CSC408H: Software Engineering – Fall 2005/2006

Student Scheduling Assistant

– Phase A – Specification and Design documents –

Team: c408h06

Roland Eck	(g3eckrol)	Part 1
Edmund Cheung	(c3cheunm)	Part 2.1, 2.2
Björn Schümann	(c5schuem)	Part 2.3, 3, 4

This contribution only refers to the initial version of the texts,
later every team member has reviewed and changed the other parts too.

Table of Contents

Table of figures

1 Information Representation

1.1 Storage Specification:

The type of storage we chose to use is a DBMS primarily due to the type of data, and how its used. First and foremost, the stored system information can be considered mostly as data-centric, as apposed to document-centric. Also, it has a fairly regular structure, which makes it ideal for databases. Furthermore, the queries performed end up being fairly simple for the system, (i.e. there is no need for doing deep recursions and such tasks.) So, it is our belief that the relationships are most efficiently managed and best represented with a DBMS.

Moreover, a DBMS requires a fairly regular implementation, there is no need for data structures that iterates through elements, instead simple SQL queries perform such tasks. Simplicity in implementation means less risk of errors, which means saving money in the long-run, assuming the DBMS is not filled with bugs. The need for writing your own code to iterate through elements takes up more implementation and testing time. Whereas a DBMS is off-the-shelf, so there is no need for such implementation and testing work.

Database technology: We chose to use an embedded database engine because it best fits the purposes of this project and the project phases. First and foremost, embeddable databases remove the overhead that client-server applications carry. We do not need tons of features that we will not use and that will slow down the environment.

Specifically, we will use an engine like *SQLite* (http://www.sqlite.org) an embeddable database engine having the following features:

- Simple to setup and administer. There is actually no configuration needed at all.
- Simple to operate and embed into a larger program. The module are self-contained, meaning there are no external dependencies. It is also lightweight, (i.e. does not use a lot of memory)
- Simple to maintain and customize
- Speed. Due to its simple framework, and the nature of the scheduling system, performance is unbeatable with embeddable databases under the following conditions:
    - databases are under several gigabytes of data,
    - concurrency is relatively low
    - databases not distributed on complex networks such as clusters, (i.e. relatively low-volume access)

Embeddable databases are simple, reliable, and at the same time, it can handle the needs of the *SSA* system. (i.e. supports databases up to 2 terabytes, most SQL92 features supported, etc...)

