Chapter 1

ConStore User Interface

The ConStore system has two separate interfaces for specifying metamodels and models. The lifecycle of a typical system modeled under a ConStore system is captured in Figure 1.1. Initially, an empty concept network is created. In an empty network, during the metamodeling phase, entity types and relation types over existing entity types are inducted. Once a metamodel is created, the concept network instance can be built. The inducted types become available after the metamodel is committed.

The ConStore lifecycle supports metamodel evolution. A new entity type or a new relation type can be inducted anytime during model construction. Only when the changes are committed, the newly inducted types become available for the model. Changes to a model similarly require a commit operation. Thus, as depicted in Figure 1.1, the lifecycle supports *batch commit* operations, wherein a sequence of changes to model or a metamodel are committed in batches. When a batch of operations is committed, the changes become available for subsequent operations.

1.1 The ConStore Programming Interface

The ConStore Programming Interface is a Java-based object oriented interface containing operations for metamodels and models. The class diagram (Figure **??**) shows the classes involved to provide the user interface for concept-net modeling.

The programming interface is further decomposed into *formation* and *navigation* interfaces. The formation interface provides operations for construction and manipulation of models and metamodels, whereas, the navigation interface supports various kinds of navigational operations for navigating through models and metamodels.

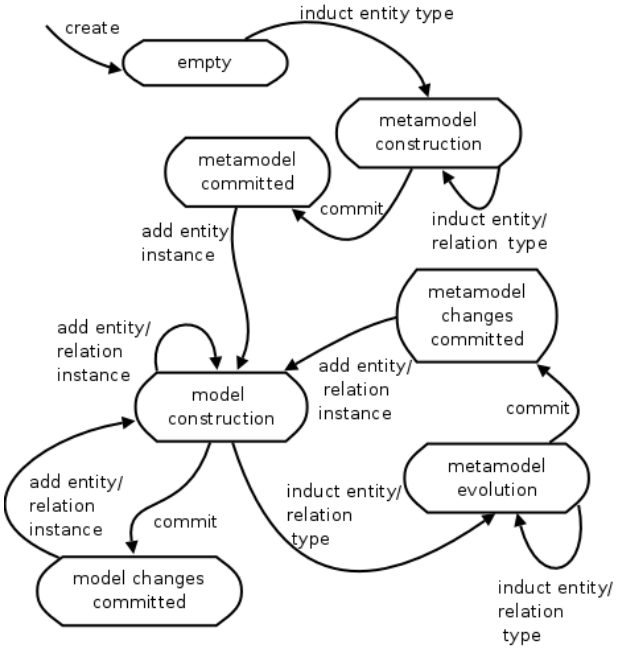

Figure 1.1: Lifecycle of a concept network

1.1.1 Formation Interface

Table 1.1 lists the operations for concept net formation under the categories of system level, metamodel level and model level operations. Some operations are provided as static class methods and some are instance methods. Creation, deletion and sessions on nets are performed via system level operations. Metamodel level operations involve creation of new entity and relation types. Model level operations include creation of entity and relation instances.

During entity and relation type construction, attribute specifications can be added to type specifications. Similarly, entity and relation instances can be constructed. The ConStore user interface facilitates data modeling in two phases of *Modeling* and *Instantiation*.

At metamodel level, once a type is created, it becomes immutable. Evolution is possible at metamodel level through creation of new entity and relation types. During metamodeling, when a new type is inducted, a corresponding class is generated for instance creation in model level programs. Modifiable attributes are members of these classes.

Updates at instance level correspond to updates to their internal attributes. At instance level, updates are possible for existing relation and entity instances. Update at instance level also includes changing relations among entity instances. A relation can be changed by updating its *left* and *right* entity instances. Existing entity and relation instances can be removed.

