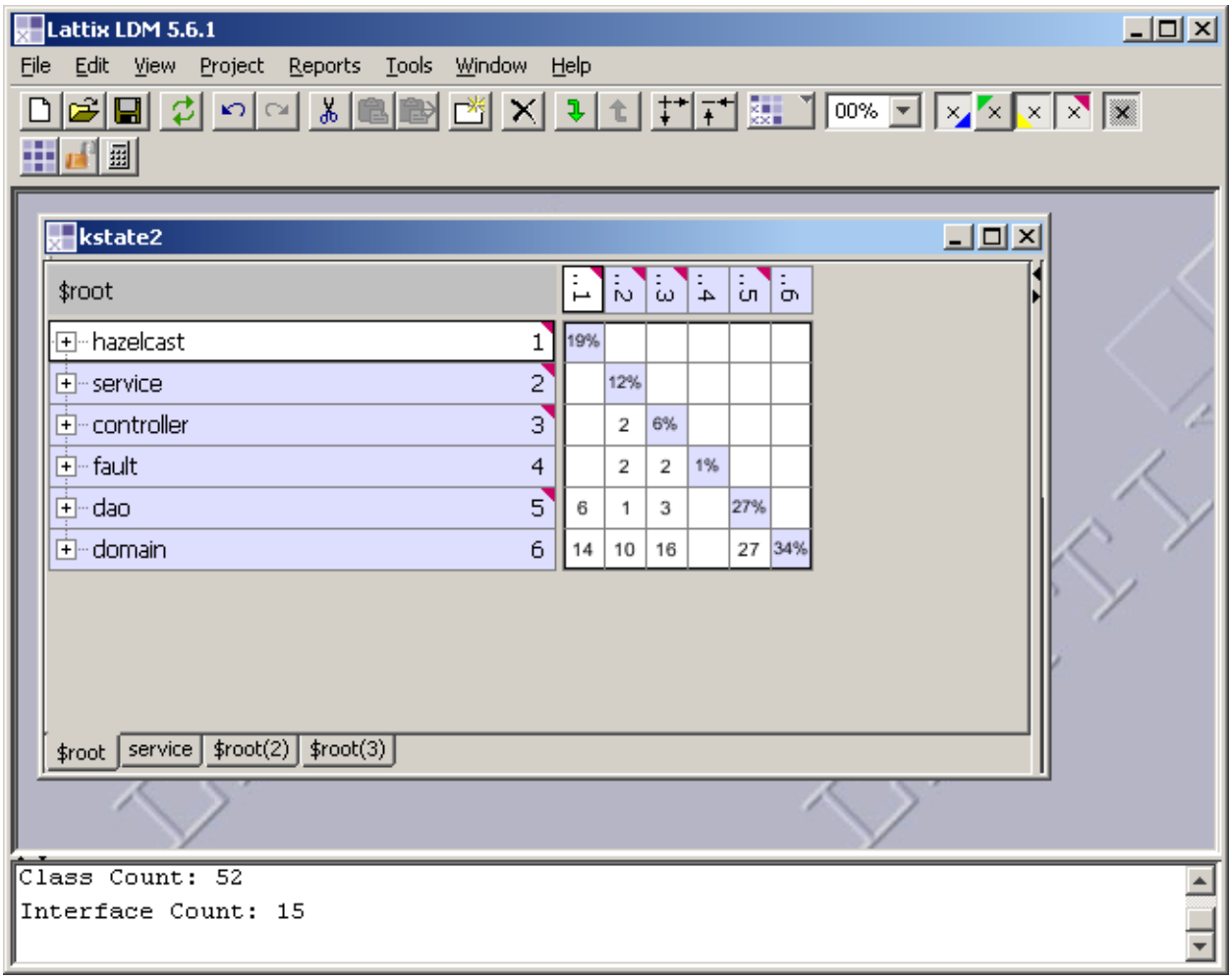


Figure 2 System Dependency State Matrix

The DSM allows a quick visual inspection that shows layered dependencies between the packages, with no circular references or package tangles. The conceptual architecture in terms of dependencies can be generated based on the DSM:

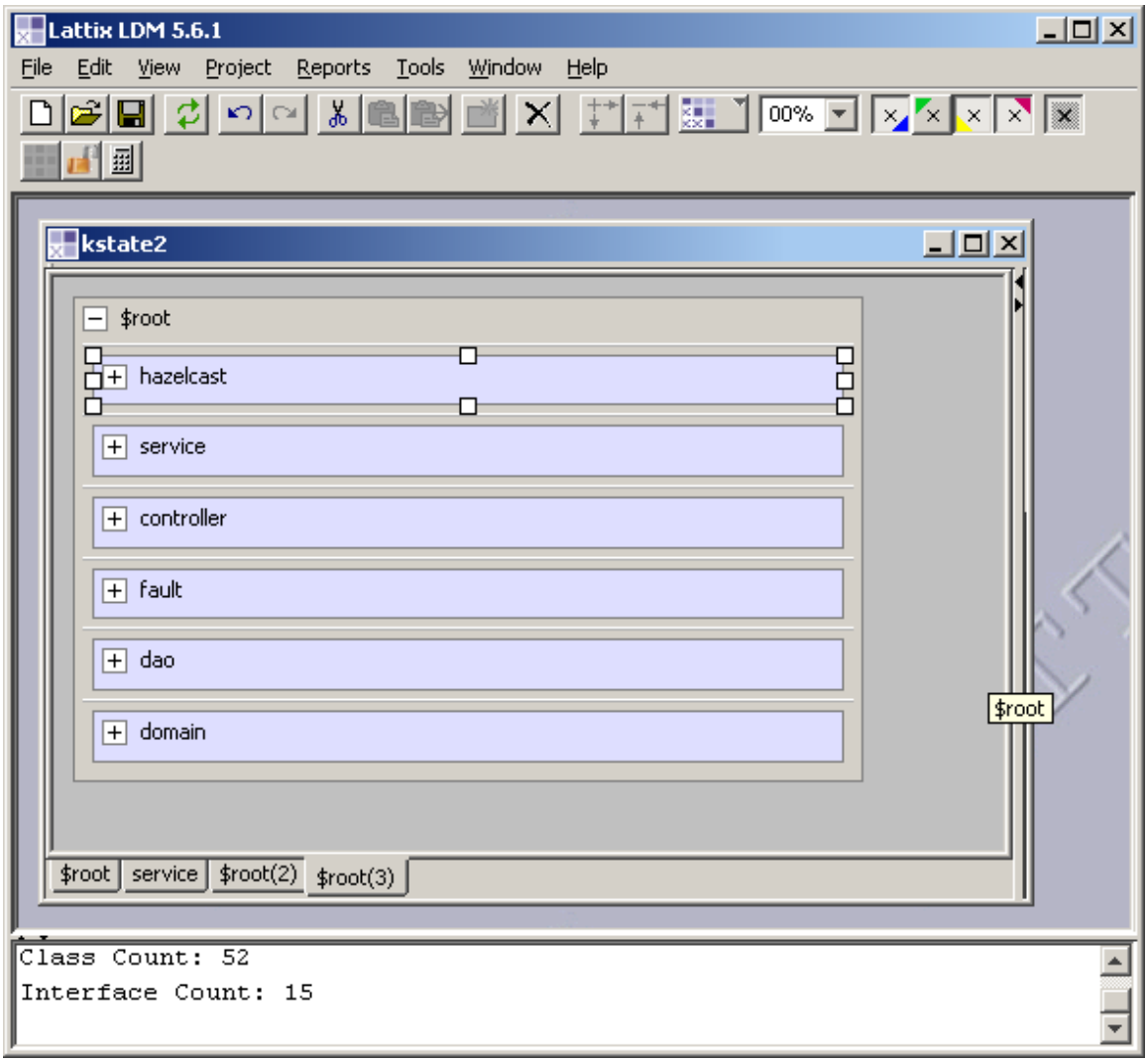


Figure 3 System conceptual architecture diagram (based on the DSM)

At the top of the hierarchy is the hazelcast package, on which no other package depends. Also at the top is the service package; no other packages in the system depend on it. Next come the controller and fault packages, then the dao and domain packages, with each package traversing from the top to the bottom having more dependencies from the above packages.

