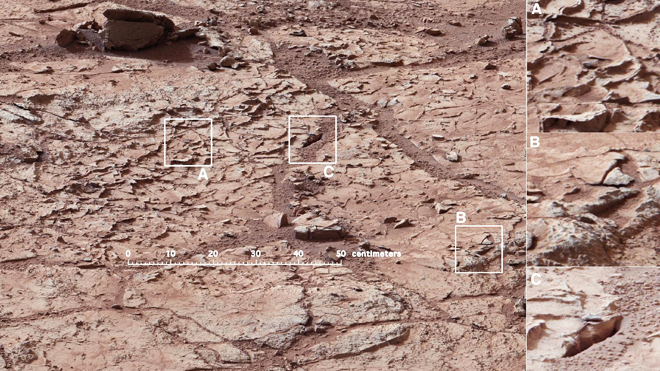
Mars Rover - #YOGOMO \*

Design Specification

320222 AdvCS Lab Course II (Software Engineering)

**Version: 1.0**

 Design Group 2

Octavian Cezar Enache

Daniel Hasegan

Yonathan Mengesha

Bishesh Tiwaree

Ernesto Gonzales

|  |  |  |
| --- | --- | --- |
| Purpose / Project | AdvCS Lab / Design Document for Mars Rover | |
| Created On / By | 22.02.2013 | Design Group 2 |
| Last Modified / by | 22.02.2013 | Design Group 2 |
| Document Availability | svn://**<username>**@tlab060.clamv.jacobs-university.de/selab/design/group**2** | |

Table 1 : Cover Sheet

*\*You Only Go On Mars Once*

**Symbols and Notations**

|  |  |
| --- | --- |
| Representation | Usage |
| "Screen Text" | For texts which are quoted from the screen ( for example System Messages , field names , menu names , buttons etc. ) |
| User Input | For exact user input |
| <variable user input> | For variable user input |
| Name | For dialogue window names , report names , program names, transaction codes , table names , keywords , file names , directories … |
| Booktitle | For references to other documents/manuals |
| [Key] | For keyboard keys (e.g. [Enter] or [Ctrl]+[F2]) |
| *Code lines* | Code |

Table 2: Symbols and Notations

Index

[Section 1: Introduction and Project Overview 5](#_Toc349339780)

[Section 2: System Architecture 6](#_Toc349339781)

[Section 2.1:Overview 6](#_Toc349339782)

[Section 2.2:Class Diagrams 6](#_Toc349339783)

[Section2.3:Class Descriptions 6](#_Toc349339784)

[Section 2.3.1: Connection 6](#_Toc349339785)

[Section 2.3.2: Parser 7](#_Toc349339786)

[Section 2.3.3: ClientSocket 8](#_Toc349339787)

[Section 2.3.4: Socket 8](#_Toc349339788)

[Section 2.3.5: Brain 10](#_Toc349339789)

[Section 2.3.6: Logger 11](#_Toc349339790)

[Section 2.3.7: ArgumentType <<enum>> 12](#_Toc349339791)

[Section 2.3.8: SpaceObject class 13](#_Toc349339792)

[Section 2.3.9: SpaceObjectRegister Class 14](#_Toc349339793)

[Section 2.3.10: VariableRegister Class 15](#_Toc349339794)

[Section 2.3.11: MapRegister Class 16](#_Toc349339795)

[Section 2.3.12: RoverDataRegister Class 17](#_Toc349339796)

[Section 2.3.13: MemoryHandler Class 18](#_Toc349339797)

[Section 2.3.14: InstructionIF 19](#_Toc349339798)

[Section 3: Language Description and Specification 21](#_Toc349339799)

[Section 3.1: Overview 21](#_Toc349339800)

[Section 3.2: Brain Requirements 21](#_Toc349339801)

[Section 3.3: Instruction Usage 22](#_Toc349339802)

[Section 3.4: Preset Variables: 23](#_Toc349339803)

[Section 3.5: Argument Types: 25](#_Toc349339804)

[Section 3.6: Instructions 26](#_Toc349339805)

[Section 3.7:Example of Brain Language Usage 33](#_Toc349339806)

[Section 4: Use Case 34](#_Toc349339807)

[Section 5: Interactions 35](#_Toc349339808)

[Section 6: Quality Assurance 38](#_Toc349339809)

[Section 6.1: Quality Requirements 38](#_Toc349339810)

[Section 6.1.2: Flexibility Requirements 39](#_Toc349339811)

[Section 6.2: Testing 41](#_Toc349339812)

[Section 7: Administrative Requirements 44](#_Toc349339813)

[Section 8: Conclusion 44](#_Toc349339814)

[Section 9: References 45](#_Toc349339815)

[Appendix 1 46](#_Toc349339816)

[Appendix 2 – Sample Test Scenario 47](#_Toc349339817)

[Scenario X: Component 47](#_Toc349339818)

[Scenario Description 47](#_Toc349339819)

[Version Control 47](#_Toc349339820)

[Test Scripts 47](#_Toc349339821)

[Case Description 47](#_Toc349339822)

[Test Components/Requirements 47](#_Toc349339823)

[Appendix 3 – Sample Test Script 48](#_Toc349339824)

[Script X.1: Something 48](#_Toc349339825)

[Script Steps 48](#_Toc349339826)

[Testing Requirements 48](#_Toc349339827)

[Teardown 48](#_Toc349339828)

[Script Steps 48](#_Toc349339829)

[Test Execution 48](#_Toc349339830)

[Script X.2: Test that fails 49](#_Toc349339831)

[Script Steps 49](#_Toc349339832)

[Testing Requirements 49](#_Toc349339833)

[Teardown 49](#_Toc349339834)

[Script Steps 49](#_Toc349339835)

[Test Execution 49](#_Toc349339836)

[Appendix 4 – Sample Test Script 50](#_Toc349339837)

# Section 1: Introduction and Project Overview

*Basic Overview and Project Description*

This document is prepared to serve the goal of making an appropriate software design for the Mars Rovers project. It was made within the time range of a week and a half, during the initial design phase of the Software Engineering Lab Course project. It involved setting up the frameworks that should be followed in order to properly execute instructions later in time, which also includes stating the specification of the language that is going to be used.

In the Mars Rover project, the Rover is placed in a fixed 2-D map, assumed to be the terrain of planet Mars, and the control software is expected to safely guide it to its home base located at the center of the map in time. Throughout the run, the rover may encounter craters, boulders and Martians, which it must avoid. The simulation starts with the user providing, to the client, the Brain file which is composed of different implementations of algorithms and schemes that are written in a language described in this document, and are geared towards leading the rover back to the home base.

Afterwards the client will independently continue to operate the rover by communicating with the server, until the end of the run. The server will send out different messages during the simulation and in the end when the round is over, it will issue the score achieved which depends upon the time spent, the condition of the rover and the penalties that have been received.

In general, our design possesses a Connection class, that acts as a mediator between the different components and classes. It listens for any type of messages from the server and parses them and passes it on to the specific component that is required, sends back the commands given by the respective component to the server and also maintains the interaction of the different classes.

The language that is found in this document is a depiction of a nice trade-off between the clarity and sophistication of the characteristics it holds and its ability to handle various implementations. It is actually based on the assembler language but has more high-level functions and features that are crafted to make the user experience better. It was designed such that it doesn't have restrictions to a particular strategy but rather a very flexible interface that one can use to try out different algorithms and plan of actions, such as future simulations and others.

The other parts of the document are composed of comprehensive description and analysis of the software design by utilizing UML diagrams, such as class, sequence and activity diagrams as well as specific use cases and test cases. These diagrams and the classes are also coupled with good descriptions of their properties and attributes to convey a good understanding of the topics that have been touched in this document.

# Section 2: System Architecture

*General description of design decisions related to architecture*

## Section 2.1:Overview

In this section of the document, we will deal with the thorough description of the numerous classes of our project. We will provide a detailed information on the attributes and methods that we think are very crucial. This does not include the setter and getter methods that have been implemented for each class.

## Section 2.2:Class Diagrams

Please refer to the appendix section of this document to obtain the class diagram of our project.

## Section2.3:Class Descriptions

### Section 2.3.1: Connection

It is the class that will dictate and mediate the overall interactions of the components of the software.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| socket | ClientSocket\* | It is the pointer to the class ClientSocket that will enable the Connection class establish a connection to the server. |
| signal | int | It is the signal that will be set to 0 or 1 depending on the availability of an incoming message from the server. |
| brain | Brain\* | It is a pointer to the Brain class that will give the possibility of controlling the Brain class, in special case such as a fatal error occurs in the Brain class. |
| logger | Logger\* | It is a pointer to the Logger class that is used to send messages to the Connection class |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return Arguments | Arguments | Description |
| Connection | (constructor) | Host:string, port:int | It creates an instance of a connection class that will handle the interaction of most of the components in the program. |
| acceptMessage | int |  | It is the method by which the Connection class retrieves an incoming message from the server. |
| sendMessageToServer | ret: int | msg: string | It is the method by which the Connection class sends back the commands generated to the server. |

### Section 2.3.2: Parser

It is the class that will take the responsibility of parsing the Brain file submitted by the Connection class. It will check if the the given Brain file is correct and creates a Brain class. The Parser class does not own any attributes and has static methods.

