A Satellite Mission Control System

Brian Schmidt Hermansen

Kongens Lyngby 2006
IMM-THESIS-2006-53

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

Summary

The Mission Control is an extension of the Ground station. The Mission Control handles the actual communication with the satellite and the Ground Station is the tracking and maintaining of the communication. This means the Ground Station controls the radio which different Mission Controls can gain access to and then use to communication with a given satellite. This means that one can service more than one satellite operator at the time if need be, but that only one can communicate at a time.

The general purpose is that operators will use the Mission Control without any prior knowledge to the Ground Station, they do however need to know a few things about the satellite which they want to use. Mainly they need to know about the commands that the satellite can comprehend, so they do not accidentally start flooding the satellite with garbage commands which might take transmission time away from the real commands. It is expected that the commands will be verified by either people or some programming qualified to do this. The operator might see it fit to make some last minute changes, these should be made before the session is passed to the Ground Station or the changes wont be able to be made as a lock will occur.

Another aspect to be considered is the storage of the data received, not as much as the medium as that has already been selected, but more which structure the database should have. The structure might change as the data becomes more complex or more fast information is needed before actually fetching the entire set of data. Furthermore then one could expect that it might variate depending on the need of the single user requesting the data later, but in this aspect one would expect changes in the software fetching the data rather than an actual

change in the database, leaving this problem to a later time in the DTUSAT2 progress.

Looking at a future aspect of the software development within the Mission Control, it means that a Mission Control can hook up to different Ground Stations running the Ground Station software to gain better coverage of the satellite. It should be mentioned that this is not necessary at this point as there is only a single Ground Station running with the software.

Resumé

Mission control er en udvidelse til ground station. Mission control håndterer kommunikationen med satelliten og ground station styrer tracking samt vedligeholdelse af kommunikationen. Dette betyder at ground station kontrollerer radioen og forskellige mission control programmer kan få adgang til den og dermed kommunikere med en satellit. Dette betyder at man kan servicere mere end en satellit operatør af gangen hvis det bliver nødvendigt, dog kan kun en enkelt af dem kommunikere af gangen.

Den generelle ide er at en operatør vil bruge mission control uden kendskab til ground station, de bliver dog nød til at kende noget til den satellit de vil bruge. Hovedsagligt skal de kende noget til de kommandoer satelliten bruger, så de ikke begynder at oversvømme satelliten med ubrugelige kommandoer, som bruger vigtig tid som andre kommandoer kunne have brugt. Det er forventet at kommandoerne vil blive verificeret af enter folk eller et program som kan gøre dette. Operatøren har måske lyst til at lave nogle små ændringer før transmissionen starter, men dette skal gøres inden en session er sendt til ground station, eller kan ændringerne ikke lade sig gøre på grund af systemmet vil låse kommandoerne.

Et andet aspekt der skal overvejes er lageringen af teledata, ikke så meget på hvordan da dette allerede er blevet valgt, men mere om hvilken struktur det skal have. Strukturen kan ændre sig efterhånden som dataerne ændre sig. Ud over dette kan man forestille sig at det kan variere sig alt efter hvordan den enkelte bruger har brug for teledataerne, men dette er dog nærmere en del af det program som henter dataerne senere. Så det er et problem vi overlader til et senere stadie i DTUSAT2s forløb.

Ser man på fremtidige ideer med hensyn på udvikling af mission control, så ser man gerne at den kan følge en satellit ved at skifte fra jordstation til jordstation. Dette skulle gøre at man opnår en bedre dækning. Det skal dog lige siges at dette ikke er en del af projektet, da der kun er en enkelt jordstation der bruger ground station softwaren.

Preface

This master thesis has been accomplished at Informatics Mathematical Modelling, the Technical University of Denmark, from November 1 2005 to April 29 2006.The project constitutes the final work of the requirement for obtaining a Master degree in Engineering at DTU. The project was supervised by Associate Professor Hans Henrik Løvengreen of the Informatics Mathematical Modelling department of DTU.

I would like to thank my advisor, Hans Henrik Løvengreen for his constant vigilance, giving me inspiration and courage during my work on this project.

Finally I would like to thank my parents and my family for keeping me in the fire during the hard days of my study. My grandfather will always stand as a shining example to me for to aspire to be one day, for his dedication to his dream and work, the first engineer of my family.

Lyngby, April 2006

Brian Schmidt Hermansen
s948259

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Introduction to the Subject

This project is a part of the larger project called DTUSAT2, which is based on the idea of sending a student made satellite into space. This satellite is then meant to track small bird migrations from Northern Europe (namely Denmark) down to north Africa. For more information visit www.dtusat.dtu.dk.

This part of the DTUSAT2 project involves analyzing and making the Mission Control part of a Ground Segment. The Ground Segment consist of the Ground Station and the Mission Control. The Ground Station handles the traffic from and to the satellite as well as the tracking of the satellite through a radio module.

The Mission Control is meant as an automatic interface to the Ground Station, which allows the operators to schedule the satellite pass ahead of time. This means that the Ground Station doesn't need to be observed by an operator at each pass, but that the system can maintain operation even without observation. Since the Mission Control is a separate part of the whole system, it could be used from any given computer with access to the network which the Ground Station operates on.

1.2 General Introduction to Ground Segments

With the increased usage of satellites in the world today, it has become increasingly important to have highly stable Ground Segments to maintain communication with the satellite or satellites that you have orbiting earth. Therefore more and more money are put into developing these systems, so they become more dependable and easier to maintain once commissioned. This is mainly due to the lower cost of having a Low Earth Orbiting satellite and the fact that you can gain a wider area coverage through having these, which is highly beneficial for research or spy satellites. The reason behind this solution being good for those purposes is that the satellite have a circle passage around the planet. This means it covers a lot of ground but also that communication with the satellite will be restricted to only when it is in radio window of the ground segment.

This project is the second part in developing a generic ground segment for a student satellite, the first part was the ground station which was made in 2005 by Yu Du as a master thesis [2]. This project will be a master thesis about a generic mission control which utilizes the ground station.

The architecture given for the ground station and the ground station can be seen in 1.1. It can be seen that a mission control can either be served by one ground station or by more, in the case of it being served by more it will switch between the ground stations to track the satellites movement. In this project a mission control will only handle all the commanding of a certain satellite, such that the satellite it commands is decided on what plugins are used.

The ground station operates with a term called session. Making a session basically means that the ground station reserves a time period at a certain time, such that the mission control can make communications with the satellite through the ground station in this period.

1.3 Scope of Project

There are three phases in every Low Earth Orbit satellite communication.

1. Pre-passage: This covers command preparation
2. Passage: Is the communication with the satellite
3. Post-passage: Telemetry data processing and dissemination

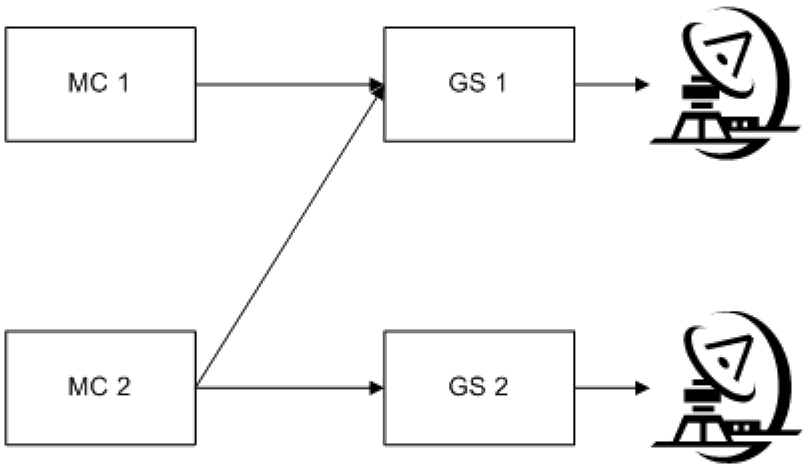

Figure 1.1: Ground Segment Architecture

The mission control will:

- Cover some of the Pre-passage, as it will use the commands but will not know what they are about
- Cover all the passage
- Cover only the storage of data in the Post-passage

In this master thesis a generic mission control is to be developed. The system should allow access to the ground station to commit a session and for ongoing communication with the satellite during its passage window. Furthermore it should have some schedule for the commands, which can be communicated to the satellite. It should also have some way to store data, such as commands and data received from the satellite. Finally it should have some automatic functionality, so it can communicate with the satellite without an operator nearby.

1.3.1 Technical Constrains

The system will be able to be deployed anywhere as long it can gain access to the ground station which is located in building 348 at DTU. The implementation of the system will be done in Java [1] due to its cross-platform stability.

1.3.2 Overview of Thesis

There are six chapters which describe the mission control system in this thesis.

- Chapter one introduces the basic knowledge of the Mission Control and outlines the scope of the project
- Chapter two looks at the requirements of a Mission Control System, looks at use cases for an operator and an automatic run. Finally it sums the decisions up at the end of the chapter.
- Chapter three contains the thoughts about the design and which practical approach should be taken.
- Chapter four is where we take the step from design to implementation and point out where things are done different than we initially wanted.
- Chapter five describes the tests done to the system and to which extend the system lives up to the requirements in these tests.
- Chapter six sums the project up with the conclusion of the thesis.

CHAPTER 2

Requirements

2.1 Introduction

Now that the system has been confined with the limitations in mind from the previous chapter, it is time to analyze the system requirements and how we want want to deal with the different issues that might occur.

2.2 General Issues

2.2.1 Mission Awareness

By mission awareness we understand the extend to which the system knows about the basic structures of commands, but not the mission specifications of the commands. This means that the basic idea behind mission awareness is that the program does not have to know about the nature of the commands and only a little about their structures. One thing there is important however is that it can keep track of the history of the commands that it as been given, so it knows where they might be at any given time in the progress of being processed.

Requirements:

- System should keep track of command history
- Mission specific handling of commanding must be done by plug-ins.