1.2 Data Model

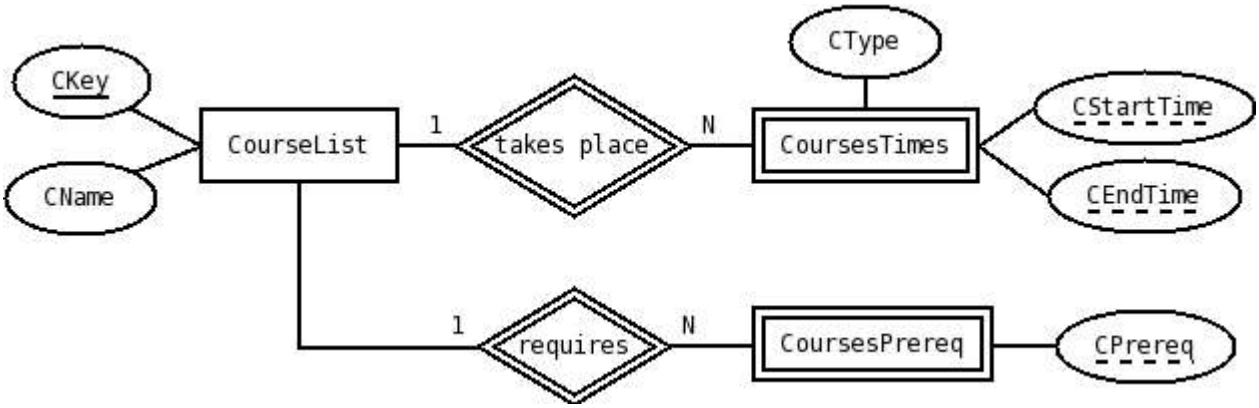

Figure 1 CourseDB entity-relationship diagram

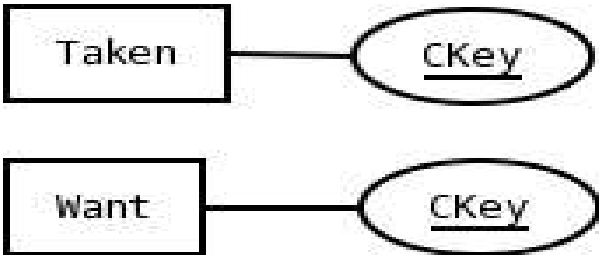

Figure 2 StudentDB entity-relationship diagram

1.3 Schema

/*

- Applies To: CKey
- Purpose: a Unique Course Key
- Size: Exactly 9 characters
- Format: DDDNNNLCW,
    - D is any uppercase character [A-Z],
    - N is any number [0-9],
    - L is either H or Y,
    - C is any number [1-9],
    - W is either F, S, or Y.

*/

/*

- Applies To: CName
- Purpose: a Course Name
- Size: Dynamic
- Format: "[\w]"

*/

/*

- Applies To: CPrereq
- Purpose: a Course Prerequisite Key
- Size: Exactly 7 characters
- Format: DDDNNNL,
    - D is any uppercase character [A-Z],
    - N is any number [0-9],
    - L is either H or Y

*/

/*

- Applies To: CStartTime
- Purpose: a Day and Time at which a Course Starts
- Size: exactly 5 characters
- Format: [1-7][0-2][0-9][0-5][0-9],
    - First digit represents the day of the week, M = 1, T = 2, ... ,Sunday = 7.
    - Second and third digit represents the hour, and is at most 24.
    - Fourth and fifth digit represents the min., and is at most 59.
    - Last 3 digits represents the time, (Hour*100 + Min), and is at most 2459.

*/

/*

- Applies To: CEndTime
- Purpose: a Day and Time at which a Course Ends
- Size: exactly 5 characters
- Format: [1-7][0-2][0-9][0-5][0-9],
    - First digit represents the day of the week, M = 1, T = 2, ... ,Sunday = 7.
    - Second and third digit represents the hour, and is at most 24.
    - Fourth and fifth digit represents the min., and is at most 59.
    - Last 3 digits represents the time, (Hour*100 + Min), and is at most 2459.

*/

/*

- Applies To: CType
- Purpose: Indicates either Lecture or Tutorial
- Size: exactly 1 character
- Format: [L|T]

*/

AdminDB Schema:

CREATE TABLE coursesdb.courseslist { CKey CHAR(7) NOT NULL, CName VARCHAR(1000) NOT NULL, PRIMARY KEY(CKey)

}

CREATE TABLE coursesdb.coursesprereq { CKey CHAR(9) NOT NULL, CPrereq CHAR(7) NOT NULL, PRIMARY KEY(CKey, CPrereq), CONSTRAINT FK_courseprereq FOREIGN KEY FK_courseprereq(CKey)

REFERENCES courseslist(CKey) ON DELETE CASCADE

}

CREATE TABLE coursesdb.coursestimes { CKey CHAR(9) NOT NULL, CStartTime INTEGER UNSIGNED NOT NULL DEFAULT 00000, CEndTime INTEGER UNSIGNED NOT NULL DEFAULT 00000, CType CHAR(1) NOT NULL DEFAULT 'L', PRIMARY KEY(CKey, CStartTime), CONSTRAINT FK_coursestimes FOREIGN KEY FK_coursestimes(CKey)

REFERENCES courseslist(CKey) ON DELETE CASCADE

}

StudentDB Schema:

CREATE TABLE mycoursesdb.mytakencourses { CKey CHAR(9) NOT NULL, PRIMARY KEY(CKey)

}

CREATE TABLE mycoursesdb.mywantcourses { CKey CHAR(9) NOT NULL, PRIMARY KEY(CKey)

}

2 Software architecture

This part explains the structure of our software in terms of modules, functions, and all the necessary building blocks.

