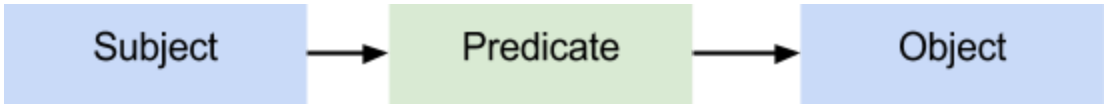
# Knowledge Graph Design Document

**Date: 8/30/2017**
Author: Eric Gieseke
Reviewers: Robert Zupko, Sytze Harkema

## Introduction

Knowledge graphs are an important method of capturing semantic information, and a core building block for the Semantic Web. Structuring information in semantic form makes it possible for automated agents to collect and process knowledge.

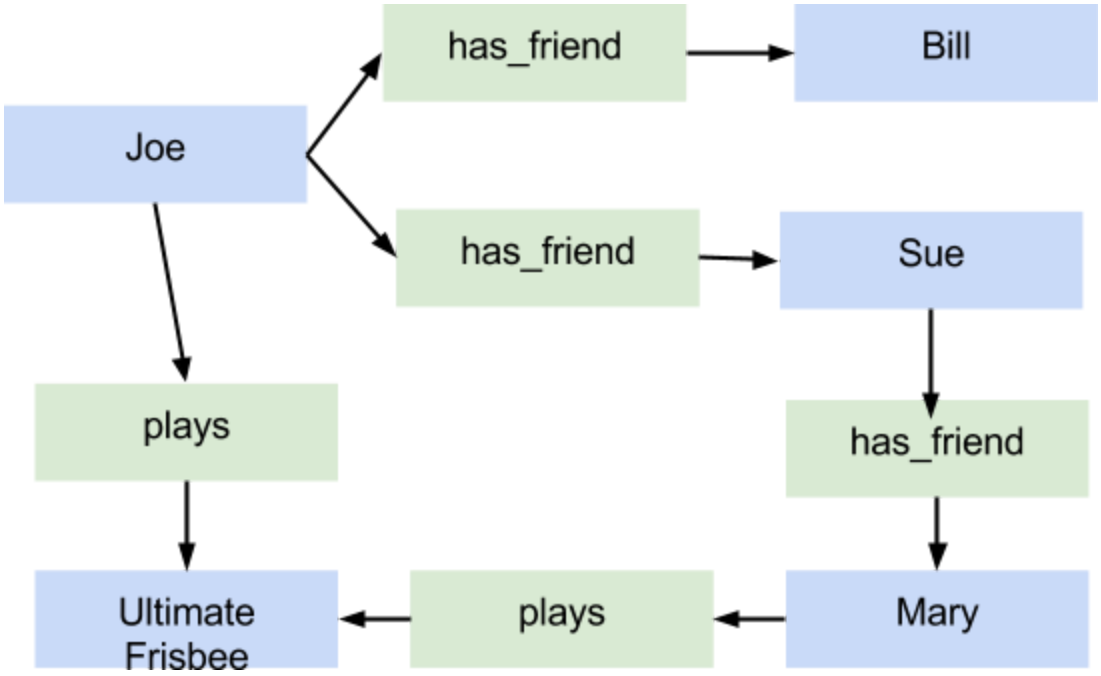
A knowledge graph is a graph of nodes, where a node can be a subject, object or both. The links between nodes are predicates. Predicates are properties of Subjects that connect Subjects to Objects.




An instance of a Subject, Predicate, and Object is referred to as a Triple. Here is an example Triple:


In this Triple, the Subject is Joe, the Predicate is has_friend, and the Object is Sue. The triple expresses a fact, and can be stated in English as “Joe has friend Sue”.

Many triples can be combined together to create a knowledge graph, where the Subjects and Objects of the graph overlap to form a lattice structure.