2.2.2 Robustness

As one of the basic system requirement we have that it needs to be stable and therefore not crash from time to time. But as there are other reasons for a system crash such as power failure or crash of the platform which the system runs on, we have to have a backup plan. This comes in the form of a storage system which will be discussed later in this chapter.

Once recovering from a system crash the vital data will be recovered from the storage system and everything should be more or less back till the instance before it went down.

2.2.3 Security

With every system there is the possibility of malicious persons wanting access to the resources controlled by it, so the way security issues is handled in this thesis is more stating which measures is already in place.

Security in this case will be limited access to the place where the system is run and of course limited access to the software developed. The software will be restricted till use only by new developers to begin with and later open for those there seeks to make use of it.

Last we want to point out that security issues are not part of this master thesis, but might be included at a later point by people working on such a project.

2.2.4 Distribution

Since this is a generalized platform we have distribution of it in mind for future development, namely letting other universities or similar institutions use and modify it as they see it fit under the normal software development rights.

Furthermore then the software will be under version control, so the earlier versions will still be available in the future.

2.2.5 Ground Station

The Ground Station has already been made, so communication with this to and from the Mission Control is already defined and will only be adjusted if the need arises due to shortcomings of the present code.

2.3 Basic Notion

In this section we will define a few terms which will be used in the rest of the report.

2.3.1 Definition of a Command

Since the command is expected to be transmitted to a satellite, the guide made for this on ESA satellites [3] was consulted to gain information and inspiration.

A command is a term for an action you want to get performed on a satellite, there are many types of commands and most of them are requests for information about the general health of the satellite. Furthermore then there is also different types of answers from the satellite depending on each of the commands there can be given, most of these are informative in the regard of where in the system a command are now and others return the answer to the request made by the command. This means that either all commands have to be unique or they should be identified in some other way. The most common way is to give them an unique identification code, either at the commands birth or at the time when they are transferred to the satellite.

2.3.2 Definition of a Pass

Pass is short for: passage of satellite. This means that it involves a time window for when satellite communication is possible. Furthermore then a pass contains the information about which commanding should be uploaded in this time period.

2.3.3 Definition of an Assignment

A command can be assigned to a pass, such an assignment is necessary as the pass stores the information about the events there takes place while the mission control communicates with the satellite through the ground station.

2.4 Pre Passage Functionality

Since we only need to know a limited amount about the command structure, we have a few choices to consider, such as if the commands are entered manually into the program and it then remembers them till they are needed or if we should have them pre-made by some external place, such that they can be received at any time. Since there might be loss of data in one way or another then the better option is to have the commands stored in some way, so they might be retrieved after a power loss or another fault which might cause the program to be temporary down. Furthermore then this will allow the program to be updated from another location, so the physical access to the computer can become more limited if desired.

Now that we know that it should be a stored medium then we have to consider the way the commands is stored. There are two obvious ways there stands out right away:

1. One or more files
2. A database

The most simple way would be to use a primitive filesystem, where the program locates new files and adds them one by one to an internal list of commands. Although this would be a simple thing to make then there is the question of stability with the files and how they should be locked by the user and most importantly how they should end up in the pre-defined folder containing them.

With a database you have the problem of accessing it easily, which might cause some problems, but otherwise it is simple to access the database from a given program from anywhere with access to the network which it is located at. The access can easily be overcome with a little program meant to store and retrieve data and once that is in place we have a simple and efficient way to store the commands and retrieve them from the program. Another bonus of doing this is that we can track a command in the system and therefore always know how far it is into the progress of being executed or just simply verified by some source so it can be considered as ready for a satellite pass.

Another thing there should be considered is the way that commands are scheduled before access to the satellite in the next pass is requested from the ground station. This is important as once we have requested the access everything will have to be in order for the pass, so they are ready to be transmitted in the correct order. Of course there is then the question of whether minor changes should be allowed in the duration before pass phase is started or if it is locked for changes in this period. We chose to only let an operator be able to change the schedule at this point and therefore lock the commands in the database from outside changes.

We also need to transmit to the ground station about our desire to schedule a pass ahead of time.

2.5 Passage Functionality

First we need to establish contact with the ground station ahead of time so we are ready for transmitting the data to the satellite and more so receiving data.

We should also consider whether we should allow the operator to access the schedule while the transmitting is in progress and how that should be handled. The schedule will be locked, so if the operator decides to change the mission parameters then he will have to first abort the schedule and then process the remaining time manually, perhaps with access to certain standard commands which he can quickly chose from a list. This brings us to the question of a safe-mode if the schedule fails to receive certain expected data or another reason. The safe-mode should just run through a given schedule, to see if the satellite is still healthy and to locate an error which might have occurred somewhere in the satellite system if possible. A safe-mode should of course be designed by one with knowledge of the satellite at a later point.

While the data is received it has to be stored, although this is normally part of the post phase then it makes more sense to do it while the transmission is going on, as then data wont be lost in the same degree if the system should fail. This will guarantee that at least some data will be stored for later retrieval. There should also be a link between the data received and the commands given for getting those data, but we will leave details of this to the design phase chapter.

2.6 Post Passage Functionality

Once the pass phase is over there is little left to do for the mission control, as more or less all of this phase is excluded from this project. The thing we do have to consider is the mission awareness from the program's view, this means that we have to create a thin red line through the program which will allow us to keep track of each of the commands current status and once we finally get here to make sure that the current command is shown as completed or failed. This could perhaps involve some information about where it failed and if possible a reason behind the failure. Furthermore one could expect a brief status report given to an operator on the data received, although no translation but only raw information, such as size.

2.7 Operator interface

Having an interface is an issue due to the fact that the mission control should if possible be up and running at all times for a stable system, so it is expected that an operator can be present before, during, after or through them all. There are a number of ways to do this, mainly that the console or GUI is always present and that the person just needs to access the computer to do whatever needs to be done. The other way is that the mission control runs as a host serving clients which operators can open whenever they need to, although this grants the system great flexibility it also causes the problem of too many people operating at the same time, which is quite undesired currently. The option is therefore to take the more simple way and create a more stable system which might include the other option in the future.

There is of course the question of unauthorized access to the system, but that would be present in both cases and in the way we have chosen it is limited to either having the program or getting access to the operators room/facility. We could implement some cryptation into the access, but that would currently be overkill for the project as we are working with a student satellite with limited functions and not a billion dollar satellite running crucial operations.

2.8 Use Cases of Mission Control

To give some idea of what we are trying to accomplish two use cases has been made.

Use case of the Mission Control Operators view

1. Starts the client
2. GUI for operator pops up
3. Operator inspects commands in the Database not yet executed
4. Operator checks the list of future passages
5. Operator chooses to assign some commands to future passages of the satellite and others to be assigned by program
6. Schedules some commands
7. Commits the changes
8. Closes client

Use case of the Mission Control behind the scene

1. Starting up
2. Checks Database schedule
3. Checks for next passage of satellite
4. Compares next passage in schedule against time
5. Establishes connection (RMI) with Ground Station and passes a session
6. Close connection (RMI) with Ground Station
7. Locks the passage in the schedule from changes
8. Establishes connection (TCP/IP) for data transfer to/from satellite
9. Transfers commands to the satellite
10. Stores data in Database as it is received
11. Closing connection (TCP/IP)
12. Cleanup

2.9 Chapter 2 summarized

In this chapter we discussed the requirements of the system, such as General Issues, Pre Passage, Passage, Post Passage and Operators Interface.

General issues was about which things the program might encounter in the future, from currently minor concerns such as security till more important concerns like robustness.

The passage sections dealt with the three different phases that the system undergoes, namely pre-pass-post passage and defines which domain we are working with in this system.

Finally there is the operators interface and what should be expected from it and this was supplicated with a few brief use cases to point out how things might be.

Chapter 3

Designing the System

In the previous chapter we discussed the requirements of the system and which options there was available to fulfill the the needs of these requirements. In this chapter we will then discuss the design which will lead to an implementation in the next chapter.

It should be noted once more that although the system works with commands, then it is the user/users that is responsible for their makeup and later verification as the system is meant to be modified to more than one satellite and therefore cannot be hard coded to every single satellite communication system there exists.

The chapter is divided up in sections which describe key concepts of the design. The first two will be about general terms and a brief introductions to the different components, the sections following this will be a more in depth discussion of the individual components and their subcomponents.

3.1 Life cycle of a Command

The basic idea is that a person can at any given time see how far a command is processed in the system. This should enable a person to track down a command later and thereby find out where it got stuck in the system if that was the case.

3.1.1 Basic Command Notion

There needs to be a clear structure of how the commands are represented in the system and the following information should be known for each command:

- Unique Identification
- Execution Time
- Command Body
- Priority (urgent/immidiate)
- Status

The unique identification is therefore backtracking a command later, such that the data can easily be paired off with a command, it also makes sure that command will not be repeated by mistake, as comparing identifications is far easier than comparing command bodies, due to the fact that the command details are not know by the system.

Execution time is set, so it roughly coincides with a future pass. The time can be altered by the system if the command did not make it the first time around.

With the command body little should be known as it mainly will just be transmitted and then the satellite will know what to do with it once it arrives.

The priority will ensure that commands deemed critical by users will be transmitted at an earlier position than those just handed in normally. This way maintenance commands or critical commands can be given at a later point and still be performed first, such as an error was just found in the last pass and now some routine is quickly put into the system for the next pass.

By having a status field we can track the command around in the system at any given time. The idea are that since the commands is crucial to the whole idea of having a Mission Control, it is necessary to come up with some way to identify each commands location in the system. So that only verified commands is actually used in transmissions to the satellite and repeats does not occur unless the command has been a new unique identification once more. This leaves us with a rough idea of which steps should be available for the commands, namely:

- Entering a command into the system
- Copy a command, verification is done
- Verifying a command
- Deleting a command
- Scheduling a command
- Canceling a command
- Transmitting a command
  - Various steps on satellite recorded
- Marking a command as completed

Now that the general idea of a command Life Cycle is in place it is possible to start designing what should be done to reach each state and what happens to the command in each state.