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| parseFile | ret: int | brain: Brain\*, brainFile: fstream, arguments: int | It is the method by which the Parser class creates a Brain class that will do the instructions and operations. It takes in a pointer to a Brain class, a Brain file and number of arguments. |
| parseImmediate | ret: int | value: void\*, type: ArgumentType\*, imed: string | This method parses the Immediate Constant kept in message 'imed' to get the value 'value' of type 'type'. |

### Section 2.3.3: ClientSocket

It is the class that primarily acts as the tool in the communication between the server and the Connection class. It provides methods that enable listening on and accepting messages from the server and at the same time send messages back to it.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| host | string | It is the address of the server the client used to create a socket and establish a connection. |
| port | int | It is the port number that is going to be used to make a connection. |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| ClientSocket | (constructor) | host: string, port: int | It is the constructor that creates an instance of a ClientSocket |
| sendclient | Int | msg: string\* | It is the method by which a message is send via the socket. It will be utilized by the Connection class. |
| recvclient | Int | msg: string\* | It is the method by which a message is accepted from the server and this will also be used by the Connection class. |
| getHost | string |  | getter method to get the host name |
| getPort | int |  | getter method to get the port number |

### Section 2.3.4: Socket

It is the class that holds the overall mechanisms and workings of the socket creation process. It will entail the implementations of the different methods that will require the use of standards libraries.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| sock | int | It is the argument that will be set depending on the success / failure of the socket creation. |
| address | sockaddr\_in | It is the argument that holds the address used to connect to the server. |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| Socket | (constructor) |  | It is the constructor used to make an instance of the socket class. |
| connectfunc | bool | host: string, port: int | It is the method by which the socket will connect to the specific address and port number. It will return a bool depending on the condition. |
| sendfunc | Bool | Msg:string | It is the method by which a message is sent to the server through the socket. |
| receive | Int | Msg:string& | It is the method by which a message is accepted from the server through the socket. |
| setNonBlocking | Void | B: bool | It is a method that will enable the Socket to return anything that is incoming or going out without getting blocked. |
| createSocket | Bool |  | It is a method by which the Socket class can create a socket that will listen for any messages that is incoming from the server. |
| bindfunc | Bool | Port:int | It is a method that will enable the Socket class to bind or tie the socket that was created to a specific port number. |
| listenfunc | Bool |  | It is a method that gives the ability to constantly check and listen for any messages that are being sent from the server. |

### Section 2.3.5: Brain

It is the class that will be created right after the Brain file has been parsed. It is the class that will perform all of the logical operations and sends out the commands to the server through the connection.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| instructionPointer | Int | It is the argument that holds the position of the instruction that is being executed. |
| instructions | Vector<InstructionIF> | It is the array or a vector of instructions that are going to be executed in the Brain class |
| memoryHandler | MemoryHandler | It is an object of the memory handler which is going to be used for retrieving arguments from and modifying or adding arguments to the memory handler. |
| signal | Int | It is the argument that will be used to control the communication between the threads, in such a way that the thread that executes the instructions checks this parameter to know if there is an incoming telemetry data. |
| logger | Logger | It is a pointer to the Logger class that will enable the Brain class to send the commmands back to the Connection class. |
| instructionBreakPointer | Int | It is the parameter that will be set if the Brain has to stop the execution of the instructions and start over from that point again using the new telemetry data. |
| instructionBreakPointer | Int | It is the parameter that will be set if an error occurred while running the instructions, such as accessing bad memory. It will do this when it encounters 'EBK' line in instruction line. |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| Brain | (constructor) |  | It is the constructor that will create an instance of a Brain classe. |
| intializeMemory | Int | Message:string | It is a method that will be used to send the string to the MemoryHandler so that the memory will be initialized. |
| parseData | Int | Message:string | It is the method that is used to parse the data that will be received from the server through the logger class, so that the Brain successfully understands the message that it gets. |
| startThread | Int |  | It is a method that will start up the thread that is responsible for constantly executing the instructions. |
| execute | Void |  | It is a private method in the Brain that will allow it to do the logical computations of the instructions that are present. |

### Section 2.3.6: Logger

It is the class that keeps the log of all the errors (both run-time and compile-time).

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| logfile | String | Stores log of all the error |
| brain | Brain\* | Keeps connection between logger and the Brain class |
| connection | Connection\* | Keeps connection between logger and the Connection class |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| Logger | (constructor) | logfile:string=errorlog.txt,brain:Brain\* | Creates a logfile named 'errorlog.txt' |
| log | void | logmessage:vector<pair<string,time\_t>> | It writes the error message with the time stamp, to the specified file. |
| sendMessageToServer | int | message:string | Sends message to server. |
| SendMessageToBrain | int | message:string | Sends message to Brain. |

\*Since every message has to pass through the logger class, both sendMessageToServer and sendMessageToBrain methods send the message to the server and Brain, respectively.

### Section 2.3.7: ArgumentType <<enum>>

Transform the information about a variable, more specifically, its type, into an integer that can be easily transported from one class to another.

|  |  |
| --- | --- |
| Type | Description |
| INT | Depicts an Int Variable |
| DOUBLE | Depicts a Double Variable |
| STRING | Depicts a String Variable |
| POSITION | Depicts a Position Variable |
| ROVER | Depicts a Rover Variable |
| HOMEBASE | Depicts a Homebase Variable |
| BOULDER | Depicts a Boulder Variable |
| CRATER | Depicts a Crater Variable |
| MARTIAN | Depicts a Martian Variable |

### Section 2.3.8: SpaceObject class

Keep the information about a SpaceObject. For a Boulder, a Crater of the Homebase we keep the speed and heading variable inaccessible and 0

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| pos | Position | The position of the SpaceObject |
| radius | double | The Radius of the SpaceObject |
| speed | double | The Speed of the SpaceObject (0 in case of Boulder or Crater) |
| heading | double | The Heading of the SpaceObject (0 in case of Boulder or Crater) |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| SpaceObject | (constructor) | message: string | Create a Space Object from the message got from the telemetry data |
| SpaceObject | (constructor) | x: double, y: double, radius: double, speed: double, heading: double | Create a Space Object from the attributes provided |
| getType | ret: int | type: ArgumentType\*, name: string | Get the type of a characteristic of the SpaceObject stored in argument: name. Returns 0 on success, 1 otherwise. |
| getValue | ret: int | value: void\*, name: string | Get the value of a characteristic of the SpaceObject stored in argument: name.  Return 0 on success, 1 otherwise. |
| setValue | ret: int | value: void\*, type: ArgumentType\*, name: string | Set the value of the characteristic 'name' of the SpaceObject to value 'value' if it is of the type 'type'. Return 0 on success, 1 otherwise. |

### Section 2.3.9: SpaceObjectRegister Class

In this class we keep all the current SpaceObjects we have in the World. That is: all boulders, all craters and all Martians that are currently in sight.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| Boulders | List<SpaceObject> | A list of all the Boulders we currently see |
| Craters | List<SpaceObject> | A list of all the Craters we currently see |
| Martians | List<SpaceObject> | A list of all the Martians we currently see |
| varsRegister | VariableRegister\* | A pointer to the VariableRegister for retrieving information about the index |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| SpaceObjectsRegister | (constructor) | varsReg: VariableRegister\* | Create a SpaceObjectsRegister class that keeps a relation with the VariableRegisterClass |
| parseSpaceObjects | ret: int | message: string | Parse the telemtry data and put in the SpaceObjectsRegister |
| clean | void |  | Clean all the Data stored in this Register |
| getType | ret: int | type: ArgumentType\*, name: string | Get the type of variable 'name' from the SpaceObjectsRegister. Returns 0 on success, 1 otherwise. |
| getValue | ret: int | value: void\*, name: string | Get the value of variable 'name' from the SpaceObjectsRegister. Returns 0 on success, 1 otherwise. |
| setValue | ret: int | value: void\*, type: ArgumentType\*, name: string | Set the value of the variable 'name' from the SpaceObjectsRegister to value 'value' if it is of the type 'type'. Return 0 on success, 1 otherwise. |

### Section 2.3.10: VariableRegister Class

The VariableRegister Class will keep track of the variables we currently have created and deleted in the Brain process. It can create, modify and delete variables.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| intMap | Map<string,int> | A map from the Int variables names to the values |
| doubleMap | Map<string,double> | A map from the Double variables names to the values |
| stringMap | Map<string,string> | A map from the String variables names to the values |
| positionMap | Map<string,Position> | A map from the Position variables names to the values |
| intArrayMap | Map<string,vector<int>> | A map from the Int Array variables names to the values |
| doubleArrayMap | Map<string,vector<double>> | A map from the Double Array variables names to the values |
| stringArrayMap | Map<string,vector<string>> | A map from the String Array variables names to the values |
| positionArrayMap | Map<string,vector<Position>> | A map from the Position Array variables names to the values |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| VariableRegister | (constructor) |  | Create an empty VariableRegister |
| clean | void |  | Clean all the Data stored in this Register |
| getType | ret: int | type: ArgumentType\*, name: string | Get the type of variable 'name' from the VariableRegister. Returns 0 on success, 1 otherwise. |
| getValue | ret: int | value: void\*, name: string | Get the value of variable 'name' from the VariableRegister. Returns 0 on success, 1 otherwise. |
| setValue | ret: int | value: void\*, type: ArgumentType\*, name: string | Set the value of the variable 'name' from the VariableRegister to value 'value' if it is of the type 'type'. Return 0 on success, 1 otherwise. |
| deleteValue | ret: int | name: string | Delete the variable 'name' from the VariableRegister. Return 0 on success, 1 otherwise. |