2.1 System's interaction with environment

A description of how our software will interact with the underlying operating system.

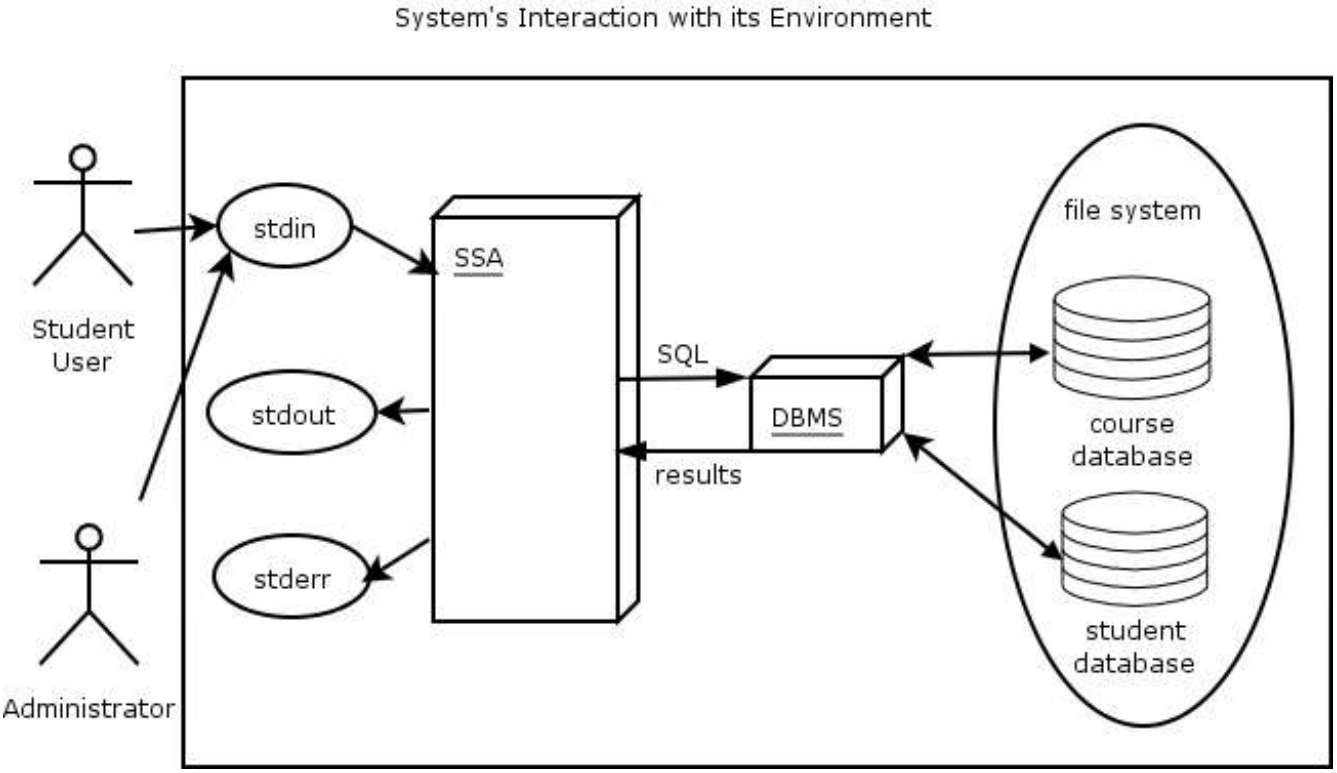

Figure 3 System's Interaction with its Environment

- The users (student or administrator) interacts with the system via standard input (normally, the keyboard).
- They get feedback from the system via the standard output and *stderr* for error messages (normally, both will be through the monitor).
- The *SSA* program will get the input and access the database via a database management system using SQL statements.
- To process these commands, the DBMS will access the databases on the file systems and return the results back into the SSA.
- The *SSA* will then return the information requested by the user out to standard output.