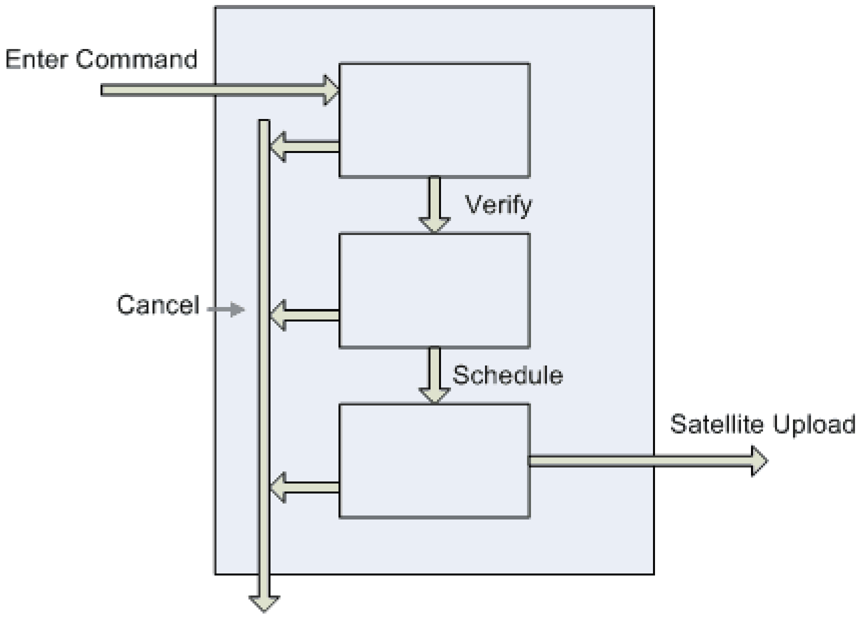

Figure 3.1: Tracking a Command

As the figure 3.1 shows then all steps are included, where the sixth step are more or less included in the first and second step put together. The sixth step could be considered a shortcut to making a verified command without need of the verification step, as it has already been verified once. By allowing the sixth step the program becomes more adaptable as people can sort through previous completed or verified commands to add, although this would of course need knowledge of commands from the users' part which is already required to begin with.

Each step will be shortly described after the rules and command list sections to ensure that there is no misunderstanding involved later.

3.1.2 Rules

The Mission Control need rules to handle the commands, so consistency is present and the system therefore become more reliable for the operator.

First thing to consider is the way that commands are treated once they have been entered, as they are entered in an unverified state. This brings us to the consideration of verification rules. A command cannot be transmitted to the satellite without having been verified and every command behind it should also be considered unverified. This means that should one enter a new command before a verified command, then the commands after the command should be set to an unverified state once more. By doing this we have to ensure that the operator knows that the first command that he has to verify is the first unverified command in a given list and that everything in front of the command is ready to be transmitted and therefore pose no trouble. The main reason behind having this rule is that putting a new command into the schedule disturbs the execution time or conditions of those following it and therefore it needs to be verified once more.

There is of course an overrule to this, as emergency commands could be given and these should not upset the verifications of the ones coming after. The reason behind this is that there would not be any time to verify the commands once more after an emergency command is put in and the previous work would be lost and pushed into the future instead. This overrule allows flexibility while the transmission is active, without the possibility of too much loss to the time window that a satellite pass got. There is of course the possibility that some commands will have to be rescheduled to a future pass, as they was last in the line in the pass.

The alternative to the overrule is that the operator just cancels the schedule and manually completes the rest of the pass. This will of course cause loss of time in the matter of not having the commands ready, but will be necessary in the case of some malfunction in the dataflow or similar events which raises an alarm to the operator.

3.1.3 Command List

A command list can be considered as a long line stretching forward in time. At the same time we have a pass schedule which contains pass objects which a command can be entered into, these pass objects is sectioned up into set number of slices which are normally time based or number based. This is because a time window to the satellite is limited and there are only room to a certain number of commands to be uploaded in this time. This can be seen in figure 3.2.

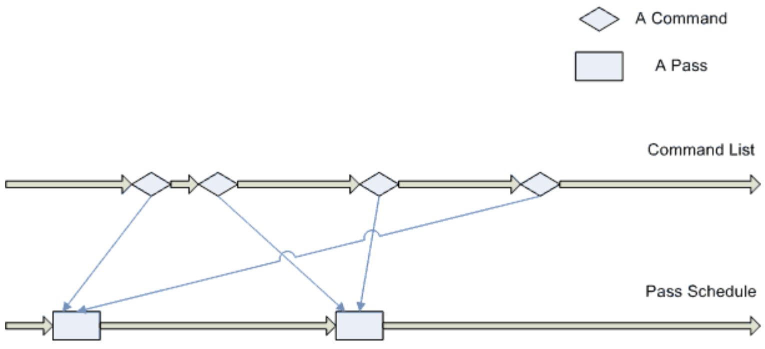

Figure 3.2: Command List Concept

The command list should be dynamic so if a command is missed then it can be pushed ahead of the present time, such that no command will ever be completely forgotten but only postponed till a later time. The actual placement of the command might not be the same order which it is transmitted if an operator chooses to change it during a transmission with the satellite, of course the actions behind this will be strictly on the operators account.

3.1.4 Entering Command or Copy a Command

As this is a bit ambiguous then it is quite important to make clear that we are talking about two separate ways to create a new command in the database. First one can enter a new command from scratch, which will in then be marked unverified or one can use the other way of making a command by simply choosing one from the list of already entered ones. Both will then be placed in the filesystem for later use.

3.1.5 Verification of Command

Here the user will get a list of currently unverified commands which need to be verified. The person doing the verification can do editing of the commands if they are faulty or discard them if they are beyond recognition.

3.1.6 Scheduling of Command

This is where a command is prepared for transmission to the satellite, namely as it is assigned a slot in a pass which is connection to a future session.

3.1.7 Uplink Command

Transmitting a command involves getting data back on its current status on the satellite. This is how a satellite might respond to getting a command:

- Command Received
- Command Started
- Command Processed

There is of course different ways and this is just to outline how the response from the satellite might be.

For interpreting these responses there is a need for a plug-in for each satellite, as there are different protocols. So some kind of open plug-in system has to be implemented so that the user knows what is currently happening.

3.1.8 Command Completion

All that is needed in this step is to mark the command as done, so that it is not accidentally done twice without the users awareness.

3.2 Architectural Design

The section will look at the components design of the system. The obvious modules are an Operator Interface, a Command Entering and Pass Control.

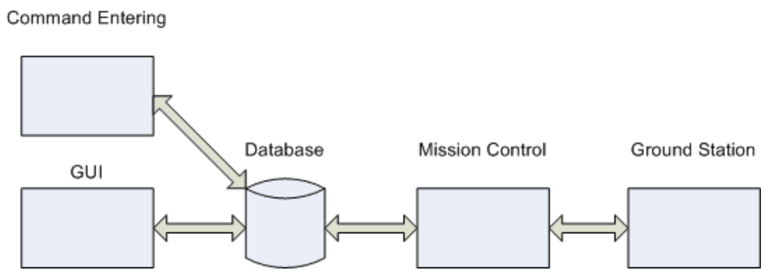

Figure 3.3: Design overview of components

On the figure 3.3 we see that the commands go into the database where they then can be obtained from the Mission Control or verified by a user. It is an easy system to set up, but there is a complex interaction between the components and the database. This complexity is due to the way that the database has no way to of informing the programs that a change to its contents has been made. Since the most obvious way to fix this problem is to lock the access from other modules to the database while one module accesses it, then this is an undesirable solution.

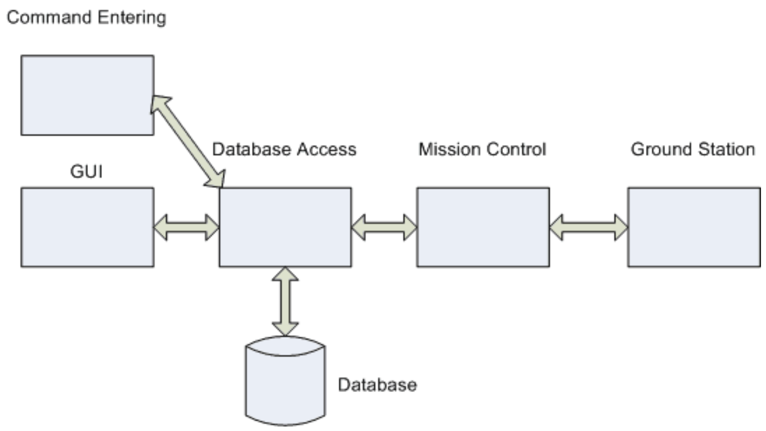

Figure 3.4: Improved Design

Changing the design slightly to accommodate for this by adding one more module, as shown in figure 3.4, we are past the problems of accessing the database as all traffic will be controlled by this module.

3.2.1 Operator Interface

This component consists of the graphical user interface which an operator can use to control the Mission Control and gain information with. It mainly works as a place where you can verify commands and schedule them, but status information about the commands will also be available here and progress of a passage. It is also here the possibility of making a termination of an ongoing passage schedule is possible, so that the operator can manually start executing his own commands. Should a user choose to terminate a schedule already in progress then there will be a list of pre-made commands at hand which can be used as manually creating new ones would spend precious time from the window of opportunity, the option to make new ones will be available still.

3.2.2 Command Entering

To enter a new command or replicating a command already on the stack this module will be used. It will be launched at the same time as the Operator Interface, such that one can switch easily between the two open windows. There is not much to be said about this one other than it will be possible to make new commands and submit them to the Mission Control System or to bring forth a list of already verified command and submit one of those onto the Mission Control once more.