Operations	Interfaces
System Level Operations	
Create new concept-net	ConStore::create
Open existing net	ConStore::open
Delete existing net	ConStore::delete
Close existing net	ConceptNet.close
Metamodel Level Operations	
Entity type creation	new Entity
Relation type creation	new Relation
Adding attributes	Entity.addAttribute
	Relation.addAttribute
Associating Entities	Relation.setLeft
	Relaton.setRight
Direction of navigation	Relation.setDirection
Inducting metamodel	ConceptNet.induct
Committing metamodel	ConceptNet.commit
Model Level Operations	
Entity instance creation	new <*EntityClass*>
Relation Instance creation	new <*RelationClass*>
Access attributes	<*EntityClass*>.<*Attrib*>
	<*RelationClass*>.<*Attrib*>
Insert entity/relation	ConceptNet.add
Retrieve entity/relation instance	ConceptNet.getInstance
Update retrieved instance	ConceptNet.update
Remove retrieved instance	ConceptNet.remove
Commit model	ConceptNet.commit

Table 1.1: ConStore formation interface

1.1.2 Navigation Interface

Table 1.2 lists some of the operations provided under the ConStore navigation interface *Query*. The operations allow retrieval of relations and entities based on type's names, instance properties and type or instance ids. The navigation interface also provides support for retrieving ids of instances. Through the id-based (Integer type used as id) navigational operations, it is possible to navigate through a network to reach an object without having to retrieve the intermediate objects. For example, the *getinstanceIds*() method has two overloadings for type name based and property based retrievals. Navigation may begin with an instance property and then continue with a cycle of (entity) id-based relation retrieval and (relation) id-based entity retrieval. Navigation direction set for a relation in metamodel formation can constrain certain navigation operations. For example, if a

left-to-right relation will be ignored by method getRelationIds() when it is invoked w.r.t. an entity that plays the *right* role.

A handle to interface *Query* is obtained from a class *ConceptNet*. This process is similar to that of the iterator pattern, in which, a traverser (iterator) is obtained from the collection. Each query instance may maintain its local state such as local caches. This design makes it possible to obtain more than one query interfaces on a concept net and use it over multiple threads. Besides the generic navigational operations supported by interface *Query*, instance-based navigational operations for navigating through relation instances are also supported. Examples of such operations are $Relation.getLeftId()$ and $Relation.getRightId()$. The *Query* interface also supports bulk or anonymous operations through which in-memory lists of all types, instances, or their ids can be constructed.

Operations on Interface *Query* obtainable from a concept net	*Type Name* or *Instance Property* based	*Id* based	Anonymous
Type Retrieval			
getType	√	√	
getAllTypes			√
getAllEntities			√
getAllRelations			√
Instance Retrieval			
getEntityInstances	√	√	
getRelationInstances	√	√	
getRelatedInstances	√	√	
getRelations	√	√	
getAllEntityInstances			√
getAllRelationInstances			√
getInstance		√	
getInstances	√	√	
getAllInstances			√
Type/Instance Id Retrieval			
getRelatedInstanceIds	√	√	
getRelationIds	√	√	
getTypeId		√	
getInstanceIds	√	√	
getAllInstanceIds			√
getAllEntityInstanceIds			√
getAllRelationInstanceIds			√
getLeftEntityId		√	
getRightEntityId		√	

Table 1.2: ConStore navigation interface

1.2 Examples

This section captures the typical lifecycle of a concept network through an example. Consider the word network shown in Figure 1.2. Code snippets in the following subsections describe the steps involved in constructing word network as concept-net.

