YODA Tutorial

David Cohen

March 30, 2015

Contents

Abstract

This tutorial introduces the YODA architecture at a high level, then walks through some of the basic features of the YODA dialog system architecture through an example Smart Home application.

1 What is YODA and Who Should Use it?

YODA is a spoken dialog system architecture - a piece of software to enable the creation of spoken dialog systems for various purposes, in various domains. Below are two check-lists that might help determine if YODA is right for you.

YODA may be good for you if some of these points describe your situation:

- You want to build an application where users can talk to a computer in a way where it remembers the context of what they've been talking about from turn to turn, allowing them to speak naturally and not specify every detail of what they mean.
- You do not have a large annotated data set of dialogs in your chosen domain / you do not have the money to buy one / you do not have the expertise to know what to collect / you do not have the time to wait while this all happens.
- Your domain has a dynamic set of objects about which your system needs to fluently converse.
- You want your system to be able to answer questions and follow commands related to the things it knows about in your domain.
- You want to have a working dialog system **fast**.
- You do not want a dialog system that evades questions it doesn't understand, acts like it understands things it doesn't, and tries to get the user to do all the work - like the classical Eliza therapist or a chat bot.

YODA, in its current state, may not be good for you if any of these things describe your situation:

- You don't trust any software tool that admits to requiring developers to "engineer knowledge" (YODA admits to this).
- You can't work with the GPL3 license.
- You have an unlimited fountain of money, data, and expertise which you plan to bring to bear on your specific dialog project.
- Your engineers (and their bosses) must exercise complete control over what the system says to create a user experience aligned with business objectives.
- You have some weird problem with the Java programming language.

2 YODA Overview

2.1 Logical Knowledge-base-centric Agents

A YODA dialog system is an agent in the tradition of other logical-knowledge-base-centric agents, as shown in figure 1. The agent acts upon the world, and updates its world model based on input from sensors.

YODA presupposes that dialog state tracking information and inferences are different enough from other types of inference and belief maintenance that they should be performed by separate

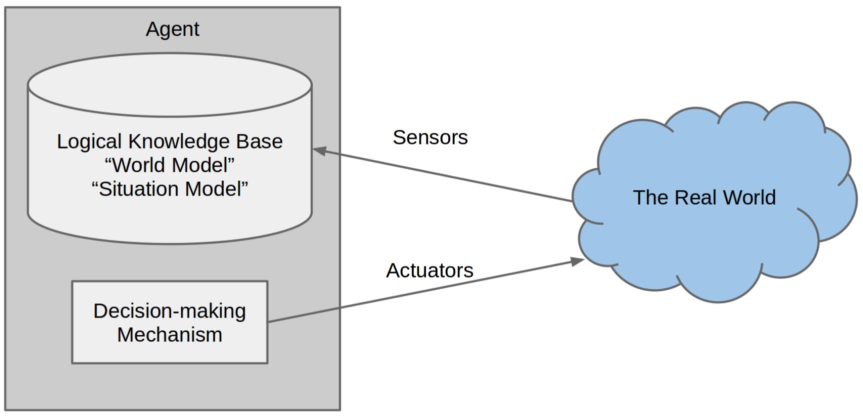


Figure 1: Logical KB-based system.

modules. Particularly, speech recognition and understanding are highly specialized sensing operations. Figure 2 shows a more accurate model of how YODA models its own position in the world. As is shown in the figure, YODA uses a separate representation for dialog state then it does for its world / situation (terms used interchangeably) model. The world model is a Sesame RDF datastore with forward-chaining inferencing. The dialog state model is set of Java objects describing a distribution of the possible ‘histories of things said’ by the user and system in the dialog up to the current moment.

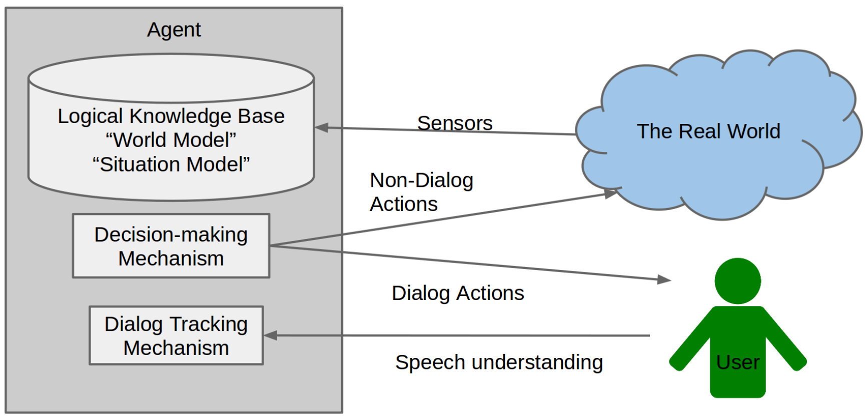


Figure 2: Separating the dialog component from a logical KB-based system.

2.2 Ontology as Architectural Glue

YODA is an ontology-driven architecture, which means it is able to discuss and interact to communicate information about world entities so long as the developer describes the relevant entities in a suitable knowledge structure a.k.a. ontology.

A YODA dialog system’s ontology acts as ‘architectural glue’ by specifying how all the other components can behave and interact with each other. A YODA system’s ontology does these things (and more):

- Language semantics (SLU valid outputs / NLG valid inputs) are restricted by the ontology.
- Dialog pragmatics modelling and inference (done by the Dialog State Tracking module) are restricted by the ontology.
- The system's world model (database) is limited by the ontology, and the ontology defines how that database should be interpreted.
- Actions considered by the dialog manager have parameters which are limited by the ontology.
- Interfaces to non-dialog tasks and sensors are built on the system ontology.

YODA provides a skeleton ontology, and a system engineer extends the ontology with classes that are specific to their domain. Domain ontology classes are defined as Java classes, as is detailed later in this tutorial.

2.3 Dialog with YODA

Many dialog systems operate by generating plausible statements and responses to keep the dialog going, in the spirit of the Eliza computer therapist or a chat-bot that attempts to pass the Turing Test. The purpose of these dialog systems tends to be to keep the user engaged, possibly while guiding them through a tree-like dialog structure to perform some task. Often, these systems have built-in strategies for ignoring the users when they ask things the system does not understand, and subtly directing the user back to the prescribed dialog tree. Supporting this type of dialog system is **not** the purpose of a YODA.

YODA attempts to enable dialog systems which actually understand their users within the scope of a domain, and respond in common helpful ways: by answering questions and obeying commands. Users will hopefully use dialog systems built with YODA because of their utility, naturalness of use, and because they are transparent and not misleading about their intelligence and capabilities.

3 Building a YODA System - Overview

Building a basic dialog system on top of the YODA architecture is a fairly simple process, consisting of the following steps:

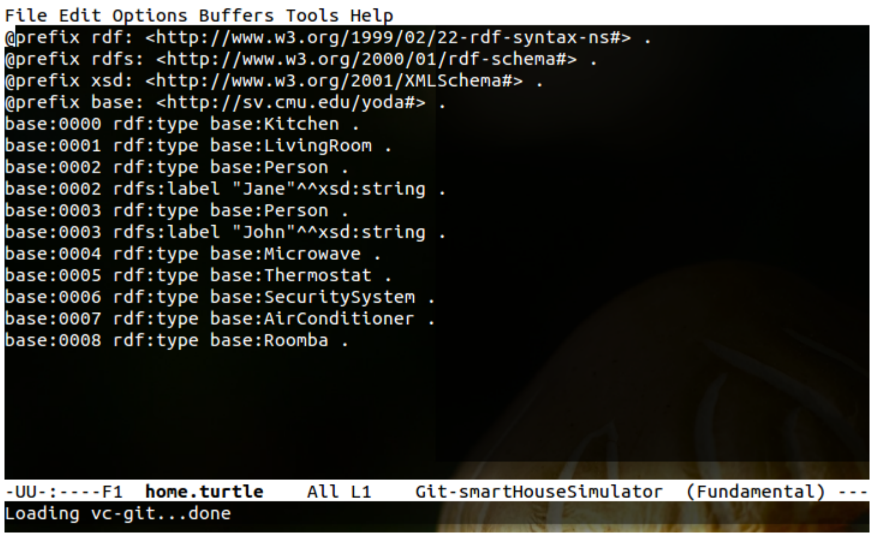
1. Build and register databases
2. Implement and register sensors
3. Implement and register non-dialog tasks
4. Build and register domain ontology
5. Provide a lexicon
6. Extend a YODA executable to include the new domain

This tutorial will explain the smart house demo system built on top of YODA. The purpose of this system is to allow a user to use, for example, a smart-phone app to stay informed about the state of their house and command various appliances remotely. The full source code for this system along with the source for several other demonstration systems, is available as part of YODA.

