Elmo User Guide

James Leigh

Elmo User Guide

by James Leigh

This document describes Elmo 1.4 and its use with *Sesame* 2.2.3. Elmo is a *JavaBean* pool implementation for the *Sesame RDF repository*, providing static *JavaBean* interfaces to RDF resources. Specifically, Elmo is a *subject-oriented RDF entity* manager that allows *JavaBean*s to be cast to different roles providing a unique context specific view of the subject.

As systems become more integrated and provide diverse roles to a broader market, traditional *object-oriented* and *component-oriented* systems can become bloated with complex interfaces and behaviours that increase expenses for software maintenance. Integrated domain driven designs often require context-specific views of shared multi-dimensional resources. These resources are a challenge to model through polymorphism alone and require a higher level of concern management.

As a *subject-oriented* system, Elmo provides a way to group common *behaviour*s and separate roles into unique interfaces and reusable classes. Domain models built with Elmo are simpler and are better able to express their design concept. This leads to lower maintenance costs and more flexible software.

Table of Contents

- ... 1
  - 1. Introduction ... 1
  - 2. Distribution ... 1
    - 2.1. License ... 1
    - 2.2. Installation ... 2
  - 3. Application Programming Interface ... 3
    - 3.1. ElmoModule ... 4
    - 3.2. SesameManagerFactory ... 4
    - 3.3. ElmoManager ... 4
    - 3.4. ElmoQuery ... 7
    - 3.5. Concepts ... 7
    - 3.6. Role Annotations ... 9
    - 3.7. Registering Concepts & Behaviours ... 10
    - 3.8. Behaviours ... 11
    - 3.9. Interceptors ... 12
    - 3.10. Factory Classes ... 13
    - 3.11. Datatypes ... 13
    - 3.12. RDF Collections ... 14
    - 3.13. DynaBeans ... 15
  - 4. Elmo Tools ... 15
    - 4.1. Compiling and Decompiling and Ontologies ... 15
      - 4.1.1. Method Declarations ... 16
      - 4.1.2. Compiling Behaviours ... 17
    - 4.2. Elmo Scutter ... 17
    - 4.3. Elmo Smusher ... 19
  - 5. Common Design Patterns ... 20
    - 5.1. Adaptor ... 20
    - 5.2. Aspects ... 20
    - 5.3. Chain of Responsibility ... 20
    - 5.4. Cohesion ... 21
    - 5.5. Context Specific Data ... 21
    - 5.6. Data Localization ... 23
    - 5.7. Hyperslices ... 23
    - 5.8. Dependency Injection ... 24
    - 5.9. Observer ... 26
    - 5.10. Persistence Ignorance ... 26
    - 5.11. Rules ... 27
    - 5.12. Serialization ... 29
    - 5.13. Strategy ... 29
  - 6. Transactions and Concurrency ... 30
    - 6.1. Transaction Isolation ... 30
    - 6.2. Undo Support ... 30
  - 7. Performance and Optimizations ... 31
    - 7.1. Serializing Literal Values ... 31
    - 7.2. AugurRepository and ReadAheadRepository ... 31

7.3. ResourceManager 32
7.4. Class Loading 32
8. Library Integration 33
8.1. Sesame 33
8.2. javax.persistence (JPA) 33
8.3. Tapestry 34
9. Examples 34
9.1. FOAF Example 34
9.2. GEDCOM Example 34
10. Known Limitations 35
10.1. Truth Maintenance for @equivalent and @inverseOf 35
10.2. @disjointWith on Concepts 35
10.3. HTTP Repository 35
11. Frequently Asked Questions 35
11.1. Do I need a database to use Elmo? 35
11.2. Are there any alternatives to creating my own concepts? 35
11.3. How can I remove a concept from a entity? 36
11.4. Why aren't Entities serializable? 36
11.5. I already have existing JavaBeans; can they be moved into an Elmo entity pool? 36
11.6. How can I have two implementations of the same method? 36
11.7. What happens when two concepts having the same property name are applied to the same subject? 36
11.8. How can I implement cross cutting concerns in Elmo? 36
11.9. How do I trigger events before and after a property change? 37
11.10. How should Entities be managed in a J2EE environment? 37
11.11. How can I implement an audit log with Elmo? 37
11.12. How do I setup a security domain over sensitive data? 37
12. Glossary 37
13. References 43

List of Figures

1. [1] 4
2. [2] 5
3. [3] 24

1. Introduction

Elmo is a role based Java persistent Bean pool. It provides a simple API to access *ontology* oriented data inside a *Sesame repository* through *concept*s and *behaviour*s that may have been designed and developed independently.

Elmo provides a static typed interface to RDF properties within an RDF store. For example, the following code accesses FOAF properties from the RDF/XML file "foaf.rdf" and prints them to standard out.

```
ElmoModule module = new ElmoModule();
SesameManagerFactory factory = new SesameManagerFactory(module);
SesameManager manager = factory.createElmoManager();
File file = new File("foaf.rdf");
manager.getConnection().add(file, "", RDFFormat.RDFXML);
for (Person person : manager.findAll(Person.class)) {
    System.out.print("Name: ");
    System.out.println(person.getFoafNames());
}
```

With Elmo's role based interface and component oriented design, software becomes more flexible and allows for novel opportunities for developing and modularizing domain driven designs. The domain model is simplified into independent concerns that are mixed together for multi-dimensional, inter-operating, or integrated applications.

Elmo is a *subject-oriented* programming library that addresses common problems in traditional *object-oriented* designs. With Elmo, application design and development can:

- Create extensions to software without modifying original source;
- Customize and integrate systems with reusable components;
- Facilitate multiple team system development by dynamically integrating independent *domain-model*s;
- Permit decentralized design and development of *concept*s and *behaviour*s;
- Simplify code by capturing design patterns into reusable *behaviour*s;
- Maintain a 1:1 mapping between design concept and implementation.

2. Distribution

2.1. License

OpenRDF Elmo is licensed under the BSD license and as such is free to use and modify as stated below.

Copyright (c) 2005-2008, James Leigh All rights reserved, unless otherwise specified.

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.

- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
- Neither the name of openRDF.org nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

2.2. Installation

Elmo stores all persistent data in a *Sesame repository* store. *Sesame* supports interchangeable *repository* stores. The one discussed in this user guide, MemoryStore, is a self-managed library and does not use a database server. The MemoryStore stores all values in memory and can save values to disk in a configured directory, it is suitable for storing 1 million values with half a gigabyte of memory. The NativeStore is another store that can scale much higher and stores values in the local file system under a configured directory. Please consult the Sesame User Guide for more details on backing RDF stores.

To use Elmo the following jars must be included in the class-path along with their dependencies.

elmo-1.4.jar

: Includes the API, composite engine, core *behaviour*s, *Sesame* repository decorators, manager, and integration classes for Spring and Tapestry.

Elmo 1.4 depends on the following libraries:

- *Sesame* 2.2.3
- javax-persistence-api
- commons-beanutils 1.7.0
- javassist 3.7.0
- slf4j 1.5.0

Elmo also ships with the following optional jars.

elmo-{ns}.jar

: Contains Elmo *concept*s for popular ontologies; just drop into your class-path and start using the *concept*s listed in the javadocs.

elmo-codegen.jar

Self executable jar to create your own *role*s and ontologies. This jar has additional dependencies on commons-cli and optionally *groovy*, asm, and antrl if the elmo:groovy annotation is used.

elmo-scutter.jar

Contains an RDF crawler that follows rdfs:seeAlso links in RDF documents.

elmo-scutter-webapp.war

Ready-to-deploy war file (downloaded separately) containing a web interface to the RDF crawler.

elmo-smusher.jar

Self executable jar to find equivalent instances in large sets of data. This jar has additional dependencies on commons-codec and commons-httpclient.

elmo-examples.war

Ready-to-deploy war file (downloaded separately) containing three example applications and all needed dependencies for use within a J2EE server.

Elmo and *Sesame* can be downloaded from openRDF.org (http://www.openrdf.org) and are also available as maven artifacts in the maven repository http://repo.aduna-software.org/maven2/releases . Below is a extract from a maven pom.xml.

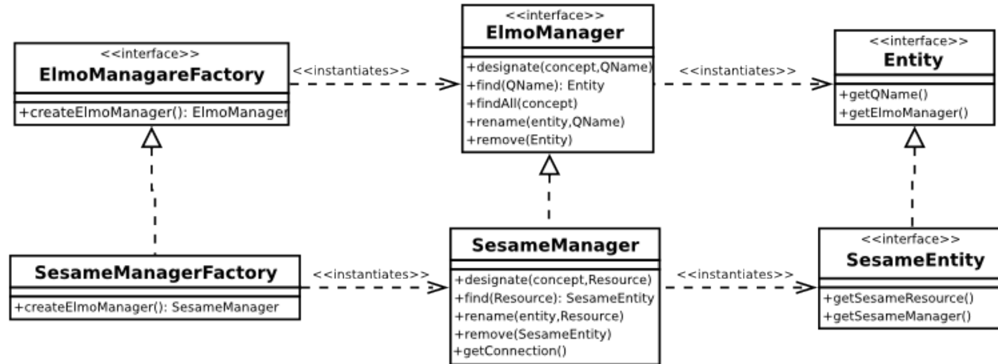
```
<dependencies>
  <dependency>
    <groupId>org.openrdf.elmo</groupId>
    <artifactId>elmo-sesame</artifactId>
    <version>1.4</version>
  </dependency>
  <dependency>
    <groupId>org.openrdf.sesame</groupId>
    <artifactId>sesame-runtime</artifactId>
    <version>2.2.3</version>
  </dependency>
</dependencies>
<repositories>
  <repository>
    <id>aduna-opensource.releases</id>
    <name>Aduna Open Source - Maven releases</name>
    <url>http://repo.aduna-software.org/maven2/releases</url>
  </repository>
</repositories>
```

3. Application Programming Interface

Elmo has three main levels of operation: ElmoManagerFactory, *ElmoManager*, and *Entity*. Normally, only one ElmoManagerFactory would exist for each domain model used with the *repository* and only one *ElmoManager* for each *repository connection*. An *Entity* would exist for each *individual* used within

a *connection*. ElmoManagerFactory and *ElmoManager* are interfaces to the classes SesameManagerFactory and SesameManager. This is shown in the following class diagram.

Figure 1. [1]



The ElmoManagerFactory can create *ElmoManager*s. When *ElmoManager*s are created, they operate on a unique *connection* to the *repository*. Through this *connection* all the properties of created entities are read and updated.

