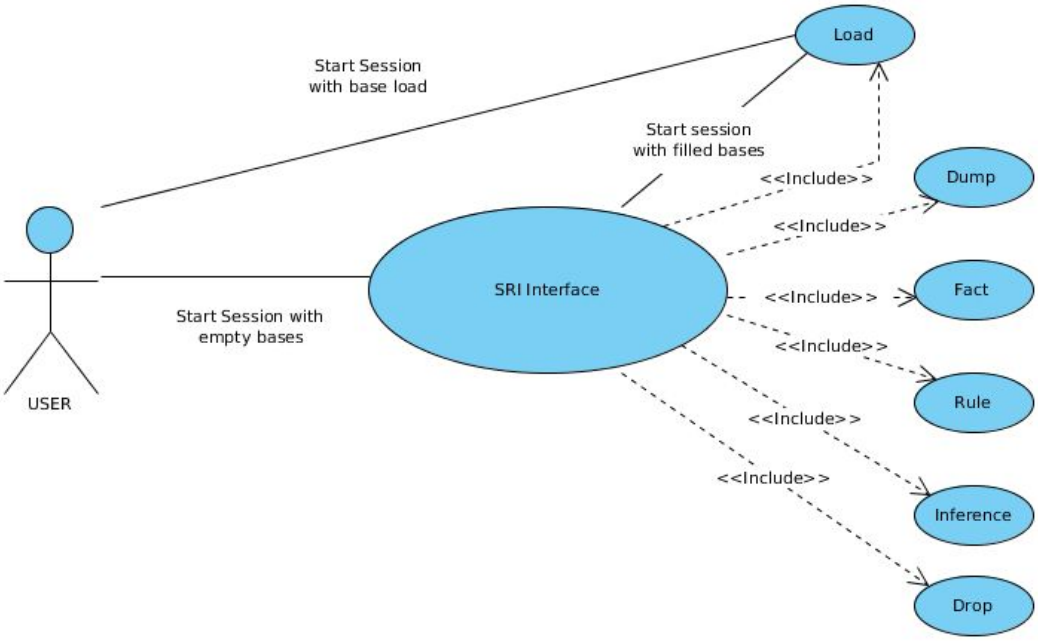
CMPS-109
Phase 1: Quarter Project
Group Members:

Daniel Thurau
Christopher Huynh
Andres Segundo

Design Document
UML Design

Use Case Diagram:

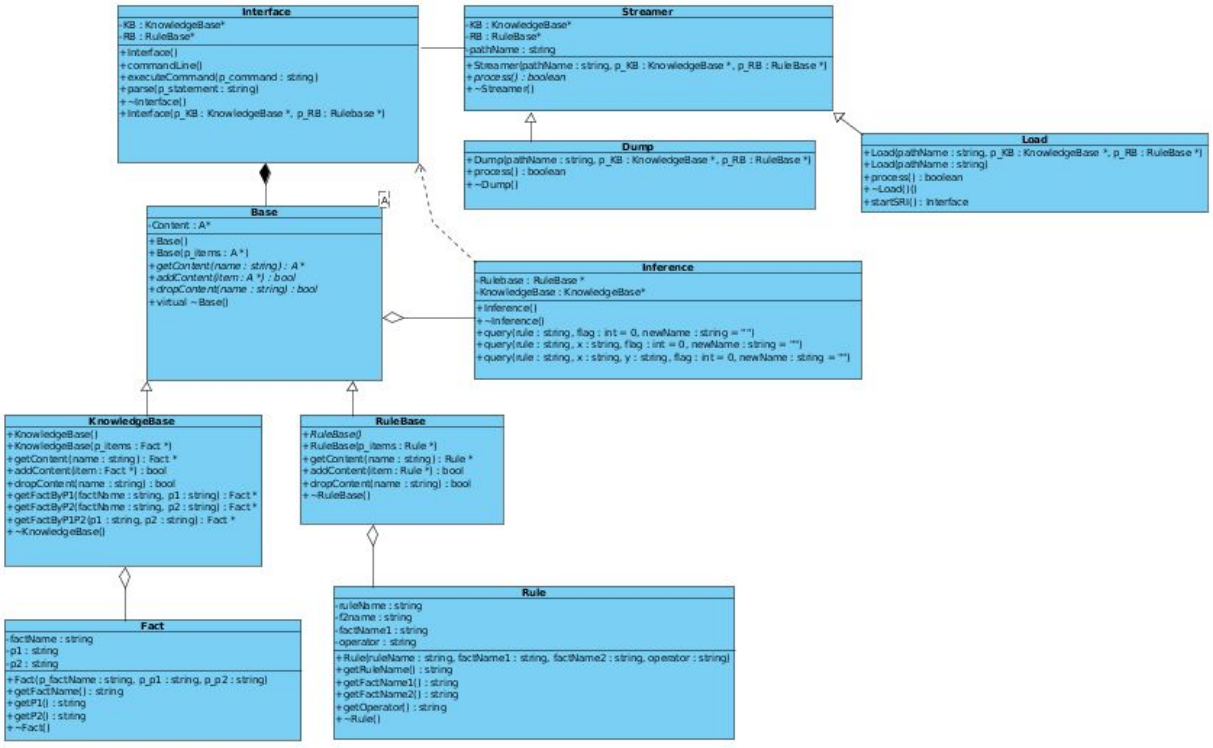


In this diagram we have the actor acting upon our SRI middleware. In the case of the SRI, the USER is the actor that triggers events to happen via two unique paths: the SRI interface, and the load command. Before we go into describing the methods that the interface and load method call, we **assume** that the entire process of triggering the SRI is done through starting an executable built to accept either one or no arguments. If

there is one argument, the SRI session is started by the LOAD command, which activates the interactive SRI session with an already populated KnowledgeBase and RuleBase, populated by the file derived from pathname (the argument for the exec). If there are no arguments, we activate the interactive SRI session with a virgin KnowledgeBase and RuleBase. Returning to the use case, the user will have direct access to only those two use cases, everything else is handled through the SRI interface, which is an interactive shell-like command line that accepts six different commands that trigger actions inside the SRI, but treats the commands as black boxes to the user. Those commands are DROP, INFERENCE, RULE, FACT, DUMP, and LOAD. For the sake of the length of the document we will address the functionality of what these commands do later in the document. Another important note is that once the user decides to start the executable with no pathname, the LOAD command is no longer an accessible command, except through the interactive SRI interface.