This example includes the Subjects: Joe, Sue, and Mary; Predicates: has_friend, and plays; and Objects: Bill, Sue, Mary, and Ultimate_Frisbee. This knowledge graph can be expressed in English, as Joe has friends Bill and Sue. Sue has friend Mary. Joe and Mary play Ultimate Frisbee.

Knowledge graphs can be extended indefinitely to include many facts from multiple sources. The graphs can contain cycles.

## Requirements

This section defines the requirements for the Knowledge Graph.

1) Read in one or more files of triples and build an in-memory knowledge graph. You can assume that the files are in a normalized form where the subjects, predicates, and objects are semantically equivalent if their identifiers are the same. Here is a sample triple file (in N-Triple format) that represents the graph above:

```
Joe has_friend Bill.
Joe has_friend Sue.
Sue has_friend Mary.
Joe plays Ultimate_Frisbee.
Mary plays Ultimate_Frisbee.
```

Each line from the file represents a Subject Predicate Object triple, space delimited and

terminated by a ".". Note that Subject, Predicate, and Object identifiers are case insensitive. The symbol "?" is a reserved keyword, and not allowed as an identifier for Subjects, Predicates, or Objects.

2) For each triple read from the input file, capture the corresponding Subject, Predicate, Object within an in-memory knowledge graph. Note that the in-memory knowledge graph should be structured in a form that can be used to support efficient queries on the knowledge graph. Persistence of the knowledge graph is not required. Capture the createDate for all Subject, Predicate, Objects and Triples as a unix time stamp.

3) Read a query file containing queries. For each query, process the query and output the query and the results. Queries will be in the form of triples, where the identifiers used to specify the Subject, Predicate, and Object can be either an identifier or the reserved symbol "?". When the "?" is specified rather than an actual identifier, this indicates a query field. The query processor should return all Triples that match the specified Subject, Predicate, Object that are not replaced with the "?".

The output should include the original query followed by the Set of all matching Triples (no duplicates). If there are no matching triples, return "<null>".

Here are some examples using the knowledge graph specified above:

input query:

```
Joe has_friend ?.
```

output:

```
Joe has_friend ?.
Joe has_friend Bill.
Joe has_friend Sue.
```

input query:

```
Joe ? ?.
```

Should output all known facts about Subject Joe.
output:

```
Joe ? ?.
Joe has_friend Bill.
Joe has_friend Sue.
Joe plays Ultimate_Frisbee.
```

input query:

```
? plays Ultimate_Frisbee.
```

Should determine all subjects that play ultimate frisbee.
output:

```
? plays Ultimate_Frisbee.
Joe plays Ultimate_Frisbee.
Mary plays Ultimate_Frisbee.
```

input query:

```
? ? ?.
```

Should output all known facts.
output:

```
? ? ?.
Joe has_friend Bill.
Joe has_friend Sue.
Sue has_friend Mary.
Joe plays Ultimate_Frisbee.
Mary plays Ultimate_Frisbee.
```

input query:

```
Joe has_friend Bill.
```

Should acknowledge the existence of the fact.
output:

```
Joe has_friend Bill.
Joe has_friend Bill.
```