Entities are referenced by a *qualified-name* (QName). Each *qualified-name* must include a namespaceURI and a localPart that is unique within the namespace. An example of a namespaceURI is "http://www.example.com/rdf/2008/data/". This namespaceURI will be referenced as "NS" throughout the examples in this document.

3.1. ElmoModule

An ElmoModule can be created using its default constructor and must be created before instantiating an SesameManagerFactory. ElmoModule provides methods to register concepts, behaviours, datatypes, and factories, it is also used to load RDF datasets, set the operating graph (context), and scan included jars. These methods set the scope of the manager factory when instantiated.

Concept and behaviour registration is described in Section 3.7. Datatype registration is described in Section 3.11. Factories are described in Section 3.10. Dataset can be loaded into and replace a context. Each ElmoModule can include a primary context, which will contain all new statements added.

3.2. SesameManagerFactory

The SesameManagerFactory is created with a module and a repository. An existing repository can be passed to the manager factory or the factory can create (and shutdown) a repository itself. A repository can be created using a repository id to a remote sesame server (passing a URL), a local data configuration directory (passing a File dataDir), or an application id, used to locate the data directory. If no repository or repository id is provided an in-memory repository will be created with no persisted storage.

3.3. ElmoManager

Elmo's *entity* manager interface provides common methods to create, designate, persist, merge, find, rename, and remove entities. It can be created using the SesameManagerFactory class or accessed by framework specific notation. For command oriented applications, a new *ElmoManager* should be created for each command (or request). The overhead for creating a new *ElmoManager* is low and resources are shared by managers created in the same ElmoManagerFactory. The entities returned by the ElmoManager are not guarenteed to be thread safe and should only be accessed from the thread they were returned in.

The manager provides a method to create (new) and designate (possibly existing) entities by a *qualified-name* (QName). Blank or anonymous entities can be created by using the overloaded create method and will operate without a *qualified-name*. The create and designate methods are called with a *concept*, and the *ElmoManager* will create a new *entity* with the subject type of the given *concept*. Calling designate with a QName that is already used will cause the subject to take on multiple roles to satisfy each designate request. If the entity already implements the *concept* given, it is equivalent to calling find and casting the result.

The SesameManager requires all namespaceURIs can contain at most one '#' and it can only appear at the end of the namespaceURI, otherwise if they contain '/' then they must end with '/', otherwise they must end with ':'. The SesameManager also requires that the localPart must not contain any '#', '/', or ':'.

Take an example of a company where an employee is an engineer, but also does sales. In Elmo this can be modelled as shown below where a subject can implement both a SalesRep and an Engineer at the same time, by calling designate with the same QName. This technique can be used to integrate and extend *domain-model*s without modifying their source code.

```
assert SalesRep.class.isInterface();
assert Engineer.class.isInterface();
ElmoModule module = new ElmoModule();
module.addConcept(Engineer.class);
module.addConcept(SalesRep.class);
factory = new SesameManagerFactory(module);
manager = factory.createElmoManager();

QName id = new QName(NS, "E340076");
Object john = manager.designate(id, Engineer.class, SalesRep.class);
assert john instanceof SalesRep;
assert john instanceof Engineer;
```

Figure 2. [2]

Elmo supports persistence ignorance. When a property is assigned to an entity, created outside of the manager and directly implements a recorded *concept*, it will be merged into the *repository*. If the object has the method getQName, it will be merged with the QName returned. If it does not have this method or the result is null, it will be merged with a newly designated anonymous resource. Merging can also be done explicitly by using the merge method on the manager. This method will merge the object into the repository, returning the managed entity. Note that the managed entity may not extend the class if it is not a recored concept. The persist method behaves like the merge method, but persist does not return the newly merged entity.

By default the interface java.util.List is registered as an interface for rdfs:Container. This means that any object created outside of the manager, which implements java.util.List, will be merged into the *repository* as a rdfs:Container when it is assigned to a property or merged explicitly.

The manager has a find method to directly load entities by a QName. Entities can also be retrieved by type or query, through findAll and createQuery methods respectively. In addition, the manager provides a method to remove the *entity* from the pool (*RDF repository*). This method will remove all references to the *entity* and its properties. If an entity includes another entity as its property value it must be removed separately, this includes anonymous entity, such as container or list entities.

The entities do not persist any values internally and all information is stored in the underlying *repository* connection. In the example below we can see that the entities john and jonny are separate Objects, but both retrieve their values from the same *repository*. The ElmoManager only persists concept properties with @rdf annotation on the getter method or on concept fields.

```
ElmoModule module = new ElmoModule();
module.addConcept(Employee.class);
factory = new SesameManagerFactory(module);
manager = factory.createElmoManager();
QName id = new QName(NS, "E340076");
Employee john = manager.designate(id, Employee.class);
```

```
Employee jonny = manager.designate(id, Employee.class);
assert john.equals(jonny);
assert john != jonny;
john.setName("John");
assert jonny.getName().equals("John");
```

Entities are not serializable and are only valid during the life of the *ElmoManager* (usually just a transaction). The Entity interface, implemented by all entities returned from the manager, provides the getQName method to allow entities to be restored at a later time by another manager. Anonymous entities return null from their getQName method and cannot be restored using the find method. Shown below is the QName property of the Entity interface.

```
factory = new SesameManagerFactory(new ElmoModule());
manager = factory.createElmoManager();
QName id = new QName(NS, "E340076");
Entity john = manager.find(id);
assertEquals(id, john.getQName());
// the subject john has the URI of
// "http://www.example.com/rdf/2008/data/E340076"
```

3.4. ElmoQuery

The *ElmoManager* can retrieve both SPARQL and SeRQL tuple queries of entity resources through the createQuery method of the *ElmoManager*. These results are then converted into Objects or arrays of Objects in an iterator. Only one query language can be configured for an *ElmoManager* at a time using the setQueryLanguage method of the SesameManagerFactory. SPARQL is the default query language, below is an example of how to use SeRQL.

```
ElmoModule module = new ElmoModule();
module.addConcept(Employee.class);
factory = new SesameManagerFactory(module);
factory.setQueryLanguage(QueryLanguage.SERQL);
manager = factory.createElmoManager();

Employee john = manager.create(Employee.class);
john.setName("John");

String queryStr = "SELECT emp FROM "
    + " {emp} <http://www.example.com/rdf/2008/model#name> {name}";
ElmoQuery<?> query = manager.createQuery(queryStr);
query.setParameter("name", "John");
for (Object emp : query) {
    assert ((Employee) emp).getName().equals("John");
}
```

The method setParameter of ElmoQuery can be used to assign literals or entities to variables in the query. To assign an entities by name, use the setQName method. To assign a class use the setType method.

3.5. Concepts

In Elmo, a *role* is defined as a *concept* or *behaviour*. A *concept*, in Elmo, is a *JavaBean* interface or class with optional Elmo annotations describing the subject type of the interface or class and the *predicate*s of each mapped Bean property or field. When using classes as concepts, property methods will be overriden if annotated, and annotated fields will be populated if used in a public method. If multiple concept classes are mapped to a single entity, none of the concept classes will be extended. Elmo ships with a collection of popular ontologies defined as Elmo *concept*s. They can be found in the elmo-concepts *module*.

Every Elmo concept property or field must be a single or a set of entities and/or *literal*s (sometimes referred to as value objects). These correspond to the *RDF* structure, where everything is a resource or a *literal* and properties have multiple values or a single value (where value is an *entity* or literal). Elmo uses *Sesame*, an *RDF repository*, for the persistence layer and all Elmo concept properties are mapped to an *RDF triple*, consisting of subject, *predicate*, and value. *RDF* repositories are *semi-structured*. For more information about *Sesame* and *RDF* please consult the *Sesame* User Guide or FAQ.

You can use any *URI* (or *URL*) you want with @rdf, but it must be unique to that property or role and every persistent concept property must have a *predicate*. Concepts properties are declared with an @rdf or @inverseOf annotation placed on a getter method or field, or map in a file named "META-INF/org.openrdf.elmo.properties". The contents of the file must be the full class name of the concept, followed by "\#" and the field name, or followed by "." and the property name followed by " = " then the predicate URI. For example the contents of the file might look like this:

mypackage.MyClass.myProperty = http://mydomain.com/myProperty

First URI from either @rdf and @inverseOf

The @rdf and @inverseOf annotations are placed on getter methods of Beans (classes or interfaces) or fields of concept classes. They are placed on methods that have no parameters, a return type, and start with "get" or if a primitive boolean "is". A setter method is not required, but if present it will also be implemented. These methods will be implemented to map to the *RDF* values in the *repository*. The fields will be populated and saved if accessed from within a local public method (such as a getter and setter) or accessed in a local method called from a local public method. The property or field type can be any registered *concept*, *literal*, or java.util.Set of the above. When a setter method is not present the property is read-only. @inverseOf annotation is used to indicate inverse relations as show below. The @rdf annotation takes precedence over the @inverseOf annotation, if both are specified, and only the first *URI* is used for reading the values from the repository.

```
@rdf("http://www.example.com/rdf/2008/model#subteam")
public Set<Team> getSubteams();
public void setSubteams(Set<Team> subteams);

@inverseOf("http://www.example.com/rdf/2008/model#subteam")
public Team getParentTeam();
```

@localized

The localized annotation can be used on properties that return a String or a Set of Strings. When this annotation is present the Locale of the *ElmoManager* is associated with assigned values, and when retrieved the closest set of values for the Locale of the *ElmoManager* are returned.

Other @rdf and @inverseOf URI values

If inferencing is enabled in the factory, through setInferencingEnabled, and the getter method of a concept property contains additional URI values, all changes to the property will also modify each listed *predicate*, and *inverseOf predicate*s will be modified with the subject and value reversed.

```
public interface Team {
    @rdf("http://www.example.com/rdf/2008/model#members")
    public Set<Employee> getMemebers(); // read-only

    @rdf({"http://www.example.com/rdf/2008/model#teamLead",
          "http://www.example.com/rdf/2008/model#members"})
    @inverseOf({"http://www.example.com/rdf/2008/model#leaderOf"})
    public Employee getTeamLead();
    public void setTeamLead(Employee value);

    @rdf({"http://www.example.com/rdf/2008/model#salesRep",
          "http://www.example.com/rdf/2008/model#members"})
    public SalesRep getSalesRep();
    public void setSalesRep(SalesRep value);

    @rdf({"http://www.example.com/rdf/2008/model#engineer",
          "http://www.example.com/rdf/2008/model#members"})
    public Engineer getEngineer();
    public void setEngineer(Engineer value);
}
```

In the example above (with inferencing enabled) the members property will be updated as the properties teamLead, salesRep, and engineer are modified. The property leaderOf of Employee will also be updated with their Team as the teamLead value is changed.