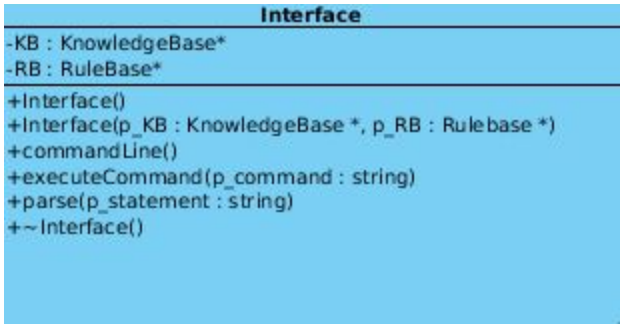
Class Diagram

Class Diagram Overview

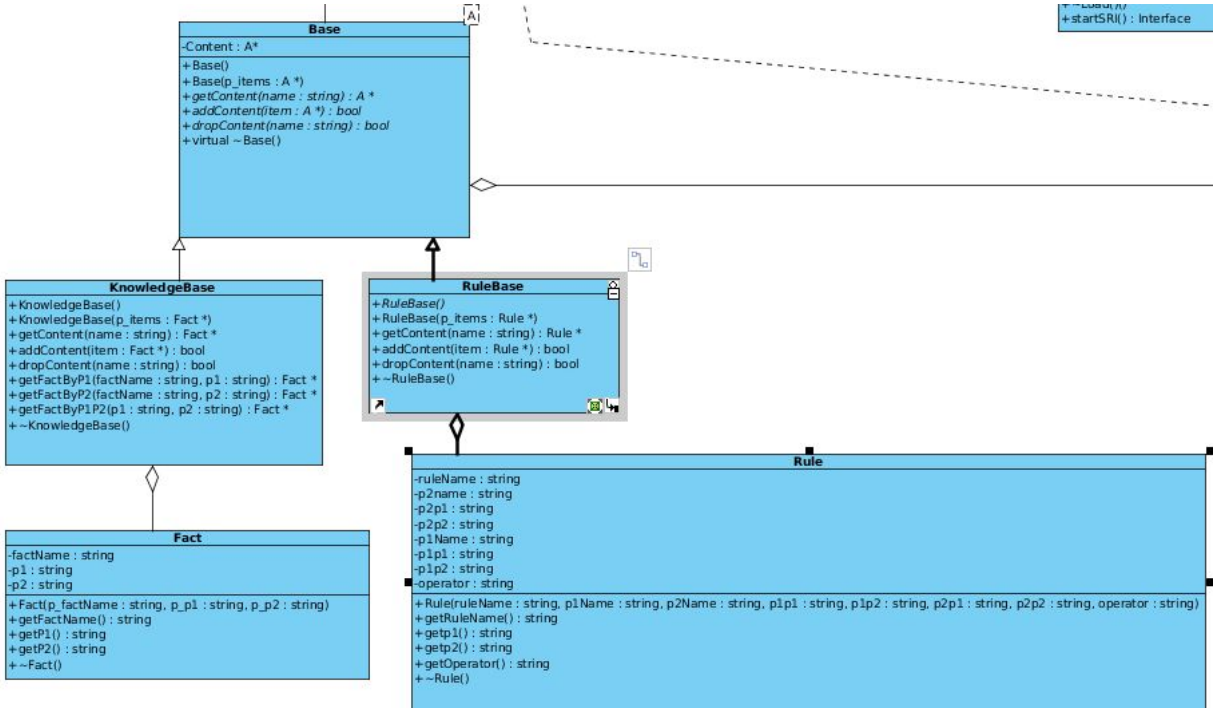


We understand that this specific image may be hard to read, and we will go into detail about each section further in the document, but this gives the best overview of what we

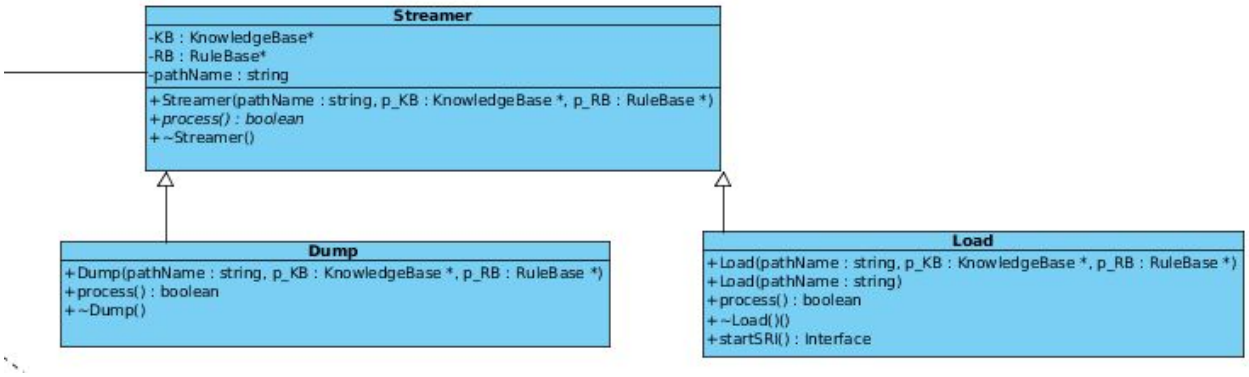
wanted to accomplish, as well as explaining relationships between classes. As you can see we have many classes, each with their own respective functionality within the SRI session. The main class represented in the diagram is the Interface() class, which when substantiated is the wrapper/buffer between the user and the rest of the middleware. Is shares several relationships with other classes, for instance the Base Class. The Base class is an abstract class that is inherited by the KnowledgeBase class and the RuleBase class. Interface and Base have a composition relationship, since one cannot exist without the other, while RuleBase and KnowledgeBase obviously have an inheritance relationship with Base. Off of each Base inheritance are two more respective classes: Fact and Rule. We decided to create these two classes because not only could we encapsulate their data and methods effectively, but it increased safety with using Rules and Facts, as well as most importantly reducing code complexity. On the right side of the diagram is the abstract class Streamer, which is associated with Interface. This class is inherited by two more classes, Load and Dump which handle I/O operations for SRI persistence between multiple session. And finally, the we have the inference class. Every time an inference is issued, we use the class Inference to be able to collect data and correctly computer solutions to the inferences. Inference is associated with Base class because of their different life cycles, and dependent on Interface.