Service Package

The following class diagram shows the classes and interfaces in the service package:

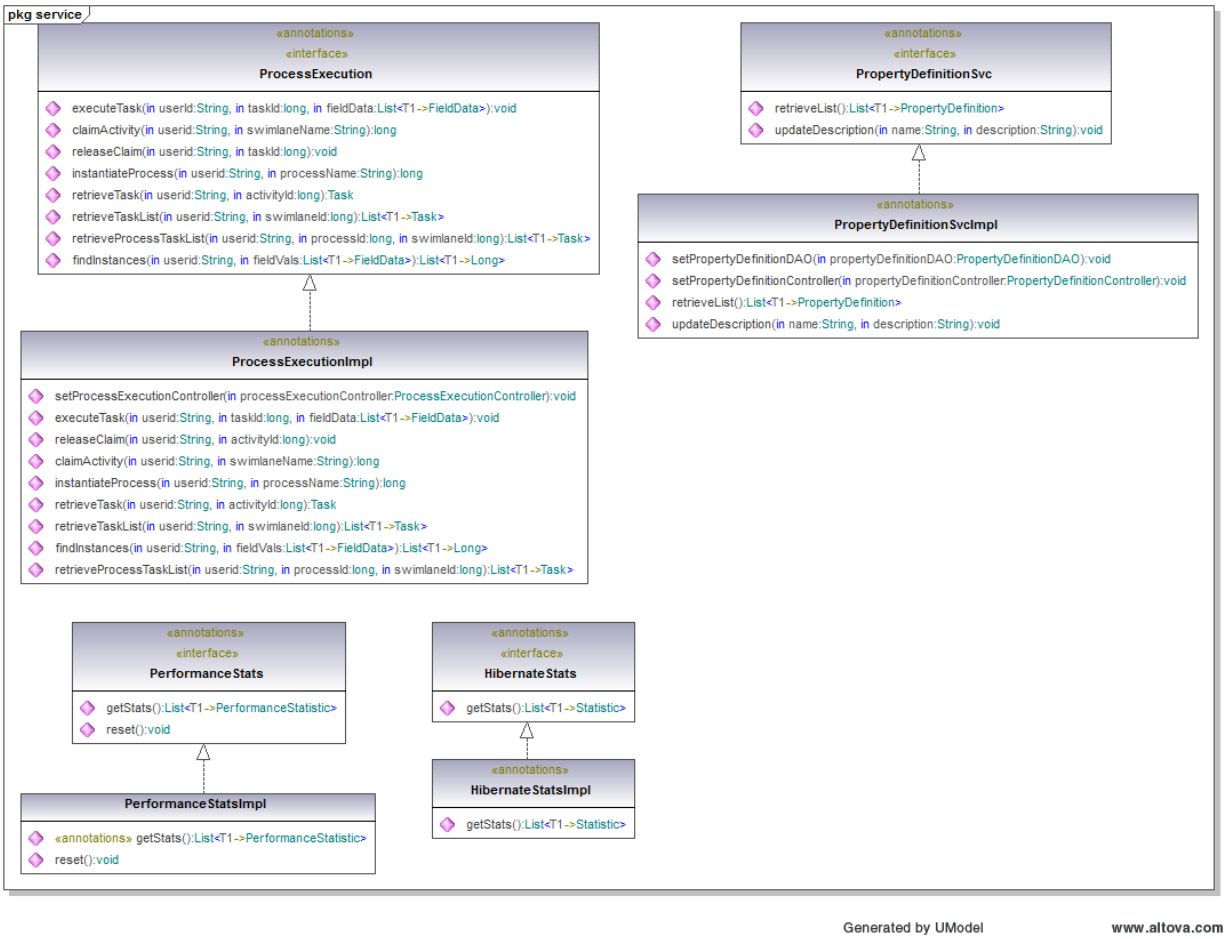


Figure 4 Service package class diagram

There are two sets of services related to the problem domain: definition services as defined by the PropertyDefinitionSvc interface, and runtime services as defined by the ProcessExecution interface. There are also services for capturing Hibernate caching statistics and for performance statistics captured by the application.

The Apache CXF framework provides the web service implementation for the system. CXF supports JAX-WS annotations, which means a service interface can be defined as a Java interface, and a Java class denoted as an implementation of an interface.

As an example, consider the HibernateStats service. The interface definition is as follows:

```
@WebService(name="hibernateStats",
targetNamespace="http://people.cis.ksu.edu/dougs")
public interface HibernateStats {
    List<Statistic> getStats();
}
```

The annotation in combination with Java reflection provides all the information needed to produce the WSDL definition for the service.

The implementation of the service looks like this:

```
@WebService(endpointInterface="service.HibernateStats")
public final class HibernateStatsImpl extends HibernateDaoSupport
     implements HibernateStats {
etc...
```

The annotation on the class informs the web service runtime infrastructure that an implementation of the service as defined in the interface is available.

CXF provides hooks to enable a servlet container to provide the HTTP binding for the service, in such a way the the Spring Framework can be used to manage the beans implementing the service using standard dependency mechanisms provided by the Spring Framework.

Enabling CXF servlet container integration is as simple as including the following in the web.xml:

```
<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-
class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
  </servlet-mapping>
```

Spring integration is as easy as including the provided CXF class path resources in a spring configuration file, followed by bean definitions and service definitions, as shown in the following sample:

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jaxws="http://cxf.apache.org/jaxws"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schemas/jaxws.xsd">

  <import resource="classpath:META-INF/cxf/cxf.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml"/>
  <import resource="classpath:META-INF/cxf/cxf-servlet.xml"/>


<bean id="hibernateStatsBean" class="service.HibernateStatsImpl">
<property name="sessionFactory" ref="mySessionFactory" />
</bean>


  <jaxws:endpoint id="hibernateStats"
   implementor="#hibernateStatsBean"
   address="/HibernateStats"/>