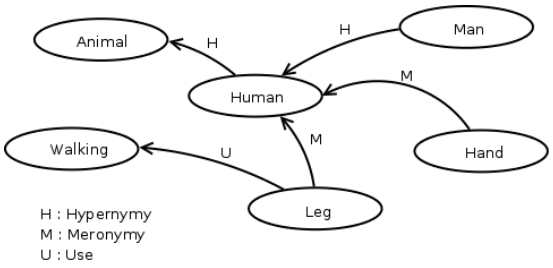

Figure 1.2: Word-Net

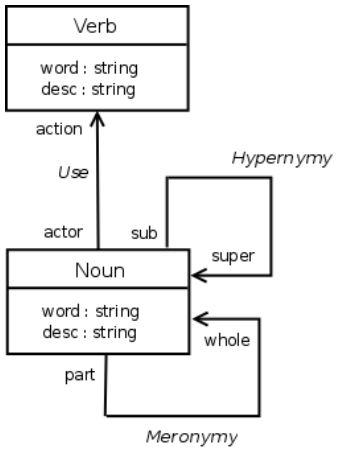

Figure 1.3: Meta model of Word-Net Fig. 1.2

1.2.1 Meta-Model Construction

Initially, user needs to construct the meta-model for the desired graphical structure (concept-net). A meta-model abstracts the graphical structure with minimal entity and relation types such that the entire graphical structure can be constructed using the meta-model. So for word network example, Figure 1.3 represents the meta-model. In meta-model, each concept is modeled as *Entity* associated with relevant attributes and the *Relations* among the entities are established with appropriate roles. The entity and relation are abstracted as *Type* class at the programming level.

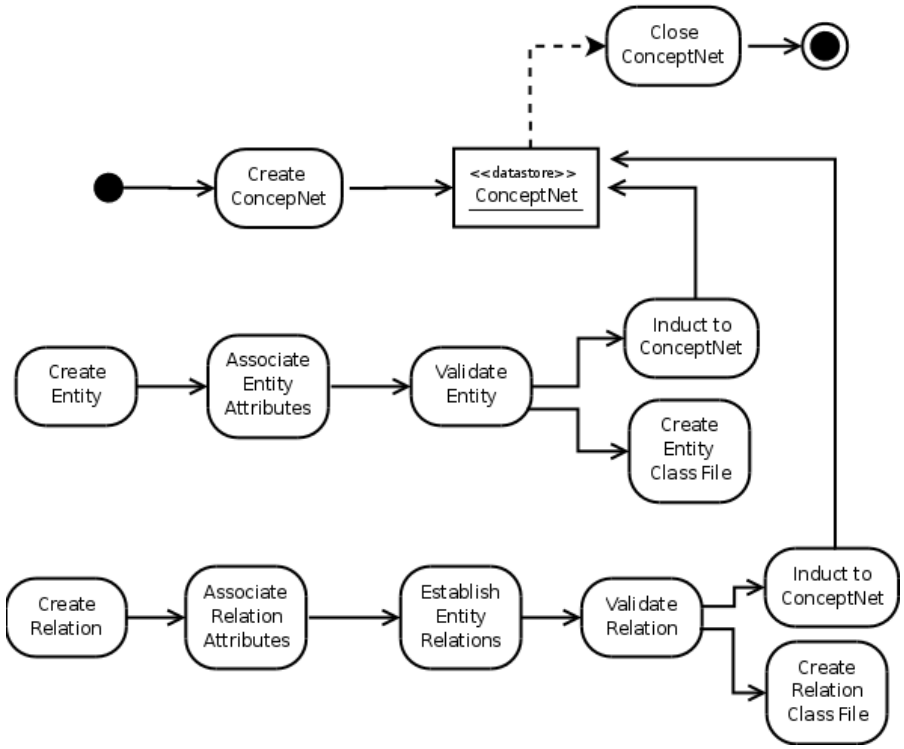

Figure 1.4: Inducting Entity and Relation Type into concept-net during modeling phase

1.2.2 Creating the Concept-Net

At storage level concept-net is set of files. The concept-net creation and management is similar to system file handling. The hierarchical structure of the storage and its elements leads to a naming convention similar to the standard file systems.

The class `ConceptNet` is responsible for the concept-net management. `ConceptNet` class abstracts and manages the *Type* and *Instance* in the concept-nets. It allows adding, updating, and removing the types and instances in the concept-nets. It also provides a high level retrieval support. Class `ConStore` provides *create* and *open* methods to get the $ConceptNet$ handle.