input query:

```
Joe has_friend Roger.
```

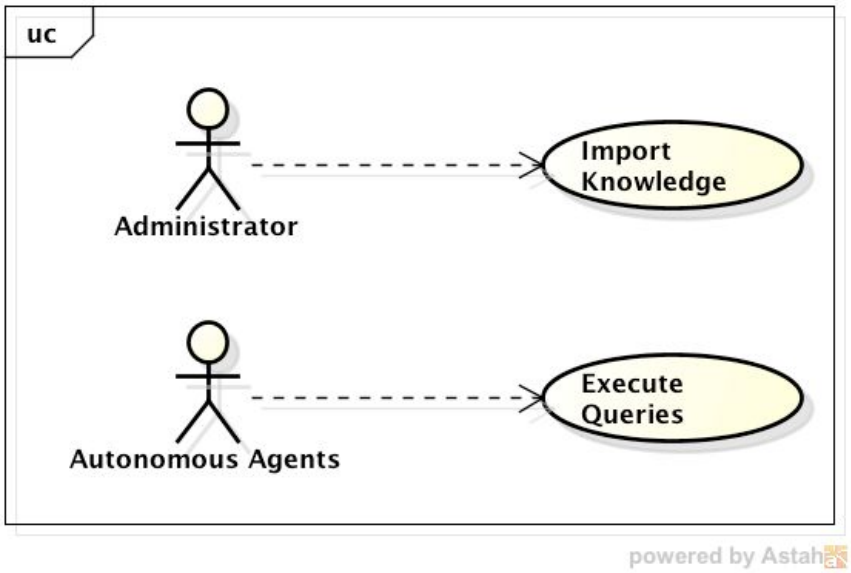
Should return null since there is no matching fact.
output:

```
Joe has_friend Roger.
<null>
```

# Use Cases

The Knowledge Graph supports 2 primary use cases:

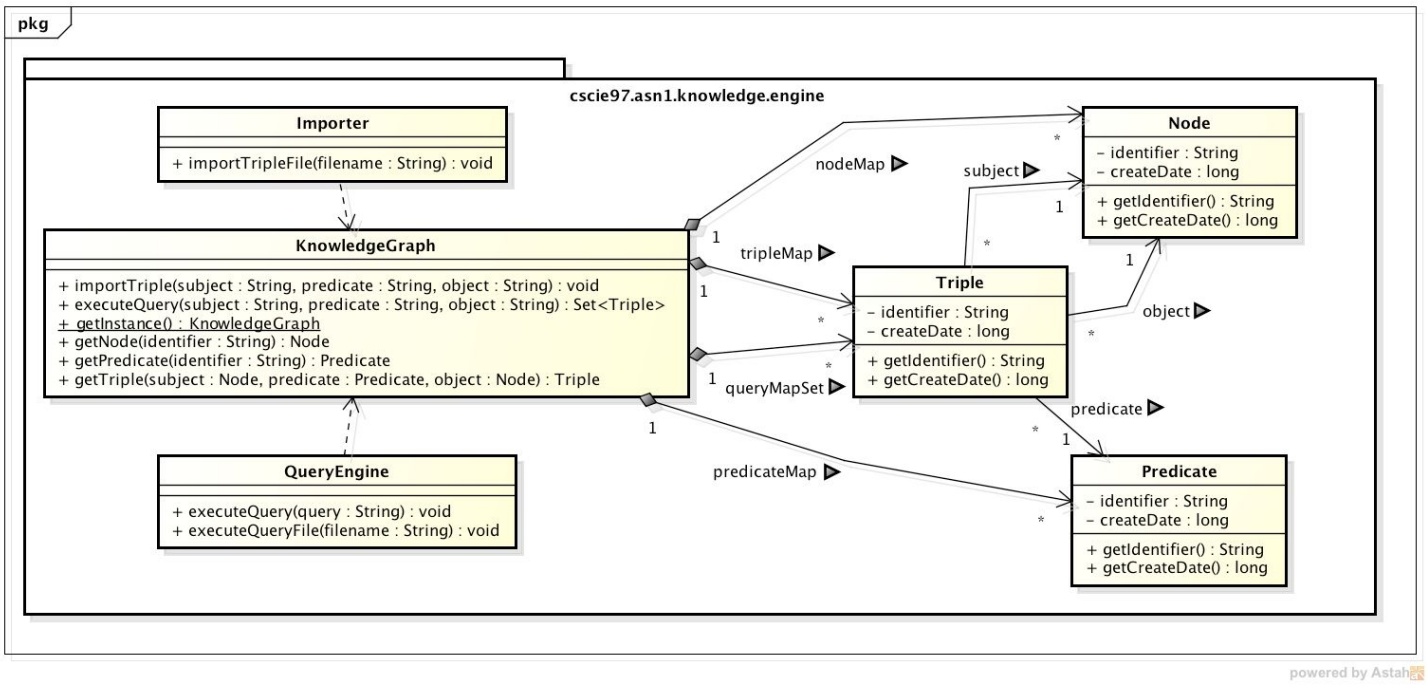
1. Administrators import Triples from formatted files into the Knowledge Graph.
2. Autonomous Agents execute queries on the Knowledge Graph.