</beans>
```

For full details, refer to the configuration information in the source code repository under src/main/webapp [2].

Property Definition Service

Due to the need to constrain scope for this project, only process field definitions were service enabled. The process definition service exposes two operations: a list service to retrieve property definitions, and service to update the description associated with a property. Property definitions are cached, thus the retrieve list and update operations provide a way to exercise caching and flushing via a service interface.

Refer to the controller package section of this document for details on the update operation. The list operation is unremarkable; it merely retrieve data via a query method on the Property Definition DAO.

Process Execution Service

The process execution service exposes operations related to instantiating and retrieving process instances, claiming, releasing, and executing tasks, and searching for items. Implementation details are provided in the controller package section.

Hibernate Statistics Service

The Hibernate statistics service provides a service front end to allow access to Hibernate second level caching statistics. This service was heavily leverage to access the performance of the first architecture, but is less relevant since the adoption of the IMDG. The IMDG's data access can be cache enabled with Hibernate, and thus service can be used to access cache statistics when that configuration is enabled.

The service itself is a service enabled wrapped of the statistics API available in Hibernate, used to retrieve the statistics and return them in the service response.

Performance Statistics Service

The performance statistics service provides methods to return performance statistics capture in the application, and to reset the statistics. The Jamon framework is used to instrument the application to capture performance statistics.

Controller Package

The controller package contains the implementations of services that require orchestrating the interactions of different objects to realize services.

ProcessExecutionController

The following class diagram shows the controller classes and iterfaces – one for process execution, one for definitions used in process execution.

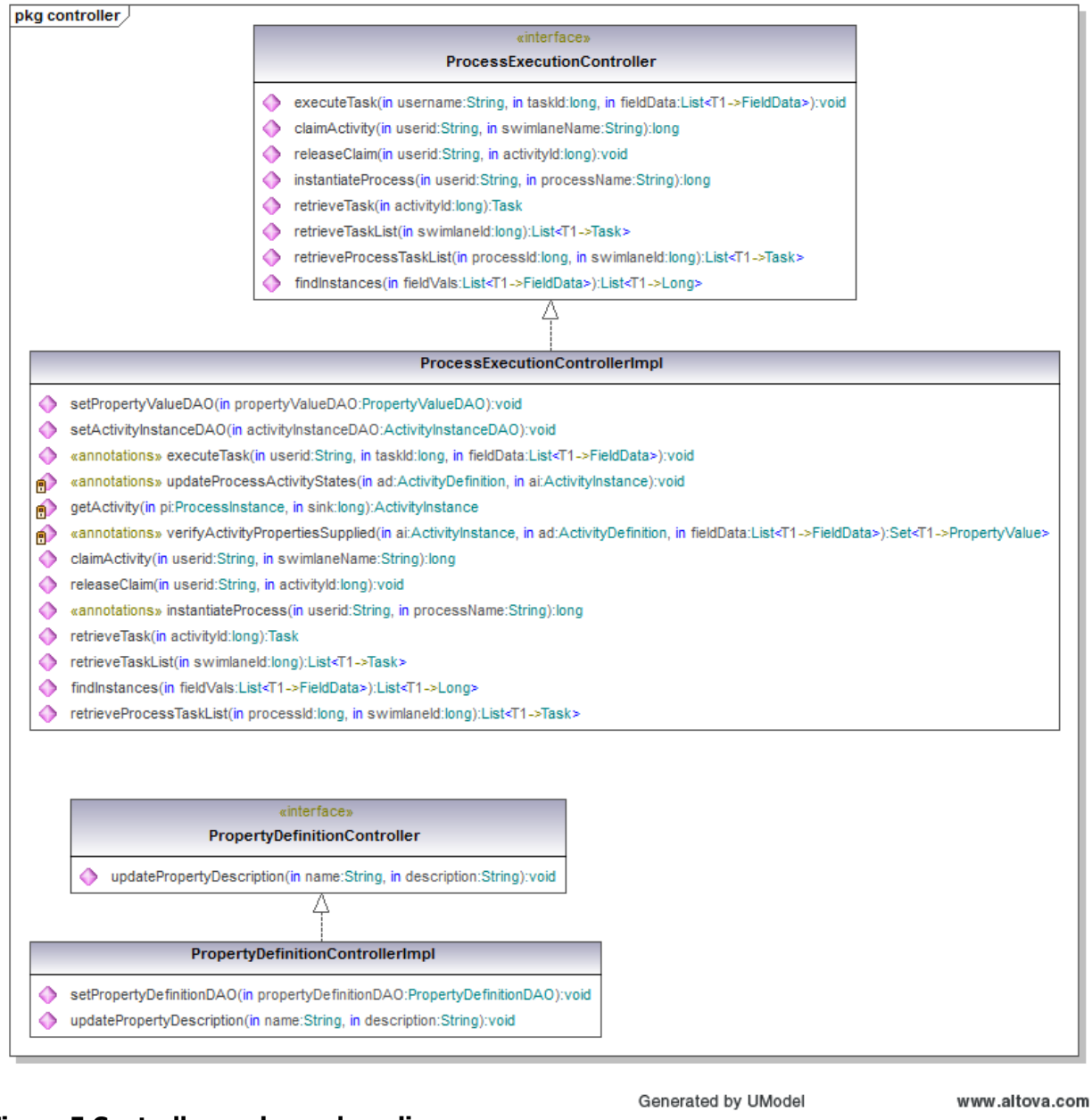


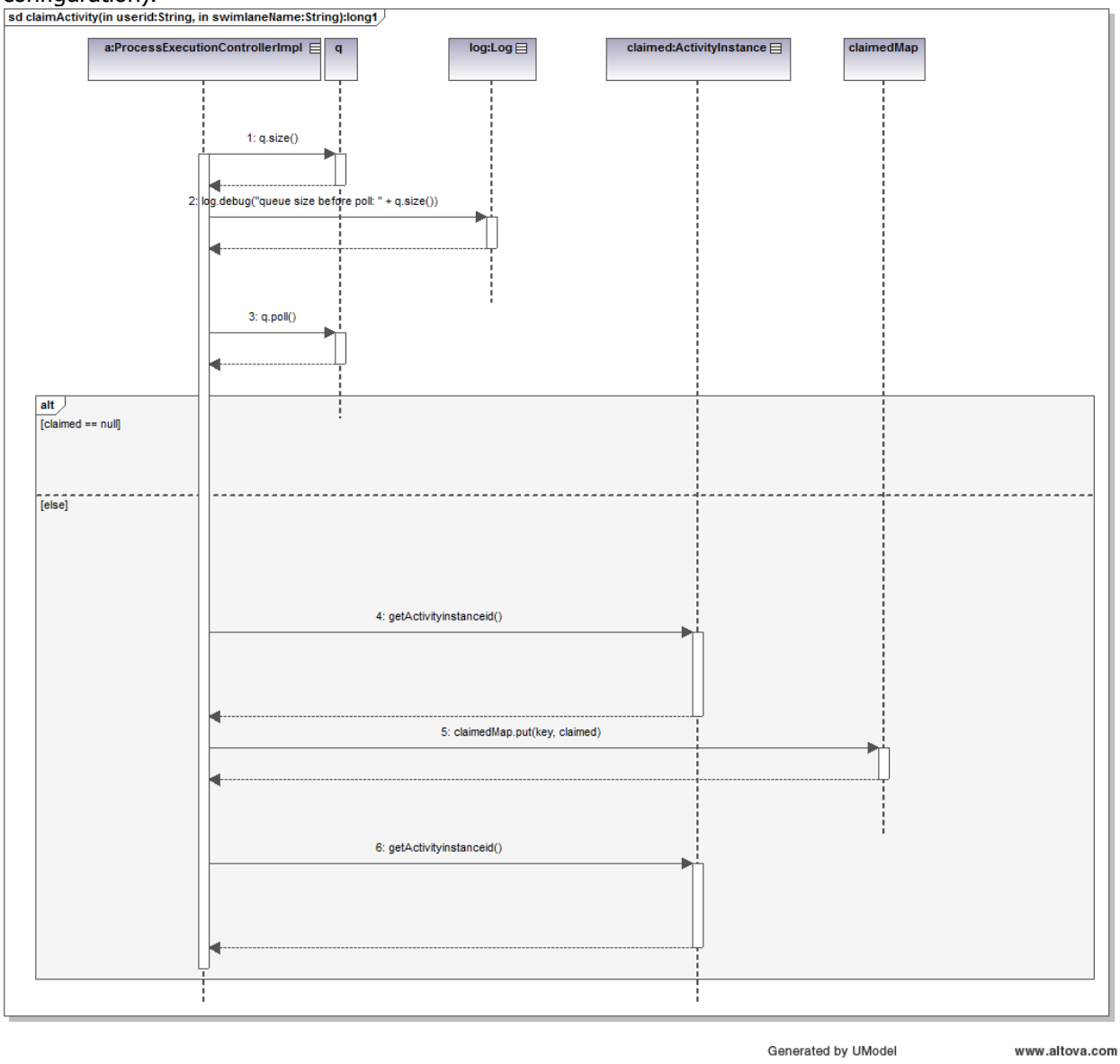
Figure 5 Controller package class diagram

The following sequence diagrams illustrate some of the more interesting methods.

Method: claimActivity

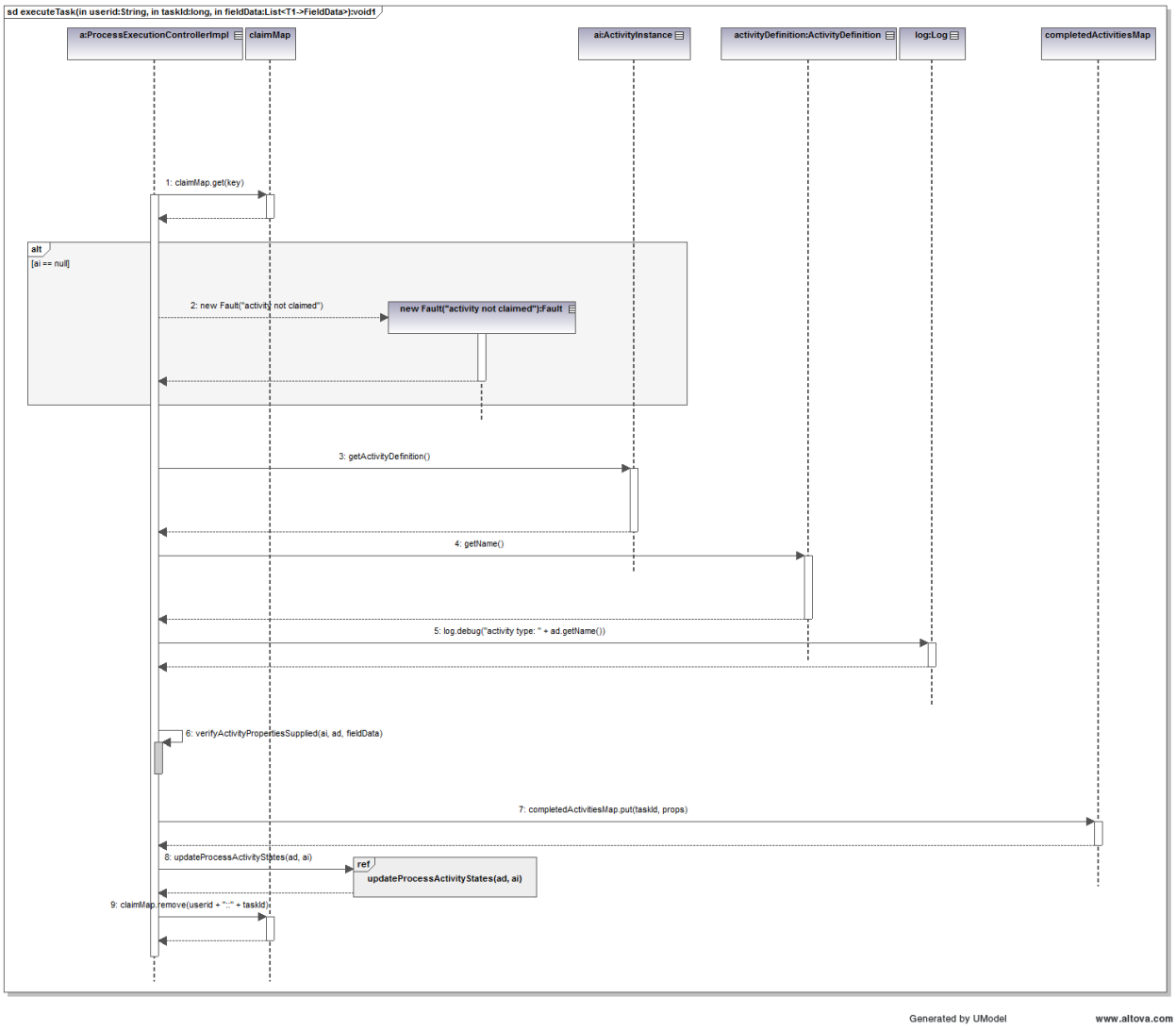
When claiming an activity, a Hazelcast distributed map is used to maintain claims. Claims are checked against the map, and written to the map if the no one else has the activity claimed.

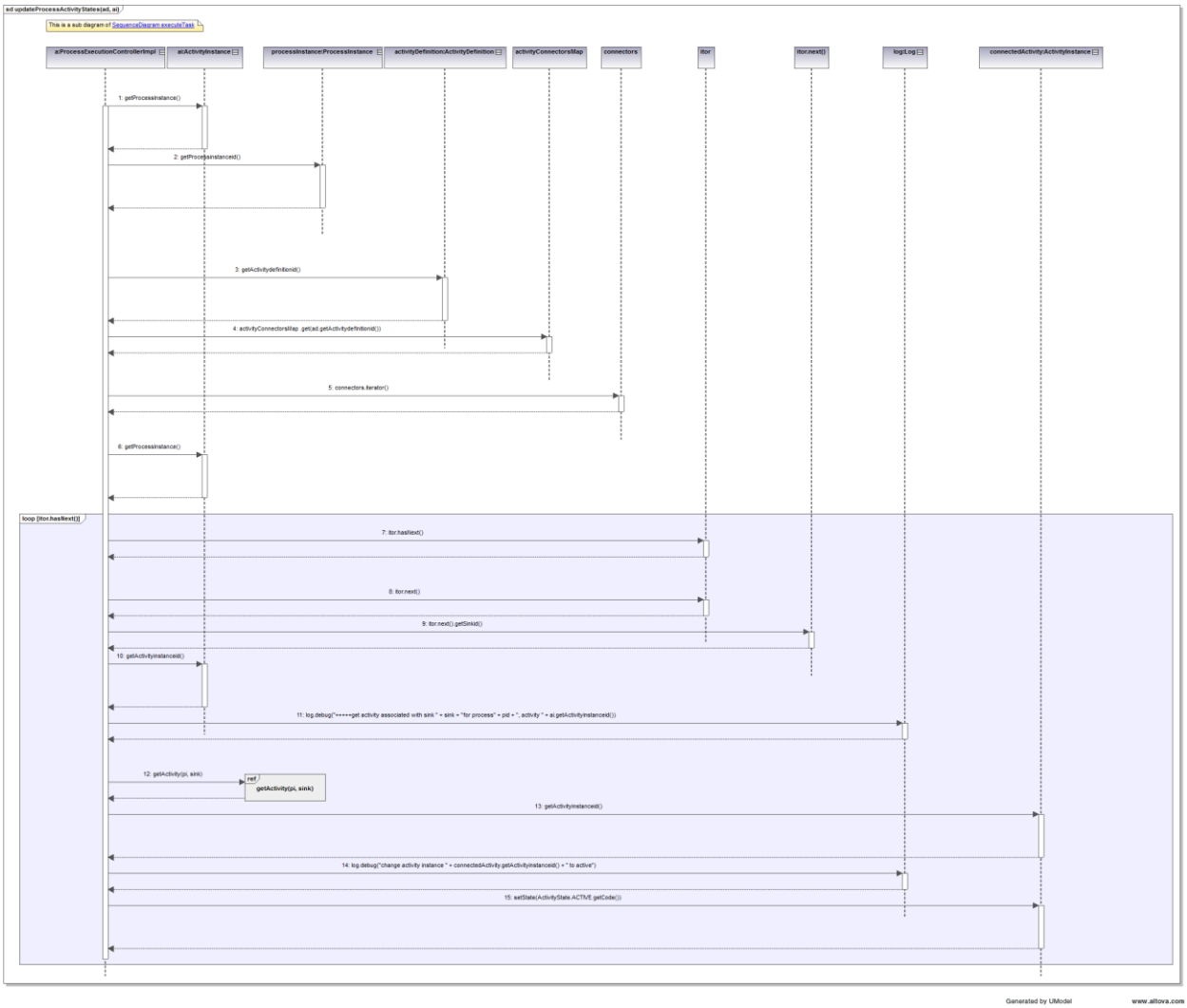
Claims are made durable (beyond replicated copies in the IMDG via the cache write behind configuration).



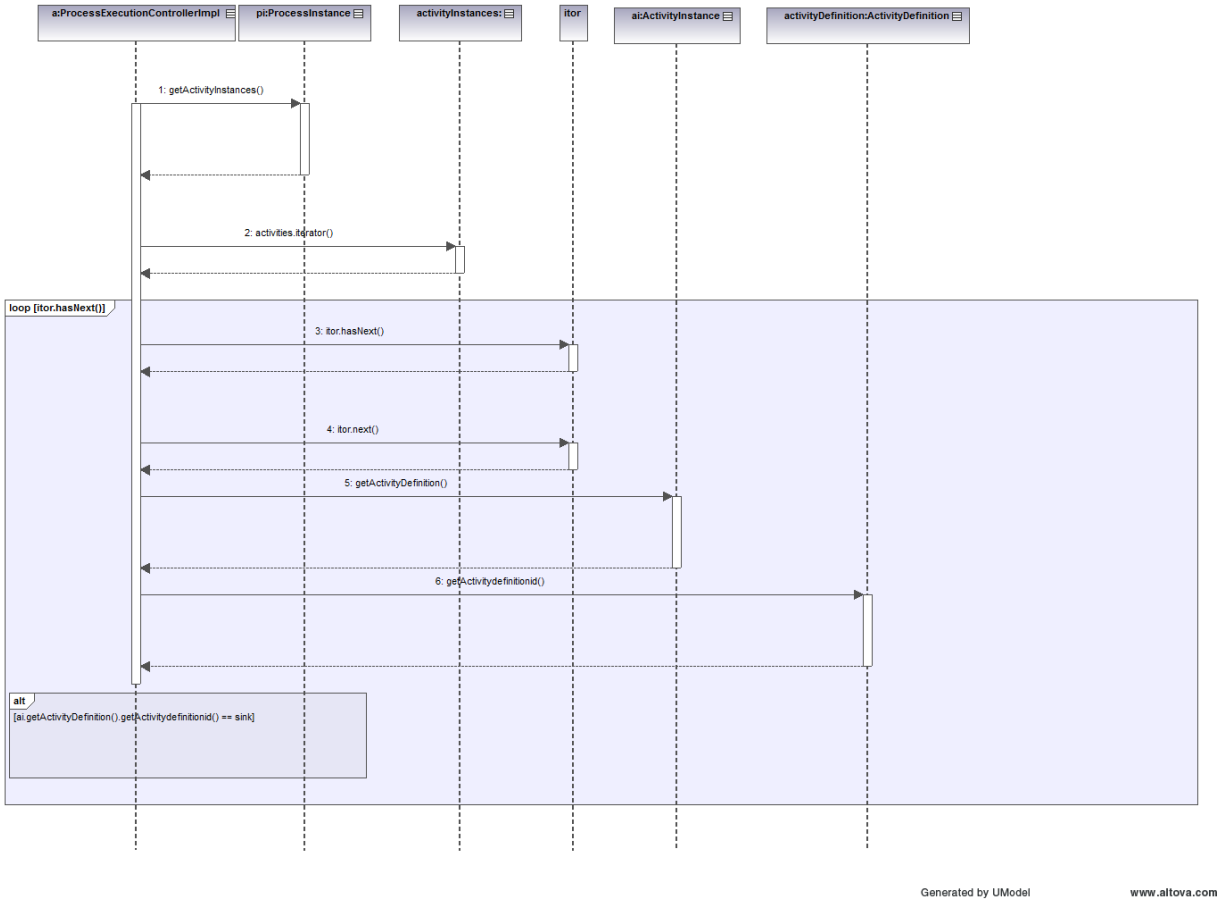
Method: executeTask