3.2.3 Command List Component

As shown in figure 3.4 then there is a need to communicate between the different modules to and from the database. This component makes sure that no overlapping communication with the database exist, so the data can only exist in a single form no matter when it is somewhere in the system.

Another important thing is that it will be responsible for storing and retrieving the actual data at startup and shutdown events, so that the data entered is not lost due to a shutdown.

3.2.4 Pass Control

This component ensures that a session is created on the Ground Station. It also needs to establish a communication link through the Ground Station once the satellite window is available, so commands can be sent to the satellite for execution and the results can be returned to the Mission Control. Furthermore it handles the schedule so it is already available once contact has been established or if changes need to be made.

3.3 Command List Component

As mentioned earlier then this component is mainly there so no overlapping exists in the uniquely identified commands, meaning that a command can only exist in a single form at any given time. Another important thing is that this component should be as simple as possible as the higher functionalities should mainly exist in the other components, such that this one does not get unnecessary complicated and therefore needs great adaptation from other components there needs to use it.

It should however contain the information about the command list and a given number of future satellite passes. The passes will then be linked to different commands from the list as the operator assigns them to different slots.

3.4 Command Insert Component

This will be a little rough graphical user interface, where new commands can be set into the Command List Component or old ones can be reused once more with new unique identification numbers. This will be running at the same time as the Operators Interface.

3.5 Operators Interface Component

Within this component we have the command manipulation and the pass scheduling for upcoming satellite passes. The operator will of course have a graphical interface which will show a list of commands in the left side of the window and the right side will adapt to the task that the operator is currently working on. The right side should show:

- Pass Schedules
- Command Editing Tools
- Telemetry Data

3.5.1 Pass Schedule description

Here we will have a list over the future passes, as one is selected the commands assigned to that pass will highlight over in the left side of the screen. Down under the list will be some tools for adding or removing commands from a selected pass.

3.5.2 Command Editing Tools description

Once the tools has been selected they will not be activated till a command is chosen, the information of the command will then appear at the top in editable text fields. In these text fields the commands can then be edited till they get the desired form which is required for a verification of them. There is of course the possibility to discard the changes and start over if need be.

3.5.3 Telemetry Data description

While the Ground Station and the satellite is communicating the data will be shown here if the option is chosen, this way the operator can get a rough idea about what is going on if the protocol is known by the person.

3.6 Pass Control Component

This component consist of subcomponents for handling different tasks, as it is the central component in making the Mission Control Center work.

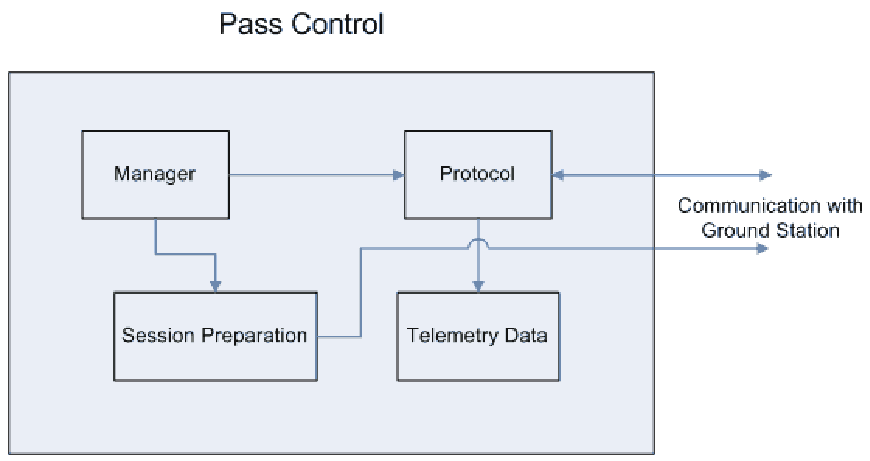

Figure 3.5: Pass Control

3.6.1 Manager

With multiple subcomponents we have a need to arrange the different information into a structure where we ensure that the different subcomponents are activated once they are required. For this we have decided that a manager which takes care of all is necessary.

A given scenario might be something like this:

- Session Scheduled
- Session Executing
    1. Wait for Pass
    2. Wait for Connection and prepare
    3. Lock Commands used
- When connected start transfer
- . . . and so forth

3.6.2 Protocol and Transmission

There are two parts to this, the protocol and the transmission. The reason behind this is the numerous different protocols there exists and our need to adapt to them all, so a generic protocol will be made which can be exchanged with a given protocol.

A protocol is a way to define how communication should be done between two parts, so that the two parts will have a given set of tools to work with. This ensures that the communication between them is well defined and cannot be miss interpreted by any of them. Another facet is that it also works as a limitation for others to break into the communication unless they know the given protocol which is used.

The transmission part will mainly just pass the data between the protocol in the Mission Control and to the Ground Station where a similar build can be found.

The component will have full access to the telemetry data component, so that the data will be stored and also passed along to the operator interface. Further more then it will have access to the pass schedule in the command list component, such that transmission can be started once the connection with the Ground Station has been established.

3.6.3 Session Activation

The session activation sends the data necessary to create a session on the Ground Station, after this is done it will notify the transmission component about it so it can start listening after a connection from the Ground Station which will indicate that a satellite pass has started.

All this component will be doing is creating the data for it once it gets the data associated with a pass from the Manager.

3.6.4 Telemetry Data

Data storage is the keyword here, it will store the data as it is received from the satellite without any form of translation done on it. It will however also be sent to the operators interface, which can then be shown when the operator makes that section active.

Chapter 4

Implementation

Now that the design has been thought through it is time to consider how things should be implemented. This chapter will therefore look at the design in a more practical way and seek out the most practical solutions to the designs.

The core of the project is still that we have some commands there should be uploaded to a satellite and then we may receive some answers, this still goes through the ground station and this means that certain parts of the system, which is implemented can only be made in a predefined way so they can interact with the ground station.

4.1 Cleanup in Ground Station

The first step of creation a working version of the Mission Control Center was to clean up in the Ground Station. This was necessary due to the fact that we do not want to include the entire Ground Station into the Mission Control, but we do need some essential parts from it to have everything work. This meant that we had to create a library which contains the overlapping components used by both systems. A rough cleanup was initially made in the beginning of the project and it was modified as was needed during the project, as the Ground

Station is under continuous development. This of course means that a final version of the changes will not be available till both projects had accomplished a satisfied state, if ever.

Everything was collected into the ”common” directory, which now serves as a cross-development ground for both projects.

4.2 Interfaces

In general interfaces provide a service which allows a class to inherit more than one interface, as a class are normally only be allowed to inherit a single super class. Another use interfaces have is that they provide a guideline for future programmers whom might find themselves working on the project and it is this feature in the interfaces which we seek.

So we have decided to use interfaces for the exchangeable parts in the system, such that it is easier for a new programmer to replace or alter current sections of the code without having to second guess the purpose of those parts or their interaction with the rest of the program.

We will now briefly discuss the interfaces which have been used in figure 4.1.

4.2.1 Command List

The command list is, as also mentioned before, a core element in this program, so the interface here is more important than in other places as it will give a better idea of what to expect from the actual code. The interface will also be quite extensive due to the massive number of operations made in the command list as every other component in the system will be heavily reliant on it.

Command list interface

- getnextPass()
- insertPass()
- removePass()
- listPass()

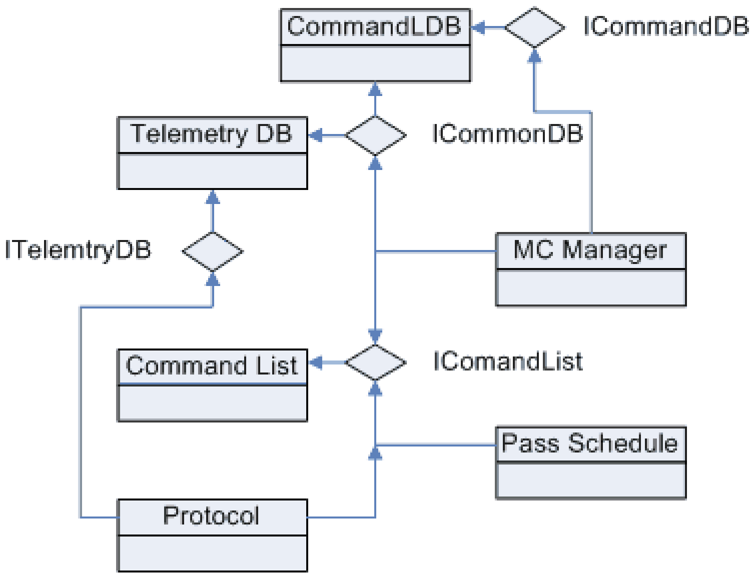

Figure 4.1: Interface overview

- getCommandObject()
- insertCommand()
- modifyCommand()
- duplicateCommand()
- checkPass()
- insertCommandInPass()
- removeCommandFromPass()
- autoInsertInPass()

With *insertCommand()* we have two different methods and there are three with the *modifyCommand()*, this is due to the fact that there is different ways to create a command in or to later modify it. Inserting a command can either be done with or without a preferred time for the command to be sent in. Modifications of the commands are based on its unique identification and the field which should be altered, this is mainly due to the fact that it is not ordinary to replace the entire command but only a section of it.

The more important structure of the rest of the commands will be discussed in the chapter Implementing the Floating Command List.

4.2.2 Database Interfaces

There are three interfaces for the database, one for the common uses of a database, one for storing the responses from the satellite and one for storing the command list. The idea between splitting them is that there are three different aspects of storing data, one will mainly just require storing (used in the protocol) and the other one will need to store, update and retrieve data (used in the command list), the third one will be used for the common methods both use.

The three interfaces are constructed as shown in the following tables.

Common database interface

- startConnection()
- closeConnection()
- destroyTable()
- list()

Command list database interface

- existsCommandTable()
- existsPassTable()
- insertCommand()
- updateCommand()
- insertPass()
- updatePass()

Storage database interface

- existsTable()

- insertData()
- retrieveData()

The reason that there is not a common *existsTable()*, is that the *existsTable()* needs to create the different tables for the commands, passages and telemetry if they do not exist. The same is true for the *update* and *insert* methods, although here it is due to the fields there needs to be retrieved.