@oneOf

The annotation *oneOf* can be placed on the setter method to assert the *URI* (value) or *label* values are correct. This check is only performed when assertion is enabled, which can be enabled with the vm argument "-ea".

```
@rdf("http://www.example.com/rdf/2008/model#gender")
public String getGender();
@oneOf(label={"M", "F"})
public void setGender(String value);
```

3.6. Role Annotations

Role annotations are placed on concepts or *behaviour*s to statically associate them with the subject types or individuals that they should be composed with.

@rdf

The @rdf annotation, in addition to being used on properties, is also used on the class/interface to associate it with a primary subject type. On class/interface declarations, alias subject types can be indicated with additional annotation values. The *role* will be included for any subject of any type in @rdf.

```
@rdf("http://purl.org/rss/1.0/channel")
public interface Channel {
    @rdf("http://purl.org/rss/1.0/link")
    public Set<String> getLink();
    public void setLink(Set<String> link);
```

Above we have defined a Channel *concept* that should be used with every resource of the type "http://purl.org/rss/1.0/channel". It has one property named link which maps to the *RDF predicate* "http://purl.org/rss/1.0/link". The same concept could also be create as a Java class.

```
@rdf("http://purl.org/rss/1.0/channel")
public class Channel {
    @rdf("http://purl.org/rss/1.0/link")
    private Set<String> link;

    public Set<String> getLink() { return link; }
    public void setLink(Set<String> link) { this.link = link; }
```

@disjointWith

To indicate restrictions of what types this *role* cannot be created with, the *disjointWith* annotation can be used. This is only checked when assertions are enabled and is only read from *concept*s passed on the create method of the *ElmoManager*.

```
@rdf({"http://www.example.com/rdf/2008/model#Customer",
      "http://www.example.com/rdf/2008/model#Client"})
@disjointWith({Employee.class})
public interface Customer {
    ...
}
```

@intersectionOf, @complementOf, and @oneOf

These annotations are an alternative to directly mapping roles to an RDF subject type. They provide a way to assign roles based on other roles of the entity or based on their *qualified-name*. In this way role specific behaviours can be defined based on a combination of roles, the absence of a role, or on a set of *individual*s.

The *intersectionOf* annotation takes a list of *role*s, that if a entity implements all of the given roles, it will also implements the declared role. The *complementOf* annotation takes a single role as argument; if any *entity* does not implement the given role, it will implement the declared role. The *oneOf* takes a list of *qualified-name*s; if any *entity* is created with one of the *qualified-name*s it will implement the declared role.

3.7. Registering Concepts & Behaviours

By registering a public interface or class with a subject type, Elmo will dynamically include this *role* with all resources of the defined type. Registering *concept*s is required in order for them to be used with the designate method. The *composition* of *role*s can facilitate decentralized design and development for an application or a suite of applications.

Concepts are statically loaded from "META-INF/org.openrdf.elmo.concepts" files and behaviours are loaded from "META-INF/org.openrdf.elmo.behaviours". These files may be included in any number of jars or paths in the Java class-path. The format of these files is the full class name optionally assigned to one or more *URI* subject type. If the @rdf class annotation is present its value will also be included. If no mapping is provided for behaviours, the role annotations of its interface concepts will be used. Here is a sample of the contents of the file.

```
org.openrdf.concepts.foaf.Person
org.openrdf.concepts.foaf.Agent
org.openrdf.concepts.dc.DcResource
```

If any of these files does not contain any classes, but does exist in a directory or jar in the class-path, the entire directory or jar will be scanned for classes with a role annotation or classes that extend or implement a annotated class and automatically register them. Concept classes cannot be register automatically. Each role will be logged at the DEBUG level when being registered. Roles can also be registered at run-time using the addConcept and addBehaviour methods of the ElmoModule.

```
ElmoModule = new ElmoModule();
module.addConcept(Engineer.class,
    "http://www.example.com/rdf/2008/model#Engineer");
// URI type of SalesRep is retrieved from the @rdf annotation
module.addConcept(SalesRep.class);
factory = new SesameManagerFactory(module);
manager = factory.createElmoManager();
```

3.8. Behaviours

A *behaviour*, also called a *mixin*, is a concrete or abstract class that implements one or more methods for an *Entity*. Behaviours provide functionality other then *RDF* property mapping. They simplify code by capturing design patterns into reusable classes. Behaviours are created with a factory, or a constructor with an interface parameter implemented by the *entity*. Behaviours are registered in the similar way that *concept*s are registered, which is described in Section 3.7.

Behaviour factories are also registered in the ElmoModule. Factories use the same role mapping as concepts and behaviours. They are used to create a behaviour for used in an *entity*. An example of using a behaviour factory can be found in Section 5.8.

Behaviours without a corresponding factory must have a default constructor or a constructor with a single parameter. This parameter can be any concept that the entity will implement, including the Entity interface itself. Behaviours have access to all properties and other behaviours within the *entity* through abstract methods or the constructor parameter.

Behaviours can be used to implement methods for the *entity*'s interfaces. Below is the EmployeeSalarySupport behaviour. Here the class implements the paySalary method for an Employee. It splits the salary amount into two bank accounts based on the properties of the employee.

```
public abstract class EmployeeSalarySupport implements Employee {
  public void paySalary(double amount) {
    double invest = getInvestmentPercent() * amount;
    BankAccount fund = getInvestmentAccount();
    BankAccount checking = getSalaryDepositAccount();
    if (invest >= amount) {
      fund.deposit(amount);
    } else if (invest > 0) {
      fund.deposit(invest);
      checking.deposit(amount - invest);
    } else {
      checking.deposit(amount);
    }
  }
}
```

3.9. Interceptors

Interceptors are *behaviour*s that have the @intercepts annotation on one or more of their methods. They are used to maintain a 1:1 mapping between design concept and implementation for cross cutting concerns. The annotation @intercepts must be placed on methods that take the same parameters as the method(s) intercepted or an InvocationContext as the parameter.

The @intercepts annotation uses pattern matching to filter which methods of the *entity* should be intercepted. By default every method of the *entity* with the same method name will be intercepted. The annotation provides six inclusive ways to select which Methods to intercept.

method

: is an regex match on the entire Method name, if not provided the method name must have the same name as the method being intercepted.

argc

: checks the number of parameters.

parameters

: ensures the parameterTypes of the method are equivalent to the classes given.

returns

: checks for equivalent or superclass of the Method's return type.

declaring

: checks if this class has this method.

conditional

is the name of a static method in the *interceptor* class that takes a Method argument and returns a boolean indicating if this method should be intercepted.

Below is an example of using an interceptor.

```
@rdf("http://www.example.com/rdf/2008/Message")
public class EmailValidator {
    @intercept(method="set.*Email.*", parameters={String.class})
    public void setEmail(InvocationContext ctx) throws Exception {
        String email = (String) ctx.getParameters()[0];
        if (email.endsWith("@example.com")) {
            ctx.proceed();
        } else {
            throw new IllegalArgumentException("Only internal emails");
        }
    }
}

ElmoModule module = new ElmoModule();
// if no URI and no @rdf, applies to all entities
module.addBehaviour(EmailValidator.class);
module.addConcept(Message.class);
factory = new SesameManagerFactory(module);
manager = factory.createElmoManager();

Message message = manager.create(Message.class);
message.setFromEmailAddress("john@example.com"); // okay
message.setToEmailAddress("jonny@invalid-example.com"); // throws exc
```

3.10. Factory Classes

Factory classes are classes that create behaviour instances. They are registered using the method addFactory in the ElmoModule or in the file "META-INF/org.openrdf.elmo.factories", using the same format as for concepts and behaviours. All factory classes must use the @factory annotation on each of their factory methods. The factory class must have either a static getInstance method or a public default constructor.

The factory classes are mapped in the org.openrdf.elmo.factories file or mapped with role annotations. The role annotations may be placed on the factory class or on the return type of the factory methods or on an interface implemented by the return type. The return type of each factory method must also be registered for, or be a concept of, the entity being constructed. If the factory method has a parameter it must be assignable from the entity being constructed.

See Section 5.8 for an example.

3.11. Datatypes

Any mapped @rdf property value, that is not an *entity*, is either a java.util.Set or a *literal* object (java.util.List is an entity). A datatype (*literal* Class) must be Serializable, have a String constructor, or a static valueOf method, if not directly supported by the LiteralManager. Elmo ships supporting most common Java *literal* Objects. Elmo also includes mapping to primitive *XML-Schema data-type*s that correspond to a standard Java Class.

Elmo, like Java5 collections, uses run-time type checking on object retrieval when casting. Elmo uses the *data-type* of *literal*s exclusively to look up the Java Object that should be instantiated and does not convert the Object to conform to a property type. If the *data-type* of the *literal* in the *repository* does not correspond to the properly type of the *concept* a ClassCastException will occur at run-time when the *literal* is retrieved. Any data that is imported into the *repository* must contain the correct *data-type*s for every *literal*, according to the org.openrdf.elmo.datatypes files. Note that no *data-type* is used for java.lang.String class.

Elmo does has some flexibility when retrieving *literal*s from the *repository*. For example the xsd:positiveInteger and xsd:integer, both resolve to java.math.BigInteger. However, in the *repository*, *literal*s of *data-type* xsd:positiveInteger do not equal *literal*s of xsd:integer, therefore if a collection contains "1"^^xsd:positiveInteger it may not contain "1"^^xsd:integer. This will cause the Set#contains(Object) to fail when searching for a match of java.math.BigInteger("1") as it will be searching for "1"^^xsd:integer.

This *data-type* mapping can be extended by including a file "META-INF/org.openrdf.elmo.datatypes". This file may be included in any number of jars or paths in the Java class-path. The format of the file is the full class name of the type assigned to the *URI* of the *data-type* that should be used when saving it to the *repository*.

```
javax.xml.datatype.XMLGregorianCalendar = http://www.w3.org/2001/XMLSchema#dateTime
java.util.Locale = http://www.w3.org/2001/XMLSchema#language
```

By default Elmo uses XMLGregorianCalendar as the Java Object for all *XML-Schema* time or date related *data-type*s. When adding an XMLGregorianCalendar to the *repository* it will use the xml schema type of the object when storing. Other date objects are supported, but by default will use the *data-type* "java:" + full class name of the Object.

3.12. RDF Collections

Elmo properties can use the collection type java.util.Set and java.util.List for property values. Properties with this type can be assigned instances from the Java collections framework, that implement one of these two interfaces. When removing entities from the repository, any property with a type of java.util.List will need to remove the list explicitly before removing the entity itself. This is not the case for java.util.Set, as it is removed along with the entity.