The following method call creates the concept-net in the specified path-name.

```
ConceptNet wordNet = ConStore.create("/usr/test/wordnet");
```

The static method `create` is called directly on the class `ConStore`. The return value of this call is the `ConceptNet` reference, named $wordNet$ in this example, which is the handle to the concept-net. The concept-net is created under a folder with name of concept-net specified by the user (here it is wordnet). This folder will hold the associated entities and relations of the concept-net. After that, the concept-net is opened in user specified

mode. The modes can be *Read* or *Write* similar to file handling modes. If the specified concept-net already exits then it throws appropriate exception.

1.2.3 Creating Entities

The meta-model reveals the entity types that need to be created. After creating the concept-net, the entities are created and inducted into concept-net. The `ConceptNet` class handles the concept-net modeling through the abstraction of *Type*. The `Entity` class extends the `Type` class as a specialization to model the concept nodes in the concept-net graph. An entity can be created directly using the `Entity` class and the below code shows the creation of `Noun` entity.

```
Entity noun = new Entity("Noun");
```

The `Entity` class constructs the entity through the constructor which takes entity name as the parameter.

1.2.4 Associating Attributes

An entity can have attributes (properties) and user might want to add attributes to the entity. The `Attribute` class holds the information about an attribute that need to be associated with the entity. The following code snippet shows the attribute creation:

```
Attribute word = new Attribute();
word.name = "word";
word.dataType = DataType.STRING;
word.repeating = false;
```

User can specify the attribute as single-valued or multi-valued (repeating) by setting the `repeating` attribute as true or false, by default it is false. User can specify the data types of String, Integer, Float, Double, Boolean, and Time to an attribute. The `DataType` class has the data types name as static constants. The constructors of `Attribute` class allow the user to construct attribute easily with the following methods.

```
Attribute(String name, int dataType);
Attribute(String name, int dataType, boolean repeating);
```

The `Type` class provides the interfaces for the attributes association. User can associate the attribute to the Entity using `addAttribute` method as shown.

```
Entity noun = new Entity("Noun");
Attribute word = new Attribute();
word.name = "word";
```

```
word.dataType = DataType.STRING;
word.repeating = false;
Result result = noun.addAttribute(word);
```

or simply user can add the attribute directly as shown.

```
Entity noun = new Entity("Noun");
Result result = noun.addAttribute("word", DataType.STRING);
```

The `Result` class is modeled to determine the status of an operation. It can also hold the information about the reason for failure of the operation along with the *Exception* information.

1.2.5 Creating Relations

Similar to entity, the `Relation` class extends the `Type` class, to support the relation modeling. A relation associates two entities with the specified roles. A Relation can be created directly using the `Relation` class using the following constructor.

```
Relation(String name, String leftRole, String rightRole);
```

The above method takes the parameters as relation name and respective role labels. The notion of left and right is meant for the user convenience to view the relation as the connection of two entities, one on left-side and other on right-side. Below code shows the creation of the *Hypernymy* relation which establishes the relation among *Noun* types.

```
Relation hypernymy = new Relation("Hypernymy","sub","super");
hypernymy.setLeft(noun);
hypernymy.setRight(noun);
hypernymy.setDirectionKind(Relation.LEFT_TO_RIGHT);
```

The `Relation` class constructs the relation through the constructor which takes the parameters, relation name (Hypernymy) with left-role name (sub) and right-role name (super). The methods `setLeft` and `setRight` sets the left and right entities of the relation. The direction of the relation can be specified using two values: 0 – *left-to-right*, and 1 – *bi-directional*. These values are the static constants in the `Relation` class.