4.3 Implementation of Mission Control Manager

This is meant to be the core which connects the different parts of the program together. Currently it just starts up the different instances of the system.

The mission control manager takes care of the command list creation and database access needed for this.

4.4 Implementation of Command List

The idea with the command list is that we have a separate place for the different modules of the program to interact through, such that the different modules can be separated from the main program and operate without the other modules. This means that the command list has to be created in a robust way and with as little actual knowledge to the surrounding modules as possible, while still dealing with these modules as the need arises.

There are two main parts to the command list:

- Commands
- Passages

Commands contains the data which we want to transmit to the satellite and some other vital information, such as time which we want the command to be transmitted and a link to a passage if it is scheduled to one. A passage or pass has to keep order over the different commands which should be sent in the next passage of the satellite, such that it is just to pull forth a pass and we got the data required to deal with a passage.

A little feature was necessary in order not to skip a future pass, in the case a new pass should be scheduled earlier than the current one that we have at hand and is checked whenever a new pass is scheduled. This made the system more flexible as one only needs to create the pass that one wants to operate on and not pre-generate all passages up to that point.

The data structures used for this are based on vectors, such as *LinkedList* and the normal *Vector*. The reason behind this is to save time on the creation of the structures, by using already existing structures.

4.5 Implementation of Ground Station Communication

A core part of the project is enabling the two programs, the mission control and ground station, to communicate with each others, such that data can be transferred between the mission control and the satellite going through the ground station.

There is two parts of communication between the mission control and the ground station:

- Requesting future sessions
- Communication during a session

To request a session to the ground station we need to use the Remote Method Invocation (RMI) which was made available to us from the ground station. There are only two methods which can be used over the RMI, namely:

- addSession()
- removeSession()

For further information about this look at [2].

The communication during a session is covered by the protocol, which will follow in the next section.

4.6 Implementation of Protocol

It was part of the project to use a set of protocols to verify that communication between the ground station and the mission control is possible. Since the AX-25 protocol has not yet been developed by the ground station people a test protocol was made. Since the protocol did not actually have a satellite to talk with yet and the AX-25 protocol will eventually be developed, then the protocol was made as simple as possible. For this purpose a library of objects called tokens was made, these can then be passed back and forth depending on which response should be made. These tokens, depending on the type, contain the information that is needed for the current communication.

The protocol use TPC/IP [5] for the connection and data transfer between each of them, as this is an easy predefined way to communicate over an internet or intranet connection.

The ground station side of the protocol needs to have a fixed structure, such that it can be used by session which was passed over earlier. This means that it should extend the abstract protocol class which was made for that purpose and thereby follow the rules set in that. Which means it should have a *connect()*, *start()*, *stop()*, *abort()*, *getStatus()* and *getProtocolType()*.

Although it was decided in the design that the ground station should be the client and the mission control the server, we reversed that in the implementation due to avoiding future problems with firewalls, should they arise. Although this is a minor detail, then it is important in the way that protocols are formed as now there is a need for a timeout in the protocol on the ground stations side. The timeout is needed to ensure that the session does not wait on a connection, there might not appear, up till the moment the session is killed, but allows the session to shut down in the proper way.

The protocol is a more advanced form of echo protocol [7], meaning that responses are based on the communication it just received. The difference is that in some cases no or a random answer is given.

4.7 Implementation of Database

Having no prior experience in MySQL [4] which was used for this project, it was decided early that spending a period time on getting acquainted with its workings was necessary. This means that a prototype for the database was made in the early stages of the project, which included a Graphical User Interface (GUI). The implementation of the actual database which is used currently was created with this prototype in mind, thereby saving the time spent earlier on in the project at a time where fast progress was needed.

It was mentioned earlier that the interfaces consist of three parts, a common interface and then one for each of the two key parts in the database workings. The reason it is split up into two parts is due to the reason that there might be two databases in the future, one for the command list and one for the telemetry data received from the satellite. This should allow greater flexibility for the mission control system and the future program there will fetch the telemetry data, as the later program will not be as dependent on the mission control systems database being online.

During the early stages of the prototype it was found out that the direct use of the connection to the database was somewhat messy, as the use of parameter was extensive. To deal with this problem a second layer was added on top of the database connection layer, a go-between which would process the commands from the mission control system and fetch the data necessary from the database to comply with the orders. This is shown in figure 4.2.

The restoring of the command list from the database caused some troubles initially when objects was retrieved from the database. They have the old values, but are created with a different value in the command list. This means that a command no longer points at the correct pass but at where the pass used to be. This problem was solved by running through the pass list, thereby every pass can re-associate the pass field in the commands.

4.8 Implementation of Operator Interfaces (GUI)

Currently this part is omitted due to lack of time at the end of the project. There needs to be designed some interfaces which can be used to establish communication between the GUIs and those parts they operate with.

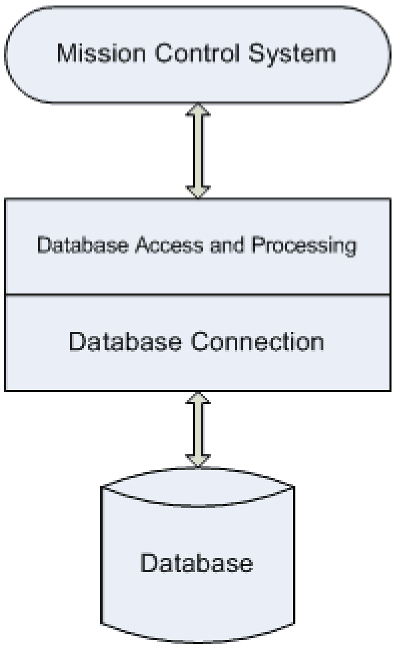

Figure 4.2: Database Access

Chapter 5

Testing

Today most software is complex, which makes functional testing of the entire system more demanding than ever before. As system development still depends on test of the final product, an alternative way to test the system was taken into use. The way that is done is to test the individual modules separately, to find out if they perform as expected and live up to the specifications set in the requirements [6].

It should be kept in mind that most of the methods we test here only have one outcome once the graphical user interface (GUI) is connected. This is is because at that point only existing commands can be chosen as that is all there is shown on the GUI which will be made available for the operators. A second GUI will be available for entering new commands into the command list.

5.1 Unit Testing

Unit testing validates that a particular module of the system conforms to its specifications and correctly performs all its required functions. It produces tests for the behavior of components of the system, to ensure their correct behavior prior to the system integration.

This section will therefor contain the individual tests of each of the modules in the system and the approach to each of the different test strategies used for the modules. The design behind the test cases can be found in the Appendix B.

5.1.1 Command List

The testing of the command list can be quite extensive unless focus is placed upon the methods which can create problems. Furthermore it will have to be split up into three parts

1. Commands
2. Passages
3. Crossover

5.1.1.1 Commands

The testing of commands is pretty simple compared to the two later parts. The testing of these would be trivial and their functionality will be tested in the Crossover section, to create work material and verify the final results.

5.1.1.2 Passages

Testing passages consists of three parts, *insertPass(Pass)*, *removePass(Pass)* and finally *getNextPass()*.

insertPass(Pass)

- There are two reasons why a pass can not be successfully inserted into the pass schedule. The first reason is that the start time of the pass lies before the current time. The second reason is that it conflicts with an already existing pass on the schedule.

  The first reason is easily tested, as it is as simple as to place the start time a little earlier than the current time. The test failed as expected so a pass with an impossible start time wont be added.

The overlapping problem are only slightly more difficult, but that is only in the way that more tests are needed to prove it. Figure 5.1 shows the four different test cases there is needed to prove the different types of overlapping problems there might arise. All four test cases fails to insert the new pass into the pass schedule.

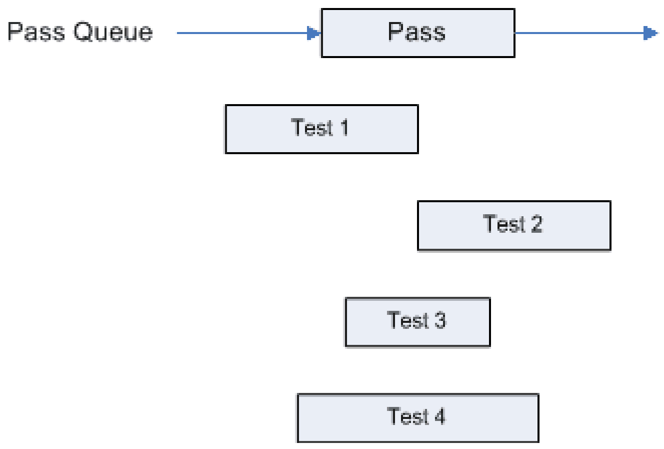

Figure 5.1: Overlapping Passages

removePass(Pass)

- There are two possibilities for a *removePass(Pass)*, either the pass exist or it does not. So the outcome is either a success or a failure in the form of an exception. It should however be noted that under normal operations of the system the *removePass(Pass)* can not be called with a non existing pass, due to the fact that the pass it selected from the pass schedule. The result of the tests are as expected once more.

getNextPass()

- We have two test scenarios there deals with *getNextPass(Pass)*, first we place a new pass after the first pass on the schedule and then check or we place a new pass before the first pass on the schedule and check. With should get the first pass back in the first case and the new pass in the second case. The result are as we believed they would be.