Through the *adaptor* SesameContainer, entities of type rdfs:Container, including rdf:Seq, rdf:Alt, and rdf:Bag will implement java.util.List. When other java.util.List objects are merged into the repository, through a setter method or explicitly with the merge method, they will have the rdf:type rdfs:Container and when loaded again they will use this adaptor.

The *adaptor* SesameList implements java.util.List for resources of type rdf:List. Often *RDF* file formats provide a short hand format for rdf:List resources. These short hands may not include the required

rdf:type statement for the list and therefore may not be imported properly into the repository for Elmo to read. In addition, rdf:Lists are implemented as a chain of rdf:Lists, so when removing them from the repository each must be removed explicitly, or cleared before being removed. Because of these three limitations for rdf:List resources, it is recommended to use one of the above rdfs:Containers as an alternative to rdf:List.

Another *RDF* collection is a set of statements that share a common subject and *predicate*. These statements are contained in the class ElmoProperty, which implements java.util.Set for the statement's objects. This collection is used for every @rdf property that returns a java.util.Set. Therefore collection properties can exist for java.util.Set, java.util.List, or one of the previously listed *RDF* concepts (rdf:Seq, rdf:Bag, rdf:Alt).

3.13. DynaBeans

An alternative to creating static concepts is the DynaClass/DynaBean interface from commons-beanutils. The DynaBeanSupport *behaviour* provides the DynaBean interface for any resource. It uses the *RDFS* /*OWL ontology* in the *repository* to resolve none-prefixed property names. When using none-prefixed property name, they will either return a single value or a set of values depending on the ontology specification of the property. When using prefixed properties, however, the ontology is not used and all properties return a set. Property can be prefixed with a namespace prefix from the repository or the entire predicate *URI*. This *behaviour* is *not* active by default. Here is an example of how this *behaviour* can be activated and used.

```
ElmoModule module = new ElmoModule();
module.addBehaviour(DynaClassSupport.class);
module.addBehaviour(DynaBeanSupport.class);
factory = new SesameManagerFactory(module);
manager = factory.createElmoManager();

QName id = new QName(NS, "E340076");
QName type = new QName("http://www.example.com/rdf/2008/model#","User");
DynaClass userClass = (DynaClass) manager.find(type);
DynaBean entity = manager.rename(userClass.newInstance(), id);
entity.set("userName", "john");
assertEquals("john", entity.get("userName"));
```

4. Elmo Tools

4.1. Compiling and Decompiling and Ontologies

The included elmo-codegen.jar can be used from the command line to create an RDF ontology file from existing JavaBeans or generate Elmo roles from an RDF ontology file. The command below will search the given jar (example-entities.jar) for classes in the package com.example.entities and output an OWL DL ontology in example-ontology.owl using the given namespace.

```
java -jar elmo-codegen.jar \
```

```
    -b "com.example.entities=http://www.example.com/rdf/2008/model#" \
    -r example-ontology.owl \
    example-entities.jar
```

In the example below, the ontology example-ontology.owl will be imported, and Elmo concepts that are defined with the given namespace will be created and compiled in example-concepts.jar. This jar will then be ready to be used for development and deployment of an Elmo application.

```
java -jar elmo-codegen.jar \
    -b "com.example.concepts=http://www.example.com/rdf/2008/model#" \
    -j example-concepts.jar \
    example-ontology.owl
```

The elmo-codegen.jar will import the ontologies and concepts from elmo-rdfs.jar and elmo-owl.jar, so they don't need to be specified on the command line. However, other dependent concepts jar files must be included at the end of the command.

4.1.1. Method Declarations

The elmo-codegen.jar will create concept interfaces from OWL classes and included getters and setters for each of the declared properties within the concept interface. Method declarations will be included in the interface if a special class hierarchy is be used. Any class that extends elmo:Message, where elmo is mapped to the namespace "http://www.openrdf.org/rdf/2008/08/elmo#", will be used to create a method declaration in a corresponding interface. There are three property restrictions that are used to determine the declaration. All values from restrictions on elmo:target determines the concept that should contain this method. Restrictions on elmo:objectResponse determine the return type of the method, use owl:Nothing for void return types. Restrictions on elmo:literalResponse determine the return type of methods with a literal response. Methods that return a primitive type should include a owl:cardinality restriction of 1 on elmo:literalResponse. Both response properties can be restricted with owl:maxCardinality of 1 to indicate they return a single result and not a Set of results. Any other properties declared for classes that extend elmo:Message will be included as parameters in the method declaration ordered by their URI.

The following RDF/XML fragment will produce an interface called "HelloWorld" with a method "hello" that returns a single String.

```
<owl:Class rdf:ID="HelloWorld"/>
<owl:Class rdf:ID="hello">
  <rdfs:subClassOf="&elmo;Message"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&elmo;target"/>
      <owl:allValuesFrom rdf:resource="#HelloWorld"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&elmo;literalResponse"/>
      <owl:allValuesFrom rdf:resource="&xsd;string"/>
    </owl:Restriction>
  </rdfs:subClassOf>
```

```
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="&elmo;literalResponse"/>
      <owl:maxCardinality>1</owl:maxCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

Classes that extends elmo:Message will also generate their own concept interfaces. These concepts will have properties that correspond to the parameters of the method. They will also extend the interface org.openrdf.elmo.Message and have behaviour that implements elmoInvoke() that correctly maps the concept properties to the method parameters invoking the method on the elmoTarget property value.

4.1.2. Compiling Behaviours

The elmo-codegen.jar tool can compile valid Java or Groovy behaviour methods while generating concepts and package them both into the generated jar. Any owl:ObjectProperties that extend elmo:method with the annotation elmo:java or elmo:groovy will be used to create a behaviour method for a corresponding concept method. The rdfs:domain indicates the concept that should implement this method and rdfs:range indicates the method declaration that is implemented by the elmo:java or elmo:groovy annotation.

If the method includes parameters they will be available as read only variables in the method body. The name will be a combination of the compiled namespace prefix of the property joined with the local name of the property in the same way they are joined to create Java properties on concepts.

The following RDF/XML fragment will create a behaviour method that implements the hello method, returning the string "World" for resources that implement the HelloWorld concept.

```
<owl:ObjectProperty rdf:ID="world">
  <rdfs:subPropertyOf rdf:resource="&elmo;method"/>
  <rdfs:domain rdf:resource="#HelloWorld"/>
  <rdfs:range rdf:resource="#hello"/>
  <elmo:java>return "World";</elmo:java>
</owl:ObjectProperty>
```

Methods with multiple parameters will also include an bridge method that accepts a java.util.Map argument. Although this bridge method has no interface it is available through reflection in the proxy entity and therefore available to interpreted languages, for example, in Groovy's named parameter syntax.

4.2. Elmo Scutter

The Elmo scutter is a generic RDF crawler that follows rdfs:seeAlso links in RDF documents, which typically point to other relevant RDF sources on the web. The Elmo scutter is based on original code by Matt Biddulph for Jena.

RDF(S) seeAlso is also the mechanism used to connect FOAF profiles and thus (given a starting location) the scutter allows to collect FOAF profiles from the Web. Several advanced features are provided to support this scenario:

- Blacklisting: sites that produce FOAF profiles in large quantities are automatically placed on a blacklist. This is to avoid collecting large amounts of uninteresting FOAF data produced by social networking and blogging services or other dynamic sources.
- White listing: the crawler can be limited to a domain (defined by a URL pattern).
- Metadata: the crawler can optionally store metadata about the collected statements.
- Filtering: incoming statements can be filtered individually. This is useful to remove unnecessary information, such as statements from unknown name-spaces.
- Persistence: when the scutter is stopped, it saves its state to the disk. This allows to continue scuttering from the point where it left off. Also, when starting the scutter it tries to load back the list of visited URLs from the repository (this requires the saving of metadata to be turned on).
- Logging: The Scutter uses slf4j to provide a detailed logging of the crawler.

The data collected by the scutter is stored in a Sesame repository. We recommend using a Native RDF repository for scuttering, because it provides the best performance for uploads.

The Scutter is available as a Java class as well as a Java servlet. The servlet provides access to all of the above features, except for filtering (which requires programming) and it can be deployed by placing the Elmo.war file in the web application directory of a Servlet/JSP container.

The servlet initialization parameters to be specified in the web.xml descriptor file are listed below. An example web.xml file is provided in the war file.

Parameter	Description	Optional/Default
server	URL of the Sesame server to store the collected data	Required
repository	Name of the repository on the server	Required
username	Username for access to the Sesame repository	Optional
password	Password for access to the Sesame repository	Optional
queue	Location of the file used to save the queue when the scutter is stopped	Required
start	URL(s) used to start scuttering. URLs should be separated by white space.	Optional
domain	Limits crawling to URLs that match the provided regular expression.	Optional

metadata	Produce reified statements containing information about the provenance of the statements and the time they were collected. Possible values: true/false	Optional, defaults to false.
autoblacklist	Enable/disable automatic blacklisting. Possible values: true/false	Optional, defaults to true (enabled).
vocab	Restrict crawling to FOAF specific vocabularies (statements with predicates from the RDF, RDFS, FOAF or WGS_84 namespaces)	Optional, only possible value is 'foaf'
focused	Collect data about a specific set of target persons. The target persons are given as foaf:Person instances in the repository.	Optional, actual value is ignored
maxThreads	Maximum number of threads allowed to be running. Must be a positive integer.	Optional, defaults to 20.

The request parameters to the server are listed in the table below. For convenience, there is an html file provided in the distribution for calling various operations on the servlet.

Parameter	Description	Optional/Default
start	Try to load the set of visited URLs and start the scutter	Parameter value ignored.
stop	Stop the scutter, save the queue to disk	Parameter value ignored.
preloadQueue	Preload the queue from the saved file	Parameter value ignored.
clear	Clear the queue and the set of visited URLs	Parameter value ignored.

A custom filtering of statements can be implemented by setting an instance of the StatementFilter interface using the setStatementFilter method of the Scutter class. See the JavaDoc for more details.

4.3. Elmo Smusher

The task of the Elmo smusher is to find equivalent instances in large sets of data. This is a very common problem when processing collections of FOAF profiles as several sources on the Web may describe a the same individual using different identifiers or blank nodes (which are always assumed to be different). While the servlet provided is specific to smushing foaf:Person instances, the underlying mechanism is generic

The smusher uses instances of ResourceComparator for comparing instances. Implementations of ResourceComparator are given for foaf:Person and swrc:Publication.

The smusher reports the results (matching instances) by calling methods on registered listeners. Listeners implement the SmusherListener interface. Two implementations of SmusherListener are provided: one writes out results in text, while the other represents matches using the owl:sameAs relationship and uploads such statements to a Sesame repository. While Sesame does not directly support OWL semantics, the semantics of this relationship (the equivalence of property values) can be easily axiomatized using Sesame's custom rule language.

5. Common Design Patterns

5.1. Adaptor

The *concept* interface can appear limited and may not fit well into existing Java interfaces. This can be resolved by creating a *behaviour adaptor*. Entities will implement all interfaces on each of its *behaviour*s. By creating annotated abstract methods on the *behaviour* to store the property values, it can adapt these values to an established Java interface.

An example of this are the *RDF* collection *behaviour*s, implementing java.util.List, described in Section 3.12.

5.2. Aspects

In *AOP*, *cross-cutting-concern*s are separated into *aspect*s that contain a small amount of behaviour used throughout a domain model. These *aspect*s are then weaved into the primary model at matching *point-cut*s. Elmo includes support for *aspect*s as a single *behaviour* or *interceptor* class, which declare their own *point-cut*s in the role annotations, method declaration, and @intercepts annotation.

5.3. Chain of Responsibility

Behaviour methods are chained together if they have the same signature. The *chain* is broken when a method returns a non-null value, true, or a non-zero primitive. Chained methods with a void return type cannot break the *chain* until all *behaviour*s in the *chain* are evaluated.