2.2 Component Decomposition

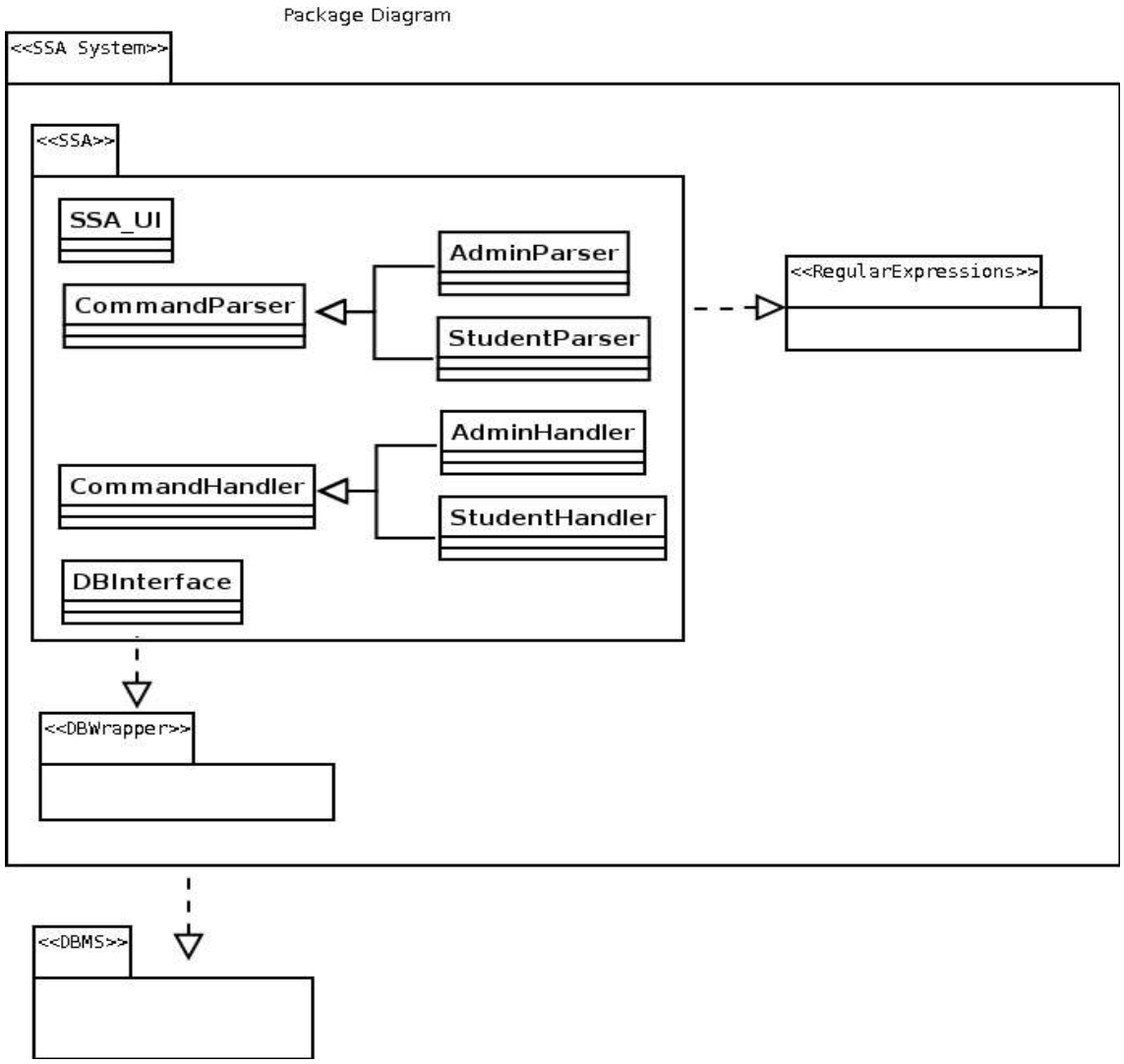

Figure 4 Package Diagram

The *SSA* package includes the *SSA_UI* class, *DBInterface* class, *AdminParser* and *StudentParser* classes, which are extensions of the *CommandParser* class. Also, the *AdminHandler* and *StudentHandler* classes, which are extensions of the *CommandHandler* class. The SSA_UI class handles the front-end duties. The Parser classes parse the inputs from the user, while the Handler classes perform the logical operations associated with the requested command(s). Finally, the *DBInterface* is for communicating with the database.