5.1.1.3 Crossover

The crossover means that we work with both commands and passages mixed together, to create a complete passage with commands. There is three methods with stands out and need to be tested, namely *insertCommandInPass(Pass, CommandObject)*, *removeCommandFromPass(Pass, CommandObject)* and *autoInsertInPass(Pass).*

insertCommandInPass(Pass, CommandObject)

- There is three variations of this, given that the pass already exists.
  - That there is room for the command in the pass
  - That the command has already been assigned to a pass
  - The pass is full and can not contain more commands

  The first test is just a standard test to see if the command is actually inserted in the pass and that the commands field containing a pass is updated. The second test is the denial of inserting a command there is already assigned to a pass and therefor should not be associated with one more pass. The final test is to see that it rejects the insertion of a new command when the pass is full. All test acted according to the expectation.

removeCommandFromPass(Pass, CommandObject)

- Testing the removal of a command from a pass is basically just to test if the cleanup is done correctly, which means that the command should not be associated with the pass anymore, nor should the pass contain the command. Once more remember that only commands already existing on the passage can be removed due to the structure of the system. The test did as expected and succeeded.

autoInsertInPass(Pass)

- This is a little more tricky to test, as there is almost an unlimited combination of tests there can be used to create a complete test of it. So it was decided that a few selective tests should be able to establish a clear picture of wether it works or not.

  First we complete a nearly full pass, for this purpose we have restricted the pass to maximum contain five commands. This is done by filling in four commands where three of them are of *low* priority and one of *normal.* Now we place pairs of commands that are not associated with any pass in the command list. Then we run the *autoInsertInPass()* command and see what happens.

For this purpose it was decided that three test cases should show enough to be certain of the functionality.

– Two urgent priority commands waiting to be assigned
– A normal priority and then an urgent priority command
– Two low priority commands

The results of the test was that in the first case both urgent commands were moved in and a low priority was taken out, in the second a low priority was taken out and both commands was put in and finally in the last one just one low was inserted as the other one did not rank high enough to replace an already scheduled command. This means that the test ran as expected.

5.1.2 Data-storage

Since the two parts of the database storage is built as nearly exact copies, it is only necessary to test the more complex one of them to see if the functionality is as expected, this means testing the command storing and retrieval. Both of these are tested at the same time, namely as the command list is stored and the program is shut down, then we fetch the entire data again to recreate the old command list. There is one more test to be preformed and that is to alter the command list and see if the changes have been made once the database is accessed the next time. Both test ran as expected.

5.1.3 Connecting with Ground Station and Protocol

The connection with the ground station was done at the same time as the protocol in a little test program made for that purpose. The test program starts by sending over a session, such that it has been scheduled, then it waits for the session to start and once it starts it tries to connect with the protocol and communicate. The test program can be found in Appendix A.6.1.

The session was passed to the ground station and at the time given in the session the protocol connected and established communication. The communication ran as expected so the testing of the protocol and RMI was successful.

5.2 Integrated Testing

With an integrated test we combine the individual modules into the final system and start testing it as an entire program to verify that the functionalities of the system is as expected. Aside from this we also test the reliability to see if a result can be reproduced each time it is run and that the performance is stable.

5.2.1 Testing against the Ground Station

This part of the testing was done up against the actual ground station running in building 348, with the protocol EchoD implemented on it, such that we could simulate a satellite passage.

To do the integrated testing everything has been coupled together in the *MCManager.java*, such that running this will use each module as it is needed. The tests have been collected in a test batch called *Test.java* in Appendix A.6.2, the test needed to be run is then fetched in the *MCManager.java*.

The two main tests are done in *createCMDL4()* and *createCMDL5()*, the *createCMDL6()* is used to clean up the tables in between tests.

createCMDL4()

- *createCMDL4()* creates a series of ten commands and two passages and stores them in the database. It also assigns some of the commands to passages, leaving the two urgent commands floating on the list. It later then auto assigns these two urgent commands to the first passage. After this is done the first of the two passages is prepared for the ground station. Meaning that a session is created and sent to the ground station and then it awaits the start of the session. Once the session is started the protocol is activated and communication is started. While this is progressing the telemetry data is stored in the database whenever an answer to a command comes back. Once the transmission is done the connection is closed and the next pass is moved up and made ready.

  The test ran flawless and the results were as expected.

createCMDL5()

- *createCMDL5()* recreates the command list from the database based on the construction made in *createCMDL4()*. It then follows the same pattern as before.

The recreation was sound and the test was a duplication from *createCMDL4()* except from the incorporated randomness involved in the protocol.

Chapter 6

Conclusion

This chapter concludes the thesis work related to the project of Satellite Mission Control System. We will first take a look at wether the things we planned for the system to do in the previous sections was actually accomplished and to which extent it lives up to the specifications.

Finally we will look at some further development there might be of interest to the system, as they got discussed during the project but not implemented.

6.1 Evaluation of the System

Through the unit testing, integration testing and finally testing up against the ground station in Chapter 5, all the functionalities of the mission control system have been thoroughly tested. The command list does what it is supposed to do, the automation works as expected and finally the interaction with the ground station has been achieved.

A command list has been created to keep track of commands and passages, such that the commands can be used in future passages and retrieved once needed by the protocol. Furthermore a database access has been created to store and retrieve data from the command list, such that it can be recreated after a shutdown.

The automated pass dealing has been accomplished, such that the system can create a session on the ground station and later connect to it through the protocol without having a user nearby. The data is then stored in a database for later retrieval.

A test protocol for both the ground station and the mission control was made to test the communication to the extend possible, the radio was still not implemented on the ground station so no actual communication with a satellite was possible, nor listening in on a satellite. Instead a semi random response was incorporated into the test protocol.

The user interfaces was not accomplished within the time limit of the project and they have to be completed before the program can be considered easy to use for people without too much prior knowledge of the rest of this system.

6.2 Ideas for further development

Protocol

Although this as been mentioned a few time through this paper then we look forward to the completion of the AX-25 protocol. And it is also worth mentioning that more protocol options should be made available as the ground station get further developed and a better way of implementing them should also be considered.

Data Security

It was mentioned in Chapter 1 that data security might become an issue in the future if a certain computer was made available for the continuous running of the mission control system. Such that a login system was made available for it if used on an easy accessible computer. Another thing worth considering is encrypted communication between the ground station and the mission control, although the data might not be vital then one could think of a situation where a third party might try to take control over the ground station to use it for their own mischievous purposes and this could end up in a loss of the privilege to transmit in the future.

Java Script

Perhaps a java script version of the program should be made such that people with access to a certain web-page could access the ground station through the mission control without having to actually be in possession of the source code, this should also allow more flexibility for those dealing with the commanding of the satellite from any given internet connection they might find themselves at. It should also be kept in mind that the data-security is made at this point then.

A Pass Generation

A pass generation for a given satellite, such that the user only needs to know the name of the satellite and it will generate a pass near a given time unless a pass is already made for that window of opportunity.

Appendix A

Source Code

A.1 Common Files

A.1.1 Tokens

Listing A.1: Basic Token

```java
package common.protocol.echo.tokens;

import java.io.Serializable;
import java.util.Date;

/**
 * @author Brian Schmidt Hermansen
 */
public abstract class BasicToken implements Serializable {

    public Date timeStamp;
    public String satellite;
}
```

Listing A.2: Command Token

```java
package common.protocol.echo.tokens;

/**
 * @author Brian Schmidt Hermansen
 */
public class CommandToken extends BasicToken {
    private static final long serialVersionUID = -5895290162697056299L;
    public String command;
    public int UID;
}
```

Listing A.3: End Token

```
package common.protocol.echo.tokens;

/**
 * @author Brian Schmidt Hermansen
 */
public class EndToken extends BasicToken {

    private static final long serialVersionUID = -9177285262989235157L;

}
```

Listing A.4: Received Token

```
package common.protocol.echo.tokens;

/**
 * @author Brian Schmidt Hermansen
 */
public class ReceivedToken extends BasicToken {

    private static final long serialVersionUID = 7850540244513677384L;
    public String command;
    public int UID;
}
```

Listing A.5: Request Token

```
package common.protocol.echo.tokens;

/**
 * @author Brian Schmidt Hermansen
 */
public class RequestToken extends BasicToken {

    private static final long serialVersionUID = 7434425652671832198L;
}
```

Listing A.6: Satellite Token

```
package common.protocol.echo.tokens;

/**
 * @author Brian Schmidt Hermansen
 */
public class SatelliteToken extends BasicToken {

    private static final long serialVersionUID = 8078924761412160371L;
    public int UID;
}
```

Listing A.7: Teledata Token

```
package common.protocol.echo.tokens;

/**
 * @author Brian Schmidt Hermansen
 */
public class TeledataToken extends BasicToken {

    private static final long serialVersionUID = 8923986302142992427L;
    public String answer;
    public int UID;
}
```

A.1.2 EchoDProtocol.java

Listing A.8: EchoD Protocol

```
package common.protocol.echo;

import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.net.SocketAddress;
import java.util.ArrayList;
import java.util.logging.Level;

import common.log.ILog;
import common.protocol.Protocol;
import common.protocol.ProtocolException;
import common.protocol.echo.tokens.EndToken;
import common.protocol.echo.tokens.ReceivedToken;
import common.protocol.echo.tokens.RequestToken;

/**
 * @author Brian Schmidt Hermansen (modification)
 */
public class EchoDProtocol extends Protocol {

    protected ServerSocket serv;
    protected Socket con;
    protected ObjectInputStream in;
    protected ObjectOutputStream out;
    protected EchoDThread thread;

    public EchoDProtocol(ILog logger){
        super(logger);
    }

    public synchronized void connect(SocketAddress sa)
    throws ProtocolException {
        try {
            con = new Socket();
            con.connect(sa);
            in = new ObjectInputStream(con.getInputStream());
            out = new ObjectOutputStream(con.getOutputStream());

        } catch (Exception e){
            throw new ProtocolException("Connection failed",e);
        }
    }

    public void connect(int port) throws ProtocolException {
        try {
            if (serv==null) serv = new ServerSocket(port);
            serv.setSoTimeout(60*1000);
            try {
                Socket con = serv.accept();
                in = new ObjectInputStream(con.getInputStream());
                out = new ObjectOutputStream(con.getOutputStream());
            } finally {
                serv.close();
            }
        } catch (Exception e) {
            throw new ProtocolException("Connection failed",e);
        }
    }

    protected void closeConnection() {
        try {
            in.close();
            out.close();
            con.close();
        } catch (Exception e) {
            logger.logMessage(mod, Level.SEVERE,
                    "Exception when closing" + e);
        }
    }

    class EchoDThread extends Thread {

        private boolean go = true;

        public void cancel() {
            go = false;
            this.interrupt();
        }

        public void run () {
            try {
                out.writeObject(new RequestToken());
                out.flush();
                while (go) {
                    Object o = in.readObject();
                    if (o==null) throw new
                        Exception("Connection closed");
                    out.writeObject(new ReceivedToken());
                    out.flush();
                }
                out.writeObject(new EndToken());
                out.flush();
            } catch (Exception e) {
                logger.logMessage(mod, Level.INFO, "Exception "
                    + e.getMessage());
            }
            closeConnection();
        }
    }

    public synchronized void start() throws ProtocolException {
        if (thread!=null) throw new
            ProtocolException("Already started");
        if (con!=null) throw new
            ProtocolException("No connection established");
        thread = new EchoDThread();
        thread.start();
    }

    public void stop() {
        if (thread!=null) thread.cancel();
        closeConnection();
    }

    public void abort() {
        stop();
    }

    public ArrayList getStatus() {
        return new ArrayList();
    }

    public String getProtocolType() {
        return "Echo";
    }
}
```