1.2.6 Inducting Entities and Relations

Once the entities and relations are created as per the meta-model, they need to be inducted into concept-net. The `induct` method of the `ConceptNet` class lets the user to induct entities and relations. But at storage level, the concept-net is a network of *Types* and *Instances* and managed by the *ConceptNet* class.

```
Result induct(Type type);
```

Thus induct method takes the parameter of `Type` class. Since `Entity` and `Relation` classes inherit the `Type` class, the same method is used for inducting entities and relations as shown below:

```
//inducting entity
Result result = wordNet.induct(noun);
if(!result.Success)
    System.out.println(result.ErrMessage);

//inducting relation
wordNet.induct(hypernymy);

//commiting the changes
wordNet.commit();
```

The entity or relation name must be unique in the concept-net and inducting a type with same name will be unsuccessful. For the sake of the example (Figure 1.3), assume the entities – *Noun* and *Verb*, and the relations – *Hypernymy*, *Meronymy*, and *Use* are created and inducted into the concept-net. User need to invoke the *commit* method to persist the types to the secondary storage.

Once the entities and relations are inducted into concept-nets, a type-safe development is provided by generating the classes (.java files) for each entity and relation types under the concept-net folder structure as shown in Figure 1.5. The classes for entities and relations can be located under the folders '{*concept-net name*}*/types/entities/*' and '{*concept-net name*}*/types/relations/*'. User can import these classes into their development environment and use it seamlessly.

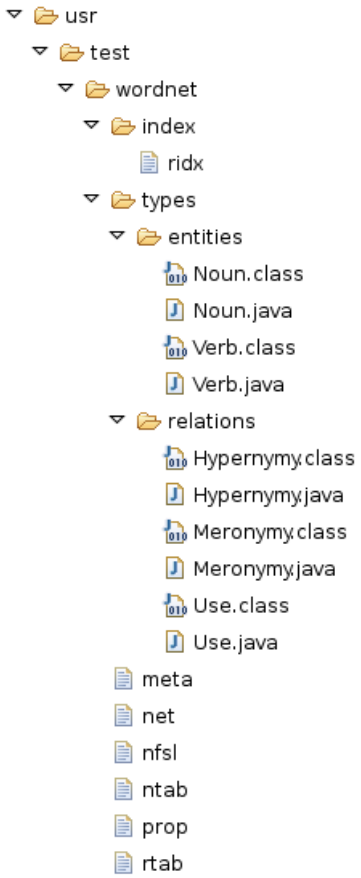

Figure 1.5: concept-net folder structure of word network example

1.2.7 Closing the concept-net

Since concept-net is handled similar to file, it is safe to close it. The concept-net can be closed using the `close` method of the concept-net handle, as shown below:

```
wordNet.close();
```

1.2.8 Instantiation

As shown in the lifecycle (Figure 1.1), the model may be created once the metamodel is in place. To construct the word network as shown in the example (Figure 1.2), user can instantiate the appropriate entities and relations of the meta-model. Figure 1.6 shows the activity of adding the Instance (entity and relation instance) during the instantiation phase.

1.2.9 Opening the Concept-Net

The concept-net need to be created only once and then the subsequent interactions happen by opening and closing the concept-net. The following method call opens an existing

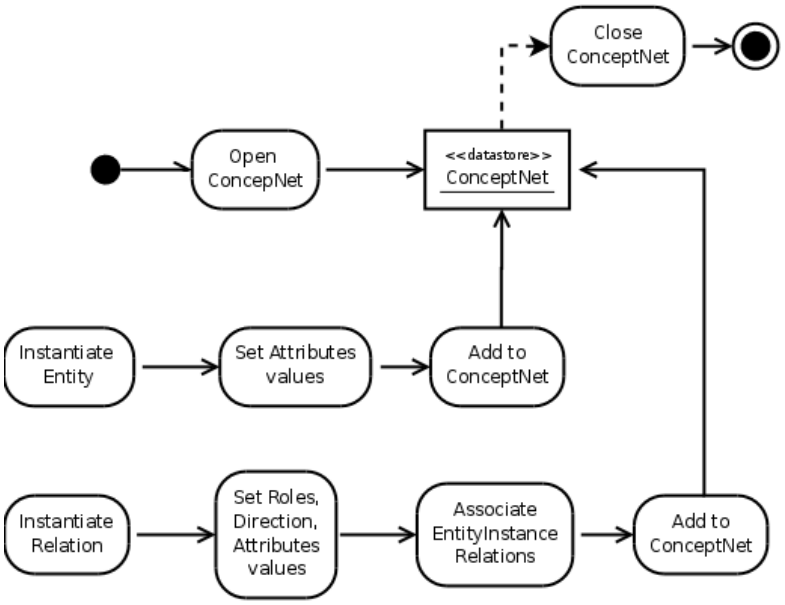

Figure 1.6: Adding Entity and Relation Instance into concept-net during instantiation phase