4 Build and Register Databases

YODA reads databases in rdf turtle format. Databases should capture the static / initial knowledge of a system. The contents of a registered database are loaded during system start-up, and the turtle file is never modified, but the facts listed in the database may change in the running system's working memory as a result of sensing.

Figure 3 shows the turtle file used to describe the initial smart house situation. There are a few appliances, rooms and people that make up the initial situation.



```
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .
@prefix base: <http://sv.cmu.edu/yoda#> .
base:0000 rdf:type base:Kitchen .
base:0001 rdf:type base:LivingRoom .
base:0002 rdf:type base:Person .
base:0002 rdfs:label "Jane"^^xsd:string .
base:0003 rdf:type base:Person .
base:0003 rdfs:label "John"^^xsd:string .
base:0004 rdf:type base:Microwave .
base:0005 rdf:type base:Thermostat .
base:0006 rdf:type base:SecuritySystem .
base:0007 rdf:type base:AirConditioner .
base:0008 rdf:type base:Roomba .
```

```
-UU-:----F1  home.turtle     All L1     Git-smartHouseSimulator  (Fundamental) ---
Loading vc-git...done
```

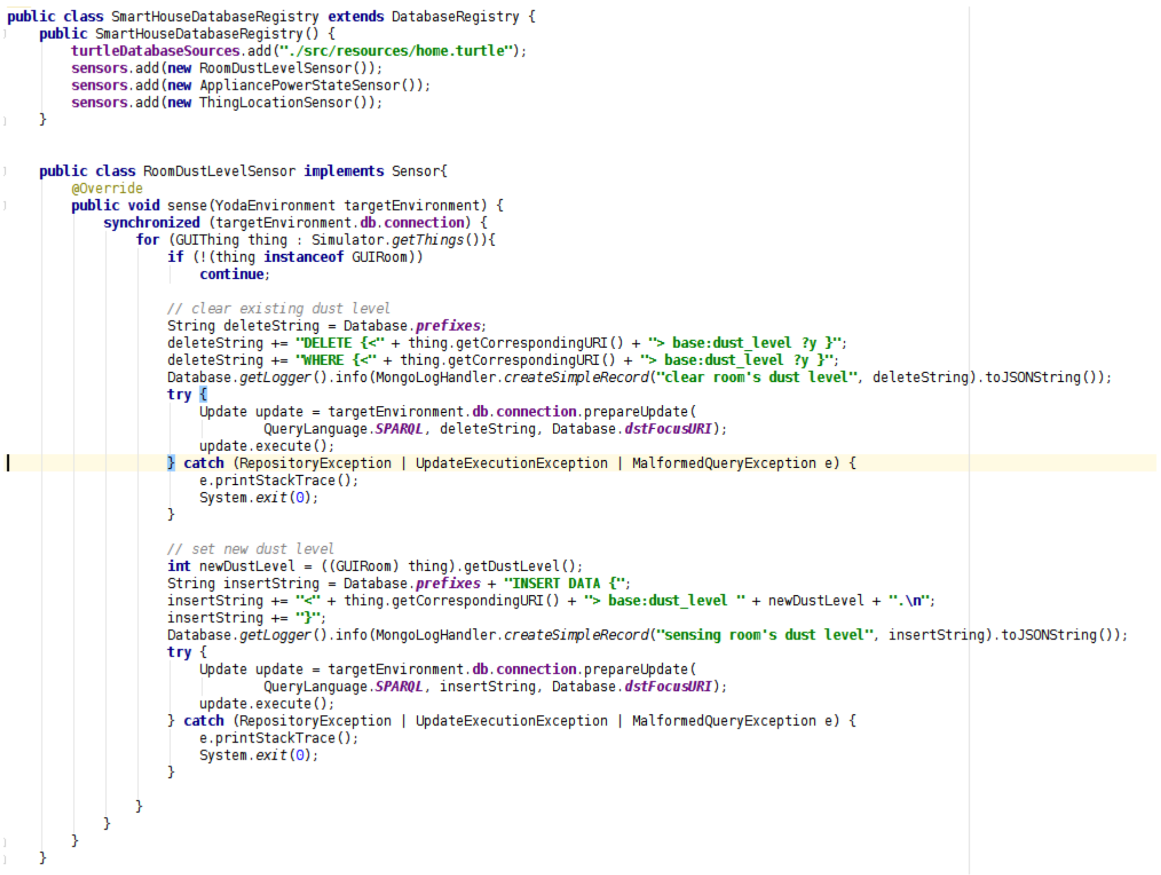
Figure 3: The rdf database file home.turtle describes the initial situation in the smart home demo system.

Some important things to remember while writing a database:

- Remember to include common prefixes in the header of your turtle file, as well as the yoda prefix (these are lines 1-4 in the example).
- Every 'entity' or 'individual' should have a unique identifier called a URI (Universal Resource Identifier). Most likely, these individuals should have a prefix of 'base:'. In the smart house example, there are 9 unique individuals, numbered 0000 to 0008. Don't confuse the URI with the 'name' or other information about an individual that may not be unique.
- Every entity should have a type, defined using the rdf:type predicate. If the system is supposed to be able to talk about this entity, it should be a type which inherits from the YODA skeleton ontology class Noun, and is registered in your ontology (see later section for how to do this). Examples in this turtle database file include base:Kitchen, base:LivingRoom, base:Person, base:Microwave, base:Thermostat, base:SecuritySystem, base:AirConditioner, and base:Roomba.
- Remember to register the database in the database registry file. This is demonstrated in the third line of code from Figure 4.

5 Build and Register Sensors

Sensors are an agent's continual inputs, which allow it to maintain an up-to-date model of the world's state. YODA provides a sensor interface which the developer can implement in their own system. Sensors are called at approximately 10Hz, soon the interface will be extended to allow for lower-period sensing.



```java
public class SmartHouseDatabaseRegistry extends DatabaseRegistry {
    public SmartHouseDatabaseRegistry() {
        turtleDatabaseSources.add("./src/resources/home.turtle");
        sensors.add(new RoomDustLevelSensor());
        sensors.add(new AppliancePowerStateSensor());
        sensors.add(new ThingLocationSensor());
    }


    public class RoomDustLevelSensor implements Sensor{
        @Override
        public void sense(YodaEnvironment targetEnvironment) {
            synchronized (targetEnvironment.db.connection) {
                for (GUIThing thing : Simulator.getThings()){
                    if (!(thing instanceof GUIRoom))
                        continue;

                    // clear existing dust level
                    String deleteString = Database.prefixes;
                    deleteString += "DELETE {<" + thing.getCorrespondingURI() + "> base:dust_level ?y }";
                    deleteString += "WHERE {<" + thing.getCorrespondingURI() + "> base:dust_level ?y }";
                    Database.getLogger().info(MongoLogHandler.createSimpleRecord("clear room's dust level", deleteString).toJSONString());
                    try {
                        Update update = targetEnvironment.db.connection.prepareUpdate(
                                QueryLanguage.SPARQL, deleteString, Database.dstFocusURI);
                        update.execute();
                    } catch (RepositoryException | UpdateExecutionException | MalformedQueryException e) {
                        e.printStackTrace();
                        System.exit(0);
                    }

                    // set new dust level
                    int newDustLevel = ((GUIRoom) thing).getDustLevel();
                    String insertString = Database.prefixes + "INSERT DATA {";
                    insertString += "<" + thing.getCorrespondingURI() + "> base:dust_level " + newDustLevel + ".\n";
                    insertString += "}";
                    Database.getLogger().info(MongoLogHandler.createSimpleRecord("sensing room's dust level", insertString).toJSONString());
                    try {
                        Update update = targetEnvironment.db.connection.prepareUpdate(
                                QueryLanguage.SPARQL, insertString, Database.dstFocusURI);
                        update.execute();
                    } catch (RepositoryException | UpdateExecutionException | MalformedQueryException e) {
                        e.printStackTrace();
                        System.exit(0);
                    }

                }
            }
        }
    }
```

Figure 4: An example from the smart house database registry, including an example sensor implementation which senses room dust level.

Figure 4 is an excerpt from SmartHouseDatabaseRegistry.java, the database and sensor registry for the smart house demo system. The current sensor interface requires the developer to directly update the system's database via SPARQL DELETE and INSERT operations. As the example illustrates, this typically involves deleting facts that are currently in the database, and replacing them with updated facts. The smart house demo system includes a home simulator, and the new facts are pulled from there; however, in a real system, these facts will be determined from various real-world sensors. This example includes the sensor for detecting room dust levels, but as is apparent, there are two other sensors registered: one to detect appliances' power states, and another to detect the location of things in the house.

