Team Captain: John Wilde
Kaitaku Takeda
Nathaniel Suriawijaya
Noriaki Nakano

CMPS 109 Project Part 1: UML Design

Here are the assumptions and design decisions that we made for the UML design of the Simple Rule Based Engine. We began the design process by trying to abstract and modularize each separate component required for the SRI Engine. We decided that the engine would need to be built on two basic building block classes: Facts and Rules. Every Fact object has a name stored as a string as well as a list of strings that contains all of the parameters associated to that fact. Rule objects have a string member for the type of operation, either “OR” or “AND”. They also have strings for the left and right predicates of the rule.

We created a container class called RuleEngine that holds a vector for Rules and a vector for Facts. We may later create separate KB and RB classes to encapsulate these vectors. The RuleEngine class also contains all of the most important functions that the engine must carry out. We used vectors because we assumed that the user can input multiple facts and rules.

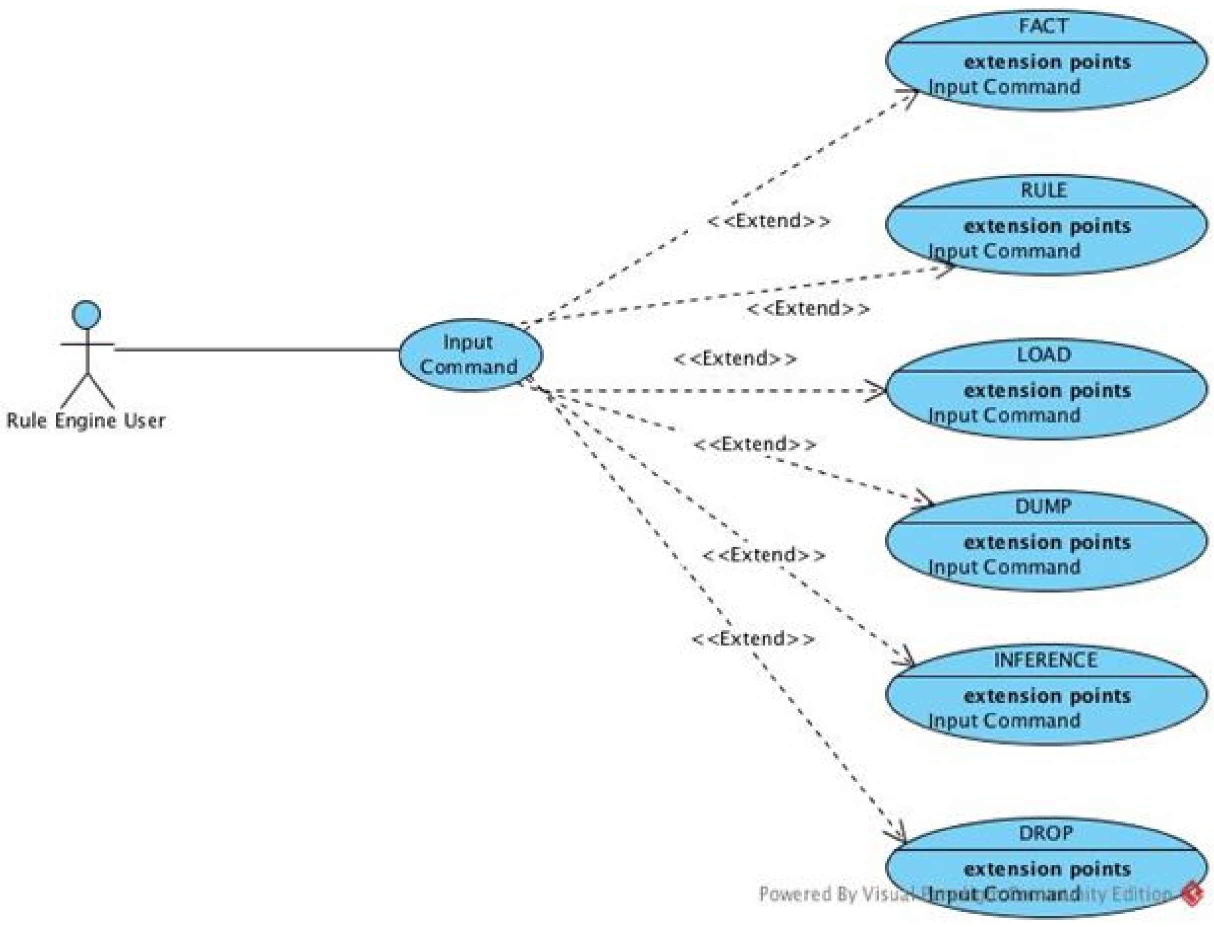
For now, we assume that the program will start in main(), and main() will have the ability to parse for command line arguments to check if the user wants to start from a .sri file or start with an empty KB and RB. If the user chooses to input commands at runtime, main() will wait for input, then call the parseInput() function after the user finishes a line of input. The parseInput() function is called via the main(), which will parse for valid user commands (FACT, RULE, etc.) and carry out the proper functions based on the output. We assume it will parse line by line, therefore it will parse a line, carryout the execution or return an error message, and then begin parsing the next line.