- Also the SSA system contains the Regular Expressions package and a *DBWrapper* package. The Regular Expressions package will be used to parse strings for matching user commands. The *DBWrapper* package acts as a bridge between the DBMS we will use and the language that SSA will be implemented in, so that they can communicate between each other.
- The final package is for the *DBMS*. This will be responsible for managing the database that acts as the data storage for both the Course database and the Student databases.

2.3 Detailed design of the system

Classes Diagram

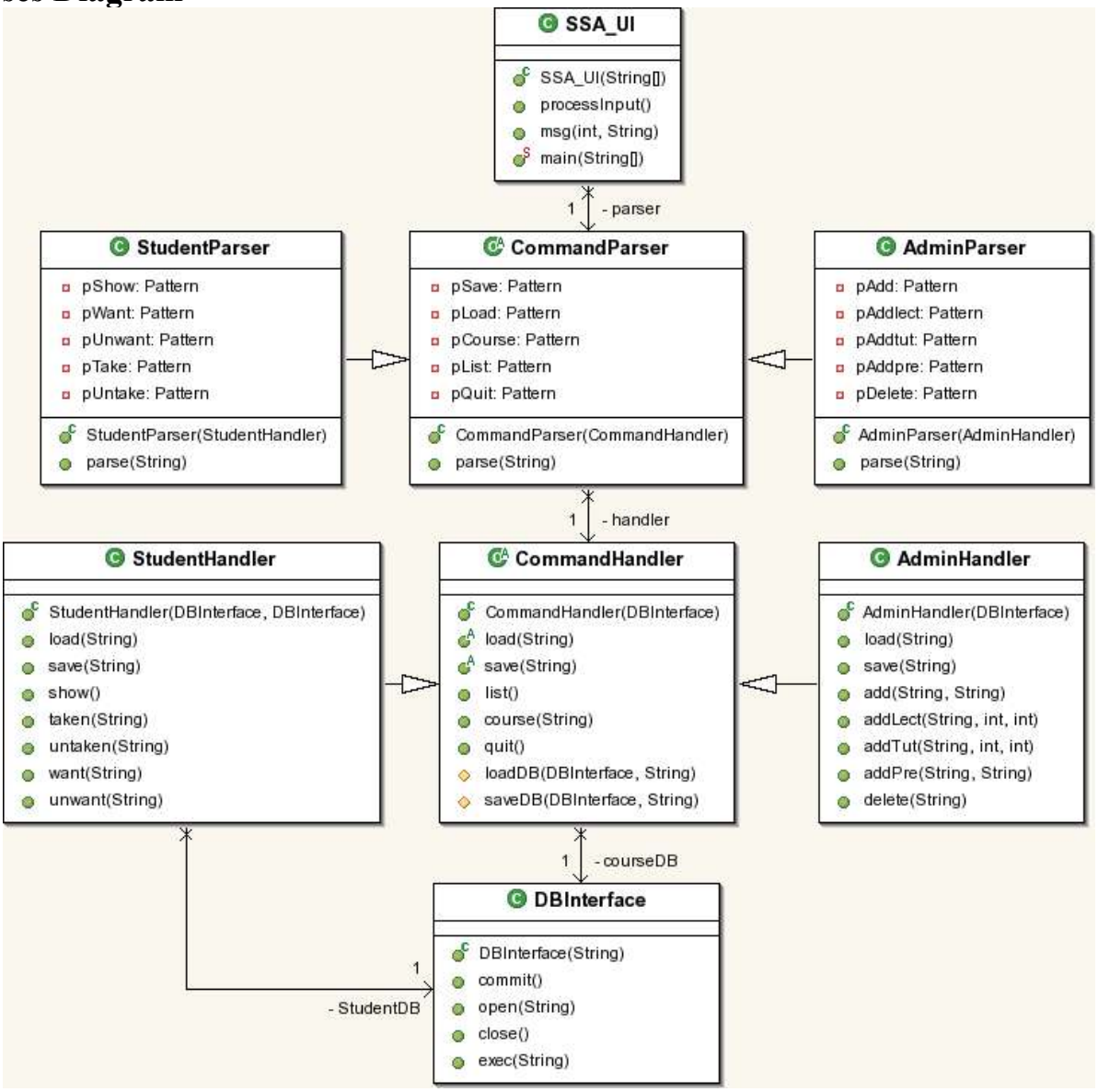

Figure 5 Class Diagram

SSA_UI contains the user interface of *SSA*. Its constructor is called from the main function with the command line arguments and initiates the *SSA* system. The other methods are for processing the input and assists in the output.

CommandParser is the superclass for every parser. A parser contains the patterns of the valid commands and its parameters. The parse method gets the user's command and with its parameters it calls the corresponding method in the handler. The parsing of the general commands which are offered for every user are implemented here, while user specific commands are implemented in a subclass.

The *StudentParser* extends the *CommandHandler* with the pattern and the parsing of the specific commands for the student mode.
The *AdminParser* extends the *CommandHandler* with the pattern and the parsing of the specific commands for the admin mode.

The patterns in the parsers are compiled from Regular Expressions of the commands. (see Appendix)