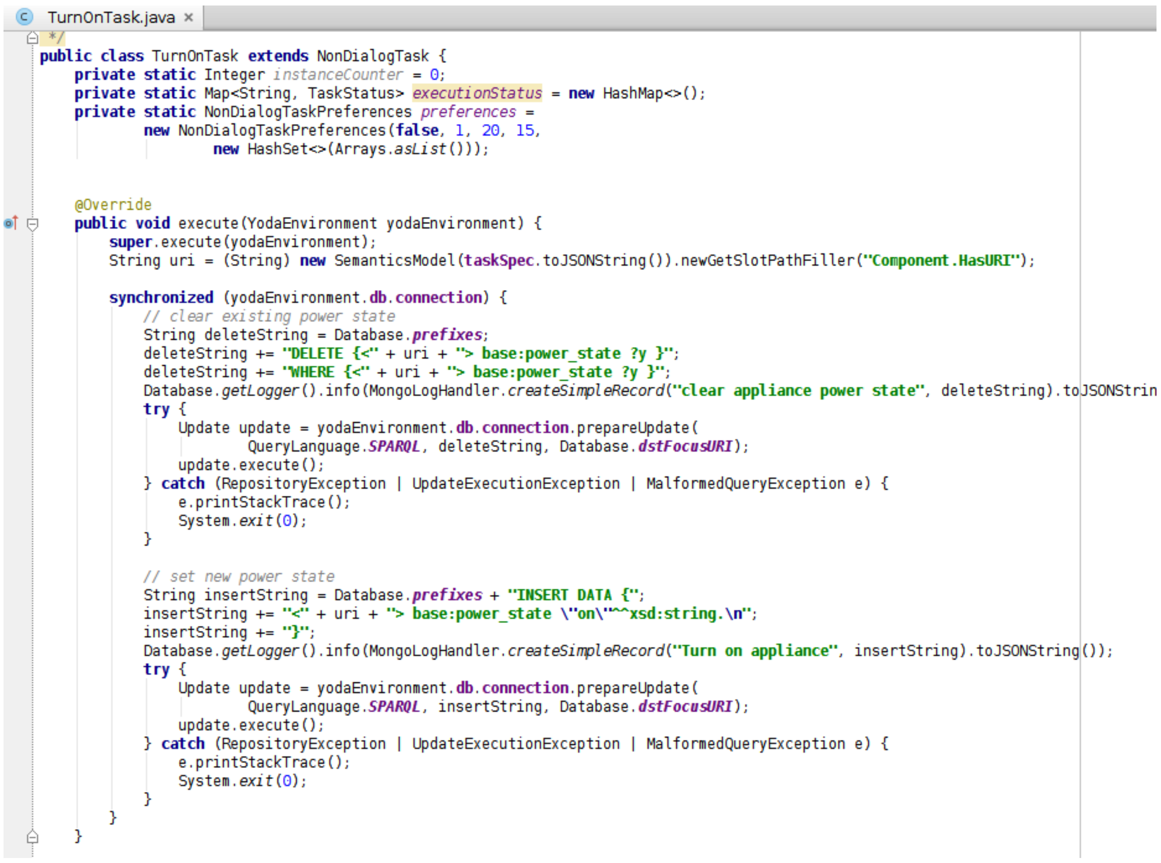
Some important things to remember while writing a sensor:

- Synchronize the database connection so that the system does not attempt to query the database contents while they're being changed.
- Delete old facts, then add new facts. Adding new facts that contradict will not delete old facts, it will create a contradictory world model.
- Register sensors in the database registry (add them to the "sensors" member variable).

6 Implement and Register Non-Dialog Tasks

TODO: UPDATE THIS SECTION (describe the simulator here)

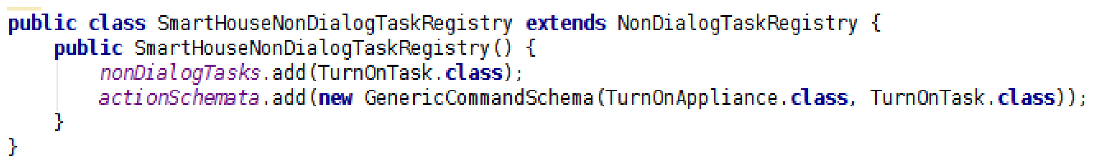
If a YODA dialog system is to do more than answer questions, the developer needs to implement some non-dialog tasks and register them with the system. Figure 5 shows the important parts of the implementation we use for this demo. The TurnOnTask non-dialog task 'turns on' an appliance by directly modifying the database. Of course, in the real world, these tasks should have real-world effects, and sensors should update the database, but this works for demonstration purposes.



```
*/
public class TurnOnTask extends NonDialogTask {
    private static Integer instanceCounter = 0;
    private static Map<String, TaskStatus> executionStatus = new HashMap<>();
    private static NonDialogTaskPreferences preferences =
            new NonDialogTaskPreferences(false, 1, 20, 15,
                    new HashSet<>(Arrays.asList()));


    @Override
    public void execute(YodaEnvironment yodaEnvironment) {
        super.execute(yodaEnvironment);
        String uri = (String) new SemanticsModel(taskSpec.toJSONString()).newGetSlotPathFiller("Component.HasURI");

        synchronized (yodaEnvironment.db.connection) {
            // clear existing power state
            String deleteString = Database.prefixes;
            deleteString += "DELETE {<" + uri + "> base:power_state ?y }";
            deleteString += "WHERE {<" + uri + "> base:power_state ?y }";
            Database.getLogger().info(MongoLogHandler.createSimpleRecord("clear appliance power state", deleteString).toJSONStrin
            try {
                Update update = yodaEnvironment.db.connection.prepareUpdate(
                        QueryLanguage.SPARQL, deleteString, Database.dstFocusURI);
                update.execute();
            } catch (RepositoryException | UpdateExecutionException | MalformedQueryException e) {
                e.printStackTrace();
                System.exit(0);
            }

            // set new power state
            String insertString = Database.prefixes + "INSERT DATA {";
            insertString += "<" + uri + "> base:power_state \"on\"^^xsd:string.\n";
            insertString += "}";
            Database.getLogger().info(MongoLogHandler.createSimpleRecord("Turn on appliance", insertString).toJSONString());
            try {
                Update update = yodaEnvironment.db.connection.prepareUpdate(
                        QueryLanguage.SPARQL, insertString, Database.dstFocusURI);
                update.execute();
            } catch (RepositoryException | UpdateExecutionException | MalformedQueryException e) {
                e.printStackTrace();
                System.exit(0);
            }
        }
    }
```

Figure 5: A sample non-dialog task - TurnOnTask turns on an appliance by directly modifying the database.

Figure 6 shows the class that registers TurnOnTask and uses the standard Command template to map the TurnOnAppliance verb to the task specification.



```java
public class SmartHouseNonDialogTaskRegistry extends NonDialogTaskRegistry {
    public SmartHouseNonDialogTaskRegistry() {
        nonDialogTasks.add(TurnOnTask.class);
        actionSchemata.add(new GenericCommandSchema(TurnOnAppliance.class, TurnOnTask.class));
    }
}
```

Figure 6: The smart home non-dialog task registry.

7 Build and Register a Domain Ontology

YODA provides a skeleton ontology, and a system developer should extend it for their own task. The major base classes are Noun, Verb, Role, TransientQuality, Adjective and Preposition, and some examples are given in the following sub-sections of extensions of these base classes for the smart house domain. Whenever you add a new class to the ontology, you need to tell the ontology registry that that class exists (it doesn't automatically add all the classes it can find, to allow for quicker experimentation with different ontology designs). The ontology registry for the smart house domain is shown in Figure 7.

An ontology registry for any other domain should include identical boilerplate code as this example, and the domain-specific classes should be added to the registry as part of the constructor. There is no need to re-register classes from the skeleton ontology, the skeleton ontology has its own registry, which will be included later.

Some advice when working on your ontology:

- Test as you go. Build related ontology items, lexicon entries, database entries and sensors together and test them together. Otherwise, there will inevitably be errors which are difficult to locate.