# Implementation

## Class Diagram

The following class diagram defines the Knowledge Graph implementation classes contained within the package “cscie97.asn1.knowledge.engine”.




# Class Dictionary

This section specifies the class dictionary for the Knowledge Graph. The classes should be defined within the package “cscie97.asn1.knowledge.engine”.

## Importer

The Importer class is responsible for reading triples from input files using N-Triple format. The Importer class creates a Triple instance for each line read from the input file and passes the resulting Triples to the KnowledgeGraph.importTriples() method. Also, only fully qualified Triples (i.e. subject, predicate, object all have identifiers) should be added to the Knowledge Graph. Trim extra leading and trailing whitespace from identifier names. The importTripleFile method throws an ImportException on error processing the input file.

***Methods***

| Method Name | Signature | Description |
|---|---|---|
| importTripleFile | (fileName:String):void | Public method for importing triples from N_Triple formatted file into the KnowledgeGraph. Checks for valid input file name. Throws ImportException on error accessing or processing the input Triple File. |

## QueryEngine

The QueryEngine class supports the execution of Knowledge Graph queries. Queries are specified as Triples in N-Triple format with the special "?" identifier representing query or "wild card". All matching Triples known by the Knowledge Engine should be printed to stdout, preceded by the query string. The Query Engine supports 2 methods, one that accepts a single Query string, and another that supports a list of queries input from a file.

Malformed queries or problems accessing the input file should result in a QueryEngineException. The QueryEngineException should include the query that caused the exception, and some details about the reason.

***Methods***

| Method Name | Signature | Description |
|---|---|---|
| executeQuery | (query:String):void | Public method for executing a single query on the knowledge graph. Checks for non null and well formed query string. Throws QueryEngineException on error. |
| executeQueryFile | (fileName:String):void | Public method for executing a set of queries read from a file. Checks for valid file name. Delegates to executeQuery for processing individual queries. Throws QueryEngineException on error. |

## KnowledgeGraph

The KnowledgeGraph manages the set of active Triples. Per the requirements, the active Triples are assumed to fit within available memory, and no persistence is required.

The KnowledgeGraph is a singleton, meaning there is only one instance of this class. A special static method (getInstance()) is provided to access the single KnowledgeGraph instance. This follows the Singleton design pattern (see http://en.wikipedia.org/wiki/Singleton_pattern).

The importTriple() method supports importing a Triple (subject, predicate, object) into the KnowledgeGraph.

The executeQuery() method supports execution of queries against the knowledge graph. The Query is specified in the form of a tuple: subject, predicate, object. Occurrences of the "?" identifier within the Query can be supported by leaving the associated link (subject, predicate or object) as null within the Triple. The executeQuery() method returns a Set of matching triples. Triples are unique based on the combination of Subject, Predicate, Object. Per the requirements, only one instance of each matching triple should be returned.