### Section 2.3.11: MapRegister Class

The MapRegister Class will keep track of the Map properties. Most of them cannot be modified. Only the drag of the map can be modified.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| length | double | The length of the Map |
| width | double | The width of the Map |
| timeLimit | int | The timeLimit imposed by the Map |
| drag | double | The drag of the Map |
| homebase | SpaceObject | The Homebase location in the Map |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| MapRegister | (constructor) |  | Create an empty MapRegister |
| parseMapDetails | ret: int | message: string | Parse the initial data and put the content in their places. |
| getType | ret: int | type: ArgumentType\*, name: string | Get the type of variable 'name' from the MapRegister. Returns 0 on success, 1 otherwise. |
| getValue | ret: int | value: void\*, name: string | Get the value of variable 'name' from the MapRegister. Returns 0 on success, 1 otherwise. |
| setValue | ret: int | value: void\*, type: ArgumentType\*, name: string | Set the value of the variable 'name' from the MapRegister to value 'value' if it is of the type 'type'. Return 0 on success, 1 otherwise. |

### Section 2.3.12: RoverDataRegister Class

The RoverDataRegister Class will keep track of Rover's properties and values.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| pos | Position | The current Position of the Rover |
| speedState | int | The current speed state of the Rover (1-3) |
| turningState | int | The current turning state of the Rover (1-5) |
| radius | double | The radius of the Rover |
| speed | double | The current speed of the Rover |
| heading | double | The current heading of the Rover |
| minSensor | double | The minimum sensor property of the Rover |
| maxSensor | double | The maximum sensor property of the Rover |
| maxSpeed | double | The maximum speed property of the Rover |
| maxTurn | double | The maximum turning speed property of the Rover |
| maxHardTurn | double | The maximum hard turning speed property of the Rover |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| RoverDataRegister | (constructor) |  | Create an empty RoverDataRegister |
| init | ret: int | message: string | Parse the initial data and put the content in their places. |
| ParseRover | ret: int | message: string | Parse the telemetry data and put the content in their places. |
| clean | void |  | Clean all the data known about the Rover |
| getType | ret: int | type: ArgumentType\*, name: string | Get the type of variable 'name' from the RoverRegister. Returns 0 on success, 1 otherwise. |
| getValue | ret: int | value: void\*, name: string | Get the value of variable 'name' from the RoverRegister. Returns 0 on success, 1 otherwise. |
| setValue | ret: int | value: void\*, type: ArgumentType\*, name: string | Set the value of the variable 'name' from the RoverRegister to value 'value' if it is of the type 'type'. Return 0 on success, 1 otherwise. |

### Section 2.3.13: MemoryHandler Class

The MemoryHandler Class will keep track of all the registers, which are preset variables and the modifiable variables. It acts like a Connection from the Register Classes and the Brain class. This class also keeps the current time, that is the time in the map that we last received from a Telemetry data.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| cTime | int | A map from the Int variables names to the values |
| spaceObjects | Map<string,double> | A map from the Double variables names to the values |
| variables | Map<string,string> | A map from the String variables names to the values |
| rover | Map<string,Position> | A map from the Position variables names to the values |
| map | Map<string,vector<int>> | A map from the Int Array variables names to the values |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| MemoryHandler | (constructor) |  | Create an empty MemoryHandler with created Memory Registers |
| init | ret: int | message: string | Parse an Initial data and send it to the respective Registers for further parsing |
| parseData | ret: int | message: string | Parse a Telemetry data and send it to the respective Registers for further parsing |
| clean | ret: int |  | Clean all the Data stored in this Registers |
| getType | ret: int | type: ArgumentType\*, name: string | Get the type of variable 'name' from the respective Register. Returns 0 on success, 1 otherwise. |
| getValue | ret: int | value: void\*, name: string | Get the value of variable 'name' from the respective Register. Returns 0 on success, 1 otherwise. |
| setValue | ret: int | value: void\*, type: ArgumentType\*, name: string | Set the value of the variable 'name' from the respective Register to value 'value' if it is of the type 'type'. Return 0 on success, 1 otherwise. |
| deleteValue | ret: int | name: string | Delete the variable 'name' from the respective Register. Return 0 on success, 1 otherwise. |

### Section 2.3.14: InstructionIF

The Instruction Interface that all Instructions will inherit from. Each instruction will be an instance of the respective class. It will have a vector of strings as argument. The instruction instance will parse the arguments at execution time and give a run-time error if there are problems.

Attributes:

|  |  |  |
| --- | --- | --- |
| Name | Type | Description |
| argc | int | The number of arguments the instruction has. |
| argv | vector<string> | The arguments kept in string format. |

Operations:

|  |  |  |  |
| --- | --- | --- | --- |
| Name | Return | Arguments | Description |
| execute | ret: int | memoryLocation: MemoryHandler\*, instrPointer: int\* | Execute the instance of this instruction using the memoryLocation and the instrPointer. Return value 'ret' is 0 on success, 1 otherwise. |
| InstructionIF | (constructor) | ret: int\*, args: vector<string> | Create an Instruction instance with the args as the list of arguments. The location pointed by ret will be 0 if the Constructor is successful, 1 otherwise. |

# Section 3: Language Description and Specification

*Detailed description of syntax and function of language*

## Section 3.1: Overview

The language was designed to provide simple operations of assembler type equipped with some very useful high level functions. Each line of the Brain File contains a single instruction. Brain Requirements are described in section (2) and the function syntax and description is presented in the revelant section.

The variable types used in the assembler are: Int, Double, Position, Label and Immediate, or Boulders, Craters, Martians or Homebase. The Brain Programmer can create some variables types and their respective arrays. To see how this works check the relevant section.

When the brain receives telemetry data from the server the received data is put in its respective registers. The data is stored in its respective registers until the server receives another telemetry data that replaces the old one. It is up to the programmer to keep information from past messages, which is easily done with the VariableRegister.

There are preset variables to access the information related to the Rover, Boulders, Craters, Martians, Homebase and Map. To view the preset variables, check the relevant section. The Brain Programmer can edit some of the values of the preset registers for easier usage of some high-level functions. This provides more flexibility and it does not corrupt memory because the modifiable data is updated with every telemetry message received.

## Section 3.2: Brain Requirements

The Brain code must meet the following requirements:

* + 1. Each line of the Brain file must contain an instruction starting with one or zero labels.
    2. A instruction is specified by the instruction name (3-letter word) and a number of arguments.
    3. The arguments are separated by a space and does not contain spaces inside them.
    4. The brain file must only contain lowercase characters, uppercase characters, numbers or one of the following {'.', '#', '\*', '[', ']'}
    5. The brain file must contain either one or no call of the function: 'brk'. The same is the case for function: 'ebk'.
    6. There must be no two labels declared with the same name.

An argument must meet the following requirements:

* + 1. An argument that starts with a lowercase or uppercase character is a Variable that relates to the memory and is handled by the MemoryHandler. In this case the argument must either be a Preset Variable or be created before the execution of this instruction.
    2. An argument that starts with a number is an Immediate constant which can be an Int or a Double. If the argument contains a '.' character then the argument represents a Double Immediate constant
    3. An argument that starts with the character '\*' represents an Immediate constant that is a String. The part of the argument after the '\*' is the represented String.
    4. An argument that starts with the character '#' represents a Label variable.
    5. The argument which is a variable and contains the '[' ']' characters must either be a preset variable, where the information inside the brackets must be an Int or an Immediate Constant, or part of an array.

## Section 3.3: Instruction Usage

In the brain code the functions have the following syntax:

[<label>] <function-name> [<argument1>] [<argument2>] … [<argument4>]

Where the <function-name> is a 3 character string that describes the function. View relevant section for all function names and individual description. Between the tags <label>, <function-name> and <argument*x*> a space is required.

All functions have a fixed number of arguments specific to each function. Check relevant section for a description of each function.

## Section 3.4: Preset Variables:

i) The Rover Register

|  |  |  |  |
| --- | --- | --- | --- |
| Preset Variable | Type | Description | Modifiable by Brain Programmer |
| Rover | Rover | The Rover | No |
| Rover.Position | Position | Position of the rover | Yes |
| Rover.SpeedState | Int | The speed state (1='b' , 2='-',3='a') | Yes |
| Rover.TurningState | Int | The Turning state  (1 = 'L' 2 = 'l' 3 = '-' 4 = 'r' 5 = 'R') | Yes |
| Rover.Radius | Double | The radius of the Rover | No |
| Rover.Speed | Double | The current velocity(speed) of the Rover | Yes |
| Rover.Heading | Double | The angle(heading) of the Rover related to the East Direction | Yes |
| Rover.MinSensor | Double | The Minimum Sensor of the Rover | No |
| Rover.MaxSensor | Double | The Maximum Sensor of the Rover | No |
| Rover.MaxSpeed | Double | The Maximum Speed of the Rover | No |
| Rover.MaxTurn | Double | The Maximum Turn of the Rover | No |
| Rover.MaxHardTurn | Double | The Maximum Hard Turn of the Rover | No |

ii) The Map Register

|  |  |  |  |
| --- | --- | --- | --- |
| Preset Variable | Type | Description | Modifiable by Brain Programmer |
| Map.Length | Double | Length of the Map | No |
| Map.Width | Double | Width of the Map | No |
| Map.TimeLimit | Int | Time limit of the Map | No |
| Map.Drag | Double | The Drag of the Map | Yes |
| Map.Homebase | Homebase | The Homebase of the Map | No |
| Map.Homebase.Position | Position | The Position of the Homebase | No |
| Map.Homebase.Radius | Double | The Radius of the Homebase | No |
| Map.Homebase.X | Double | The X Coordinate of the Homebase | No |
| Map.Homebase.Y | Double | The Y Coordinate of the Homebase | No |