concept-net by taking the path-name and mode either read-only(r) or read-write(rw) as the parameter. The permissions are applied till the concept-net is closed for operations.

```
ConceptNet wordNet = null;
try {
      wordNet = ConStore.open("wordnet","rw");
    }catch (FileNotFoundException fe) {
      fe.printStackTrace();
    }
```

The static method `open` is invoked directly on the class `ConStore`, which returns the `ConceptNet` reference, named `wordNet` in this example. If the specified concept-net does not exists, then appropriate exception is thrown.

1.2.10 Instantiating Entities

To construct the word network as shown in the example (Figure 1.2), user can instantiate the appropriate entities and relations of the meta-model. For example, to model the *Animal* node in the word network example, user needs to create the instance of *Noun* entity and set its attributes. As mentioned in subsection 1.2.6, user can import entity

and relation types as Java classes into their development environment with the following *import* statements.

```
import types.entities.*;
import types.relations.*;
```

To expedite this, the following inheritance structure is used.

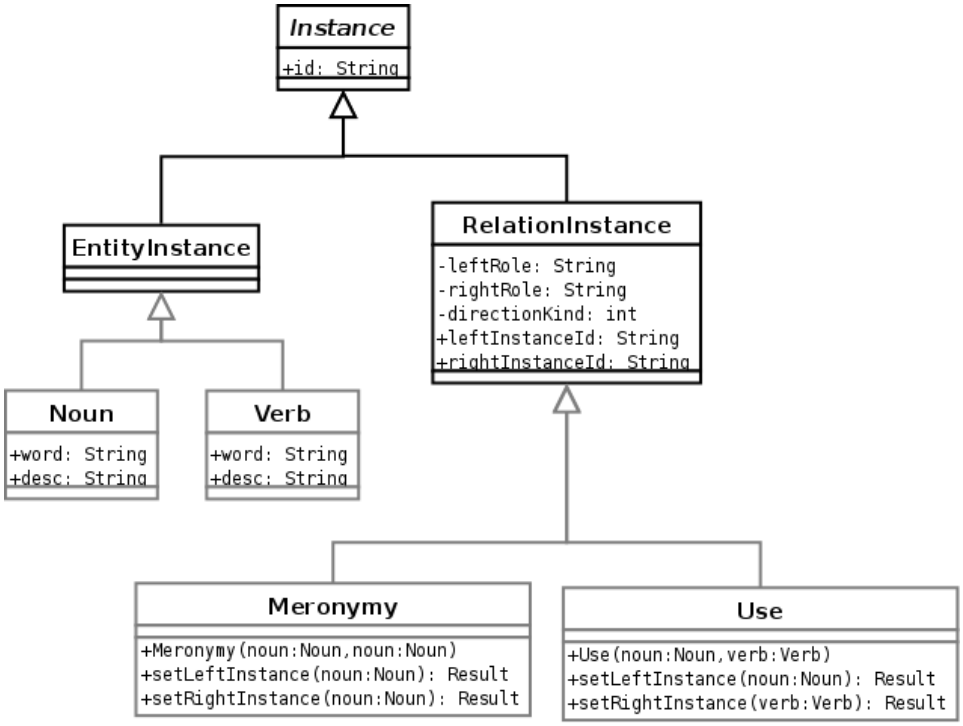

Figure 1.7: Class Diagram of showing the inheritance relation after the *Entity* and *Relation* types for word network are inducted into concept-net