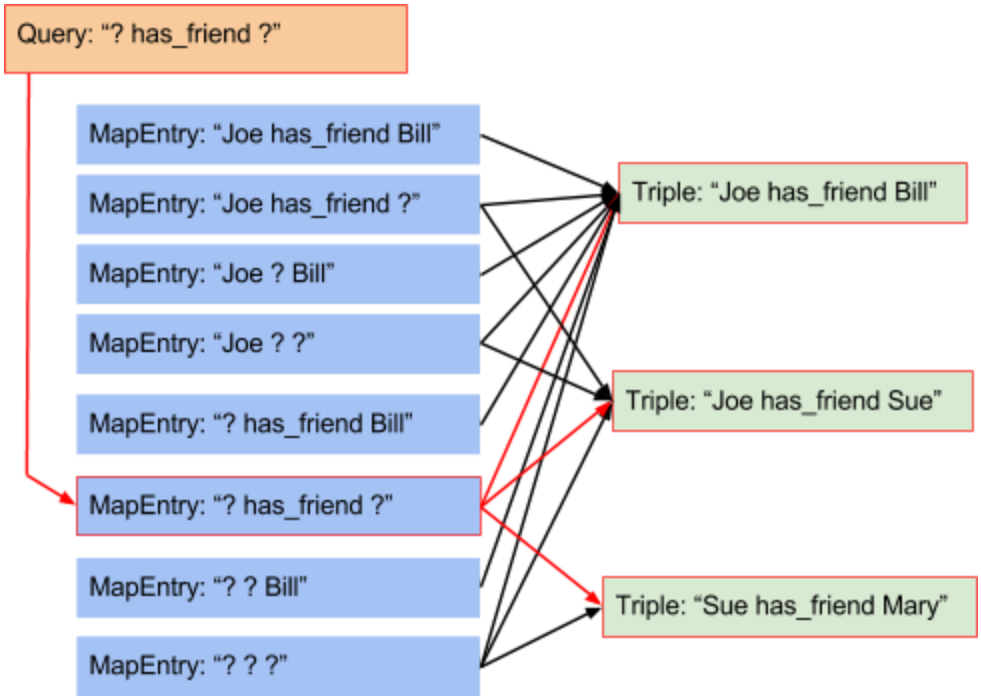
To improve query performance, an additional queryMapSet association is defined between the KnowledgeGraph and the Triples. As Triples are added to the Knowledge Graph, compute the permutations for all possible replacements of subject, predicate, object with "?". For each permutation add an entry to the queryMap if it does not already exists, and then add the Triple to the Triple Set for each of the queryMap entries.

In this way, all possible queries are pre-computed, resulting in O(1) query performance, due to the efficiency of the HashMap in looking up the MapEntry for the given Query. Simple brute force search for matching Triples would result in O(n) or worse performance.

For example, the Triple "Joe has_friend Bill", has the following possible variations.

"Joe has_friend Bill"
"Joe has_friend ?"
"Joe ? Bill"
"Joe ? ?"
"? has_friend Bill"
"? has_friend ?"
"? ? Bill"
"? ? ?"

The following diagram provides an example of how the query map is used to locate Triples that match the query “? has_friend ?”. In this case, any triples that include the predicate “has_friend” should be returned.




***Methods***

| Method Name | Signature | Description |
|---|---|---|
| importTriple | (subject:String, predicate: String, object:String):void | Public method for adding a Triple to the KnowledgeGraph. The following associations must be updated: nodeMap, tripleMap, queryMapSet, predicateMap to reflect the added Triple. There should be one Triple instance per unique Subject, Predicate, Object combination, so that Triples are not duplicated. |
| executeQuery | (subject:String, predicate: String, object:String):Set<Triple> | Use the queryMapSet to determine the Triples that match the given Query. If none are found return an empty Set. |
| getInstance | ():KnowledgeGraph | This method returns a reference to the single static instance of the KnowledgeGraph. |
| getNode | (identifier:String):Node | Return a Node Instance for the given node identifier. Use the nodeMap to look up the |

| | | |
|---|---|---|
| | | Node. If the Node does not exist, create it and add it to the nodeMap. Node names are case insensitive. |
| getPredicate | (identifier:String):Predicate | Return a Predicate instance for the given identifier. Use the predicateMap to lookup the Predicate. If the Predicate does not exist, create it and add it to the predicateMap. Predicate names are case insensitive. |
| getTriple | (subject:Node, predicate:Predicate, object:Node):Triple | Return the Triple instance for the given Object, Predicate and Subject. Use the tripleMap to lookup the Triple. If the Triple does not exist, create it and add it to the tripleMap and update the queryMapSet. |