CommandHandler is the superclass for every handler. A handler contains the logic for a user modes. General commands which are offered for every user are implemented here, while user specific commands are implemented in a subclass.

The *AdminHandler* extends the *CommandParser* with the logic of the specific commands for the admin mode.
The *StudentHandler* extends the *CommandParser* with the logic of the specific commands for the student mode.

DBInterface encapsulates the operations for accessing the wrapper for the database. Since the interface of each wrapper differs, this class makes sure that switching to another database is as easy as possible.

Collaboration Diagram

To get a quick overview of the behavior of the objects, take a look at the collaboration diagram below.

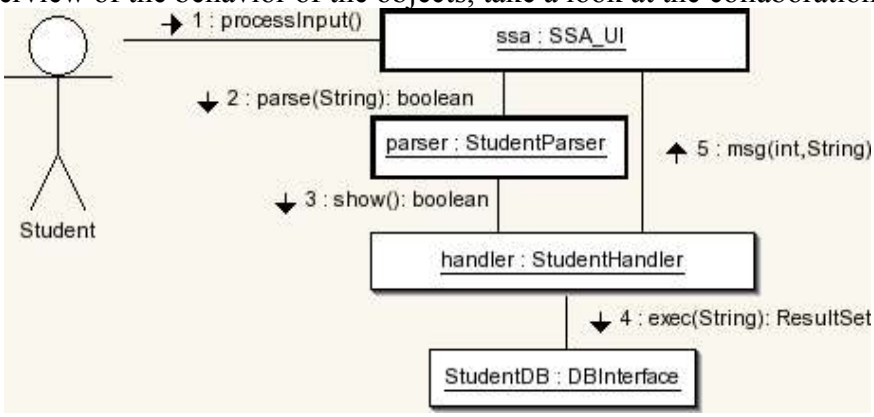

Figure 6 Collaboration Diagram

1. The user's input is processed by *processInput()* of *SSA_UI*. It gets one whole line of the input and calls the *parse()* method of the current handler.
2. *parse()* of the *parser* object tries to match the user's command against the patterns of the commands.
3. The corresponding method of the handler object is then called (e.g. *show*).
4. The handler method runs the logic of the process using the *exec* method of the *DBInterface* to execute SQL commands.
5. The results are processed and for the output the *msg* method of the *SSA_UI* is called.

Sequence Diagram

For a more detailed view of initialization and the behaviour of the SSA system take a look at the sequence diagram below.