iii) The SpaceObjects Register

|  |  |  |  |
| --- | --- | --- | --- |
| Preset Variable | Type | Description | Modifiable by Brain Programmer |
| NumberBoulders | Int | The Number of the Boulders we currently have in range | No |
| NumberCraters | Int | The Number of the Craters we currently have in range | No |
| NumberMartians | Int | The Number of the Martians we currently have in range | No |
| Boulder[i] | Boulder | The 'i'th Boulder where i > 0 and i <= NumberBoulders | No |
| Crater[i] | Crater | The 'i'th Crater where i > 0 and i <= NumberCraters | No |
| Martian[i] | Martian | The 'i'th Martian where i > 0 and i <= NumberMartians | No |
| Boulder[i].Position | Position | The Position of the 'i'th Boulder | Yes |
| Boulder[i].Radius | Double | The Radius of the 'i'th Boulder | Yes |
| Boulder[i].X | Double | The X Coordinate of the 'i'th Boulder | Yes |
| Boulder[i].Y | Double | The Y Coordinate of the 'i'th Boulder | Yes |
| Crater[i].Position | Position | The Position of the 'i'th Crater | Yes |
| Crater[i].Radius | Double | The Radius of the 'i'th Crater | Yes |
| Crater[i].X | Double | The X Coordinate of the 'i'th Crater | Yes |
| Crater[i].Y | Double | The Y Coordinate of the 'i'th Crater | Yes |
| Martian[i].Position | Position | The Position of the 'i'th Martian | Yes |
| Martian[i].Radius | Double | The Radius of the 'i'th Martian | Yes |
| Martian[i].X | Double | The X Coordinate of the 'i'th Martian | Yes |
| Martian[i].Y | Double | The Y Coordinate of the 'i'th Martian | Yes |
| Martian[i].Speed | Double | The Speed of the Martian | Yes |
| Martian[i].Heading | Double | The Heading of the Martian | Yes |

iv) Other Registers used

|  |  |  |  |
| --- | --- | --- | --- |
| Preset Variable | Type | Description | Modifiable by Brain Programmer |
| CurrentTime | Int | The Time that we received with the Telemetry Data | No |

## Section 3.5: Argument Types:

There are 6 argument types that can be used in the low level functions. The project also has 5 high level argument types that can be used. Types are described in the following table:

|  |  |  |  |
| --- | --- | --- | --- |
| Use | Argument Type | Description | Comment |
| low-level | Int | Integer value |  |
| Double | Rational Value |  |
| String | List of characters |  |
| Position | Pair of two rational values | Can access each of its values |
| Immediate | Immediate constant | constant that depicts an int, a double or a string without using a variable |
| Label | A string representing a number | is changed during compile time with an int immediate constant representing the line to jump to |
| high-level | Rover | Rover class |  |
| Homebase | Homebase class |  |
| Boulder | Boulder class |  |
| Crater | Crater class |  |
| Martian | Martian class |  |

## Section 3.6: Instructions

#### Section 3.6.1: Arithmentic Operations

**- Function-name: add**

- Name meaning: addition

- Number of Arguments: 3

- Argument1:

Possible Type: Int, Double, Position

- Argument2:

Possible Type: Int, Double, Position, Immediate

- Argument3:

Possible Type: Int, Double, Position, Immediate

- Result: Argument1 = Argument2 + Argument3

- Description: Get Argument2 and Argument3 from memory, add them and store the result in the Argument1 Variable.

If Argument1, Argument2 or Argument3 are not immediate and are not created before this instruction call, there will be a run-time error.

- Argument Type possibilities:

1) Argument1, Argument2 an Argument3 are all of the same type: Int, Double or Position

2) Argument1 is of type Int and Argument2 and Argument3 are of type Int or Immediate(representing an Int)

3) Argument1 is of type Double and Argument2 and Argument3 are of type Double or Immediate(representing a Double)

**- Function-name: sub**

- Name meaning: subtractions

- Number of Arguments: 3

- Argument1:

Possible Type: Int, Double, Position

- Argument2:

Possible Type: Int, Double, Position, Immediate

- Argument3:

Possible Type: Int, Double, Position, Immediate

- Result: Argument1 = Argument2 - Argument3

- Description: Get Argument2 and Argument3 from memory, substract them and store the result in the Argument1 Variable.

If Argument1, Argument2 or Argument3 are not immediate and are not created before this instruction call, there will be a run-time error.

- Argument Type possibilities:

1) Argument1, Argument2 an Argument3 are all of the same type: Int, Double or Position

2) Argument1 is of type Int and Argument2 and Argument3 are of type Int or Immediate(representing an Int)

3) Argument1 is of type Double and Argument2 and Argument3 are of type Double or Immediate(representing a Double)

**- Function-name: mul**

- Name Meaning: multiplication

- Number of Arguments: 3

- Argument1:

Possible Type: Int, Double, Position

- Argument2:

Possible Type: Int, Double, Position, Immediate

- Argument3:

Possible Type: Int, Double, Immediate

- Result: Argument1 = Argument2 \* Argument3

- Description: Get Argument2 and Argument3 from memory, multiply them and store the result in the Argument1 Variable.

If Argument1, Argument2 or Argument3 are not immediate and are not created before this instruction call, there will be a run-time error.

- Argument Type possibilities:

1) Argument1, Argument2 an Argument3 are all of the same type: Int or Double

2) Argument1 is of type Int and Argument2 and Argument3 are of type Int or Immediate(representing an Int)

3) Argument1 is of type Double and Argument2 and Argument3 are of type Double or Immediate(representing a Double)

4) Argument1 and Argument2 are of type Position and Argument3 is of type Double or Immediate(representing a Double)

**- Function-name: div**

- Name Meaning: division

- Number of Arguments: 3

- Argument1:

Possible Type: Int, Double, Position

- Argument2:

Possible Type: Int, Double, Position, Immediate

- Argument3:

Possible Type: Int, Double, Immediate

- Result: Argument1 = Argument2 / Argument3

- Description: Get Argument2 and Argument3 from memory, divide them and store the result in the Argument1 Variable.

If Argument1, Argument2 or Argument3 are not immediate and are not created before this instruction call, there will be a run-time error.

- Argument Type possibilities:

1) Argument1, Argument2 an Argument3 are all of the same type: Int or Double

2) Argument1 is of type Int and Argument2 and Argument3 are of type Int or Immediate(representing an Int)

3) Argument1 is of type Double and Argument2 and Argument3 are of type Double or Immediate(representing a Double)

4) Argument1 and Argument2 are of type Position and Argument3 is of type Double or Immediate(representing a Double)

**Section Conditional Functions**

**- Class of Functions with function-name: if[Lle]**

- Functions: ifL, ifl, ife

- Name Meaning:

ifL = if x1 is less or equal than x2 then …

ifl = if x1 is less than x2 then …

ife = if x1 equals x2 then …

- Number of Arguments 3

- Argument1:

Possible Type: Int, Double, Immediate

- Argument2:

Possible Type: Int, Double, Immediate

- Argument3:

Possible Type: Label, Immediate

- Result: if (Argument1 OP Argument2) then jump to line Argument3.

if function-name is ifL then OP is <=

if function-name is ifl then OP is <

if function-name is ife then OP is =

- Description: Perform a conditional statement. In case the condition is true the Instruction Pointer is set to the Argument3 variable.

If Argument1 or Argument2 are not immediate and are not created before this instruction call, there will be a run-time error.

- Argument types:

1) Argument1 and Argument2 are one of the types: Int or Immediate(representing an Int)

2) Argument1 and Argument2 are one of the types: Double or Immediate(representing a Double)

**- Class of Functions with function-name: ie[Lle]**

- Functions: ieL, iel, iee

- Name Meaning:

ieL = if x1 is less or equal than x2 then … else ...

iel = if x1 is less than x2 then … else ...

iee = if x1 equals x2 then … else ...

- Number of Arguments 4

- Argument1:

Possible Type: Int, Double, Immediate

- Argument2:

Possible Type: Int, Double, Immediate

- Argument3:

Possible Type: Label, Immediate

- Argument4:

Possible Type: Label, Immediate

- Result: if (Argument1 OP Argument2) then jump to line Argument3 else jump to line Argument 4.

if function-name is ifL then OP is <=

if function-name is ifl then OP is <

if function-name is ife then OP is =

- Description: Perform a conditional statement. In case the condition is true, the Instruction Pointer is set to the Argument3 variable, else it is set to Argument4.

If Argument1 or Argument2 are not immediate and are not created before this instruction call, there will be a run-time error.

- Argument types:

1) Argument1 and Argument2 are one of the types: Int or Immediate(representing an Int)

2) Argument1 and Argument2 are one of the types: Double or Immediate(representing a Double)

#### Section 3.6.3:Jump functions

**- Function-name: jmp**

- Name Meaning: jump

- Number of Arguments: 1

- Argument1:

Possible Type: Label, Immediate

- Result: Jump to position Argument1

#### Section 3.6.4:Memory functions

**- Function-name: put**

- Name Meaning: put

- Number of Arguments: 2

- Argument1:

Possible Type: Int, Double, Position, String

- Argument2:

Possible Type: Int, Double, Position, String, Immediate

- Result: Argument1 = Argument2

- Description: Get Argument2 from memory, and store it into the Argument1 Variable.