Executing tasks involves updating task state based on the data passed in, checking the supplied data against the required data as specified by the task definition metadata. State is written to the IMDG, which is eventually persisted via IMDG write behind integration. Of interest here (detailed in the hazelcast package section) is the queuing of activiated activities: they are made available in the grid's distributed queues during the cache write behind of the updated state. This means there is a period of time where the newly activated tasks are not seen via queries. Eventually, the global view of the process is made consistent when the state in the database is updated and newly activiated activites are available in the IMDG queues.

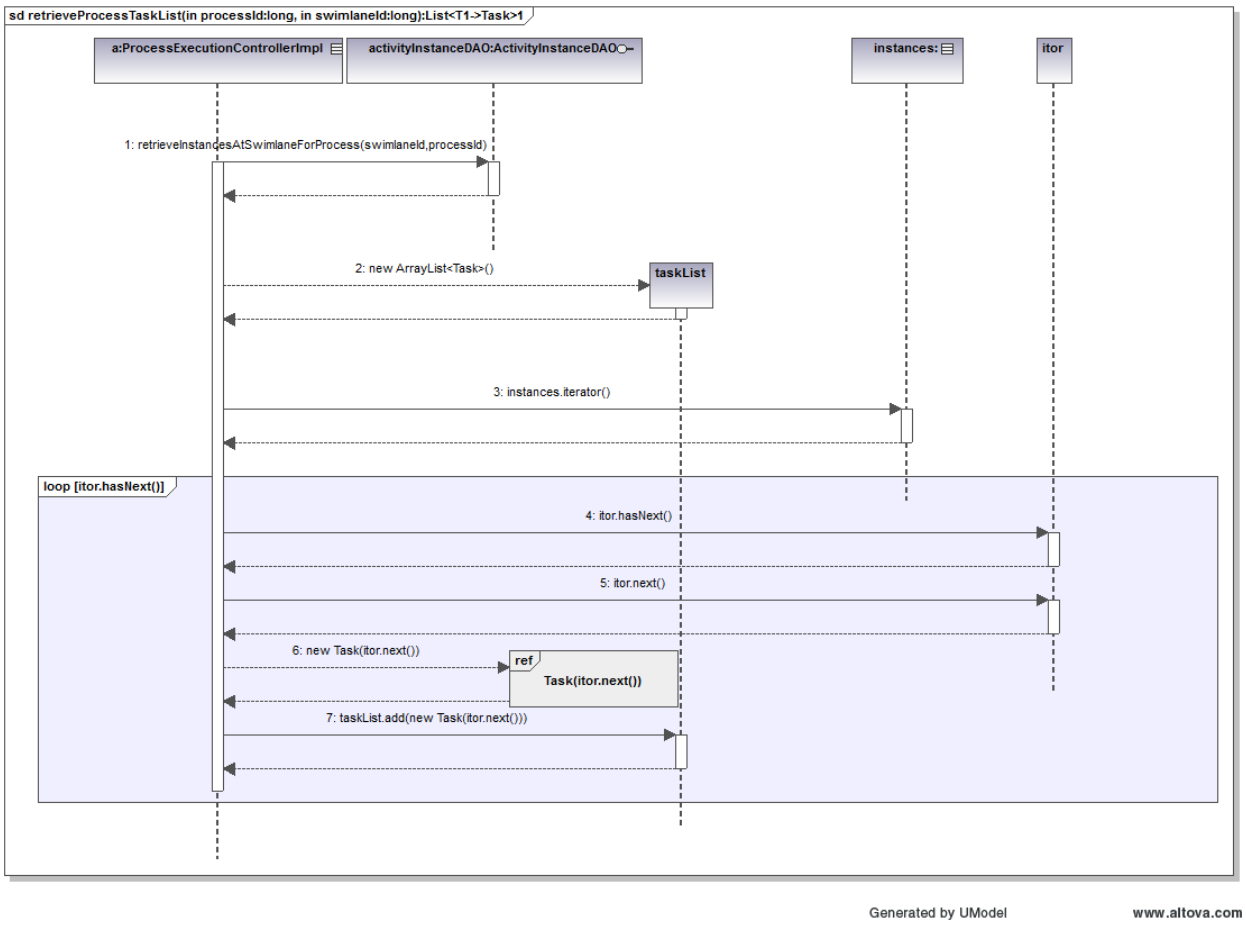

sd executeTask(in userid:String, in taskId:long, in fieldData:List<T1->FieldData>):void1
a:ProcessExecutionControllerImpl
claimMap
ai:ActivityInstance
activityDefinition:ActivityDefinition
log:Log
completedActivitiesMap
1: claimMap.get(key)
alt
[ai == null]
2: new Fault("activity not claimed")
new Fault("activity not claimed"):Fault
3: getActivityDefinition()
4: getName()
5: log.debug("activity type: " + ad.getName())
6: verifyActivityPropertiesSupplied(ai, ad, fieldData)
7: completedActivitiesMap.put(taskId, props)
8: updateProcessActivityStates(ad, ai)
ref
updateProcessActivityStates(ad, ai)
9: claimMap.remove(userid + ":" + taskId)
Generated by UModel
www.altova.com



Method: getActivityInstances

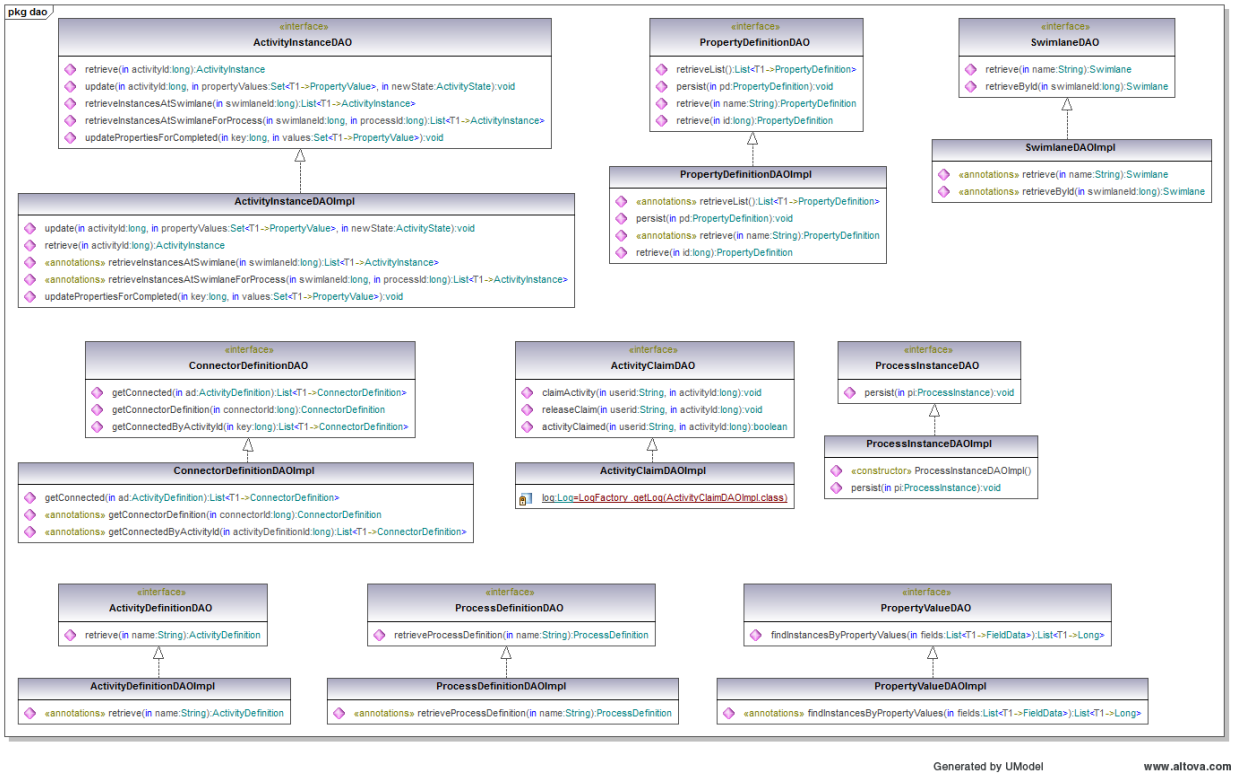


Method: retrieveProcessTaskList

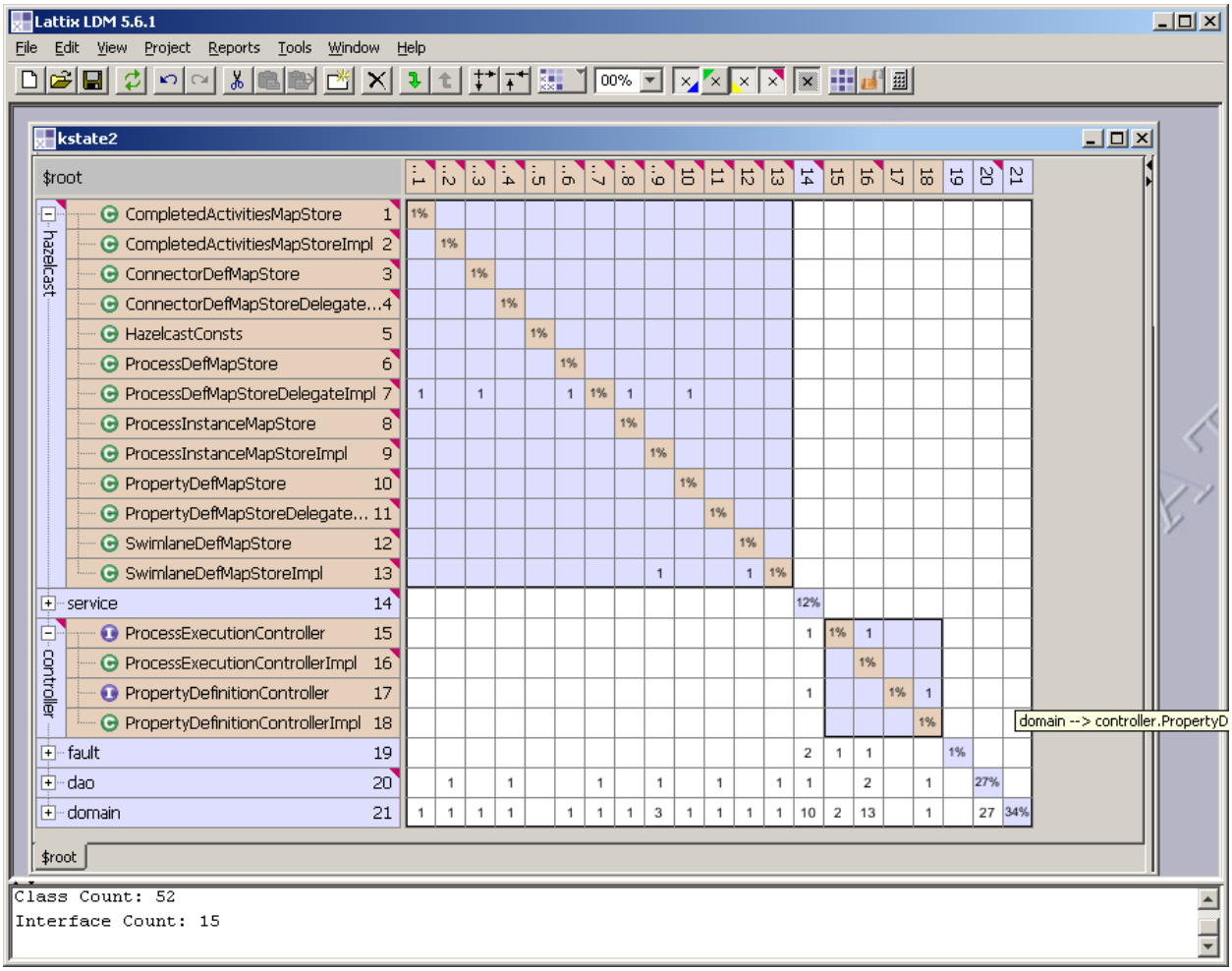


DAO Package