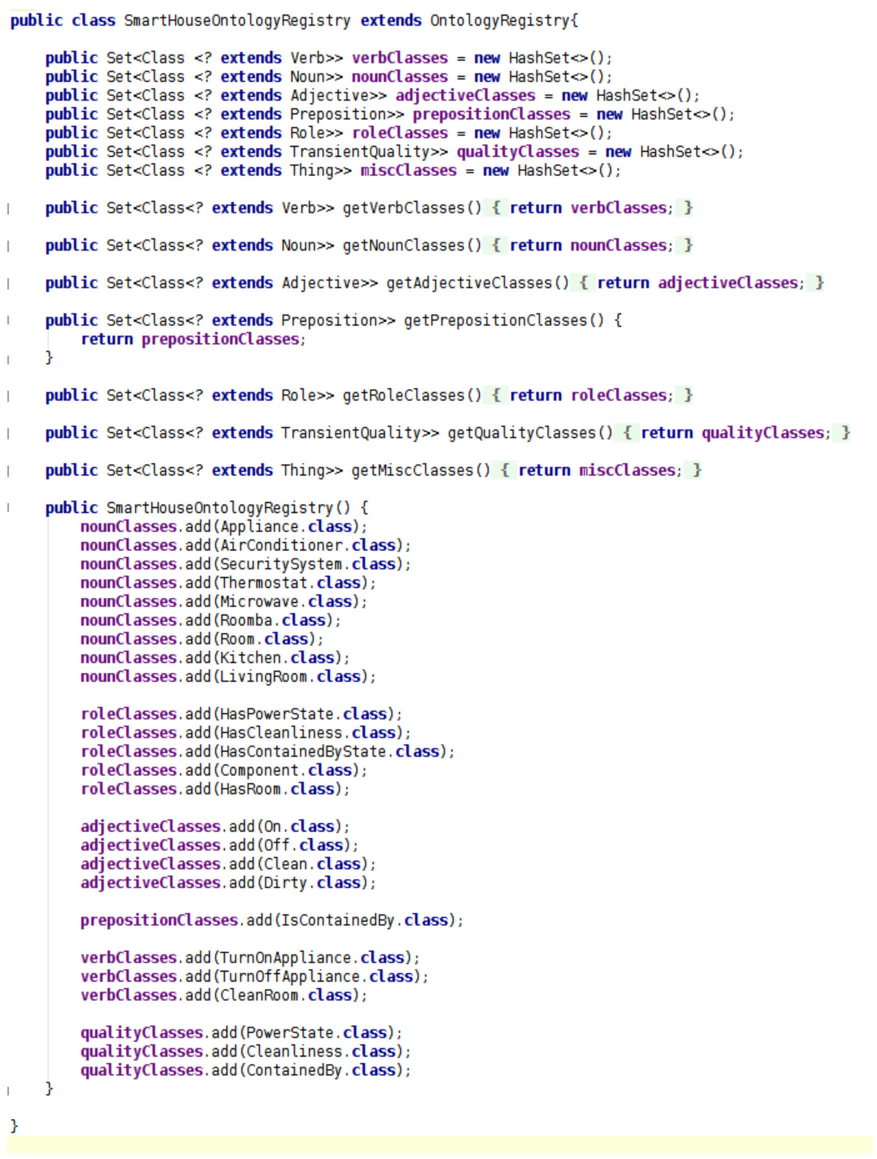
- Remember to **register your ontology classes** as you create them.

7.1 Noun

Nouns are the concepts which we usually describe as nouns in language - particularly the physical and abstract objects which make up a domain. There is not much work to do to define a noun, simply create a Java class that inherits from the appropriate parent within the Noun sub-tree. Two examples are shown in Figure 8. Appliance is a high-level class, which inherits from the YODA skeleton ontology's PhysicalNoun class. AirConditioner is a more specific appliance sub-class, so it inherits from Appliance.

7.2 Transient Qualities, Adjectives and Prepositions

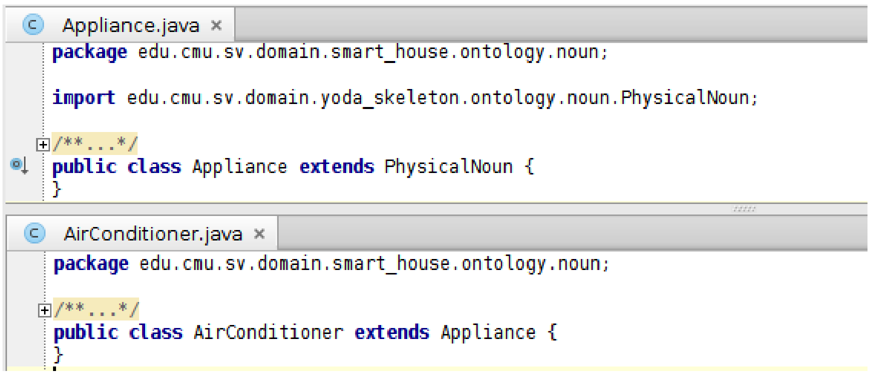
A dialog system needs to be able to determine to what degree an entity matches a description (reference resolution), and generate unambiguous descriptions to refer to objects (reference generation). For example, if the user asks the smart house dialog system "Is the kitchen dirty?", the dialog system needs to figure out what "the kitchen" is referring to, and whether that thing is dirty. Similarly, if



```java
public class SmartHouseOntologyRegistry extends OntologyRegistry{

    public Set<Class <? extends Verb>> verbClasses = new HashSet<>();
    public Set<Class <? extends Noun>> nounClasses = new HashSet<>();
    public Set<Class <? extends Adjective>> adjectiveClasses = new HashSet<>();
    public Set<Class <? extends Preposition>> prepositionClasses = new HashSet<>();
    public Set<Class <? extends Role>> roleClasses = new HashSet<>();
    public Set<Class <? extends TransientQuality>> qualityClasses = new HashSet<>();
    public Set<Class <? extends Thing>> miscClasses = new HashSet<>();

    public Set<Class<? extends Verb>> getVerbClasses() { return verbClasses; }

    public Set<Class<? extends Noun>> getNounClasses() { return nounClasses; }

    public Set<Class<? extends Adjective>> getAdjectiveClasses() { return adjectiveClasses; }

    public Set<Class<? extends Preposition>> getPrepositionClasses() {
        return prepositionClasses;
    }

    public Set<Class<? extends Role>> getRoleClasses() { return roleClasses; }

    public Set<Class<? extends TransientQuality>> getQualityClasses() { return qualityClasses; }

    public Set<Class<? extends Thing>> getMiscClasses() { return miscClasses; }

    public SmartHouseOntologyRegistry() {
        nounClasses.add(Appliance.class);
        nounClasses.add(AirConditioner.class);
        nounClasses.add(SecuritySystem.class);
        nounClasses.add(Thermostat.class);
        nounClasses.add(Microwave.class);
        nounClasses.add(Roomba.class);
        nounClasses.add(Room.class);
        nounClasses.add(Kitchen.class);
        nounClasses.add(LivingRoom.class);

        roleClasses.add(HasPowerState.class);
        roleClasses.add(HasCleanliness.class);
        roleClasses.add(HasContainedByState.class);
        roleClasses.add(Component.class);
        roleClasses.add(HasRoom.class);

        adjectiveClasses.add(On.class);
        adjectiveClasses.add(Off.class);
        adjectiveClasses.add(Clean.class);
        adjectiveClasses.add(Dirty.class);

        prepositionClasses.add(IsContainedBy.class);

        verbClasses.add(TurnOnAppliance.class);
        verbClasses.add(TurnOffAppliance.class);
        verbClasses.add(CleanRoom.class);

        qualityClasses.add(PowerState.class);
        qualityClasses.add(Cleanliness.class);
        qualityClasses.add(ContainedBy.class);
    }

}
```

Figure 7: SmartDomainOntologyRegistry.java, registering the domain ontology with YODA.

there are multiple entities in the discourse (as there usually are), and the system needs to describe one of them, it might select some quality which disambiguates its true referent from the other possibilities. To get the most out of YODA's support for this, a developer is required to implement transient qualities and the corresponding adjectives and prepositions that make up their domain.



```
package edu.cmu.sv.domain.smart_house.ontology.noun;

import edu.cmu.sv.domain.yoda_skeleton.ontology.noun.PhysicalNoun;

/**...*/
public class Appliance extends PhysicalNoun {
}
```

```
package edu.cmu.sv.domain.smart_house.ontology.noun;

/**...*/
public class AirConditioner extends Appliance {
}
```

Figure 8: Two Java classes to define two ontology Noun classes.

7.2.1 Overview

Adjectives and prepositions are 'grouped' by the quality which maps to them. For example, the adjectives 'cheap' and 'expensive' are grouped with the quality 'price range', and the prepositions 'close to' and 'far from' are grouped with the quality 'distance'. The basic relationship between qualities and adjectives / prepositions, along with the basic procedure for inferring the truth value of any one adjective / preposition are shown in Figure 9. The system's database encodes everything it needs to determine whether an adjective / preposition holds. The system runs a SPARQL query on the database to determine the quality's degree, then applies a piecewise linear map to determine the truth degree of the adjective / preposition.