We made the assumption that facts can have any number of parameters. Similarly, rules can have any number of parameters. Therefore, we designed our Fact class constructor and our Rule class constructor to be variadic. We also assumed that user’s input valid command line argument. For invalid commands, the program will throw an error message. We have a util class for possible use in the future, but as of now it is empty.

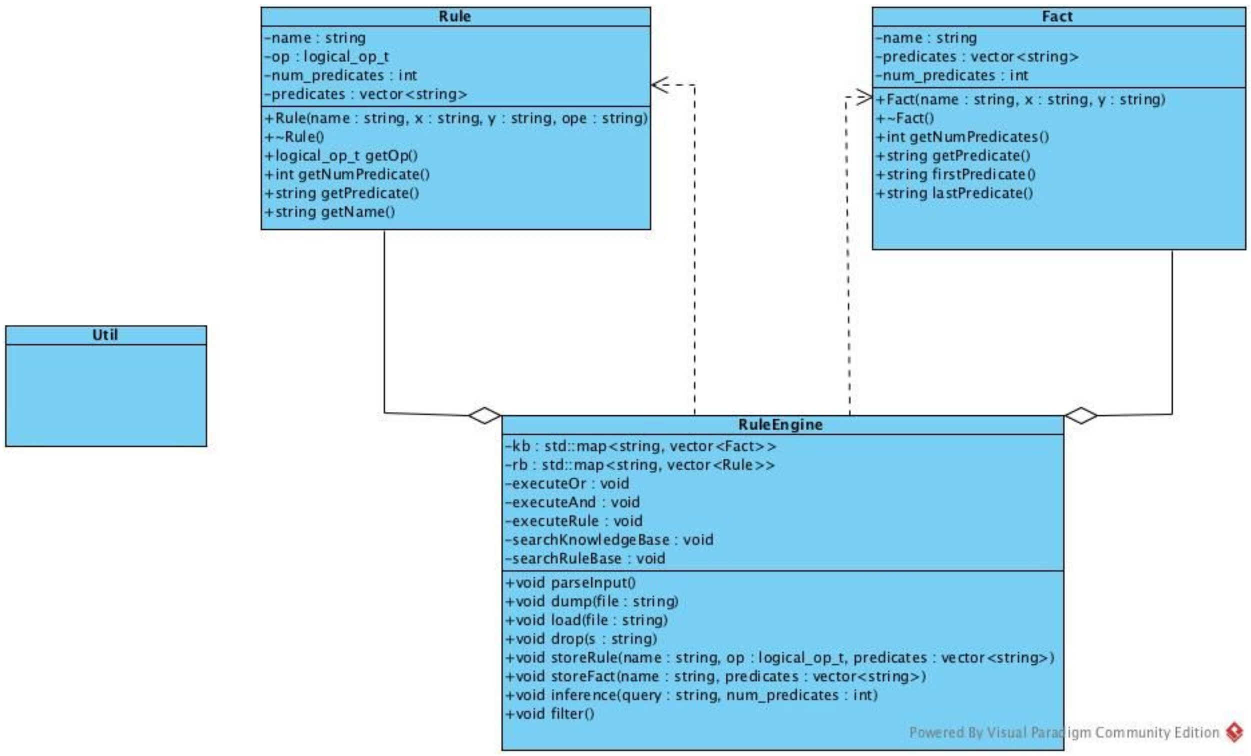
The other key design decision we had was for the executeRule() function. This function needs to have a Rule object passed to it, which it will then use to execute either the executeOr() or executeAnd() functions based on the value of its rule.operator string. These functions will then read the left and right predicates of the rule and traverse the KB and RB looking for matches. It will continue to execute rules if it finds rules, and print related facts back to the user.

Lastly, we designed the other important functions such as dump(), load(), and drop(). These will all carry out the necessary behavior with their functions and should be fairly straightforward to implement.