A.1.3 NoTrackingSession.java

Note that this one is from the ground station and is only included to show its functionality.

Listing A.9: No Tracking Session

```
package common.session;

import java.util.Calendar;
import java.util.logging.Level;

import common.satellite.Satellite;
import common.protocol.Protocol;
import common.protocol.ProtocolException;

/**
 * @author hhl
 */
public class NoTrackingSession extends AutomatedSession {

    private static final long serialVersionUID = -5923765917048141261L;
    protected int protocolPort;
    protected int initialDelay = 10;
    protected int slack = 10;
    protected Calendar stoptime;
    protected Thread myThread;

    public NoTrackingSession(String name, Calendar start, Calendar end,
            Class<? extends Protocol> protocolClass, int port)
            throws InvalidSessionException {
        super(name, start, end, protocolClass,
                new Satellite("CUTE-1") // Just a dummy
        );
        protocolPort = port;
    }
    public void setInitialDelay(int d) {
        if (d >= 0) initialDelay = d;
    }

    public void setSlack(int d) {
        if (d >= 0) slack = d;
    }

    @Override
    public synchronized void abort() {
        super.abort();
        if (myThread!=null)
            myThread.interrupt();
    }

    @Override
    public void run() {
        try {
            myThread = Thread.currentThread();
            logger.logMessage(mod, Level.INFO,
                "Connecting protocol ...");
            try {
                protocol.connect(protocolPort);
                logger.logMessage(mod, Level.INFO, "Connected!");
            } catch (ProtocolException pe) {
                logger.logMessage(mod, Level.SEVERE,
                    "Session '"+getSessionName()+
                        "' " + "could not establish connection at " +
                        protocolPort + ":" +
                        pe.getMessage());
                return;
            }
            Thread.sleep(initialDelay*1000);
            logger.logMessage(mod, Level.INFO,
                "Starting protocol ...");
            try {
                protocol.start();
            } catch (ProtocolException pe) {
                logger.logMessage(mod, Level.SEVERE,
                    "Session '"+getSessionName()+"' "
                        + "could not start protocol: "
                        + pe.getMessage());
                return;
            }
            stoptime = (Calendar) endTime.clone();
            stoptime.add(Calendar.SECOND,-slack);
            long waitingtime = stoptime.getTimeInMillis();
            waitingtime -= System.currentTimeMillis();
            if (waitingtime > 0) Thread.sleep(waitingtime);
            logger.logMessage(mod, Level.INFO, "Stopping protocol ...");
            protocol.stop();
        } catch (InterruptedException e) {
            logger.logMessage(mod, Level.WARNING,
                "Session interrupted.");
        }
    }
}
```

A.2 Command List

A.2.1 ICommandList.java

Listing A.10: Command List Interface

```
package commandList;

import java.util.Calendar;
import java.util.LinkedList;
import java.util.Vector;

/**
 * @author Brian Schmidt Hermansen
 */
public interface ICommandList {

    public Pass getNextPass();
    public boolean insertPass(Pass pass);
    public boolean removePass(Pass pass);
    public LinkedList<Pass> listPass();

    public CommandObject getCommandObject(int identification);
    public boolean insertCommand(Priority priority, String command);
    public boolean insertCommand(Priority priority, String command,
        Calendar time);
    public boolean modifyCommand(CommandObject co, Status status);
    public boolean modifyCommand(CommandObject co, Calendar time);
    public boolean modifyCommand(CommandObject co, Pass pass);
    public int duplicateCommand(CommandObject co);

    public Vector<Boolean> checkPass(Pass pass);
    public boolean insertCommandInPass(Pass pass, CommandObject co);
    public boolean removeCommandFromPass (Pass pass, CommandObject co);
    public boolean autoInsertInPass(Pass pass);
}
```

A.2.2 CommandList.java

Listing A.11: Command List

```
package commandList;

import java.util.Calendar;
import java.util.LinkedList;
import java.util.Vector;

/**
 * @author Brian Schmidt Hermansen
 */
public class CommandList implements ICommandList {
    private LinkedList<Pass> passSchedule = null;
    private LinkedList<CommandObject> commandList = null;
    public boolean passScheduleChanged = false;

    public CommandList() {
        passSchedule = new LinkedList<Pass>();
        commandList = new LinkedList<CommandObject>();
    }

    public Pass getNextPass() {
        int size = passSchedule.size();
        if (size >= 0) {
            Pass nextPass = passSchedule.getFirst();
            passScheduleChanged = false;
            return nextPass;
        }
        return null;
    }

    public boolean insertPass(Pass pass) {
        for (int i = 0; i < passSchedule.size(); i++) {
            Pass tmpPass = passSchedule.get(i);
            if (tmpPass.getTime().firstElement().getTimeInMillis() >
            pass.getTime().firstElement().getTimeInMillis()) {
                if (tmpPass.getTime().firstElement().getTimeInMillis() >
                pass.getTime().lastElement().getTimeInMillis()) {
                    passSchedule.add(i, pass);
                    if (i == 0) passScheduleChanged = true;
                    return true;
                } else if (tmpPass.getTime().lastElement().
                            getTimeInMillis() <
                            pass.getTime().firstElement().
                            getTimeInMillis()) {
                                return false;
                }
                return false;
            }
        }
        passSchedule.add(pass);
        return true;
    }

    public boolean insertPass(int i, Pass pass) {
        passSchedule.add(i, pass);
        if (i == 0) passScheduleChanged = true;
        return true;
    }

    public boolean removePass(Pass pass) {
        Pass tmpPass;
        int size = passSchedule.size();
        int i;
        for (i = 0; i < size; i++) {
            tmpPass = passSchedule.get(i);
            if (tmpPass.equals(pass)) {
                int tmp = pass.numberOfCommands();
                for (int j = 0; j < tmp; j++) {
                    int UID = pass.getID(0);
                    CommandObject co = getCommandObject(UID);
                    removeCommandFromPass(pass, co);
                }
                passSchedule.remove(i);
                if (i == 0)
                    passScheduleChanged = true;
                return true;
            }
        }
        return false;
    }

    public LinkedList<Pass> listPass() {
        return passSchedule;
    }

    public CommandObject getCommandObject(int identification) {
        boolean notFound = true;
        int index = 0;
        CommandObject co = null;
            while(notFound) {
                co = commandList.get(index);
                ++index;
                if (identification == co.getIdentification()) {
                    notFound = false;
                }
            }
        return co;
    }

    public boolean insertCommand(Priority priority, String command) {
        int nextUIN;
        nextUIN = commandList.size();
        CommandObject newCommand = new CommandObject(nextUIN, priority,
                command, Status.NEW);
        commandList.add(newCommand);
        return true;
    }

    public boolean insertCommand(CommandObject co) {
        int nextUIN;
        nextUIN = commandList.size();
        CommandObject newCommand = new CommandObject(nextUIN,
                co.getPriority(),
                co.getCommand(), Status.NEW);
        commandList.add(newCommand);
        return true;
    }

    public boolean insertCommand(Priority priority, String command,
        Calendar time) {
        int nextUIN;
        nextUIN = commandList.size();
        CommandObject newCommand = new CommandObject(nextUIN, priority,
                command, Status.NEW);
        newCommand.setTime(time);
        commandList.add(newCommand);
        return true;
    }

    public boolean modifyCommand(CommandObject co, Status status) {
        int index = commandList.indexOf(co);
        co.setStatus(status);
        commandList.set(index, co);
        return true;
    }

    public boolean modifyCommand(CommandObject co, Calendar time) {
        int index = commandList.indexOf(co);
        co.setTime(time);
        commandList.set(index, co);
        return true;
    }

    public boolean modifyCommand(CommandObject co, Pass pass) {
        int index = commandList.indexOf(co);
        co.setPass(pass);
        commandList.set(index, co);
        return true;
    }

    public int duplicateCommand(CommandObject co) {
        CommandObject tempCommand = commandList.getLast();
        int lastUIN = tempCommand.getIdentification();
        int nextUIN = 1 + lastUIN;
        CommandObject newCommand = new CommandObject(nextUIN,
                co.getPriority(), co.getCommand(), co.getStatus());
        commandList.add(newCommand);
        return nextUIN;
    }

    public Vector<Boolean> checkPass(Pass pass) {
        Vector<Boolean> vt = new Vector<Boolean>();
        int constrain = pass.constrain;
        int size = pass.numberOfCommands();
        if (size == constrain) {
            vt.add(true);
            vt.add(false);
        }
        if (size < constrain) {
            vt.add(false);
            vt.add(false);
        }
        if (size == 0) {
            vt.add(false);
            vt.add(true);
        }
        return vt;
    }

    public boolean insertCommandInPass(Pass pass, CommandObject co) {
        int indexS = passSchedule.indexOf(pass);
        int indexC = commandList.indexOf(co);

        if ((co.getExecutedTime() == null) && (co.getPass() == null) &&
                co.getStatus() == Status.NEW) {
            int identification = co.getIdentification();
            boolean constrainSatisfied =
                pass.queueCommand(identification);
            if (constrainSatisfied) {
                passSchedule.set(indexS, pass);
                co.setPass(pass);
                co.setStatus(Status.PASS);
                commandList.set(indexC, co);
                return true;
            } else {
                Boolean inserted = reSchedulePass(pass, co);
                if (inserted)
                    return true;
                return false;
            }
        }
        return false;
    }

    public boolean removeCommandFromPass (Pass pass, CommandObject co) {
        int indexS = passSchedule.indexOf(pass);
        int indexC = commandList.indexOf(co);

        int identification = co.getIdentification();
        pass.dequeueCommand(identification);
        passSchedule.set(indexS, pass);
        co.setPass(null);
        if (co.getStatus() != Status.DONE)
            co.setStatus(Status.NEW);
        commandList.set(indexC, co);
        return true;
    }

    public boolean autoInsertInPass(Pass pass) {
        int indexS = passSchedule.indexOf(pass);
        int sizeCL = commandList.size();
        Boolean status = true;
        for (int i = 0; i < sizeCL; i++) {
            System.out.println(i);
            CommandObject co = commandList.get(i);
            if (co.getPass() == null && co.getStatus() != Status.DONE
                    && (co.getTime() == null
                    || co.getTime().getTimeInMillis()
                    < pass.getTime().lastElement().getTimeInMillis())) {
                Boolean notFull = checkPass(pass).firstElement();
                System.out.println("Not full? : " + notFull);
                if (!notFull) {
                    insertCommandInPass(pass, co);
                } else if (co.getPriority() == Priority.URGENT) {
                    Boolean change = reSchedulePass(pass, co);
                    if (change)
                        status = false;
                }
            }
        }
        passSchedule.set(indexS, pass);
        return status;
    }

    public int findPassLocation(Pass pass) {
        int i = 0;
        int size = passSchedule.size();
        for (i = 0; i <= size; i++) {
            if (passSchedule.get(i).equals(pass))
                return i+1;
        }
        return -1;
    }

    public void updateCommandsFromPass() {
        int size = passSchedule.size();
        for (int i = 0; i <= size; i++) {
            Pass tmp = passSchedule.get(i);
            for (int j = 0; j <= tmp.numberOfCommands(); j++) {
                CommandObject co = getCommandObject(tmp.getID(j));
                modifyCommand(co, tmp);
            }
        }
    }

    private boolean reSchedulePass(Pass pass, CommandObject co) {
        int i;
        int priority = priorityToInt(co.getPriority());
        CommandObject tmpC = null;
        if (priority > 0) {
            for (i = 0; i < pass.constrain; i++) {
                int UID = pass.getID(i);
                CommandObject tmpCo = commandList.get(UID);
                // Taking the last instance of the lowest priority
                if (priorityToInt(tmpCo.getPriority()) <= priority) {
                    priority = priorityToInt(tmpCo.getPriority());
                    tmpC = tmpCo;
                }
            }
        }
        if (tmpC != null) {
            pass.dequeueCommand(tmpC.getIdentification());
            reScheduleCommand(tmpC);
            pass.queueCommand(co.getIdentification());
            return true;
        }
        return false;
    }

    private Integer priorityToInt(Priority priority) {
        if (priority == Priority.LOW)
            return 0;
        if (priority == Priority.NORMAL)
            return 1;
        if (priority == Priority.HIGH)
            return 2;
        if (priority == Priority.URGENT)
            return 3;
        return -1;
    }

    private void reScheduleCommand(CommandObject co) {
        co.setTime(null);
        if (co.getPriority() == Priority.LOW)
            co.setPriority(Priority.NORMAL);
        if (co.getPriority() == Priority.NORMAL)
            co.setPriority(Priority.HIGH);
    }
}
```