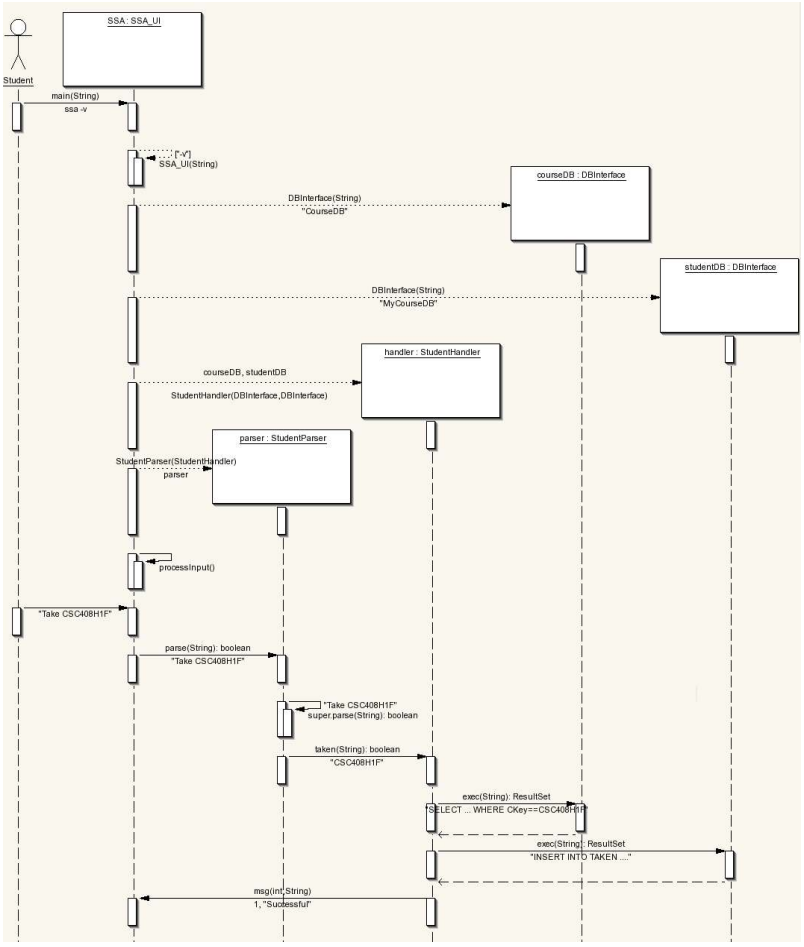

Figure7 Sequence Diagram

1. The *main* function starts by creating a new *SSA_UI* object.
2. In the *SSA_UI* constructor, the command line arguments are first parsed
3. The *DBInterface* objects that are needed to be created are created after command line parsing.
4. During the construction, the *DBInterface* object(s) establishes connections with the database(s).
5. Then the *SSA_UI* constructor initializes the handler corresponding to the mode the user asked for.
6. With this handler, a new parser can be constructed.
7. Now the processing of the input can begin. So the main function calls the *processInput* method of the new *SSA_UI* object.
8. When the user issues a new command to standard input, the parser is called with the whole line.
9. First the parser tries to match the command against the common commands patterns.
10. And if this fails it tries its own pattern for the mode specific commands.
11. If a command pattern matches, the parameters are extracted and the corresponding method in the handler object is called.
12. The logic associated with the command is processed, including SQL statements on the *DBinterface* objects.
13. The result, error messages, and additional information is then presented to the user using the *msg* method of the *SSA_UI* object.

3 Errors and error handling

This document discusses errors that may arise during the use of the system and how our software will handle such errors.

3.1 Critical errors

- If the initial loading of the database fails, the program immediately stops. The database is a critical component of our system, and without it, there is nothing to be done.
- If a database fails to load, the system will attempt to fall back on the previously active database. If there are no databases to fall back on, the program terminates. Again, the database is a critical component that is required for any work to be done.

3.2 Recoverable errors

- If the program is started with illegal options, a usage notification will get printed out and the program will immediately terminate.
- If parsing the current command fails because the command is unknown or the number of parameters is incorrect, we will abort the processing of this command and print information indicating which inputs were not expected. Then, the program will continue with the next command. Any command that does not match with the expressions given on the regular expressions specifications will result in error. Refer to the specifications for the complete list.
- If loading a new database fails, we will warn the user and try to reload the last active database.
- If saving the current state of a database fails, we will output an error message and give the user options on how to proceed. The options will be whether to save to another file, cancel, or exit the program.
- If processing a single command fails, we will output an error message and continue with the next command. Examples of processing errors include the following:
    - adding a course that already exists
    - adding times where the start time is later than an end time
    - adding times for a course that does not exist
    - adding a prerequisite that does not exist
    - adding a prerequisite for a course that does not exist
    - deleting a course that does not exist
    - wanting a course that does not exist
    - unwanting a course that is not on the want list
    - taking a course that does not exist
    - untaking a course that is not on the taken list