The interface class provides the interactive part of the SRI middleware. It's two variables are pointers to the two Base's used, and are either substantiated in the default constructor, or connected to after the Load class does the initial creation on a ./SRI /path_to_file startup (this method of startup will be discussed in another section). After the Base's are connected to the Interface, the commandLine method can be used to start the interactive session. This method handles the looping and passing of entered user commands to itself, invoking the next method, parse(p_statement: string). This method will take whatever the user has inputted into the SRI, and parse it for correctness, and pass it to executeCommand(p_command : string), which depending on what the parser has passed it, will act accordingly to execute one of the six commands.



This section of the class diagram is the backbone of SRI. Without it, there would be nothing to search against. The Base Class is an abstract method utilizing a template to increase the amount of reusable code available for its children classes. Using A as a typename, we want to be able to use the variable Content : A* (becoming Content : Fact * and Content : Rule *) to be a data structure to hold the Fact and Rule objects in their respective bases. We also want to be able to inherit the similar functions getContent(name : string): A *, addContent(item : A*) bool, and dropContent(name : string) : bool which perform similar actions in both the RuleBase and KnowledgeBase. KnowledgeBase will need additional setter functions to be able to search it's rules via its parameter values in the horn clause, so we plan to implement those as well. Other than that, both bases need matching default constructors and Default parameter constructors for unpopulated and populated substantiations. Fact is a much simpler class than Rule is, only containing a name, and two parameter variables. We provide getter functions inside of Fact to reduce code complexity in methods gathering information about facts, as well as providing safety to the object (can only touch the values in a fact on creation). The Rule class is more complicated because of what a Rule is composed of . It contains fields for the Rule Name as well ad separate variables for both of the rules "rule target" and their parameter (we included this for now because of issues we predict will arise when we attempt pipelining). Rule follows the same format as Fact by including just getter methods for security and reducing of code complexity.



This part of the diagram handles the LOAD and DUMP commands, that are needed for persistence of the SRI. Streamer is the abstract class that had one method called process(). Process needs to be implemented in class Dump to use its pointers given to it upon substantiation to iterate through both the RuleBase and KnowledgeBase, retrieve all entries in both, and syntactically correctly enter them into a file denoted by pathName. This syntax of dumping is important, otherwise any Load of that file would cause failure. Load is an important class relative to our design of the SRI. The only case in which the user would not interact with the interface is to use load. Before we address that, Load is nearly the same as Dump, but in the opposite direction. It takes in a pathname as well as two pointers to the base's and will load everything from the file denoted by pathname into the Bases. We plan to parse each line, create a pointer to a newly substantiated Fact/Rule parsed from the line, and use the pointers given to the class in the constructor to enter the newly formed object into the bases. This method does have key difference from its parent and sibling class, the startSRI() method which returns an object of type Interface. The trace of how this function works is simple: we have an executable file compiled from a main.cpp file that has a few lines of code that starts our session of the SRI. The executable needs to be able to accept up to one argument, and perform a check upon start, that check will see if there is an command line argument passed. If this evaluates as true, our executable will next see if this command line argument is a path to a file and if that file is a legal .sri file. If that check resolves as true, then a Load Object is created, using the default constructor that takes in only one parameter (the pathname we just checked). This constructor will create a new substantiation of a KnowledgeBase and RuleBase (Interface hasn't been created so there is no bases created yet). If this object is created correctly, we have a load object that has the only pointers to the bases as well as a pathname to a file that holds data to populate a KnowledgeBase and a RuleBase. From within the executable we now can now call startSRI with this Load object, which will call the process method, and substantiate a new Interface object and use its Default parameter constructor to give the

address' of the now populated bases to Interface AND RETURN THE INTERFACE. We are now outside of the if/else statement in the executable and with our new Interface object that can call the same functions we would call as if we had just created a Interface without populating it.