In Figure 1.7, for the word network example, the `Noun` and `Verb` are entity types which inherit the *EntityInstance* class. Similarly, `Meronymy` and `Use` are relation types inherits the *RelationInstance* class. The respective .java files for the *Type* classes like Noun, Meronymy, are created while inducting the types into the concept-net. The entity's attributes are declared as public attributes in the respective Java class. This empowers the user to use these types directly into their development enabling type-safety. User can instantiate respective entity and set its attributes directly as shown below:

```
Noun animal = new Noun();
animal.word = "Animal";
animal.desc = "A Living Being";
```

1.2.11 Instantiating Relations

Similar to entity, relations can be instantiated directly from *Relation* class and set its attributes as shown below:

```
Noun human = new Noun();
Noun hand = new Noun();
Meronymy meronymy = new Meronymy(human, hand);
```

1.2.12 Adding Instances

The instantiated entities and relations are treated as *Instances* in the concept-net. The `add` method of the `ConceptNet` class enables to add the entity and relation instances to the concept-net, as shown below:

```
Result result = wordNet.add(animal);
if(!result.Success)
    System.out.println(result.ErrMessage);
```

To add the relation instance to the concept-net, the entities that associated with the relation need to be added first in the concept-net before adding the Relation, otherwise the operation will be unsuccessful.

```
wordNet.add(human);
wordNet.add(hand);
Result result = wordNet.add(meronymy);
if(!result.Success)
    System.out.println(result.ErrMessage);
```

1.2.13 Updating Instances

User can update the attribute values on entity and relation instances and update back to concept-net using the `update` method of `ConceptNet` class as shown below:

```
animal.desc = "A major group of multicellular, eukaryotic organisms";
Result result = wordNet.update(animal);
if(!result.Success)
    System.out.println(result.ErrMessage);
```

1.2.14 Removing Instances

Instances can be removed using the `remove` method of `ConceptNet` class as shown below:

```
wordNet.remove(animal);
wordNet.remove(meronymy);
```

1.2.15 Deleting the concept-net

The concept-net can be deleted by invoking the static method `delete` of `ConStore` class. The user has to pass the concept-net path name as the parameter.

```
ConStore.delete("/usr/test/wordnet");
```

Deleting a concept-net implies deleting the concept-net directory including its contents.

1.2.16 Type and Instance Retrieval

Type and instance retrieval using navigation interface (Section 1.1.2) is presented with code snippets in following subsections. The interface *Query* supports property or id-based retrieval of types and instances. Student concept network as shown in the Figure 1.8 is used for explanation.

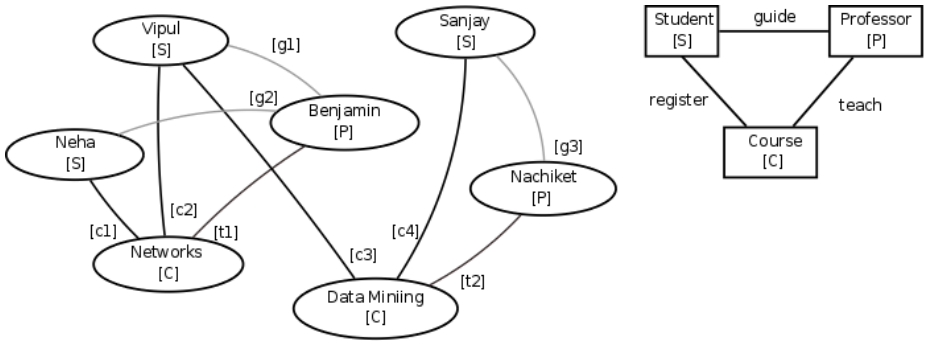

Figure 1.8: Student-Net

Type Retrieval

The type retrieval signifies the data retrieval from the metamodel. The below code snippet shows the type retrieval based on the type name.

```
Query query = studentNet.query();
Type type = query.getType("Student");
//returns the 'Student' type
```

The *getType* method is overloaded with *type-id*, where user can retrieve type based on *type-id*.