```
public class SupportAgentEmailReader {
    public boolean readEmail(Message message) {
        if (message.getToEmailAddress().equals("help@support.example.com")) {
            // process email here
            return true;
        } else {
            return false;
        }
    }
}
```

```
public abstract class PersonalEmailReader implements EmailUser {
    public boolean readEmail(Message message) {
        String un = getUserName();
        if (message.getToEmailAddress().equals(un + "@example.com")) {
            // process email here
           return true;
        } else {
           return false;
        }
    }
}

String agentType = "http://www.example.com/rdf/2008/model#SupportAgent";
String userType = "http://www.example.com/rdf/2008/model#User";
ElmoModule module = new ElmoModule();
module.addConcept(SupportAgent.class, agentType);
module.addBehaviour(SupportAgentEmailReader.class, agentType);
module.addBehaviour(PersonalEmailReader.class, userType);
module.addConcept(EmailUser.class, userType);
factory = new SesameManagerFactory(module);
manager = factory.createElmoManager();

QName id = new QName(NS, "E340076");
manager.designate(id, User.class);
manager.designate(id, SupportAgent.class);
EmailUser user = (EmailUser) manager.find(id);
user.setUserName("john");

Message message = manager.create(Message.class);
message.setToEmailAddress("john@example.com");
if (user.readEmail(message)) {
    // email has been read
}
```

5.4. Cohesion

Typical OO design tends to implement communicational *cohesion* - that is, the methods are grouped together into a class because they operate on the same data. With Elmo the higher level of functional *cohesion* is achieved by separating the class even further into functional behaviours. Test driven development becomes much easier at this level of *cohesion*, making each unit test small and easily understood. This allows the design to meet the single responsibility principle, giving each behaviour class one, and only one, reason to change.

5.5. Context Specific Data

Elmo also supports context specific entity managers. We have already demonstrated how a entity can implement both a SalesRep and an Engineer, but what if someone is a SalesRep only within a specific context and not otherwise?

Entity managers can be created with multiple inclusive contexts. The ElmoModule class is optionally associated with a context, described as a QName. When these *module*s include other *module*s they also inherit a read-only view of the included contexts. New factories created against a module with a context will only read property values from that context and any of the included contexts. When a property is changed to a new value, the old value is removed only if it is in the primary context. When using contexts, each triple statement is stored as a 4-tuple with the primary context as the fourth value.

In the example below we can see that the employee E340076 is a SalesRep in the first period and an Engineer in the second with a different salary. Any manager with the common context will have E340076 as a employee named John, but only in period1 will John be a SalesRep with a salary of 90, and only in period2 will John be an Engineer with a salary of 100. In this way contexts can also be used to setup security domains around sensitive data.

```
QName c = new QName(NS, "Period-common");
QName p1 = new QName(NS, "Period-1");
QName p2 = new QName(NS, "Period-2");
ElmoModule module = new ElmoModule();
module.addConcept(Employee.class);
module.addConcept(SalesRep.class);
module.addConcept(Engineer.class);
module.addBehaviour(SalesRepBonusBehaviour.class);
module.addBehaviour(EngineerBonusBehaviour.class);
module.setGraph(c);
ElmoModule m1 = new ElmoModule().setGraph(p1).includeModule(module);
ElmoModule m2 = new ElmoModule().setGraph(p2).includeModule(module);
repository = new SailRepository(new MemoryStore());
repository.initialize();
factory = new SesameManagerFactory(module, repository);
factory1 = new SesameManagerFactory(m1, repository);
factory2 = new SesameManagerFactory(m2, repository);
common = factory.createElmoManager();
period1 = factory1.createElmoManager();
period2 = factory2.createElmoManager();

Employee emp;
Object obj;
QName id = new QName(NS, "E340076");

emp = common.designate(id, Employee.class);
emp.setName("John");

SalesRep slm = period1.designate(id, SalesRep.class);
slm.setTargetUnits(10);
slm.setUnitsSold(15);
slm.setSalary(90);

Engineer eng = period2.designate(id, Engineer.class);
eng.setBonusTargetMet(true);
eng.setSalary(100);

obj = common.find(id);
assertTrue(obj instanceof Employee);
assertFalse(obj instanceof SalesRep);
```

```
assertFalse(obj instanceof Engineer);
emp = (Employee) obj;
assertEquals("John", emp.getName());
assertEquals(0.0, emp.getSalary());

obj = period1.find(id);
assertTrue(obj instanceof Employee);
assertTrue(obj instanceof SalesRep);
assertFalse(obj instanceof Engineer);
emp = (Employee) obj;
assertEquals("John", emp.getName());
assertEquals(90.0, emp.getSalary());
assertEquals(6.75, emp.calculateExpectedBonus(0.05), 0);

obj = period2.find(id);
assertTrue(obj instanceof Employee);
assertFalse(obj instanceof SalesRep);
assertTrue(obj instanceof Engineer);
emp = (Employee) obj;
assertEquals("John", emp.getName());
assertEquals(100.0, emp.getSalary());
assertEquals(5, emp.calculateExpectedBonus(0.05), 0);
```

5.6. Data Localization

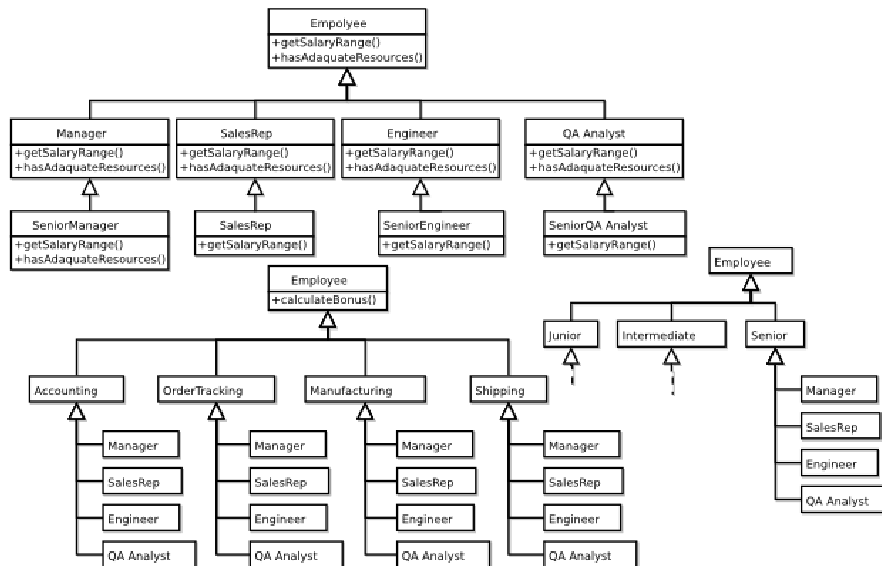
Elmo supports seamless data *localization* for String values. This is activated on concept properties with the @localized annotation. By setting the locale of the *ElmoManager*, the property values will be retrieved and stored for the closest language that matches the manager's locale.

```
factory = new SesameManagerFactory(new ElmoModule());
english = factory.createElmoManager(Locale.ENGLISH);
french = factory.createElmoManager(Locale.FRENCH);

QName id = new QName(NS, "D0264967");
document = (DcResource) english.find(id);
document.setTitle("Elmo User Guide");

document = (DcResource) french.find(id);
assert "Elmo User Guide".equals(document.getTitle());
document.setTitle("Elmo Guide de l'Utilisateur");
assert "Elmo Guide de l'Utilisateur".equals(document.getTitle());

english = factory.createElmoManager(Locale.ENGLISH);
document = (DcResource) english.find(id);
assert "Elmo User Guide".equals(document.getTitle());
```

5.7. Hyperslices

Hyperslices are used to capture multi-dimensional *separation-of-concerns*. They can be thought of as *aspect*s or *program-slice*s managed at a higher level of concern. A hyperslice is a complete object behaviour hierarchy intended to encapsulate concerns in dimensions other than the dominant one. The behaviour classes within a hyperslice are weaved using the annotations defined in Section 3.6.

Figure 3. [3]



Above we can see examples of three dimensional hyperslices. The top hyperslice shows an object hierarchy defining the salary range and adequate resources behaviours. These behaviours use polymorphism to extend or override the super-classes behaviour. The bottom left captures the complex concern of calculating bonuses. In this hyperslice, each department maintains its own calculation for assigning bonuses, which can differ within a department and by job. The hyperslice on the right shows how a third dimension could also be captured within this domain model.

Without hyperslicing, classes would have to be created for each combination of the dimensions (experience level, department, and job). For example SeniorAccountingManager, JuniorShippingEngineer, ..etc. The behaviours described above might potentially be scattered throughout the hierarchy. However, by using hyperslicing in Elmo, these individual concerns are separated into distinct manageable packages, simplified to only the relevant dimension, and weaved together as needed.

5.8. Dependency Injection

Not all behaviours are self sufficient; some need additional support from other objects or components. In the example below we can see the behaviour BankAccountSupport delegating the method calls to the injected BankAccountService.