If Argument1 or Argument2 are not immediate and are not created before this instruction call, there will be a run-time error.

- Argument Type possibilities:

1) Argument1 and Argument2 are both of the same type: Int, Double, Position, String

2) Argument1 is of type: Int, Double, Position or String and Argument2 is the respective Immediate variable

**- Function-name: crt**

- Name Meaning: create

- Number of Arguments: 2

- Argument1:

Possible Type: Immediate(String)

- Argument2:

Possible Type: Immediate(String)

- Result: Create Variable Argument2 of type Argument1 and initialize it with the standard initializing value.

- Description: Argument1 and Argument2 are Immediate constants representing a string. Argument2 must represent a good variable name. If Argument2 is already created, there will be a run-time error.

**- Function-name: cra**

- Name Meaning: create array

- Number of Arguments: 3

- Argument1:

Possible Type: Immediate(String)

- Argument2:

Possible Type: Immediate(Int)

- Argument3:

Possible Type: Immediate(String)

- Result: Create Variables Array Argument3 of type Argument1 of Argument2 elements. Initialize it with the standard initializing value.

- Description: Argument1 and Argument3 are Immediate constants representing a string. Argument2 is an Immediate constant representing an Int. Argument3 must represent a good variable name. If Argument3 Variable is already created, there will be a run-time error. Also, a wrong representation of the Immediate constants will lead to a run-time error.

**- Function-name: del**

- Name Meaning: delete

- Number of Arguments: 1

- Argument1:

Possible Type: Immediate(String)

- Result: Delete variable represented by Immediate Constant Argument1.

- Description: Argument1 is an Immediate Constant representing a String which variable is to be deleted. The variable can be an Array, an Int, a Double, a Position or a String.

- Comment: The program waits for an Immediate String instead of a specific variable name in order to be able to delete Arrays of variables.

#### Section 3.6.5:Communication with the brain

**- Function-name: ret**

- Name Meaning: return

- Number of Arguments: 1

- Argument1:

Possible Type: String, Immediate(String)

- Result: Send a message Argument1 to the Server

- Description: If Argument1 is not Immediate or it is not created, there will be a run-time error.

**- Function-name: wat**

- Name Meaning: wait

- Number of Arguments: 0

- Result: Wait for a message

- Description: Wait for telemetry data from the Server

**- Function-name: brk**

- Name Meaning: break

- Number of Arguments: 0

- Result: Break point

- Description: If the Brain executor is not waiting for Telemtry data by using the wat function, the instruction pointer will jump to the first instruction after this instruction.

- Constraints: There can be only one function of this type in the whole brain file.

**- Function-name: ebk**

- Name Meaning: error break

- Number of Arguments: 0

- Result: Error Break point

- Description: If the Brain executor occured a run-time error the Instruction pointer will jump to the first instruction after this instruction.

- Constraints: There can be only one function of this type in the whole brain file.

**- Function-name: cln**

- Name Meaning: clean

- Number of Arguments: 0

- Result: Clean the Variable Registers

- Description: Delete all the memory allocated up to this point.

#### Section 3.6.6:High level functions

**- Function-name: DST**

- Name Meaning: Distance

- Number of Arguments: 3

- Argument1:

Possible Type: Double

- Argument2:

Possible Type: Rover, Homebase, Boulder, Crater, Martian, Position

- Argument3:

Possible Type: Rover, Homebase, Boulder, Crater, Martian, Position

- Result: Argument1 becomes the distance between Argument2 and Argument3

- Description: Calculate the distance between Argument2 and Argument3 at this point in time and store the result into Argument1. If Argument1, Argument2 or Argument3 does not exist there will be a run-time error.

**- Function-name: SIM**

- Name Meaning: Simulate

- Number of Arguments: 3

- Argument1:

Possible Type: Position

- Argument2:

Possible Type: Int, Immediate(Int)

- Argument3:

Possible Type: Rover, Martian

- Result: Calculate Argument3's position after Argument2 seconds and store it in Argument1

- Description: Given the current information known about the Argument3, find at what position will it be after Argument2 seconds and store the Position into Argument1. If Argument1, Argument2 or Argument3 are not Immediate or does not exist there will be a run-time error.

**- Function-name: GTA**

- Name Meaning: Get Angle

- Number of Arguments: 3

- Argument1:

Possible Type: Double

- Argument2:

Possible Type: Position

- Result: Calculate Argument2's angle related to point (0,0) and store it in Argument1

- Description: Get the angle of the Argument2 Position related to the Position (0,0). Store the result in the Argument1 variable.

If Argument1 or Argument2 does not exist there will be a run-time error.

**- Function-name: GAP**

- Name Meaning: Generate All Possibilities

- Number of Arguments: 3

- Argument1:

Possible Type: Immediate(String)

- Argument2:

Possible Type: Immediate(String)

- Argument4:

Possible Type: Int, Immediate(Int)

- Result: Simulate all possible directions after Argument3 seconds and put the results in newly created Array of Positions Argument1 and Array of Strings Argumen2, both of length 9.

- Description:

- create Argument1 as Position Array of length 9

- create Argument2 as String Array of length 9

- for all possible messages to send to the Rover modify the Rover's direction and find where it will be after Argument3 seconds.

- Put the Positions in the Argument1 variable and the respective direction in the Argument2 variable.

### Section 3.7:Example of Brain Language Usage

We can write the Brain file:

#Beginning crt \*Double \*ang

crt \*Position \*npos

#Start sub npos Map.Homebase.Position Rover.Position

GTA ang npos

sub ang Rover.Heading ang

ifL 180.0 ang #TurnLeft

ifl ang 0.0 #TurnLeft

ife ang 0.0 # GoOn

#TurnRight ret \*r

jmp #NextStep

#TurnRight ret \*l

jmp #NextStep

#GoOn crt \*Int \*state

put state Rover.TurningState

ife state 1 #ShiftDoubleRight

ife state 2 #ShiftRight

ife state 3 #Accelerate

ife state 4 #ShiftLeft

ife state 5 #ShiftDoubleLeft

#ShiftDoubleRight ret \*r

#ShiftRight ret \*r

jmp #Accelerate

#ShiftDoubleLeft ret \*l

#ShiftLeft ret \*l

jmp #Accelerate

#Accelerate put ang Rover.Speed

put state Rover.SpeedState

iee state 3 #Finish #Acc

#Acc ret \*a

#Finish del \*state

jmp #NextStep

#NextStep wat

jmp #Start

#Error Handling brk

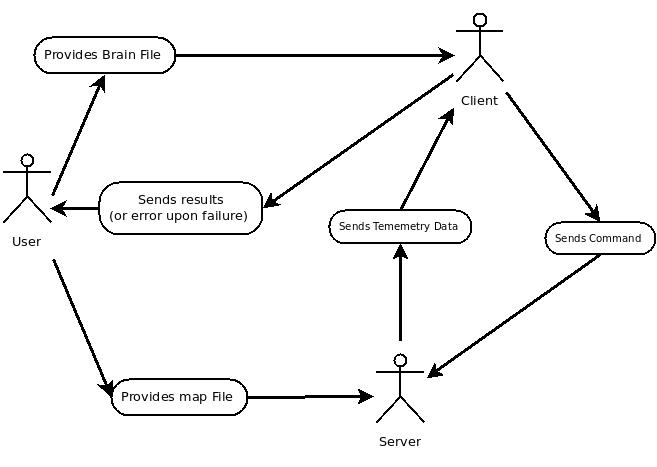
ebk

cln

jmp #Beginning

# Section 4: Use Case

*Possible usage scenarios that give insight into system workflow*



Use case diagram is the representation of user interaction with the system. User, Client and Server are the actors in the use case diagram above. Client consists of components like Brain, Connection, Parser, Logger and Socket. The underlying functionalities of all these classes are explained in the later part.

Basically, User gives a brain file to the client, and the client has to act upon the brain file. The communication between server and client is in term of exchanging of telemetry data and commands.

# Section 5: Interactions

*Description of how components interact with each other inside the system in sequence*

In order to have better management of the processes of handled by our program, two threads were implemented : one for the thread that waits for data sent through the server and another one for waiting for handling sudden messages from he server in the brain class, which is the class that is created after the Brain File has been compiled.