The DAO package contains data access objects that encapsulate access to the database. The use of the Hibernate object/relational mapping API is confined to this package, including any generated exceptions which are normalized by Spring to the Spring exception hierarchy.



The DAO classes are mostly accessed from the hazelcast package via integration from the backing store. There is some direct use of the DAOs from the services or the controllers; direct use is done for some types of queries, with write performed via Hazelcast integration.



Domain Package

The domain package defines entities corresponding to the classes outlined in the formal specification. Note this domain model is what is known as an anemic domain model; the behavior is contained in other entities such as the controllers and service implementations. Refactoring the system to leverage an architecture such as Command Query Responsibility Segregation (CQRS) or apply the patterns of Domain Driven Design (DDD) would eliminate the anemic domain model.

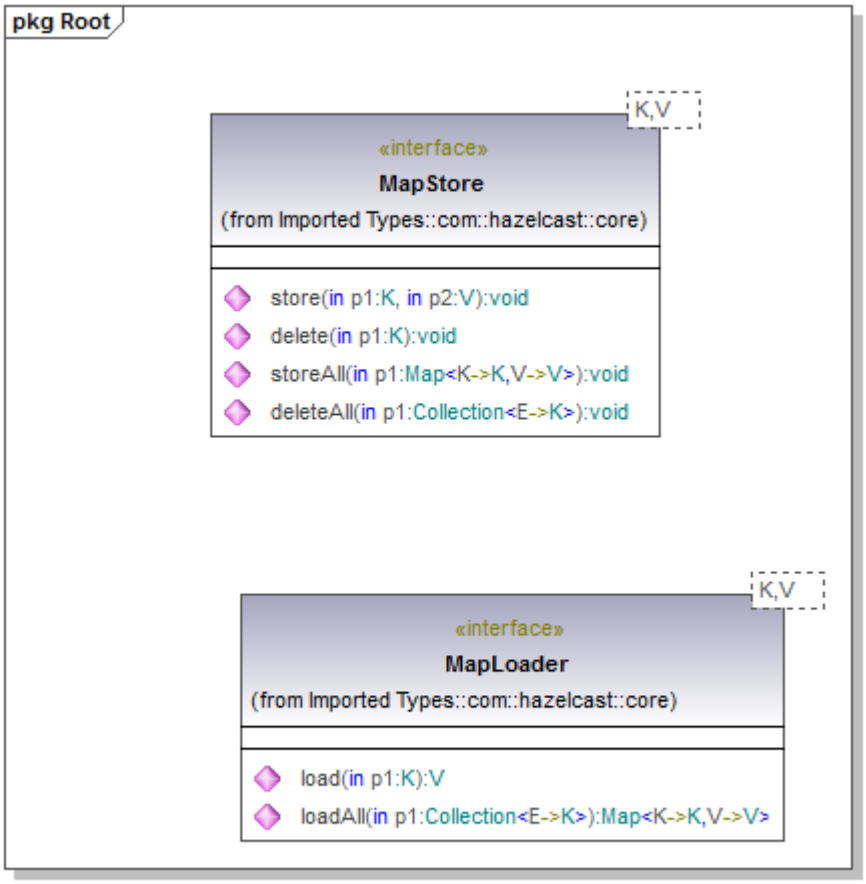
Refer to the Software Architecture description document for details [1].

Fault Package

This package contains a single class (Fault) which is used to convey a fault message and code back to web service clients when an exception is generated in the service implementation.

Hazelcast Package

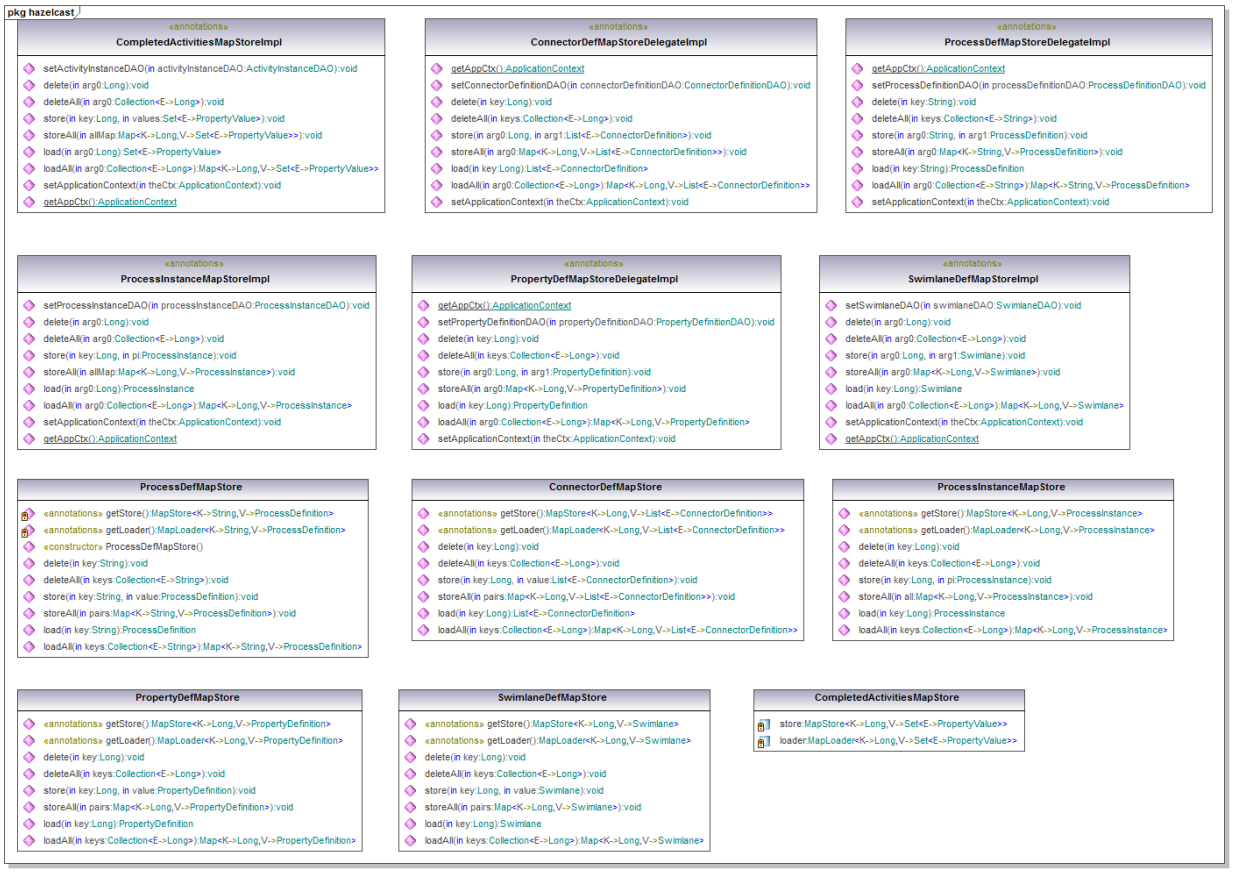
The hazelcast package contains implementations of the Hazelcast MapLoader and MapStore interfaces. Hazelcast defines the iterfaces as follows:



MapLoader and MapStore implementations are configured using Hazelcast's configuration mechanism:

```
<map name="processInstanceMap">
<!--
Number of backups. If 1 is set as the backup-count for example, then
all entries of the map will be copied to another JVM for fail-safety.
Valid numbers are 0 (no backup), 1, 2, 3.
-->
<backup-count>2</backup-count>
<!--
Valid values are: NONE (no eviction), LRU (Least Recently Used), LFU
(Least Frequently Used). NONE is the default.
-->
<eviction-policy>NONE</eviction-policy>
<!--
Maximum size of the map. When max size is reached, map is evicted
based on the policy defined. Any integer between 0 and
Integer.MAX_VALUE. 0 means Integer.MAX_VALUE. Default is 0.
```

```
-->
<max-size>0</max-size>
<!--
When max. size is reached, specified percentage of the map will be
evicted. Any integer between 0 and 100. If 25 is set for example, 25%
of the entries will get evicted.
-->
<eviction-percentage>25</eviction-percentage>
<map-store enabled="true">
<!--
Name of the class implementing MapLoader and/or MapStore. The class
should implement at least of these interfaces and contain
no-argument constructor.
-->
<class-name>hazelcast.ProcessInstanceMapStore</class-name>
<!--
Number of seconds to delay to call the MapStore.store(key, value).
If the value is zero then it is write-through so MapStore.store(key,
value) will be called as soon as the entry is updated. Otherwise it
is write-behind so updates will be stored after write-delay-seconds
value by calling Hazelcast.storeAll(map). Default value is 0.
-->
<!-- Set to 0 for unit tests -->
<write-delay-seconds>5</write-delay-seconds>
</map-store>
</map>
```



Integration of the MapLoader and MapStore implementations in the system has an interesting twist. The classes that I wrote in the construction of the system are all Spring managed, with the Spring Framework used to control bean life cycle, inject dependencies, manage transactions and database connections, and so on. Hazelcast, however, directly instantiates classes that are configured as loader and store implementations.

To enable the integration of Hazelcast loader and store implementations with my Spring managed beans, I implemented the Hazelcast classes as shims, which lookup the actual implementations using the Spring API, then delegate calls to the Spring managed implementation, taking advantage of Springs database session management, transactions, etc.

Example – shim class loader and store lookup methods:

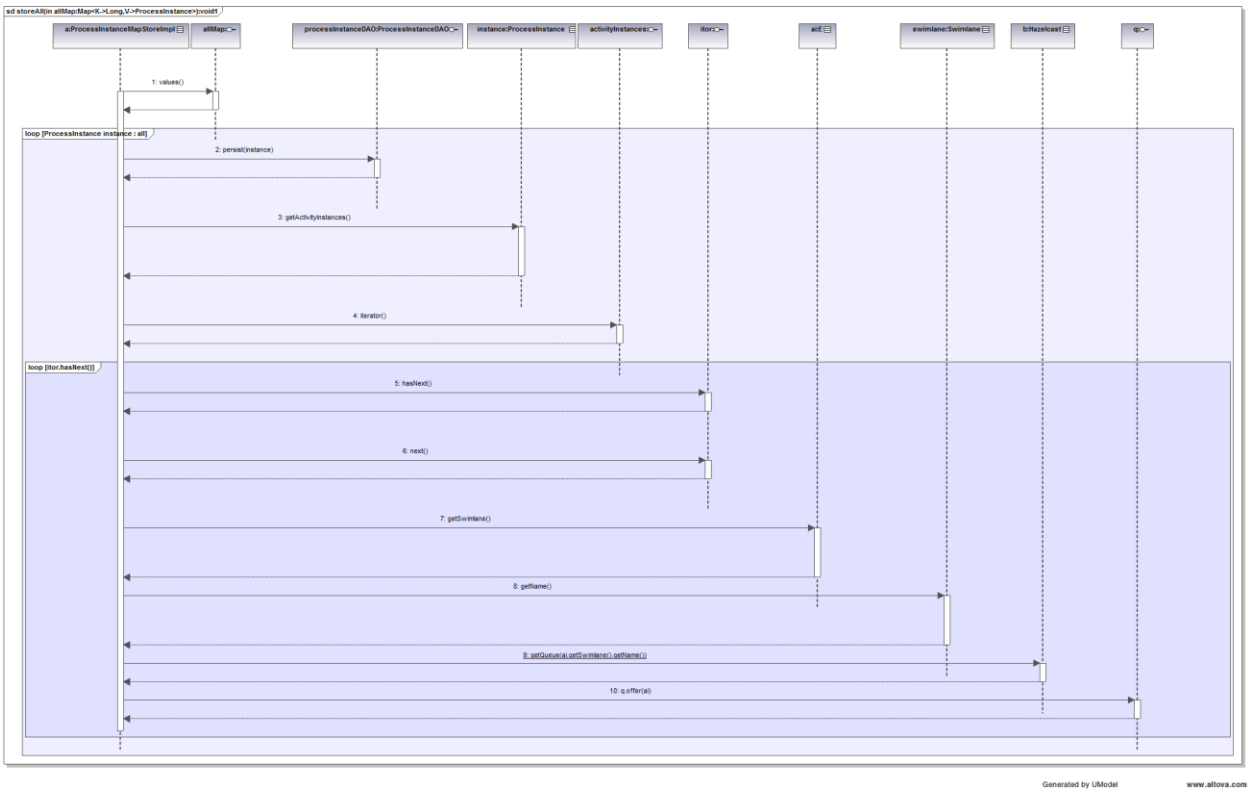
```
public final class ProcessInstanceMapStore implements MapStore<Long,
ProcessInstance>, MapLoader<Long, ProcessInstance> {