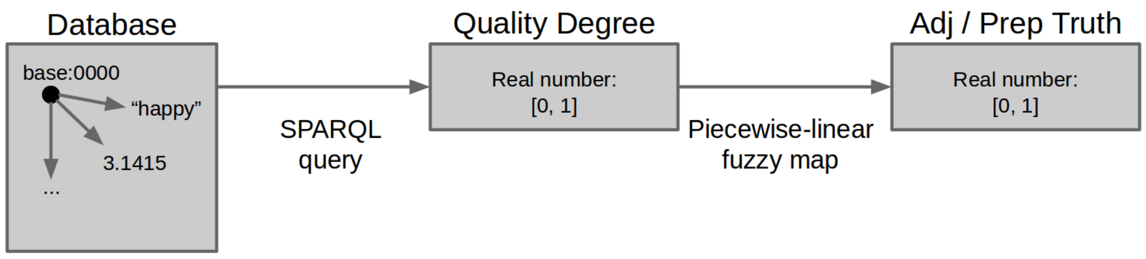


Figure 9: Diagram of the procedure to compute the truth value of some adjective w.r.t an entity or some preposition w.r.t. two entities.

Specifying the query to run is part of defining a custom TransientQuality in the ontology, while parameterizing the piecewise linear mapping is part of defining a custom Adjective / Preposition in the ontology.
To enable YODA to make use of a group of related adjectives or prepositions for reference resolution and generation, the developer needs to do the following:

- Extend TransientQuality to create a new quality, implement its getQualityCalculatorSPARQLQuery() method.
- Define an adjective or preposition 'group' by extending Adjective or Preposition (depending on whether your quality is unary or binary) with a new abstract class which will encapsulate

all the adjectives and prepositions in this group, and implement its getQuality() method.

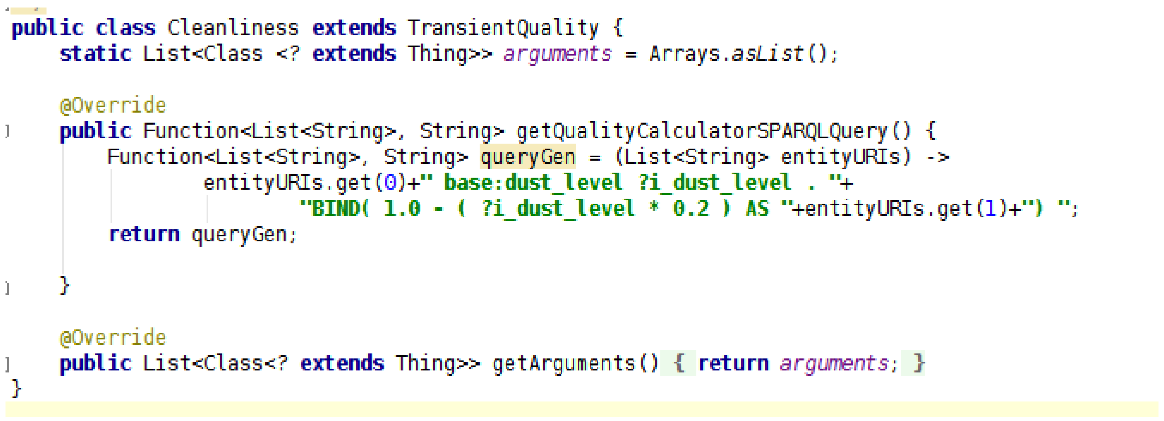
- Extend Adjective or Preposition and override getCenter() and getSlope() for each adjective / preposition in the group.
- Extend HasQualityRole with a new role and give it a domain and range appropriate for the new quality.
- Register all of these ontology classes!

7.2.2 Extending TransientQuality

Every custom TransientQuality will need to override the getQualityCalculatorSPARQLQuery(). This method returns a function which generates part of a SPARQL query given a list of arguments. If the quality is unary (for adjectives), there will be two arguments: the first argument is the entity who the quality is being determined for, and the last argument is the variable which should have the result bound to it. If the quality is binary (for prepositions), there will be three arguments: the first two are the entities which the quality is being determined for, and the last argument is the variable which should have the result bound to it.

The implementation of getQualityCalculatorSPARQLQuery() depends intimately on how the quality is sensed / encoded in the database. This subsection presents two examples which a large fraction of anticipated use cases will resemble. There are several other TransientQuality implementations included in the smart house and Yelp Phoenix demo systems, which you should use as additional references.

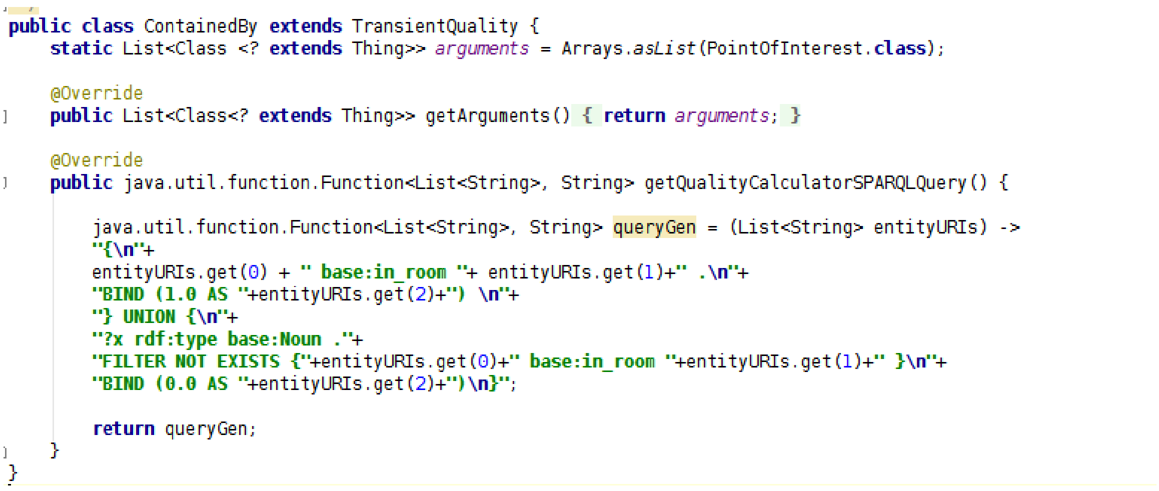
A simple example from the smart house domain is the Cleanliness unary quality, which supports the Clean and Dirty adjectives. Figure 10 shows the ontology definition of the Cleanliness quality. The smart home system has some way of sensing the dust level of rooms within the house, and it is encoded as an integer from 0 to 5 using the base:dust_level predicate. The SPARQL query in this example maps that to the range between 0 and 1, where 1 is higher cleanliness by taking $1.0 - 0.2 * d$ where d is the dust level. This way, if the dust level is 0, then Cleanliness is 1.



```
public class Cleanliness extends TransientQuality {
    static List<Class <? extends Thing>> arguments = Arrays.asList();

    @Override
    public Function<List<String>, String> getQualityCalculatorSPARQLQuery() {
        Function<List<String>, String> queryGen = (List<String> entityURIs) ->
                entityURIs.get(0)+" base:dust_level ?i_dust_level . "+
                        "BIND( 1.0 - ( ?i_dust_level * 0.2 ) AS "+entityURIs.get(1)+") ";
        return queryGen;

    }

    @Override
    public List<Class<? extends Thing>> getArguments() { return arguments; }
}
```

Figure 10: Ontology definition of the Cleanliness TransientQuality. This quality is later mapped to the Clean and Dirty adjectives. The definition includes a SPARQL query which determines a float value between 0 and 1, which is the "degree of cleanliness" for an input individual.

A more complicated example is the ContainedBy quality, which supports the preposition IsContainedBy (relating things to the room that contains them). The ontology definition of ContainedBy is shown in Figure 11. In this case, the quality can basically be determined as an objective fact from the database; the base:in_room predicate relates the thing to the room which contains it. Unfortunately, the SPARQL code to map this to 0 and 1 is a bit cryptic, which is why I've included the example. Since this is a binary quality, which relates two entities, the input list of entity URIs is longer than for a unary quality.



```
public class ContainedBy extends TransientQuality {
    static List<Class <? extends Thing>> arguments = Arrays.asList(PointOfInterest.class);

    @Override
    public List<Class<? extends Thing>> getArguments() { return arguments; }

    @Override
    public java.util.function.Function<List<String>, String> getQualityCalculatorSPARQLQuery() {

        java.util.function.Function<List<String>, String> queryGen = (List<String> entityURIs) ->
        "{\n"+
        entityURIs.get(0) + " base:in_room "+ entityURIs.get(1)+" .\n"+
        "BIND (1.0 AS "+entityURIs.get(2)+") \n"+
        "} UNION {\n"+
        "?x rdf:type base:Noun ."+
        "FILTER NOT EXISTS {"+entityURIs.get(0)+" base:in_room "+entityURIs.get(1)+" }\n"+
        "BIND (0.0 AS "+entityURIs.get(2)+")\n}";

        return queryGen;
    }
}
```

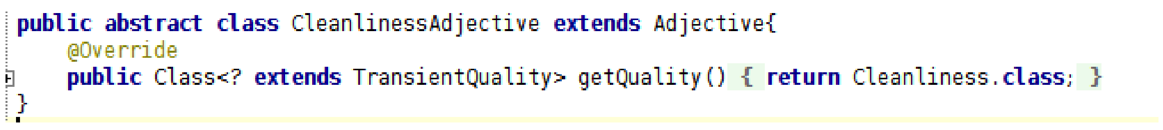
Figure 11: Ontology definition of the ContainedBy TransientQuality. This quality is later mapped to the IsContainedBy preposition, which relates an appliance or person to the room in the house that it is in.