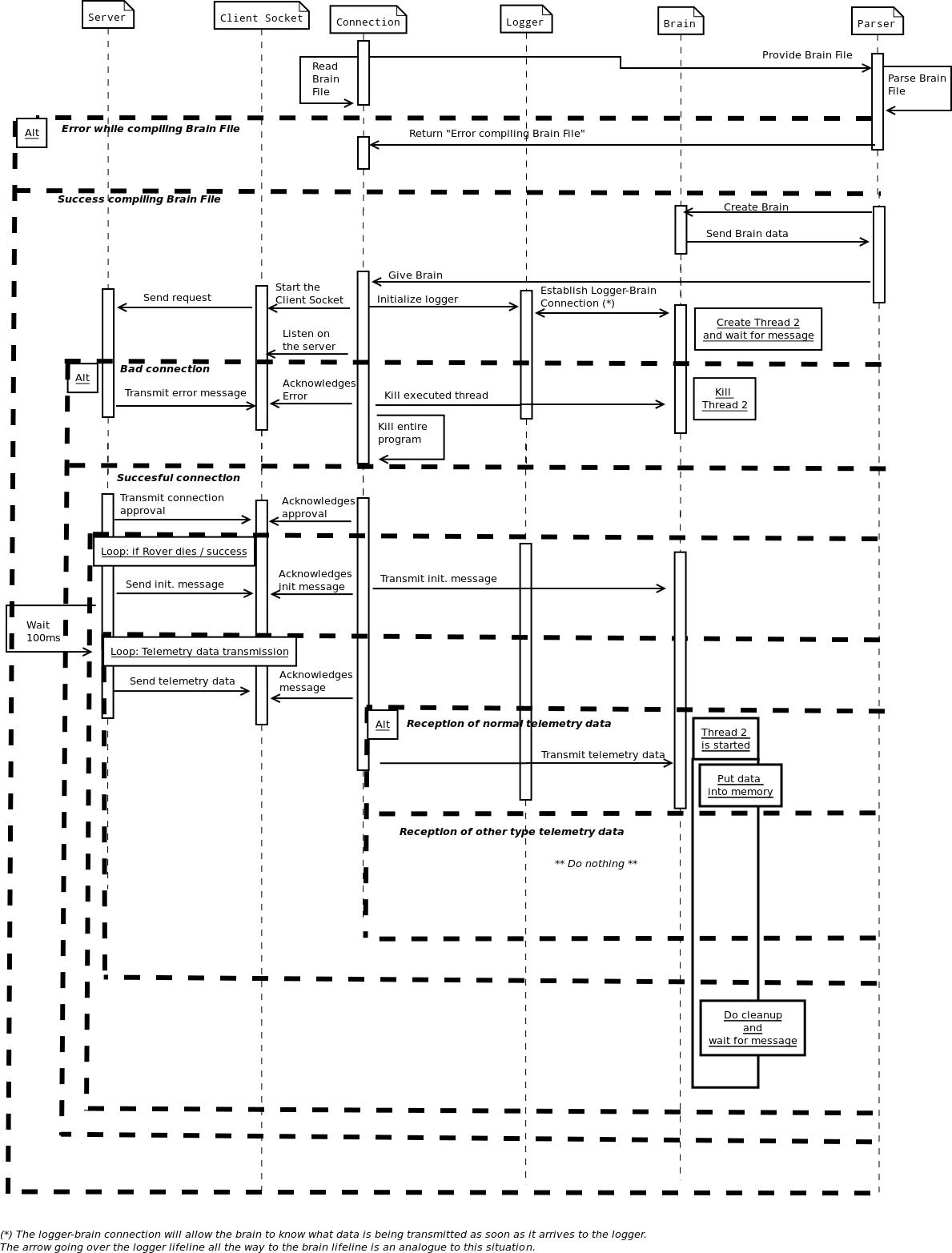
In the first sequence diagram, we started by having the connection providing the Brain File before we started the if cases. We looked for describing the possible errors and how it is handled at each stage of communication between the Brain and the Server, which, as it can be seen, ended in nested if statements and nested loops. In the end (either a failure or a success in on returning to the home base), we make sure to do a cleanup to clean most of the memory for preparing for the next round.

The second sequence diagram is basically an enhanced version of the loop we see in the “Reception of Telemetry Data” if-case in the first diagram. We check for the given by thread 1 and then we perform the instructions, and after that we check for proper memory handling. In case there is not any errors, we send a confirmation message and hold on for the next instruction.

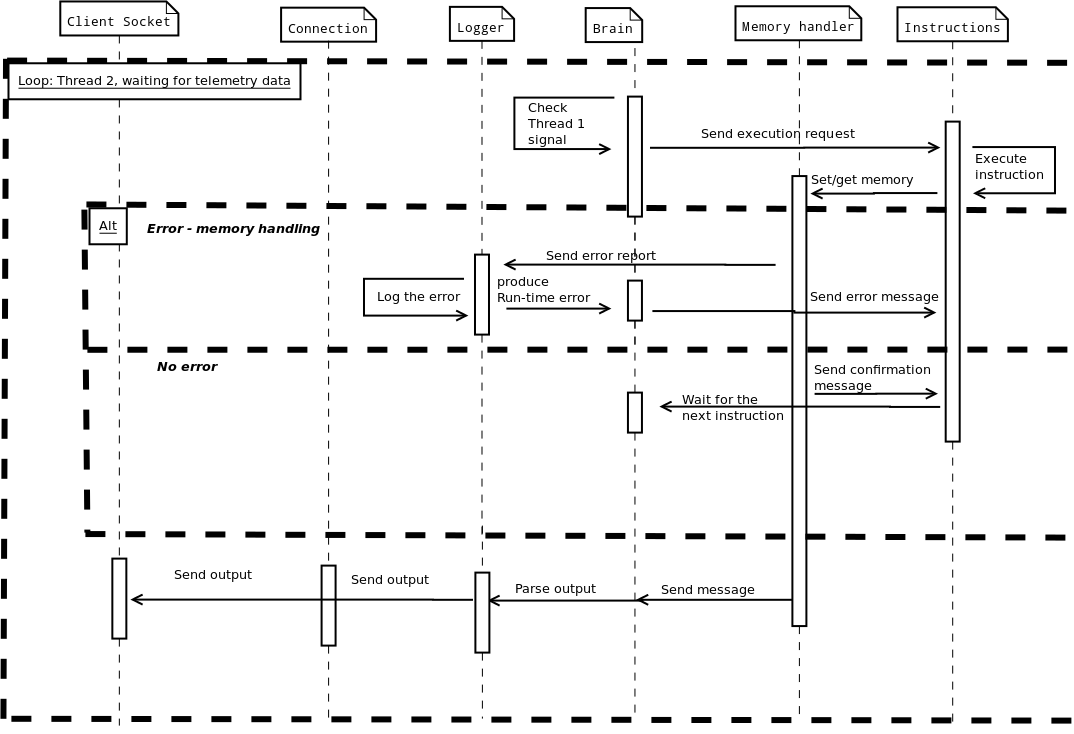
It is important to point out that in both sequence diagrams the logger plays a very important role saving all the communication that is being done from starting the Rover until finishing the mission. We located the logger in between the Connection and the Brain and in both diagrams for a better depiction of its functionality.

See the diagram on the next page for the first thread’s sequence information. (Digram 1)

See the diagram on the second next page for the second thread’s sequence information. (Diagram 2)

***Diagram 1***

***Diagram 2***



# Section 6: Quality Assurance

*Discussion about how the design fulfills requirements and testing approach*

## Section 6.1: Quality Requirements

Section 6.1.1: General Discussion

One of paradigms of the design that has been considered of high importance is flexibility, robustness and reliability. These traits should not only describe the language itself , but also the interpreter, and ideally the whole system\* and all its integrating parts. In this sense , this part of the document is concerned with quality assurance of this project , meaning:

* Initial “client” requirements and how these are addressed by the design;
* Requirements that were not immediately clear from the initial description but which the design team considers natural implications of these initial requirements ,and how they might be addressed;
* Emergent requirements that have come up as a consequence of design decisions and system integration considerations and how these are addressed;
* Testing and validation of the design.

The reader should note that where possible, these ways of addressing the requirements are given only as suggestions from the design team, and are not integral parts of the system itself ( for example, the system could in theory function perfectly fine without a logger, but much of the robustness is lost ).

The section on testing is also meant to be only a guideline and does not need to be strictly followed by the developer. However, the implementer should take into consideration that these guidelines are written by the original designers, who have a good understanding of their own design.

Overall can also be used as a sort of quick reference in case some features are not completely clear after reading the more abstract part of the documentation ( if it is not completely clear what the purpose of some components is , then these requirements and the testing guidelines should provide more insight ).

The appendix section also includes 2 templates which can help with quality assurance – a test script template and a test case/use case scenario template **.**

It is also important to note that quality assurance is a continuous process, and the considerations from this section should be updated and re-checked throughout all the phases of production ( design ,coding ,integration, implementation , validation … ) , and through the product’s lifetime.

### Section 6.1.2: Flexibility Requirements

This design is based on language flexibility. Namely, this design allows:

* Easy configuration of important parameters of the system and scalability :

-initial conditions can easily be set and modified , and the function of the system is not dependent on this;

-multi-thread approach allows good configurability and task division within the system;

-memory is highly scalable and easily modifiable, both from the “user” point of view

( compile-time and run-time variable declaration ) and from the “implementer” point of view

( component separation and clear delimiting allows easy “modding” of the system architecture , especially of the memory component, by adding registers for example ) ;

-having the logger at any communication between the Connection and the Brain allows flexible and clear modification of logging options , implementation of new logging possibilities ( logging to special formats for example ) and flexible error management .

* Possibility of implementation of theoretically any kind of algorithm, lower and higher level:
* high language flexibility, allowing usage of low-level and high-level functions at the same time, including important mathematical operations and tools ( basic arithmetic operations but also very comprehensive switching statements and easily configurable jump statements ) allow creation of loops , checking of conditions and complex calculations;
* flexible memory management allows compile-time and run-time creation of variables and data storage in all forms that can be used in calculations in this context ( integers , doubles, and even strings for labling and for output ) ;
* special memory register for storing data values with time-stamps , allows time-dependent algorithms and a way to track history if this is needed in calculations or algorithm logic;
* highly modular and moddable instructions allow easy rectification of design in case new requirements arise or it is found that some functions are missing/outdated/deprecated/ etc .
* High level atomicity with logical background:
* atomic objects are split in a logical way that relates to the communication between them,their functions, dependencies on eachother , and general system integration;
* it is easy to add new actors or modify existing ones , with no or minimal effect on other classes or the system as a whole;

One must be aware that there are also some disadvantages to having high flexibility, but the pros outweigh the cons in this case. Some disadvantages might be :

* Less user-friendliness ;
* Steeper learning-curve for implementer;
* Possible security risks ;
* Higher development cost ( especially time-wise) .