```
@rdf("http://www.example.com/rdf/2008/model#BankAccount")
public interface BankAccount extends BankAccountBehaviour {
  @rdf("http://www.example.com/rdf/2008/model#accountNumber")
```

```
  public long getAccountNumber();
  public void setAccountNumber(long number);
}
public interface BankAccountBehaviour {
  public double getBalance();
  public void withdraw(double amount);
  public void deposit(double amount);
}
public class BankAccountSupport implements BankAccountBehaviour {
  private AccountReference account;
  private BankAccountService service;
  public void setAccountReference(AccountReference account) {
    this.account = account;
  }
  public void setBankAccountService(BankAccountService service) {
    this.service = service;
  }
  public double getBalance() {
    return service.getBalanceOfAccount(account);
  }
  public void withdraw(double amount) {
    service.withdrawFormAccount(account, amount);
  }
  public void deposit(double amount) {
    service.depositToAccount(account, amount);
  }
}
@rdf("http://www.example.com/rdf/2008/model#BankAccount")
public class BankAccountFactory {
  @factory
  public BankAccountBehaviour createBankAccountBehaviour(BankAccount account) {
    BankAccountSupport behaviour = new BankAccountSupport();
    long number = account.getAccountNumber();
    assert number > 0;
    BankAccountService service = BankAccountService.getInstance();
    behaviour.setAccountReference(service.getAccountReference(number));
    behaviour.setService(service);
    return behaviour;
  }
}

ElmoModule module = new ElmoModule();
module.addConcept(BankAccount.class);
module.addFactory(BankAccountFactory.class);
factory = new SesameManagerFactory(module);
manager = factory.createElmoManager();
long number = 10000001;
QName qname = new QName(NS, Long.toString(number));
BankAccount account = manager.designate(qname, BankAccount.class);
account.setAccountNumber(number);
double balance = account.getBalance(); // will call accountService
```

In the example above the BankAccount interface is the Elmo concept with the mapped concept property "accountNumber", and the BankAccountBehaviour interface is the behaviour interface of BankAccountSupport. This behaviour has the factory BankAccountFactory, which is responsible for creating the support class instances and injecting them with the BankAccountService instance.

5.9. Observer

When static events need to be triggered from methods called on a entity, such as their property values changing, an *interceptor* or *behaviour* should be created. Interceptors are described in Section 3.9. For dynamic *observer*s that are interested in property modification, the PropertyChangeNotifierSupport *interceptor* can be used by registering it with the subject types that need to be observed.

The PropertyChangeNotifierSupport *interceptor* implements the PropertyChangeNotifier interface, tracking PropertyChangeListeners (*observer*s) and notifying them when a setter property method has been called on the entity or a Set has been modified. If the notifier *repository* is used and the manager is not in auto-flush mode when the property is modified, the entity will not inform its listeners until the *connection*'s state has changed (rollback or commit), otherwise the listeners will be informed before the setter method returns. Only when the event is fired from within the setter method is the listener informed of the property and new value of the change. This *behaviour* is *not* active by default, and can be activated at run-time (shown below) or placed into a resource file "META-INF/org.openrdf.elmo.behaviours" as stated in Section 3.7.

```
ElmoModule module = new ElmoModule();
module.addBehaviour(PropertyChangeNotifierSupport.class,
        "http://www.example.com/rdf/2008/model#Employee");
repository = new SailRepository(new MemoryStore());
repository = new NotifyingRepositoryWrapper(repository, true);
repository.initialize();
factory = new SesameManagerFactory(module, repository);
manager = factory.createElmoManager();

QName id = new QName(NS, "E340076");
Employee employee = manager.designate(id, Employee.class);
TestListener listener = new TestListener();
((PropertyChangeNotifier) employee).addPropertyChangeListener(listener);

manager.setAutoFlush(false);
employee.setName("john");
assertFalse(listener.isUpdated());
manager.flush();
assertTrue(listener.isUpdated());
```

5.10. Persistence Ignorance

Elmo supports two main type of concepts: interfaces and classes. Concept interfaces are smaller and can be joined with other concepts, making them more flexible and easier to combine. However, interface concepts must be instantiated through the ElmoManager's create or designate methods, which ties the implementation to the ElmoManager. For simple concepts, that are fully disjoint from any other concept

not in the hierarchy, a concept class can be used instead. Concept classes have the advantage that they can be created at any time without an ElmoManager and persisted in later. This allows the implementation to be more independent and ignorant of the persistence layer (persistence ignorance). Most POJOs (Plain Old Java Object) can be used as concept classes. When class names and properties are mapped in the appropriate files, the class constructor can be used to create a new instance and persisted into the ElmoManager with the persist method. Concept classes without a getQName() method will be persisted anonymously. Concept classes must be registered with the ElmoModule using the addConcept method or mapped in the "META-INF/org.openrdf.elmo.concepts" file. The concept fields or properties must be mapped in the "META-INF/org.openrdf.elmo.properties" file.

```
# concepts/Person.java
package concepts;
public class Person {
    protected String name;
    public String getName() {
        return name;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getFirstInitial() {
        return name.substring(0, 1);
    }
}
# META-INF/org.openrdf.elmo.concepts
concepts.Person = http://www.example.com/rdf/2008/data/Person
# META-INF/org.openrdf.elmo.properties
concept.Person\#name = http://www.example.com/rdf/2008/data/name
```

5.11. Rules

There are two ways to implement rules in Elmo. One way is to used goal driven, or backward chaining, approach. A virtual read-only property is created on an interface that is implemented by a *behaviour* rule. The rule is executed whenever the property is read and the state of the property is calculated and returned. By using Elmo's build-in method chaining, these rules can be executed in turn until the result is complete. A good example of this are validation rules. Take for example the following example, a rule that states a team must have a team lead before it can be given a project.

```
public abstract class TeamHasNoTeamLeadRule implements Team {
  public boolean isRejectingProjects() {
    return getTeamLead() == null;
  }
  public void collectReasonsForRejectingProjects(List<String> reasons) {
    if (getTeamLead() == null) {
      reasons.add("Team does not have a team lead.");
    }
  }
}
public abstract class TeamAlreadyHasProjectRule implements Team {
  public boolean isRejectingProjects() {
```

```
    return getProject() != null;
  }
  public void collectReasonsForRejectingProjects(List<String> reasons) {
    if (getProject() != null) {
      reasons.add("Team already has project.");
    }
  }
}
```

Another technique that is used is called data driven, or forward chaining. These types of rules have the advantage of persisting their state and can also be used in a filter of a query. Below is a behaviour that is triggered when the expenses change on a project, if the expenses are over budget the overbudget state is modified. This triggers the overbudget rules, which will alert the manager using the injected messaging service.

```
@rdf("http://www.example.com/rdf/2008/model#Project")
public interface Project {
  @rdf("http://www.example.com/rdf/2008/model#manager");
  public Manager getManager();
  public void setManager(Manager manager);
  @rdf("http://www.example.com/rdf/2008/model#expenses");
  public double getExpenses();
  public void setExpenses(double expenses);
  @rdf("http://www.example.com/rdf/2008/model#budget");
  public double getBudget();
  public void setBudget(double budget);
  @rdf("http://www.example.com/rdf/2008/model#overBudget");
  public boolean isOverBudget();
  public void setOverBudget(boolean over);
}
public abstract class ExpensesOverBudgetRule implements Project {
  public void setExpenses(double expenses) {
    if (expenses > getBudget()) {
      setOverBudget(true);
    } else if (isOverBudget()) {
      setOverBudget(false);
    }
  }
  public void setBudget(double budget) {
    if (budget < getExpenses()) {
      setOverBudget(true);
    } else if (isOverBudget()) {
      setOverBudget(false);
    }
  }
}
public class AlertManagerRule implements Project {
  public void setOverBudget(boolean over) {
    MessagingService messaging = MessagingService.getInstance();
    if (over) {
      messaging.alert(getManager(), "Project is over budget");
    } else {
      messaging.inform(getManager(), "Project is within budget");
```

```
    }
  }
}
```

5.12. Serialization

Entities are not serializable. They store all their properties in the *repository* and are just a *facade* to the underlying *connection*. To store and retrieve an *Entity*, the method getQName will return a QName (or null if an anonymous entity), which is serializable. This can later be used to access the entity with find(QName) method. By serializing the QName of the entity instead of the entity itself, the entity can be restored by another manager on another *connection*.

Below is an example of how to serialize and deserialize the entire *repository*. For more information about how to selectively export or to export in other formats like XML, please see the *Sesame* User Guide.

```
File file = new File("repository.rdf");
Writer writer = new FilteWriter(file);
manager.getConnection().exportStatements(new RDFXMLWriter(writer));
writer.close();

File file = new File("repository.rdf");
manager.getConnection().add(file, "", RDFFormat.RDFXML);
```

5.13. Strategy

Elmo dynamically includes appropriate *behaviour*s for entities based on subject type mapping. In the example below, the calculateExpectedBonus method on Employee has two implementations, one for SalesRep and one for Engineer. Elmo will include the appropriate *strategy* (or *behaviour*) for the given subject.

```
public abstract class SalesRepBonusSupport implements SalesRep {
    public double calculateExpectedBonus(double percent) {
        int units = getUnitsSold();
        int target = getTargetUnits();
        if (units > target) {
            return percent * getSalary() * units / target;
        } else {
            return 0;
        }
     }
}
public abstract class EngineerBonusSupport implements Engineer {
    public double calculateExpectedBonus(double percent) {
        boolean target = isBonusTargetMet();
        if (target) {
            return percent * getSalery();
        } else {
            return 0;
        }
```

```
    }
}

ElmoModule module = new ElmoModule();
module.addConcept(SalesRep.class);
module.addConcept(Engineer.class);
module.addBehaviour(SalesRepBonusSupport.class);
module.addBehaviour(EngineerBonusSupport.class);
factory = new SesameManagerFactory(module);
manager = factory.createElmoManager();

QName id = new QName(NS, "E340076");
Engineer eng = manager.designate(id, Engineer.class);
eng.setBonusTargetMet(true);
eng.setSalery(100);

Employee employee = (Employee) manager.find(id);
double bonus = employee.calculateExpectedBonus(0.05);
assertEquals(5.0, bonus);
```

6. Transactions and Concurrency

6.1. Transaction Isolation

The ElmoManager is in auto-commit mode by default with no active transaction. To activate a transaction call the begin method on the transaction object returned from getTransaction - manager.getTransaction().begin(). The transaction is closed by calling manager.getTransaction().commit() or manager.getTransaction().rollback(). It is recommended to use only one active transaction per manager.

6.2. Undo Support

ElmoManager implements the memento design pattern. Because Elmo is transaction based, only operations that were performed by the entity pool the memento was created in, will be undone. When using multiple entity pools, or using the ThreadProxyRepository with multiple *connection*s, multiple memento objects must be created for each. Memento support is only available when the *repository* stack contains a notifying *repository*. Below is an example of how to use the undo feature in Elmo.