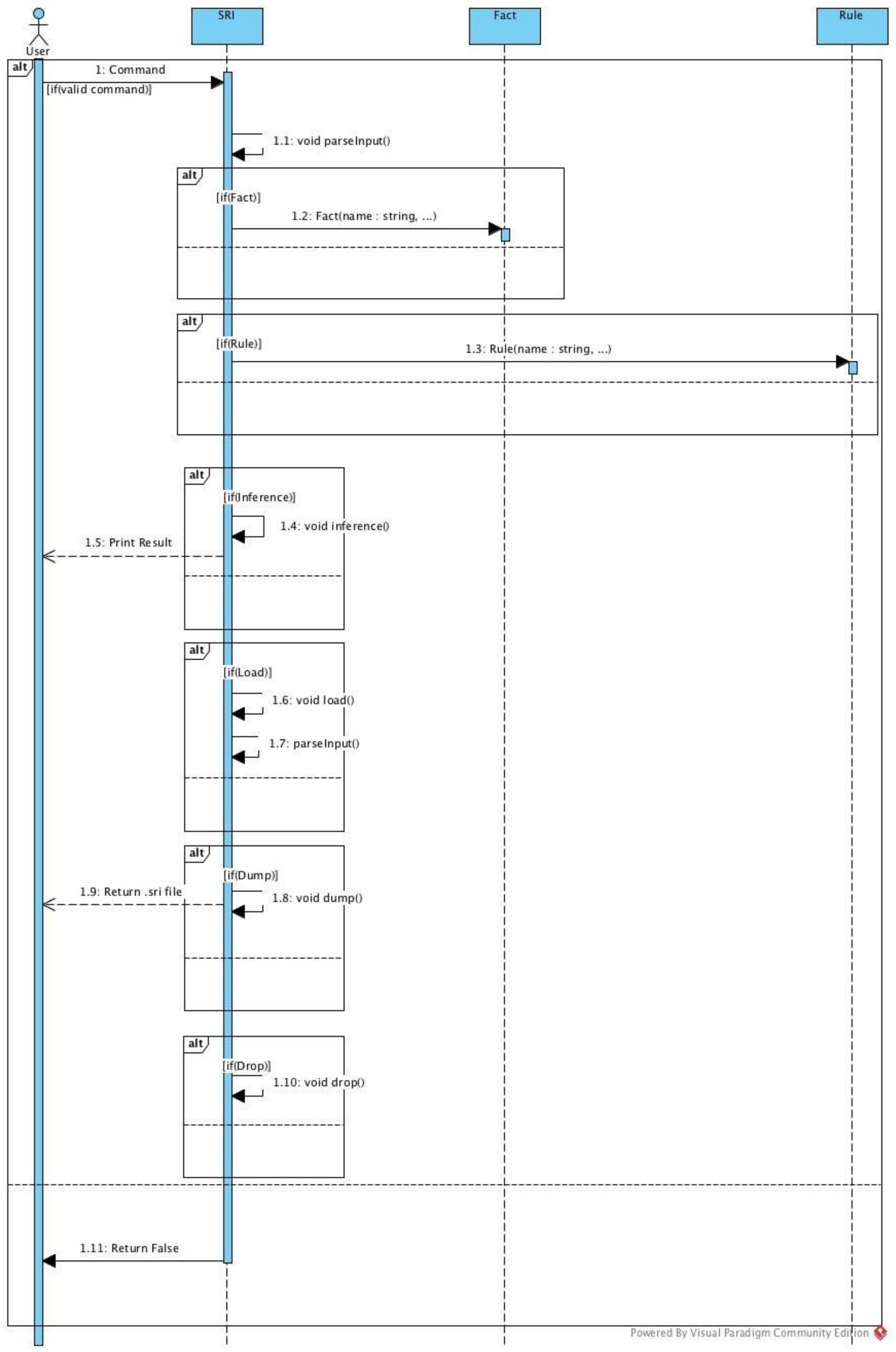
Use Case Diagram



Class Diagram


Rule
-name : string
-op : logical_op_t
-num_predicates : int
-predicates : vector<string>
+Rule(name : string, x : string, y : string, ope : string)
+~Rule()
+logical_op_t getOp()
+int getNumPredicate()
+string getPredicate()
+string getName()
Fact
-name : string
-predicates : vector<string>
-num_predicates : int
+Fact(name : string, x : string, y : string)
+~Fact()
+int getNumPredicates()
+string getPredicate()
+string firstPredicate()
+string lastPredicate()
Util
RuleEngine
-kb : std::map<string, vector<Fact>>
-rb : std::map<string, vector<Rule>>
-executeOr : void
-executeAnd : void
-executeRule : void
-searchKnowledgeBase : void
-searchRuleBase : void
+void parseInput()
+void dump(file : string)
+void load(file : string)
+void drop(s : string)
+void storeRule(name : string, op : logical_op_t, predicates : vector<string>)
+void storeFact(name : string, predicates : vector<string>)
+void inference(query : string, num_predicates : int)
+void filter()
Powered By Visual Paradigm Community Edition

Sequence Diagram


User
SRI
Fact
Rule
alt
1: Command
[if(valid command)]
1.1: void parseInput()
alt
[if(Fact)]
1.2: Fact(name : string, ...)
alt
[if(Rule)]
1.3: Rule(name : string, ...)
alt
[if(Inference)]
1.4: void inference()
1.5: Print Result
alt
[if(Load)]
1.6: void load()
1.7: parseInput()
alt
[if(Dump)]
1.8: void dump()
1.9: Return .sri file
alt
[if(Drop)]
1.10: void drop()
1.11: Return False
Powered By Visual Paradigm Community Edition