All these disadvantages are ,however, taken into account and addressed as much as possible:

-this extensive documentation also includes example documents and much writing on a sample or suggested implementation and testing;

-all communications pass through the logger where they are parsed and checked for validity , and memory management has its own handler which makes sure no invalid operations are performed ;

-high atomicity ensures ease of development and testing in a component-driven fashion , in spite of the larger number of classes.

#### Section 6.1.3: Robustness Requirements

Robustness is also an important requirement of this project, and a very good level of robustness is achieved with two design approaches:

Having an intermediate step between all inter-component communication ( the logger and the memory handler ) ;

High atomicity and minimal dependence;

FIrstly , the intermediate steps assure that all information is parsed before being handled , so that any communication error can be correctly handled ( however this is desired by the implementer ) ,and the system can continue functioning without catastrophic failure.

It can be seen from the sequence diagrams that at any point , the messages between the Communication component and the Brain component all pass through the logger. The logger parses the info and makes sure it respects the expected format . Thus,incorrect or mailicious imput cannot be injected into the brain file . At the same time , the Brain file is not aware of the connection’s existence,and cannot be influenced by it .It only knows about the logger ,and communicates through this.

A similar situation happens “inside” the brain file when the instructions are executed : all instructions that require memory access will have to access the memory through the memory handler, which checks whether the actions first of all make any sense ,and if they are valid . It is thus here also not possible to inject malicious code or mistakenly break down the operation of the system.

in case of faulty implementation and one of the components does break down , all other components are separate , and the logger also handles every line of execution , so it can accurately see where the breakdown has occurred.

It is up to the decision of the implementer how to treat this logging (either to file,to console,to another stream ….) and how to treat exceptions or errors.

#### Section 6.1.4: Reliability Requirements

Thorough documentation and comprehensive design help build up reliability, but ultimately this quality comes down to how thorough and relevant the testing is. This is discussed in the next section.

## Section 6.2: Testing

#### Section 6.2.1: General Testing Discussion and Approach

Testing does not have to be more complicated than a simple “check results against expectations” algorithm. This is all there is to it . What matters is testing consistency , and answers to 4 questions :

* What to test?
* What NOT to test?
* How to test?
* When to test?

From the perspective of the item under test, this design team sees the tests as being split into these categories:

1. **Positive Tests**

-tests that expect a SUCCESS

1. **Conformance Test ( High Priority )**

Does the item under test fulfill its expected action correctly?

NOTE: A conformance test passes if for ALL CORRECT INPUTS, there is a CORRECT OUTPUT/BEHAVIOUR

**Example:**

Test scenario:Unit test of SpaceObject class

Item under test:Getters and setters of SpaceObject class

Expected behaviour: Before any values are set, the getters should return the default values, in the correct format , without raising any exceptions or setting any error codes.

Setters should set a given value in correct format and without rasing any exceptions or setting any error codes.

After a value is set, getters should return the set value , in the correct format , without raising any exceptiosn or setting any error codes.

1. **Stress Tests ( High priority,where applicable )**

Does the item under test behave as expected under high load ?

1. **Negative tests**

-tests that expect a FAILURE and correct treatment of this error

**a. Negative conformance test ( High priority )**

Does the item under test correctly handle wrong input ?

***NOTE:*** A negative conformance test passes if for ALL WRONG INPUTS,there is a CORRECT OUTPUT/BEHAVIOUR.

**b.Connection Fault Management (Medium Priority )**

Does the item under test correctly handle loss of communication with another component ?

**c.Configuration Error Management ( High Priority )**

Does the item under test correctly recognize and react to wrong configuration?

**d.Tear-Down Error Management**

Does the component correctly handle tear-down operation ? ( disposal of itself and all other classes that it should dispose ? )

Example: checking for memory leaks;

From the perspective of how many items are currently under test or affected by the test, there are other categories:

1. Unit Tests

Only one component is under test. This test should not include or affect any other components.

1. Integration Test

One or more components are under test and the focus is on how the components interact with each other.

1. System Test

The whole system is the item of the test .

From the perspective of whether or not access to the code at runtime is required , there can be white-box or black-box tests.

From the perspective of whether or not the tests are critical for validation of the component , they can be critical or non-critical.

It is important to carefully label tests according to these categories,so that not only the testers themselves but also anyone interested in information about the test have a better insight into what the respective test actually does.

#### Section 6.2.2: What to Test

For this project , stress testing is not extremely relevant , but all the others should be high priority.

#### Section 6.2.3: What Not to Test

Since there is no access to the inner workings of the server , this part of the system cannot be covered by tests from this project.

Testing tools,if any used, are also not part of the scope of the tests from this document and are assumed to be working correctly.

TCP protocol communication and the sockets are also not part of this design and thus are not covered by the tests.

Wrong input from these should still be covered by the other component tests though.

#### Section 6.2.4: How to test

Abstracting away from the testing environment, these steps should be followed:

Always after the development of a component is complete, unit tests scenarios should be written for it , with coverage as close to 100% as possible. Next, integration tests should be written to make sure this component correctly communicates with the other components in the system .

After critical tests are passed, only then the component is considered “ complete” .

In a more concise way:

* Finish implementation of component ( component now enters testing phase )
* Write Unit tests and integration test scenarios ( each with multiple scripts )
* Go through the test,fix errors
* When all critical tests pass,the component is considered “released” from testing.

A testing environment can make testing much more user-friendly. One such tool is CppTest,which is open source and freely available .

A sample test scenario template with sample script can be found in the appendix .

#### Section 6.2.5: When to Test

As previously mentioned , a component is not considered “ ready for deployment” until it has passed all the critical tests. This means that components should be tested as soon as they are coded.

#### Section 6.2.6: Criteria for Entering Test Phase

Testing is considered passed once all critical tests for the component have been passed. Which tests are critical and which are not are up to the developers.

#### Section 6.2.7: Criteria for release

Testing is considered passed once all critical tests for the component have been passed. Which tests are critical and which are not are up to the developers.

#### Section 6.2.8: Test Information Management

it is important that the information gathered from the testing phase is not lost in between design or development stages. In the current approach , the person who makes the test scenario and test case must also include information about what to do after the test, such as where to store the information.

Allthough irrelevant for the design phase, it is suggested that in the development phase, a new document is written which has general testing information management agreements which have to be followed by all development teams working on the project.

Centralization of all test information is paramount to the continuous reliability of the project and not only serves as a proof of development team thoroughness but also greatly helps evolution and future development of the software. it is also very important that all teams follow the same standards on notation, labeling and procedure ( a suggested course of action is given in 6.2.1 and following subsections ).

# Section 7: Administrative Requirements

*Miscellaneous non-functional information such as role assignment and timetable*

Roles:

* Developer
* Tester
* Validator

Documentation/Supporting documents:

* Design Documentation ( present )
* Testing Centralization document ( for ex xls file )
* Test Scenarios
* Test Scripts

Software:

* IDE for developing in C++
* C++ compiler
* CLAMV Access
* The server simulation software and maps
* CPPTest for testing ( recommended )

Hardware:

* UNIX comptabile machine

# Section 8: Conclusion

*Closing Remarks*

Our team looked for robustness and flexibility in the implementation. We created a Brain Instruction Language that resembles that of the assembler language but also uses some high-level functions that provides the programmer more functionality with less coding. The Brain language creates a logical flow of computation by providing many functionalities such as labels usage, dynamically created memory and functions that are able to handle several types of errors. This set of instructions provides the programmer flexibility and robustness by using basic instructions that are available for every low level programming language. The system design that we provide offers great functionality and easiness. For instance, we made emphasis on the use of the concept of the logger in order to register everything going in the flow of information between the major instances of the implementation. Also, in our documentation we took really good care on indicating how to test the system for several situations.

In a nutshell, we looked for a balance of robustness, little restrictiveness and proper error handling in the design of our system.