Some notes for extending TransientQuality:

- Look through all the available example implementations to see if your use case closely resembles them.
- Use the sparql_playground git repository to edit a turtle file and try out your own queries in a simpler environment than a full-fledged YODA dialog system.
- Implement the static "arguments" boilerplate, which should be an empty list for unary qualities, and should be the type of Noun being related to for binary qualities. Also override the getArguments() method.

7.2.3 Defining Adjective / Preposition Groups

Frequently several related adjectives / prepositions are mapped from the same quality. In this case, it reduces required boilerplate to create a single abstract adjective / preposition class, and to make all the concrete classes inherit from it. Figure 12 shows the group class for the adjectives derived from the Cleanliness quality. Make sure to override the getQuality() method to return the appropriate quality class.



```java
public abstract class CleanlinessAdjective extends Adjective{
    @Override
    public Class<? extends TransientQuality> getQuality() { return Cleanliness.class; }
}
```

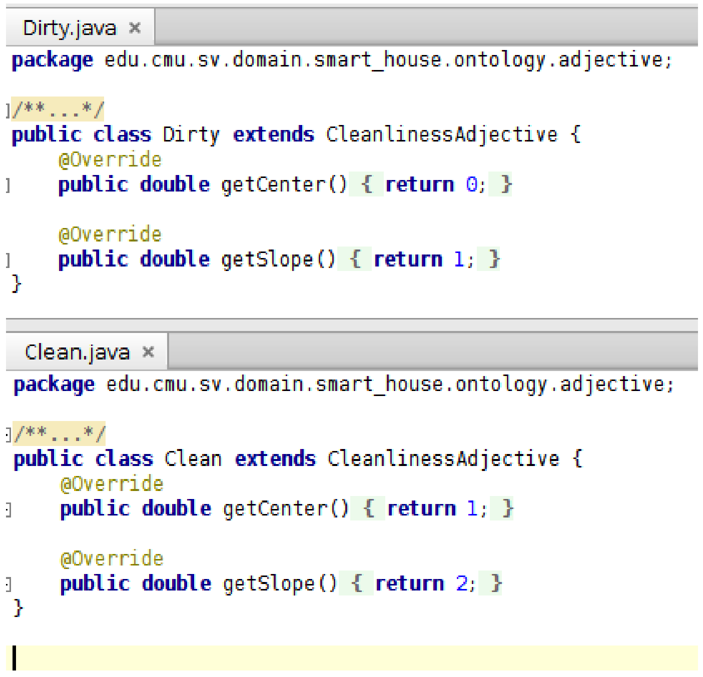
Figure 12: Ontology definition of the CleanlinessAdjective group. This is an abstract class, but still must be registered with the ontology.

7.2.4 Adding Custom Adjectives / Prepositions

Once the quality and group have been defined, all that needs to be done to define a custom adjective / preposition is:

- Extend the abstract group class with the custom adjective / preposition.
- Overwrite getCenter() and getSlope() to specify the linear fuzzy map for the given adjective / preposition relative to the group's quality.

The piecewise linear fuzzy map is a function which maps a quality degree to an adjective / preposition truth. It is a pyramid function with value 1 at the center, and sloping downward in both directions until bottoming out at zero. The resulting truths are also in the range [0,1].



Dirty.java ×

```java
package edu.cmu.sv.domain.smart_house.ontology.adjective;

/**...*/
public class Dirty extends CleanlinessAdjective {
    @Override
    public double getCenter() { return 0; }

    @Override
    public double getSlope() { return 1; }
}
```

Clean.java ×

```java
package edu.cmu.sv.domain.smart_house.ontology.adjective;

/**...*/
public class Clean extends CleanlinessAdjective {
    @Override
    public double getCenter() { return 1; }

    @Override
    public double getSlope() { return 2; }
}
```

Figure 13: Ontology definitions of Clean and Dirty. Notably, these inherit from the group's abstract adjective class, and include parameters for the piecewise linear map from Cleanliness.

Figure 13 shows the definitions of the Dirty and Clean adjectives. Notably, these inherit from the group's abstract adjective class, and include parameters for the piecewise linear map from Cleanliness. The fuzzy maps for these two adjectives are illustrated in Figure 14. As the figure shows,

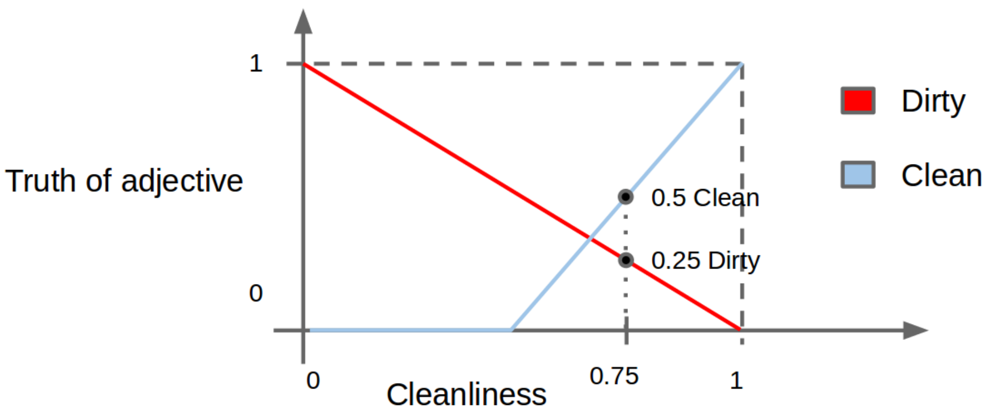
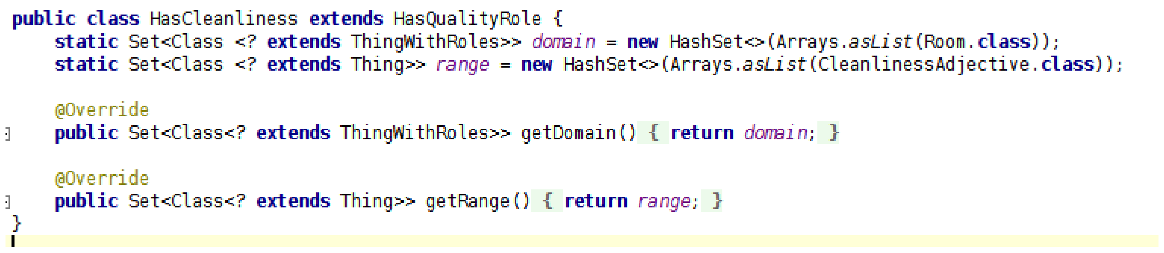


Figure 14: Piecewise-linear fuzzy mapping from the Cleanliness transient quality to the Clean and Dirty adjectives. When the degree of Cleanliness is .75, then Clean is .5 true, and Dirty is .25 true. Drawn in red and blue are the piecewise-linear mapping functions, which are parametrized by a center point and a slope downward from that center point.

Clean has a higher slope than dirty, and it will have a truth value of 0 for any Cleanliness value <= 0.5. The choice of center and slope are somewhat arbitrary, and they depend on what scale the quality itself has. Ideally, choosing parameters should be an empirical task based on measuring the quality when adjectives are used by people.

7.2.5 Adding Roles and Registering

Roles need to be added for completeness. For every adjective or preposition group, you need to implement a HasQualityRole that acts as a semantic connector between nouns and the group. An example for the Cleanliness adjective group is shown in Figure 15.



```
public class HasCleanliness extends HasQualityRole {
    static Set<Class <? extends ThingWithRoles>> domain = new HashSet<>(Arrays.asList(Room.class));
    static Set<Class <? extends Thing>> range = new HashSet<>(Arrays.asList(CleanlinessAdjective.class));

    @Override
    public Set<Class<? extends ThingWithRoles>> getDomain() { return domain; }

    @Override
    public Set<Class<? extends Thing>> getRange() { return range; }
}
```

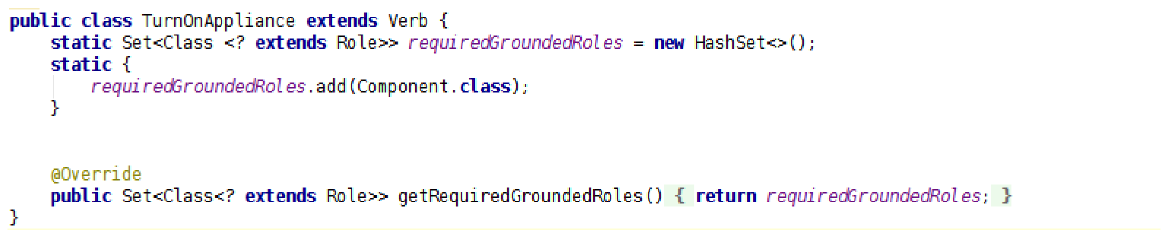
Figure 15: The HasCleanlinessRole is the semantic connector between nouns and cleanliness adjectives.