```
ElmoModule module = new ElmoModule();
module.addConcept(Employee.class);
repository = new SailRepository(new MemoryStore());
repository = new NotifyingRepositoryWrapper(repository, true);
repository.initialize();
factory = new SesameManagerFactory(module, repository);
manager = factory.createElmoManager();
```

```
Employee emp = manager.create(Employee.class);
Memento memento = manager.createMemento();
emp.setName("John");
assertEquals("John", emp.getName());

manager.undoMemento(memento);
assertNull(emp.getName());
```

The memento object does not lock any modified resources. If the same resources have been modified outside of the memento object and cause a conflict when the memento is undone, unexpected results can appear in the repository, so use with caution.

7. Performance and Optimizations

7.1. Serializing Literal Values

Datatypes must be able to store their state in character format, but may have repository conversion optimizations as MemoryStore does. When using non-String *literal* property values, overhead of conversion can occur particularly if no String constructor is present and no static valueOf method exists. The default LiteralManager ships with common Java conversions and also supports valueOf method, String constructors, and Java serialization. Java serialization is provided as a catch-all solution and often under-performs compared to object-specific converters. When using non-standard *literal* objects, ensure that they have a static valueOf method or String constructor (if this is not possible, consider using a custom LiteralManager).

7.2. AugurRepository and ReadAheadRepository

When using non-local repositories, overhead costs exist for each *repository* hit. If concept properties are retrieved in nested loops, the overhead cost can add up quickly. In this case the *repository* client can be wrapped in an AugurRepository or ReadAheadRepository. The AugurRepository can reduce the number of hits to the delegate *connection* from o(m^n) to o(n) by tracking requests and anticipating related information. The ReadAheadRepository reduces the number of hits to the repository for the same subject.

Augur is best used when the complete results are expected to fit into memory and not all the properties will be read. ReadAhead is best used when the complete results may not fit into memory and most of the concept properties will be retrieved. Below is an example of using the AugurRepository or ReadAheadRepository in a SesameManagerFactory:

```
repository = new HttpRepository("http://server.example.com/openrdf-sesame/", "default");
repository = new AugurRepository(repository); // or new ReadAheadRepository(repository);
repository.initialize();
ElmoModule module = new ElmoModule();
Module.addConcept(Employee.class);
factory = new SesameManagerFactory(module, repository);
manager = factory.createElmoManager();
```