***Associations***

| **Association Name** | **Type** | **Description** |
|---|---|---|
| nodeMap | Map<String, Node> | Private association for maintaining the active set of Nodes (i.e. Subjects and/or Objects). Map key is the node identifier and value is the associated Node. Node identifiers are case insensitive. |
| predicateMap | Map<String, Predicate> | Private association for maintaining the active set of Predicates. Map key is the predicate identifier and value is the associated Predicate. Predicate identifiers are case insensitive. |
| tripleMap | Map<String, Triple> | Private association for maintaining the active set of Triples. Map key is the Triple identifier and value is the associated Triple. |
| queryMapSet | Map<String, Set<Triple>> | Private association for maintaining a fast query lookup map. Map key is the query string (e.g. “Bill ? ?”), and value is a Set of matching Triples. |

## Triple

The Triple class represents a unique Triple (Subject, Predicate, Object) within the KnowledgeGraph. A Triple contains 3 references: Subject, Predicate and Object. The Triple is uniquely identified as the concatenation of the identifiers for the associated Subject, Predicate and Object. (e.g. “Joe has_friend Bill”)

***Methods***

| Method Name | Signature | Description |
|---|---|---|
| getIdentifier | ():String | Returns the Triple identifier. |
| getCreateDate | ():long | Returns the Triple creation date. |

***Properties***

| Property Name | Type | Description |
|---|---|---|
| identifier | String | Private unique non mutable identifier for the Triple. Of the form: subject.identifier + “ “ + predicate.identifer + “ “ + object.identifier. |
| createDate | long | Unix time stamp when triple was created |

***Associations***

| Association Name | Type | Description |
|---|---|---|
| subject | Node | Private non mutable association to the associated Subject instance. |
| predicate | Predicate | Private non mutable association to the associated Predicate instance. |
| object | Node | Private non mutable association to the associated Object instance. |

## Node

The Node class represents instances of Subjects and Objects. A Node has a unique String identifier (e.g. “Bill”, or “Ultimate_Frisbee”). Note that a single instance of a Node can represent both a Subject and an Object within the Knowledge Graph.

***Methods***

| Method Name | Signature | Description |
|---|---|---|
| getIdentifier | ():String | Returns the Node identifier. |
| getCreateDate | ():long | Returns the Node creation date. |

***Properties***

| Property Name | Type | Description |
|---|---|---|
| identifier | String | Private unique non mutable identifier for the Node. Node identifiers are case insensitive. |
| createDate | long | Unix time stamp when node was created. |

## Predicate

The Predicate class represents the predicate portion of a Triple. Like Node, the Predicate includes a unique String identifier that uniquely identifies the predicate (e.g. “has_friend”).

***Methods***

| Method Name | Signature | Description |
|---|---|---|
| getIdentifier | ():String | Returns the Predicate identifier. |
| getCreateDate | ():long | Returns the Predicate creation date. |

***Properties***

| Property Name | Type | Description |
|---|---|---|
| identifier | String | Private unique non mutable identifier for the Predicate. Predicate identifiers are case insensitive. |
| createDate | long | Unix time stamp when predicate was created. |

**Triple, Node, and Predicate Instance Management:**

Because of the in-memory nature of this implementation, to optimize memory usage, there

should only be one instance for each unique Triple, Node and Predicate object. This follows the FlyWeight design pattern (see http://en.wikipedia.org/wiki/Flyweight_pattern).

## Testing

Implement a test driver class called TestDriver that implements a static main() method. The main() method should accept 2 parameters, an input Triple file, and an Input Query file. The main method will call the Importer.importTripleFile() method, passing in the name of the provided triple file. After loading the input triples, the main() method will invoke the executeQueryFile() method passing in the provided query file name. The TestDriver class should be defined within the package “cscie97.asn1.test”.

## Risks

Because of the in memory implementation, the number of triples is limited by the memory allocated to the JVM.