7.3 Role

Roles are relations between two entities

7.4 Verb

This domain defines one new verb TurnOnAppliance, which is used to issue commands. Figure 16 shows the class definition, which includes a list of required grounded roles. TurnOnAppliance is not considered semantically complete unless the Component role has been provided. This information is used to guide slot-filling dialog and to help the system decide when to ground certain noun phrases. In the non-dialog task registry, we later specify that the standard mapping is used to turn command dialog acts containing the TurnOnAppliance verb into the TurnOnTask non-dialog task. This is discussed in a later section.



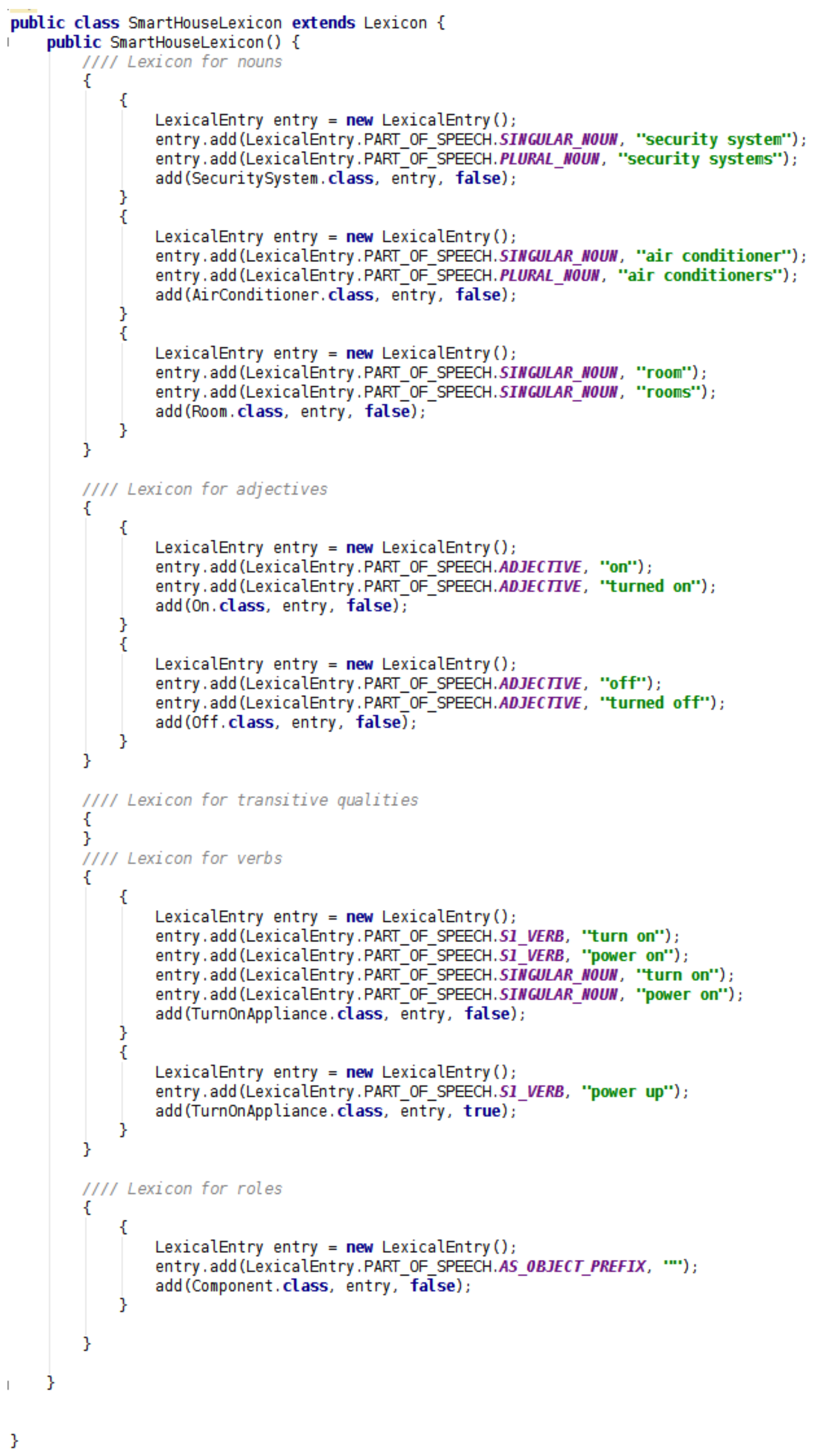
```
public class TurnOnAppliance extends Verb {
    static Set<Class <? extends Role>> requiredGroundedRoles = new HashSet<>();
    static {
        requiredGroundedRoles.add(Component.class);
    }


    @Override
    public Set<Class<? extends Role>> getRequiredGroundedRoles() { return requiredGroundedRoles; }
}
```

Figure 16: Definition for the TurnOnAppliance verb.

8 Provide a lexicon

TODO: UPDATE THIS SECTION

YODA's NLG and SLU components rely on a developer-provided lexicon, which provides words and phrases to represent the different ontology concepts in different parts of speech. Figure 17 shows the lexicon used in the smart home tutorial. Both the NLG and SLU components make use of the information in the lexicon. Often, we want our system to use formal words for generation, but to understand humans in whatever words they choose. Accordingly, we can add a word to the lexicon for understanding but not for generation by setting the third parameter of the Lexicon.add() function to 'true'. In this case, the term "power up" will be understood by the system, but not generated.