//ProcessDefMapStoreDelegate delegate;
private MapStore<Long, ProcessInstance> store;
private MapLoader<Long, ProcessInstance> loader;



@SuppressWarnings("unchecked")
public MapStore<Long, ProcessInstance> getStore() {
if(store == null) {
ApplicationContext ctx = ProcessDefMapStoreDelegateImpl.getAppCtx();
store = (MapStore<Long,
ProcessInstance>)ctx.getBean("processInstanceMapDelegate");
}
return store;
}


@SuppressWarnings("unchecked")
public MapLoader<Long, ProcessInstance> getLoader() {
if(loader == null) {
ApplicationContext ctx = ProcessDefMapStoreDelegateImpl.getAppCtx();
loader = (MapLoader<Long,
ProcessInstance>)ctx.getBean("processInstanceMapDelegate");
}
return loader;
}
```

Example – used of delegation in the shim class method:

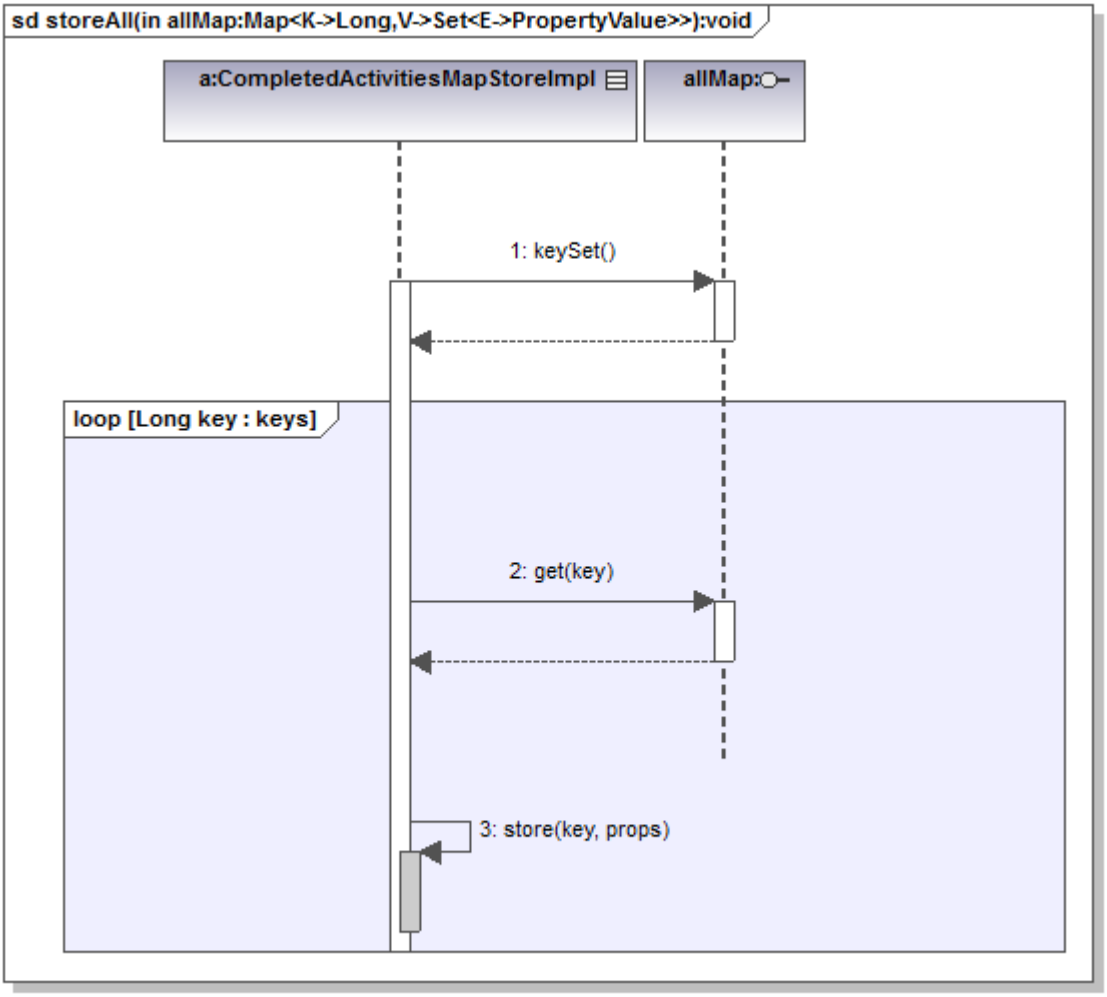
```
public Map<Long, ProcessInstance> loadAll(Collection<Long> keys) {
return getLoader().loadAll(keys);
}
```

The following sequence presented in this section of the document show a couple of the more interesting scenarios.

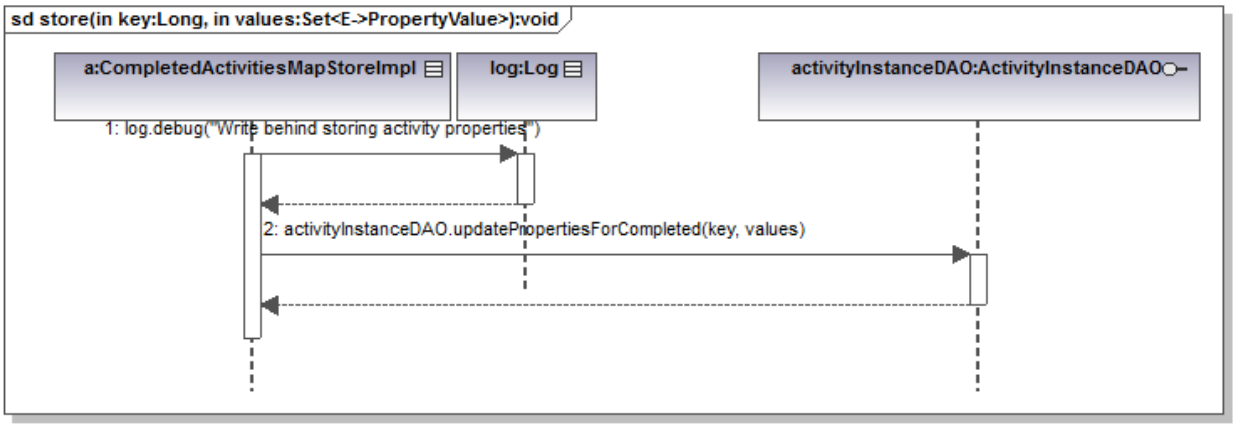
Method: ProcessInstanceMapStoreImpl.storeAll



Method: CompletedActivitiesMapStoreImpl.storeAll



Generated by UModel www.altova.com



Generated by UModel www.altova.com