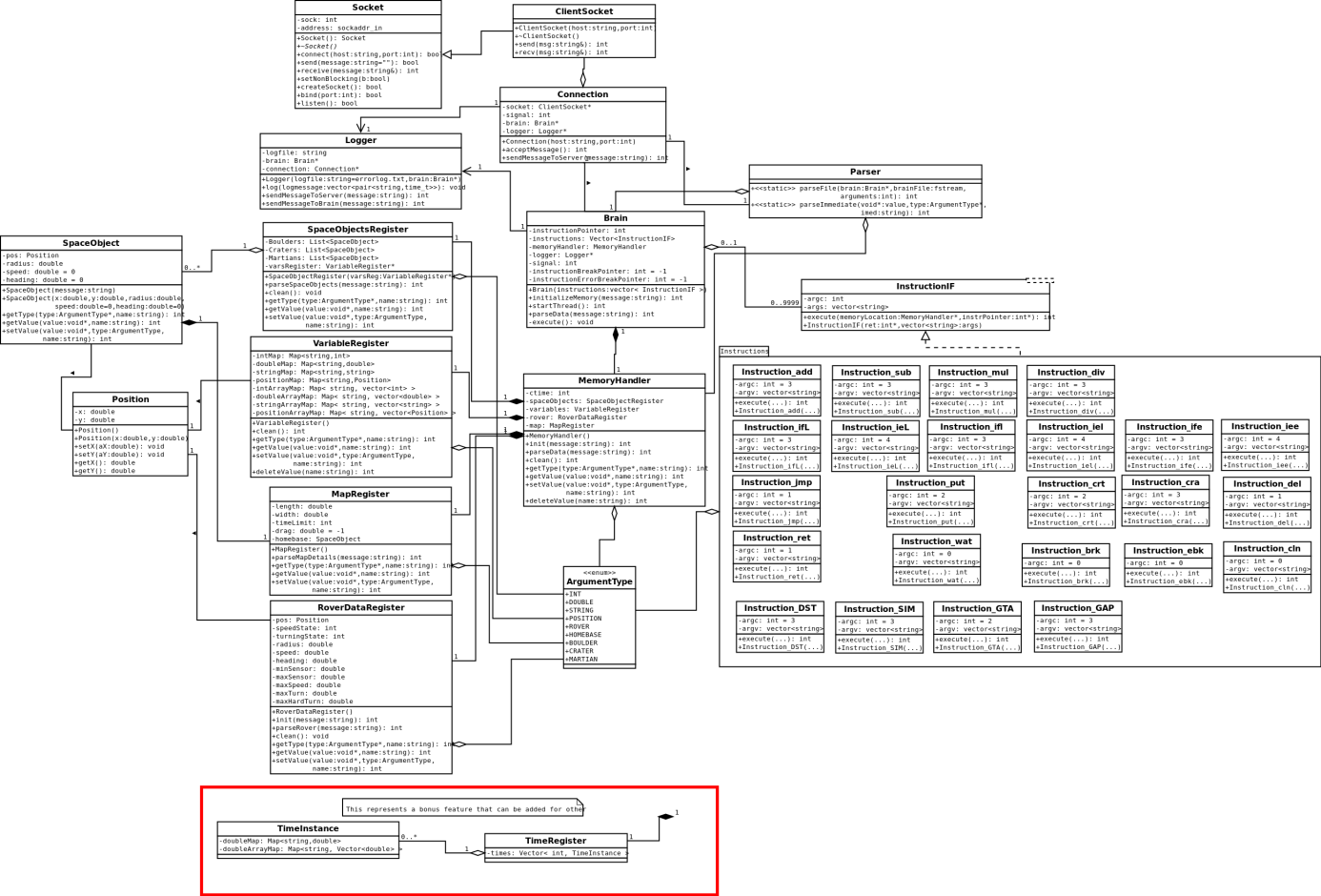
# Section 9: References

* Introduction to the Unified Modeling Language, Advanced Computer Science Lab Course II, Jacobs University Bremen
* <http://www.uml.org/>
* Project Specification - Mars Rover, Software Engineering Lab Spring 2013, Jacobs University Bremen

<http://www.faculty.jacobs-university.de/bmahleko/320222_advanced_computer_science_II_lab/slides/SELab-Spring2013-Specification.pdf>

* <http://global.fncstatic.com/static/managed/img/Scitech/mars%20drilling%20site.jpg>

# Appendix 1



# Appendix 2 – Sample Test Scenario

# Scenario X: Component

## Scenario Description

* Description of use case/test scenario

## Version Control

|  |  |  |  |
| --- | --- | --- | --- |
| Version # | Date | Author | Description |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Test Scripts

The following scripts will cover this scenario:

* x.1 Something
* x.2 Something else

## Case Description

* What the user/tester does or should do

## Test Components/Requirements

Which general requirements are needed for this test/use case

* stuff
* other stuff

# Appendix 3 – Sample Test Script

## Script X.1: Something

### Script Steps

* The steps the user/tester must do , detailed enough to be able to be followed, and including expected results and expected action after each step

### Testing Requirements

Specific test/use case requirements for this particular script

* stuff
* other stuff

### Teardown

* Expected actions after termination of use case application/testing , and if applicable instructions regarding what where and how to document ( results for example )

### Script Steps

| **Step #** | **Test Action** | **Expected Results** | **Pass/ Fail** |
| --- | --- | --- | --- |
| 1 | Build Solution | Successful | Pass |
| 2 | Start SMSimulator | console appears, starts listening | Pass |
| 3 | Run test client with some command | client correctly registers new event | Pass |
| 4 | Run test client with some other command | Client correctly executes command  (see attached test1.png ) – if steps require taking pictures or noting something in other documents | Pass |

### Test Execution

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date/Time | Tester | Test ID | Test Phase | Status |
| 14/03/13 11:00 am | Who Ever | N/A | N/A | Passed |

## Script X.2: Test that fails

### Script Steps

* steps

### Testing Requirements

* reqs for the script

### Teardown

* If the test fail document the information from NUnit. Otherwise, close NUnit and document the results in attached xcel sheet.

### Script Steps

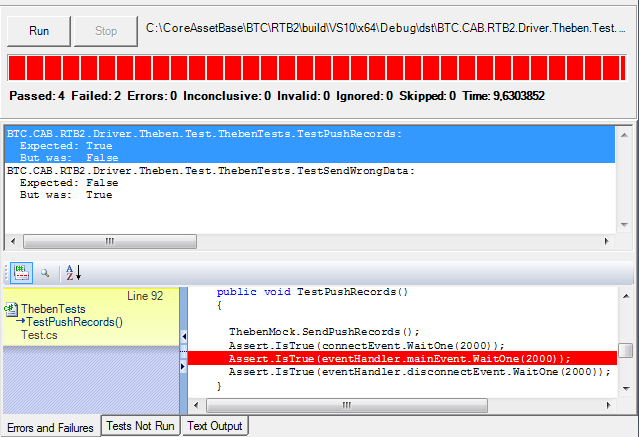
| **Step #** | **Test Action** | **Expected Results** | **Pass/ Fail** |
| --- | --- | --- | --- |
| 1 | Build Solution | Successful | Pass |
| 2 | Start Nunit as admin ,add assembly | NUnit starts, assembly correctly read, tests added | Pass |
| 3 | Run the tests | Test pass ( check nunit.jpg ) | Pass |

### Test Execution

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Date/Time | Tester | Test ID | Test Phase | Status |
| 14/05/13 11:28 am | Some Dude | N/A | N/A | Passed |
| 18/12/13 14:00 pm | Other Dude | N/A | N/A | **FAIL** |

**Test Failure Information:**

Detailed description of the step at which testing fails , and in conformance with teardown instructions ( in this case to add a photo ) . as detailed info as possible.

If applicable and known , here should be documented possible causes of the tests , possible ways to fix , and possible course of action to solve test failure ( test other components that failed , change parameters of test,change system ,etc … )

# Appendix 4 – Sample Test Script

|  |  |  |  |
| --- | --- | --- | --- |
| **Version** | **Author** | **Time** | **Comment** |
| r24 | dhasegan | 2013-02-17 13:47:56 | First Diagram and list of function |
| r48 | dhasegan | 2013-02-19 00:25:07 | Added new functions with high level functionality. Added methods to most of the brain classes. |
| r51 | egonzaleshuaman | 2013-02-19 14:40:25 | sequence diagram overview |
| r53 | btiwaree | 2013-02-19 14:47:46 | Diagram\_with logger |
| r59 | egonzaleshuaman | 2013-02-19 15:20:26 | first activity diagram |
| r64 | oenache | 2013-02-19 15:31:12 | Index added. Main Document started |
| r74 | egonzaleshuaman | 2013-02-19 16:17:01 | corrected sequence diagram |
| r99 | ymengesha | 2013-02-19 21:42:04 | Connection classes added and updated |
| r172 | dhasegan | 2013-02-21 11:24:36 | Added some instruction functions, setter and getters and constructors where missing |
| r174 | dhasegan | 2013-02-21 14:07:53 | Added Documentation of the Brain instructions and overview. |
| r178 | dhasegan | 2013-02-21 16:51:54 | Added the preset variables in the Documentation and revised some things |
| r190 | ymengesha | 2013-02-21 20:35:15 | added the final structure of the design document along with the index and introduction |
| r193 | ymengesha | 2013-02-21 20:55:03 | just another version of FinalDoc with .docx |
| r196 | dhasegan | 2013-02-21 21:01:03 | Minor modifications in the FinalDoc. |
| r247 | dhasegan | 2013-02-22 11:00:41 | Created Brain Example and Added a new Function |
| r281 | oenache | 2013-02-22 17:17:32 | added testchapter.docx which contains QA and test info |
| r299 | egonzaleshuaman | 2013-02-22 17:54:51 | first draft of Complex Sequence Diagram 1st thread |
| r304 | egonzaleshuaman | 2013-02-22 18:05:29 | minor fixes to Sequence Diagram 1st thread |
| r307 | ymengesha | 2013-02-22 18:20:12 | Updated the introduction of the final document |
| r329 | dhasegan | 2013-02-22 20:19:21 | Added system architecture classes and modifications in the Class diagram |
| r341 | egonzaleshuaman | 2013-02-22 20:47:23 | draft - second diagram 2nd thread |
| r345 | ymengesha | 2013-02-22 20:59:44 | modified the class diagram |
| r999 | ALL | 2013-02-22 23:54:50 | Uploaded Everything #yolo |