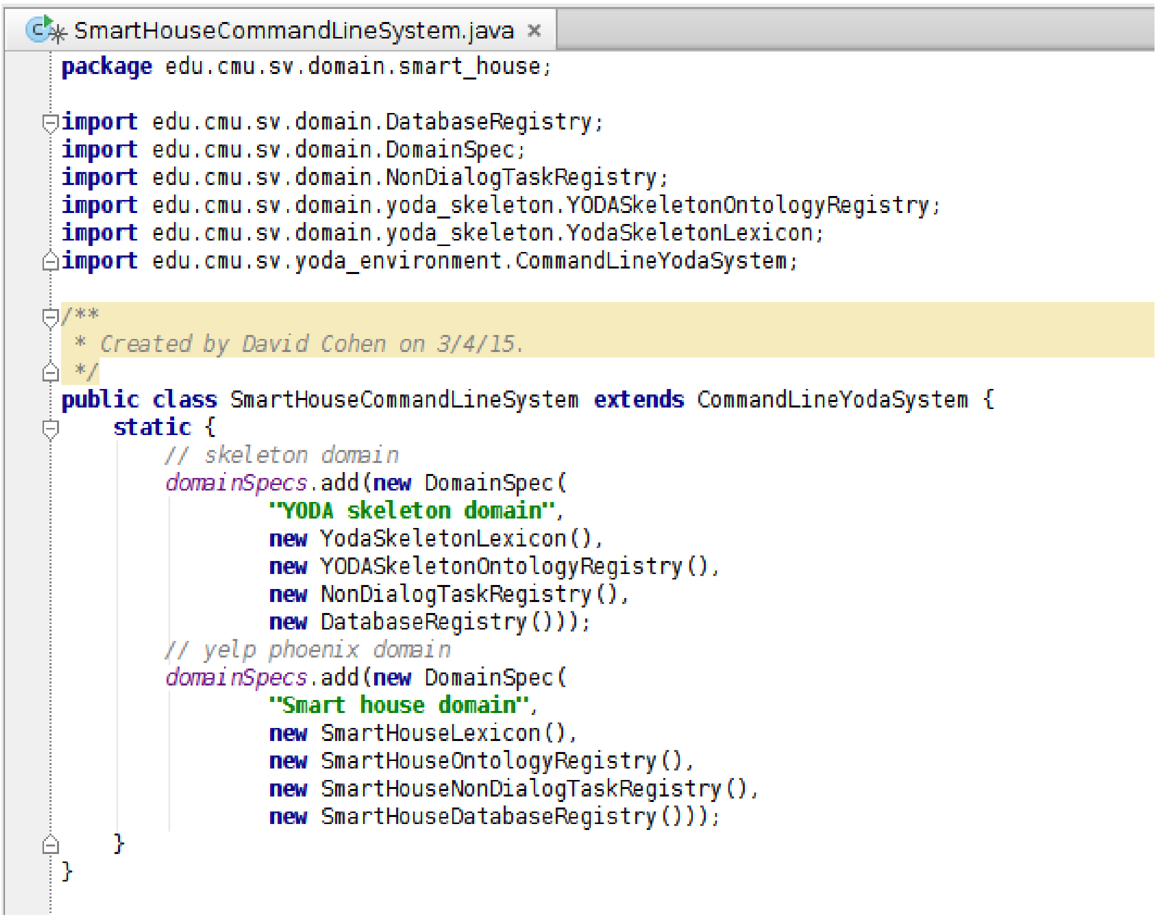
```java
public class SmartHouseLexicon extends Lexicon {
    public SmartHouseLexicon() {
        //// Lexicon for nouns
        {
            {
                LexicalEntry entry = new LexicalEntry();
                entry.add(LexicalEntry.PART_OF_SPEECH.SINGULAR_NOUN, "security system");
                entry.add(LexicalEntry.PART_OF_SPEECH.PLURAL_NOUN, "security systems");
                add(SecuritySystem.class, entry, false);
            }
            {
                LexicalEntry entry = new LexicalEntry();
                entry.add(LexicalEntry.PART_OF_SPEECH.SINGULAR_NOUN, "air conditioner");
                entry.add(LexicalEntry.PART_OF_SPEECH.PLURAL_NOUN, "air conditioners");
                add(AirConditioner.class, entry, false);
            }
            {
                LexicalEntry entry = new LexicalEntry();
                entry.add(LexicalEntry.PART_OF_SPEECH.SINGULAR_NOUN, "room");
                entry.add(LexicalEntry.PART_OF_SPEECH.SINGULAR_NOUN, "rooms");
                add(Room.class, entry, false);
            }
        }

        //// Lexicon for adjectives
        {
            {
                LexicalEntry entry = new LexicalEntry();
                entry.add(LexicalEntry.PART_OF_SPEECH.ADJECTIVE, "on");
                entry.add(LexicalEntry.PART_OF_SPEECH.ADJECTIVE, "turned on");
                add(On.class, entry, false);
            }
            {
                LexicalEntry entry = new LexicalEntry();
                entry.add(LexicalEntry.PART_OF_SPEECH.ADJECTIVE, "off");
                entry.add(LexicalEntry.PART_OF_SPEECH.ADJECTIVE, "turned off");
                add(Off.class, entry, false);
            }
        }

        //// Lexicon for transitive qualities
        {
        }
        //// Lexicon for verbs
        {
            {
                LexicalEntry entry = new LexicalEntry();
                entry.add(LexicalEntry.PART_OF_SPEECH.S1_VERB, "turn on");
                entry.add(LexicalEntry.PART_OF_SPEECH.S1_VERB, "power on");
                entry.add(LexicalEntry.PART_OF_SPEECH.SINGULAR_NOUN, "turn on");
                entry.add(LexicalEntry.PART_OF_SPEECH.SINGULAR_NOUN, "power on");
                add(TurnOnAppliance.class, entry, false);
            }
            {
                LexicalEntry entry = new LexicalEntry();
                entry.add(LexicalEntry.PART_OF_SPEECH.S1_VERB, "power up");
                add(TurnOnAppliance.class, entry, true);
            }
        }

        //// Lexicon for roles
        {
            {
                LexicalEntry entry = new LexicalEntry();
                entry.add(LexicalEntry.PART_OF_SPEECH.AS_OBJECT_PREFIX, "");
                add(Component.class, entry, false);
            }

        }

    }


}
```

Figure 17: The lexicon used to power the SLU and NLG in the smart home demo system.

9 Extend a YODA Executable

TODO: UPDATE THIS SECTION: include a more exciting example from the current system

The final step of creating a YODA dialog system is to extend one of the provided executable dialog systems to include the new domain. Figure 18 shows the executable which includes the smart house domain. All that is required in the executable is to extend a YODA base executable and add the yoda skeleton domain and the smart house domain in a static context.

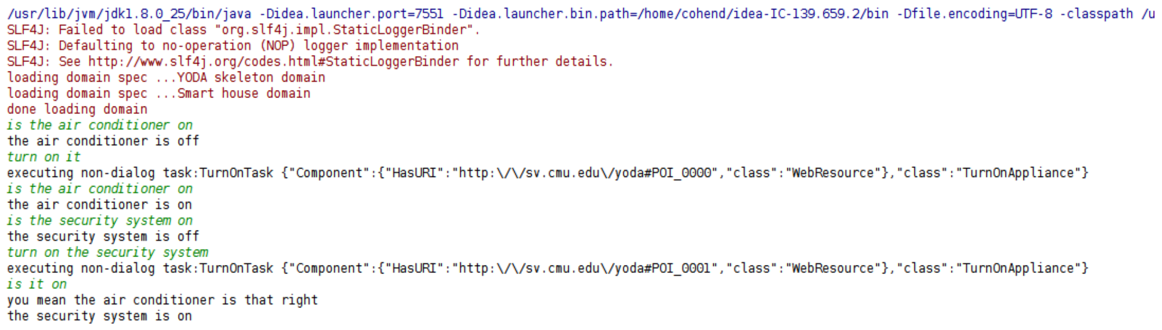


SmartHouseCommandLineSystem.java

```java
package edu.cmu.sv.domain.smart_house;

import edu.cmu.sv.domain.DatabaseRegistry;
import edu.cmu.sv.domain.DomainSpec;
import edu.cmu.sv.domain.NonDialogTaskRegistry;
import edu.cmu.sv.domain.yoda_skeleton.YODASkeletonOntologyRegistry;
import edu.cmu.sv.domain.yoda_skeleton.YodaSkeletonLexicon;
import edu.cmu.sv.yoda_environment.CommandLineYodaSystem;

/**
 * Created by David Cohen on 3/4/15.
 */
public class SmartHouseCommandLineSystem extends CommandLineYodaSystem {
    static {
        // skeleton domain
        domainSpecs.add(new DomainSpec(
                "YODA skeleton domain",
                new YodaSkeletonLexicon(),
                new YODASkeletonOntologyRegistry(),
                new NonDialogTaskRegistry(),
                new DatabaseRegistry()));
        // yelp phoenix domain
        domainSpecs.add(new DomainSpec(
                "Smart house domain",
                new SmartHouseLexicon(),
                new SmartHouseOntologyRegistry(),
                new SmartHouseNonDialogTaskRegistry(),
                new SmartHouseDatabaseRegistry()));
    }
}
```

Figure 18: SmartHouseCommandLineSystem.java is a fully-functioning command line dialog system in the smart house domain.

This creates a command-line dialog system, which takes words as input, and outputs words (and non-dialog tasks). An example dialog with this system is shown in Figure 19.



```
/usr/lib/jvm/jdk1.8.0_25/bin/java -Didea.launcher.port=7551 -Didea.launcher.bin.path=/home/cohend/idea-IC-139.659.2/bin -Dfile.encoding=UTF-8 -classpath /u
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
loading domain spec ...YODA skeleton domain
loading domain spec ...Smart house domain
done loading domain
is the air conditioner on
the air conditioner is off
turn on it
executing non-dialog task:TurnOnTask {"Component":{"HasURI":"http:\/\/sv.cmu.edu\/yoda#POI_0000","class":"WebResource"},"class":"TurnOnAppliance"}
is the air conditioner on
the air conditioner is on
is the security system on
the security system is off
turn on the security system
executing non-dialog task:TurnOnTask {"Component":{"HasURI":"http:\/\/sv.cmu.edu\/yoda#POI_0001","class":"WebResource"},"class":"TurnOnAppliance"}
is it on
you mean the air conditioner is that right
the security system is on
```

Figure 19: A sample dialog with the program SmartHouseCommandLineSystem.java.

Glossary

ASR - Automated Speech Recognition A computer program that takes raw audio signals as input and delivers a sequence of words as output. YODA does not provide an ASR component..

Domain The scope of things and their properties that are of interest for an application. In YODA, a domain is formalized through an ontology..

FOL - First Order Logic A formalism for describing a state of belief and concluding other beliefs through proof..

KB - Knowledge Base A portion of computer memory which contains a logical model of what a program knows..

Ontology A formal, written description of a domain. Typically this includes information such as a class hierarchy and the set and type restrictions of predicates. In YODA, a system's domain is specified with an ontology built by extending some core Java classes..

SLU - Spoken Language Understanding A computer program that takes a sequence of words as input and delivers some semantic representation as output. This can be a set of slot-value pairs (known as shallow SLU), or a structured logical representation (as in the SLU component in YODA)..

TTS - Text To Speech A computer program that takes a sequence of words as input and delivers audio of those words being spoken as output. This is often the last module in a spoken dialog system pipeline. YODA does not provide a TTS engine..