```
// By using Augur all information is gathered in 6 hits.
// By using ReadAhead: 1 + number of employees.
// Otherwise it would hit: 1 + 5 * number of employees.
for (Employee emp : manager.findAll(Employee.class)) {
    String name = emp.getName();
    String address = emp.getAddress();
    String phone = emp.getPhoneNumber();
    String email = emp.getEmailAddress();
    String details = name + address + phone + email;
    System.out.print(details);
}
```

Above, five properties are retrieved for each employee. By using just the AugurRepository the *repository* will be hit for:

1. a list of employees,
2. the type of each employee,
3. the name of each employee,
4. the address of each employee,
5. the phone number of each employee, and finally
6. the email address of each employee.

With just the ReadAheadRepository every property will be retrieved at once for each entity. Without anything the *repository* will be hit for every method called.

There is extra overhead for tracking what properties need to be and have been retrieved. When using a local *repository*, such as MemoryStore or NativeStore directly, the extra overhead does not outweigh the benefits.

7.3. ResourceManager

The subject types of every entity are stored in the *repository* using the predicate "http://www.w3.org/1999/02/22-rdf-syntax-ns#type". The designate method adds the types to the *repository* and are read during create and find methods. *RDF* data by design is semi-structured, making it necessary to read the type of each entity. However, in some *domain*s, reading the type of the entity may not be necessary. By overriding the default resource manager, *domain* specific optimizations can be performed.

7.4. Class Loading

Elmo creates new entity proxy classes dynamically at run-time to account for the different combinations of *role*s and to implement the *RDF* property mapping. These classes are only created once for each combination, and are reused for subsequent entities. When the necessary *role* combinations are known in advance and compiled classes are included in the Java class-path, no classes will be created at run-time. When the system property "elmobeans.target" is set, every dynamic class created will be saved to the

folder value of the property. These class files can be packaged into a jar and deployed with an Elmo application to avoid new classes being created at run-time.

There are three types of classes created: mappers, entities, and behaviours. They are created in the package elmobeans.mappers, elmobeans.proxies, and elmobeans.behaviours respectively. The class names are hash combinations of the names of the *role*s they implement. The mappers are created by ElmoMapperClassFactory implementing ElmoMapperResolver, the proxies and factories are created by ElmoEntityCompositor implementing ElmoEntityResolver. Both implementations attempt to load existing classes using ClassFactory before creating a new copy. Elmo uses Javassist to generate the Java byte-code. If all *role* combinations are compiled in advance, these implementations can be substituted.

8. Library Integration

8.1. Sesame

When using the SesameManagerFactory, all entities created will implement the interface *Entity* and SesameEntity. The *Entity* has access to the *qualified-name* of the resource as a QName and the ElmoManager. The SesameEntity interface provides access to the the *Sesame* resource for this entity and the SesameManager instance used to create it. The SesameManager provides access to the *Sesame repository connection* used by this manager.

The entity's non-functional properties (type of java.util.Set) stream the results from the repositories and should be closed after every iteration to clean up temporary resources. They will close themselves automatically once all the results have been read. However, the manager also has a close(Iterator) method to close opened iterators. This can be used when not all results have been read.

8.2. javax.persistence (JPA)

Elmo provides a subset of the JPA implementation for easier integration into existing frameworks. EntityManager is an Enterprise *JavaBean* persistence interface for relational mapping of Entity Objects. Elmo ships supporting a subset of this interface. This interface provides a standard way for applications to interact with the Bean pool. Because entities are dependent on the *repository connection* they can only be used within a transaction.

To use Elmo through JPA, use the provider org.openrdf.elmo.sesame.SesamePersistenceProvider with the property repositoryId, one of dataDir or applicationId, with optional property resources, which are described below.

dataDir

: the Sesame data directory, stores multiple repository configurations and persistence data.

applicationId

: when no dataDir is provided use this unique id to identify a system specific dataDir location.

serverUrl

the remote Sesame server URL, an alternative to dataDir and applicationId.

repositoryId

the repository configuration id stored in the dataDir or serverUrl, if none exists a persisting MemoryStore repository configuration will be created with this id.

resources

load RDF datasets from resource files - file can either be a supported RDF file or a property file listing RDF resources to load, optionally assigned to a context.

See the FOAF example for how to setup Elmo's JPA implementation in Spring.

Elmo does not support detached entities (unlike other JPA implementations). Therefore entities can only be used within a JPA transaction. Elmo supports foreign entities being merged into the pool if they implement an registered interface, however, they are only copied and cannot be attached to the session. In Elmo the merge and persist method do the same thing. Once an entity has been persisted any further changes to it will not be reflected in the repository.

8.3. Tapestry

Elmo includes a set of configuration and classes that can be used to allow HiveMind to manage the *ElmoManager* as a singleton service. It also includes some configuration points that can be used to provide additional configuration inside a hivemind.xml file.

See the GEDCOM example for how to setup Tapestry and Elmo.

9. Examples

Working examples of the above code samples can be found under the src folder in the file UserGuideTest.java. The elmo-examples.war includes three small applications to help better understand how the different interfaces can be used.

RSS Example

User can input parameters, through channel.jsp or on the command line, which are then used to create a *RSS* feed. This example demonstrates the use of an application managing *ElmoManager* itself.

9.1. FOAF Example

Using the local name of a *FOAF* entry, the servlet prints a listing of their friends and provides navigation along the social graph. In this example the application is using the EntityManager interface and the Spring container manages the entity pool.

9.2. GEDCOM Example

With a search input the application lists all royal people with the given substring in their name from the *repository*. The application provides a browse and an edit screen. Here the entity pool is configured and managed by HiveMind. It uses DynaBean interface to set and retrieve concept properties.

10. Known Limitations

10.1. Truth Maintenance for @equivalent and @inverseOf

The two inferencing annotations, when enabled, simply duplicate every add and remove call to the other *predicate*s. There is no truth maintenance on removed statements as a *reasoner* would do. When two properties have the same *equivalent* (or *inverseOf*) *predicate* and the same value, but only one is removed, its *equivalent predicate* is also removed. If truth maintenance is necessary, it is recommend to add a *reasoner* to the *repository* sail stack instead.

10.2. @disjointWith on Concepts

The *disjointWith* annotation is only read on the *concept* that is passed on the create method. This is not a guarantee that the *concept*s will be disjointed, but only serves as a assertive check within the create method. This check is only performed when assertions are enabled (-ea JVM flag).

10.3. HTTP Repository

The HTTP repository in Sesame 2.x does not include its own uncommitted changes when evaluating queries. The memory and native stores, however, do include their own uncommitted changes. All transactions are isolated from uncommitted changes in other transactions.

11. Frequently Asked Questions

11.1. Do I need a database to use Elmo?

Elmo can use any *Sesame repository* that may or may not need a database. The two repositories used in this user guide, NativeStore and MemoryStore, do not use a database. Other repositories that use a backing database may scale better under higher volume or load.

11.2. Are there any alternatives to creating my own concepts?

The generator *module* can facilitate creating your own *concept*s, but Elmo also includes support for DynaBeans that allow accessing *predicate*s by their local name, without using a *concept* interface.

11.3. How can I remove a concept from a entity?

Use the method removeDesignation on the manager to remove a concept from an entity.

11.4. Why aren't Entities serializable?

*Entity*s keep their state in the *repository* and cannot be serialized in isolation. However, their QName can be serialized, and the entire *repository* can also be serialized.

11.5. I already have existing JavaBeans; can they be moved into an Elmo entity pool?

Elmo *concept*s must first be created, see Section 4.1. Since Elmo *concept*s can be annotated Java interfaces or classes, a existing domain model could also be adapted to act as Elmo concepts, by adding annotations (or adding concepts and properties files), this would allow the same model to work in both Elmo and other entity pools.

11.6. How can I have two implementations of the same method?

When two *behaviour*s implement the same method and are being integrated into the same class, Elmo *chain*s the methods together calling one after another in no particular order until a value is returned or all methods have been called. See Section 5.3 for more information.

11.7. What happens when two concepts having the same property name are applied to the same subject?

If the property types are the same, both setter methods will be called when the property is set and only one will be retrieved when the property is read. If the property types are not the same an exception is thrown when the concepts are combined.

11.8. How can I implement cross cutting concerns in Elmo?

Elmo supports *interceptor*s that can be matched to some or all methods implemented by any *role*. Through this, cross cutting concerns like logging and security can be implemented in an Elmo entity pool.

11.9. How do I trigger events before and after a property change?

By using *interceptor*s to wrap a method invocation, events can be triggered before and after a method call. Elmo also provides the PropertyChangeListener stated in Section 5.9.

11.10. How should Entities be managed in a J2EE environment?

On servlet initialization, the *repository* and manager factory should be initialized. For each request a new *ElmoManager* should be created and its *Entity*s can only be used within the request they were created with. The manager should be closed after the request is complete. The manager factory should be closed when the servlet is destroyed.

11.11. How can I implement an audit log with Elmo?

Create an *interceptor* to log the operations of every type and every method.

11.12. How do I setup a security domain over sensitive data?

Elmo supports data contexts. In Section 5.5 it was shown how property values can be stored within a context. This can be used to separate security *domain*s.

12. Glossary

adaptor

: adapts one interface into another that a client expects. An adaptor allows classes to work together that normally could not because of incompatible interfaces by wrapping its own interface around that of an already existing class or interface.

aspect

: is a part of a program that cross-cuts its core concerns, therefore violating its separation of concerns.

AOP

: (Aspect Oriented Programming) attempts to aid programmers in the separation of concerns, specifically cross-cutting concerns, as an advance in modularization.

behaviour

: is a reusable block of computer code or script that, when applied to an object causes it to respond to input in meaningful patterns. In Elmo a behaviour is a class associated to a subject type.

chain

: is a series of processing objects. Each processing object contains a set of logic that describes the types of command objects that it can handle, and how to pass off those that it cannot to the next processing object in the chain. Behaviours with the same signature are chained together until a value is returned.

cohesion

: is an ordinal type of measurement. Modules with high cohesion tend to be preferable because high cohesion is associated with several desirable traits of software including robustness, reliability, re-usability, and understandability whereas low cohesion is associated with undesirable traits such as being difficult to maintain, difficult to test, difficult to reuse, and even difficult to understand.

complementOf

: is analogous to logical negation: the class extension consists of those individuals that are NOT members of the class extension of the complement class.

component-oriented

: is a field of study within software engineering. It claims that software components, like the idea of hardware components, used for example in telecommunications, can ultimately be made interchangeable and reliable.

composition

: is a way and practise to combine simple objects or data types into more complex ones.

concept

: is a description of supported operations on a type, including syntax and semantics. In Elmo the semantics specify the subject type and property predicates of a Java interface or class.

connection

: allows client software to talk to repository server software, whether these exist on the same machine or not. A connection is required to send commands and receive answers, usually in the form of a result set. All JavaBean properties are stored and retrieved over a repository connection.

cross-cutting-concern

: are aspects of a program which affect (crosscut) other concerns. These concerns often cannot be cleanly decomposed from the rest of the system in both the design and implementation, and result in either scattering or tangling of the program, or both.

data-type

: is a constraint placed upon the interpretation of data in a type system in computer programming.

disjointWith

: will assert that a subject that is being assigned to one type cannot simultaneously be an instance of a specified other type.

domain

indicates the relevant set of entities that are being dealt with by quantifiers.

domain-model

can be thought as a conceptual model of a system. A domain model will tell about the various entities involved and their relationships. The domain model is created to understand the key concept of the system and to familiarize with the vocabulary of the system.

ElmoManager

is a connection to a JavaBean pool. Concept properties are stored, retrieved and converted through an ElmoManager.

Entity

is any JavaBean returned from an ElmoManager.

entity

is a resource tracked by unique identity and not by its value.

equivalent

ties together a set of component ontologies as part of a another. It is frequently useful to indicate that a particular class or property in one ontology is equivalent to a class or property in a second ontology.

facade

is an object that provides a simplified interface to a larger body of code.

FOAF

(Friend of a Friend) is a project for machine-readable modelling of homepage-like profiles and social networks.

GEDCOM

(GEnealogical Data COMmunication) is a specification for exchanging genealogical data between different genealogy software.

groovy

is an object-oriented programming language for the Java Platform as an alternative to the Java programming language. It can be viewed as a scripting language for the Java Platform, as it has features similar to those of Python, Ruby, Perl, and Smalltalk.

hyperslices

encapsulate module concerns in dimensions other than the dominant one.

individual

is a single resource.

interceptor

is behaviour that can be inserted around a method invocation. It allows you to transparently apply cross-cutting behaviour, like logging and metrics, to an existing object model.

intersectionOf

describes a class for which the class extension contains precisely those individuals that are members of the class extension of all class descriptions in the list.

inverseOf

associates two properties with inverted subject and value. i.e. If a property, P1, is tagged as the inverseOf P2, then for all x and y: P1(x,y) iff P2(y,x)

inversion-of-control

is an object-oriented programming principle that can be used to reduce coupling inherent in computer programs it is also know as dependency injection.

JavaBean

is a reusable software component that can be manipulated visually in a builder tool.

join-point

is a point in the control flow of a program, like a method call.

label

is the string value of a literal.

literal

is the most primitive value type represented in RDF, typically a string of characters. The content of a literal is not interpreted by RDF itself and may contain additional XML markup. Literals are distinguished from Resources in that the RDF model does not permit literals to be the subject of a statement (a subject, predicate, value triple).

localization

is the adaptation of an object to a locality. In Elmo only String localization is supported.

mixin

programming is a style of software development where units of functionality are created in a class and then mixed in with other classes.

module

is a self-contained component of a system, which has a well-defined interface to the other components.

object-oriented

programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs.

observer

: is to observe the state of an object in a program.

oneOf

: is a restriction on a property value that must be one of the listed values.

ontology

: is a data model that represents a set of concepts within a domain and the relationships between those concepts. It is used to reason about the objects within that domain.

OWL

: (Web Ontology Language) is a language for defining and instantiating Web ontologies. An OWL ontology may include descriptions of classes, along with their related properties and instances. OWL is designed for use by applications that need to process the content of information instead of just presenting information to humans.

point-cut

: is a set of join points. Whenever the program execution reaches one of the join points described in the pointcut, a piece of code associated with the pointcut (called advice) is executed.

predicate

: is a trait or aspect about that resource that is being described. In Elmo every persistent property has a predicate.

program-slice

: consists of the parts of a program that (potentially) affect the values computed at some point of interest, referred to as a slicing criterion.

qualified-name

: is an unambiguous name that specifies specifically which object, function, or variable a call refers to absolutely.

RDF

: (Resource Description Framework) is a family of World Wide Web Consortium (W3C) specifications originally designed as a metadata model using XML but which has come to be used as a general method of modelling knowledge, through a variety of syntax formats (XML and non-XML).

RDFS

: or RDF Schema is an extensible knowledge representation language, providing basic elements for the description of ontologies, otherwise called RDF vocabularies, intended to structure RDF resources.

reasoner

: performs the act of using reason to derive a conclusion from certain premises, using a given methodology.

repository

is a central place where data is stored and maintained.

role

is a concept, behaviour, interface of a behaviour, or interceptor that an entity can be a composition of.

RSS

is a family of web feed formats used to publish frequently updated digital content, such as blogs, news feeds or pod-casts.

semantic-web

is an evolution of the World Wide Web in which information is machine processable (rather than being only human oriented), thus permitting browsers or other software agents to find, share and combine information more easily.

semi-structured

is a database model. In this model, there is no separation between the data and the schema, and the amount of structure used depends on the purpose.

separation-of-concerns

(SoC) is the process of breaking a program into distinct features that overlap in functionality as little as possible.

Sesame

is an open source Java framework for storing, querying and reasoning with RDF and RDF Schema. It can be used as a database for RDF and RDF Schema, or as a Java library for applications that need to work with RDF internally.

strategy

pattern whereby algorithms can be selected on-the-fly at run-time.

subject-oriented

programming is a method of computer program composition that supports building object-oriented systems as compositions of subjects, extending systems by composing them with new subjects, and integrating systems by composing them with one another.

triple

is an n-tuple with n being 3. A triple is a sequence of three elements. It is not a set of three elements, as the ordering of the elements matters, and an element can be present more than once in the same triple.

tuple

is a finite sequence (also known as an "ordered list") of objects, each of a specified type. A tuple containing n objects is known as an "n-tuple".

URI

(Uniform Resource Identifier) is a compact string of characters used to identify or name a resource. The main purpose of this identification is to enable interaction with representations of the resource over a network.

URL

is a Uniform Resource Locator, a uniform syntax for global identifiers of network-retrievable documents.

URN

is a Uniform Resource Identifier (URI) that uses the *urn* scheme, and does not imply availability of the identified resource. Both URNs (names) and URLs (locators) are URIs, and a particular URI may be a name and a locator at the same time.

XML-Schema

was published as a W3C Recommendation in May 2001, is one of several XML schema languages.

13. References

- Dirk Baumer, Dirk Riehle, Wolf Siberski, and Martina Wulf. The Role Object Pattern (http://st-www.cs.uiuc.edu/%7Ehanmer/PLoP-97/Proceedings/riehle.pdf). 1997.
- Martin Fowler. Dealing with Roles (http://www.martinfowler.com/apsupp/roles.pdf). 1997.
- John Vlissides. Subject-Oriented Design (http://citeseer.ist.psu.edu/392989.html). 1998.
- P. Tarr, H. Ossher, W. Harrison and S.M. Sutton, Jr. N Degrees of Separation: Multi-Dimensional Separation of Concerns (http://www.acm.org/pubs/citations/proceedings/soft/302405/p107-tarr/). Proceedings of the International Conference on Software Engineering (ICSE'99), May, 1999.
- Subject-oriented programming (http://www.research.ibm.com/sop/). 1999.
- Robert C. Martin. Agile Software Development, Principles, Patterns, and Practices. 2002.
- Kazimierz Subieta, Andrzej Jodlowski, Piotr Habela, and Jacek Plodzien. Conceptual Modeling of Business Applications with Dynamic Object Roles (http://www.users.pjwstk.edu.pl/~habela/publications/conceptualModeling_Roles.pdf). 2003
- Alan Cyment, Nicolas Kicillof, and Fernando Asteasuain. Enhancing model-based AOP with behavior representation (http://www.kircher-schwanninger.de/workshops/MDD&AOSD/old/ECOOP2006/papers/CKA.pdf). 2006.
- Nilsson, Jimmy. Applying Domain-Driven Design and Patterns (http://domaindrivendesign.org/books/index.html#DDD_apply). 2006.

Notes

1. ElmoManager Class Diagram
2. Entity Class Composition
3. Hyperslices Example