Instance Retrieval

Instances can be retrieved based on their property or id. The below code shows the property-based instance retrieval where a *student* is retrieved based on the *name*.

```
List<Instance> instances = query.getInstance("Student", "name", "Sanjay");
//returns the 'Sanjay' student instance
```

The *getInstance* method is overloaded with *instance-id*, where user can retrieve an instance based on *instance-id*. The *getEntityInstances* and *getRelationInstances* methods allow retrieving the entity and relation instance specifically based on their property or id. The relation instances associated with an instance can be retrieved using the *getRelations* method. For example, the below code demonstrates the retrieval of all the relations associated with the student Vipul.

```
List<RelationInstance> relInstances
           = query.getRelations(<instance-id of Vipual instance>);
//returns the g1, c2, and c3 relation instances
```

The above retrieval can be filtered by relation type name as shown below.

```
List<RelationInstance> relInstances
          = query.getRelations(<instance-id of Vipual instance>, "register");
//returns the c2, and c3 relation instances
```

The *getRelatedInstances* retireves the instances which are associated through the relations for the given instance-id or its property For example, the following code shows the retrieval of related instances student Neha.

```
List<EntityInstance> eInstances
       = query.getRelatedInstances(<instance-id of Neha instance>);
//returns the Network and Benjamin entity instances
```

The above retrieval can be filtered by relation type name as shown below.

```
List<EntityInstance> eInstances
      = query.getRelatedInstances(<instance-id of Neha instance>, "guide");
//returns the Benjamin entity instance
```

Bulk Retrieval

The bulk retrieval methods facilitates in the retrieving all the types and instances in the concept network in bulk or retrieving the instances in group based on their type. The methods *getAllEntites* and *getAllRelations* provide the retrieval of all entity and relation types in the metamodel of the concept network. Similarly, the methods *getInstances*, *getAllEntityInstances*, and *getAllRelationInstances* provide bulk retrieval of entity and relation instances of concept network. For example, the following code retrieves all the entity instances of the student-net.

```
List<Instances> allInstances
          = query.getAllEntityInstances();
//returns all the entities instances (Vipul, Neha, Snajay,
//Benjamin, Nachiket, Networks, Data Mining)
```

Entity and relation instances can be retrieved based on their type using the overloaded methods *getEntityInstances* and *getRelationInstances*. The following example shows retrieval of relation instances of type *guide*.

```
List<RelationInstance> relInstances
         = query.getRelationInstancs("guide");
//returns the g1, g2, and g3 relation instances
```

Id Retrieval

The id-based retrieval allows retrieving instance-ids based on their property which is helpful if user wants to navigate the concept network without retrieving the full instance. The *selectivedeserialization* technique explained in Section **??** is used to retrieve individual attribute's value without needing full instance deserialization. Table 1.2 listed the id-based retrieval methods. The following example demonstrates property-based instance id retrieval and further using *getInstance* method the actual instance can be retrieved if required.

```
List<Integer> instanceIds = query.getInstanceIds("Course", "name", "Networks");
//returns the 'Networks' course instance id as list

List<Instance> instances = query.getInstances(instanceIds);
//returns the 'Networks' course instance
```

Navigation through a concept-net

Using the id retrieval, user can navigate the concept-net. The following code retrieves and prints the value of property *name* of the instances of all entities related to the guides of a student with the value of property *name* as *vipul*.

```
List<Integer> instanceIds =
               query.getInstanceIds("student", "name", "vipul");
for(Integer sid : instanceIds) {
    List<Integer> guideIds =
               query.getRelatedInstanceIds(sid, "guide");
    for(Integer  gid: guideIds) {
         List<Instance> relInstances =
```

```
            query.getRelatedInstances(gid);
          for(Instance sinst: relInstances)
              System.out.print(sinst.getAttributeValue("name"));
    }
}

//outputs ‘Neha, Networks’
```