3.3 Warnings

- If an empty database is loaded, we will provide the user with a message that warns the user that there are no records in the database. Then the program will continue to run.
- If the data the user wants to add seems to be unusual, we will provide a warning message and explain that the user may be doing something they are not intending to do and how they can undo the operation if they want. Then, we will continue with the operation. An example of this may be if the user wanted to add a class with a very short time session.
- If the student wants to add a course to the take or want list, that already exists, we will provide a warning.
- If an admin adds prerequisites for a course that includes the course itself, we will provide a warning.

Note: The level of error messages output will of course depend on the verbose flag. When a commands fail, the level of output will also depend on the debug level.

4 Testing plans

To prevent and find errors as early as possible and to produce bugles code, we want to follow a two-stage testing strategy.

Stage 1: Unit testing of the individual classes

Before any code of a class is written, a corresponding test class implementing a suite of test cases has to be developed by someone else. This test class should check each individual method of the class and its result.

After this is done, the class is passed on to someone else within the team who fixes the code until every test cases completes successfully.

Stage 2: External testing of the system

After the implementation, each external functional requirement is tested again using the input and output of the whole system. This can be done by developing another test suite for the whole system. The test cases for this suite are developed by the third person within out group.

Tools

To make the best use of our time, we decided to use Eclipse together with a Unit Testing. (For example JUnit, if Java is the language we use). This way, all test suites and their test cases can be effectively managed. The external testing process will be semi-automated. Sample sets of expected results based on a sequence of inputs will be saved in files. The inputs will be saved to files and will be redirected as inputs for the program and the outputs will be redirected into output files. Finally, the sample expected output files will be compared with the actual results using diff. This whole process will be scripted to minimize the amount of manual operations performed during the external testing.

Appendix

Regular expressions for the parsers

```
pws = [ \t]

pmonday    = M|MO|MON|MOND|MONDA|MONDAY
ptuesday   = T|TU|TUE|TUES|TUESD|TUESDA|TUESDAY
pwednesday = W|WE|WED|WEDN|WEDNE|WEDNES|WEDNESD|WEDNESDA|WEDNESDAY
pthursday  = R|TH|THU|THUR|THURS|THURSD|THURSDA|THURSDAY
pfriday    = F|FR|FRI|FRID|FRIDA|FRIDAY
psaturday  =   SA|SAT|SATU|SATUR|SATURD|SATURDA|SATURDAY
psunday    =   SU|SUN|SUND|SUNDA|SUNDAY
pday       = pmonday|ptuesday|pwednesday|pthuersday|pfriday|
             psaturday|psunday

phour      = [0-1][0-9]|2[0-3]
pminute    = [0-5][0-9]
ptime      = (pday) pws (phour)(:(pminute))? pws* - pws* (phour)(:
             (pminute))?

pckey      = [A-Z]{3}[0-9]{3}[HY][1-9][FSY]
pprereq    = [A-Z]{3}[0-9]{3}
pcname     = [\w]+
pdbname    = [\w]+       (the only one, which is case sensitive)

psave      = SAVE (pws+ (pdbname))?
pload      = LOAD (pws+ (pdbname))?
pcourse    = COURSE pckey
plist      = LIST
pquit      = QUIT

padd       = ADD pckey pcname
paddlect   = ADDLECT pckey ptime (, ptime)*
paddtut    = ADDTUT pckey ptime (, ptime)*
paddpre    = ADDPRE pckey pprereq (, pprereq)*
pdelete    = DELETE pckey

pshow      = SHOW
pwant      = WANT pckey
punwant    = UNWANT pckey
ptaken     = TAKEN pckey
puntaken   = UNTAKEN pckey
```