A.2.3 CommandObject.java

Listing A.12: Command Object

```
package commandList;

import java.util.Calendar;

/**
 * @author Brian Schmidt Hermansen
 */
public class CommandObject {

    private int identification = 0;
    private Priority priority;
    private String command;
    private Status status;
    private Calendar time;
    private Calendar executedTime;
    private Pass pass;
    public CommandObject(int identification , Priority priority ,
        String command, Status status) {
        this.identification = identification;
        this.priority = priority;
        this.command = command;
        this.status = status;
    }

    public String toString() {
        if (time != null) {
            if (pass != null) {
                return identification + "␣" + priority + "␣" + command
                + "␣" + status + "␣" + time.toString() + "␣queued";
            }
            return identification + "␣" + priority + "␣" + command + "␣"
            + status + "␣" + time.toString() + "␣not␣queued";
        }
        return identification + "␣" + priority + "␣" + command + "␣"
        + status;
    }

    public int getIdentification() {
        return identification;
    }

    public Priority getPriority() {
        return priority;
    }

    public String getCommand() {
        return command;
    }

    public Status getStatus() {
        return status;
    }

    public Calendar getTime() {
        return time;
    }

    public Calendar getExecutedTime() {
        return executedTime;
    }

    public Pass getPass() {
        return pass;
    }

    public boolean setStatus(Status newStatus) {
        status = newStatus;
        return true;
    }

    public boolean setTime(Calendar time) {
        this.time = time;
        return true;
    }

    public boolean setExecutedTime(Calendar executedTime) {
        this.executedTime = executedTime;
        pass = null;
        return true;
    }
```

```java
    public boolean setPass(Pass setPass) {
        pass = setPass;
        return true;
    }

    public boolean setPriority(Priority p) {
        priority = p;
        return true;
    }
}
```

A.2.4 Pass.java

Listing A.13: Pass Object

```java
package commandList;

import java.util.Vector;
import java.util.Calendar;

/**
 * @author Brian Schmidt Hermansen
 */
public class Pass {

    public final int constrain = 5;
    private Vector<Integer> passVector = new Vector<Integer>();
    private Calendar passStart = null;
    private Calendar passStop = null;

    public Pass(Calendar startTime, Calendar endTime) {
        passStart = startTime;
        passStop = endTime;
    }

    public boolean queueCommand(int identification) {
        if (checkConstrain()) {
            passVector.add(identification);
            return true;
        }
        return false;
    }

    public void dequeueCommand(int identification) {
        int index = passVector.indexOf(identification);
        passVector.remove(index);
    }

    public Vector<Calendar> getTime() {
        Vector<Calendar> vt = new Vector<Calendar>();
        vt.add(passStart);
        vt.add(passStop);
        return vt;
    }

    public int numberOfCommands() {
        return passVector.size();
    }
    public int getID(int i) {
        return passVector.get(i);
    }

    private boolean checkConstrain() {
        if (passVector.size() >= constrain) {
            return false;
        }
        return true;
    }
}
```

A.2.5 Priority.java

Listing A.14: Priority Enumeration

```
package commandList;

/**
 * @author Brian Schmidt Hermansen
 */
public enum Priority {
    LOW,
    NORMAL,
    HIGH,
    URGENT;
}
```

A.2.6 Status.java

Listing A.15: Status Enumeration

```
package commandList;

/**
 * @author Brian Schmidt Hermansen
 */
public enum Status {
    NEW,
    PASS,
    VERIFIED,
    GROUNDSTATION,
    SATELLITE,
    DONE;
}
```

A.3 Operator

A.3.1 MainGUI.java

Listing A.16: Main GUI

```
package operator;

import commandList.CommandList;
/**
 * @author Brian Schmidt Hermansen
 */
public class MainGUI {

    private CommandList cmdl;

    public MainGUI(CommandList cmdl) {
        this.cmdl = cmdl;
    }
}
```

A.3.2 InsertGUI.java

Listing A.17: Insert GUI

```
package operator;

import commandList.CommandList; /**
 * @author Brian Schmidt Hermansen
 */
public class InsertGUI {

    private CommandList cmdl;

    public InsertGUI(CommandList cmdl) {
        this.cmdl = cmdl;
    }
}
```

A.4 Storage

A.4.1 ICommonDataBase.java

Listing A.18: Common Database Interface

```
package storage.interfaces;

import java.sql.Connection;
import java.sql.ResultSet;

/**
 * @author Brian Schmidt Hermansen
 */
public interface ICommonDataBase {

    public Connection startConnection();
    public Connection closeConnection(Connection conn);
    public void destroyTable(Connection conn, String tableName);
    public ResultSet list(Connection conn, String tableName);
}
```

A.4.2 IStorageData.java

Listing A.19: Telemetry Database Interface

```
package storage.interfaces;

import java.sql.Connection;
import storage.DataObject;

/**
 * @author Brian Schmidt Hermansen
 */
public interface IStorageData {

    public void existsTable(Connection conn, String tableName);
    public void insertData(Connection conn, String tableName,
            DataObject daob);
    public DataObject retriveData(Connection conn, String tableName,
            int UID);
}
```

A.4.3 ICommandBase.java

Listing A.20: Command List Database Interface

```
package storage.interfaces;

import java.sql.Connection;

import commandList.CommandObject;
import commandList.Pass;

/**
 * @author Brian Schmidt Hermansen
 */
public interface ICommandBase {

    public void existsTableCommand(Connection conn, String tableName);
    public void existsTablePass(Connection conn, String tableName);
    public void insertCommand(Connection conn, String tableName,
            CommandObject co);
    public void updateCommand(Connection conn, String tableName,
            CommandObject co);
    public void insertPass(Connection conn, String tableName,
            Pass pass);
    public void updatePass(Connection conn, String tableName,
            Pass pass, int passLocation);
}
```

A.4.4 StorageData.java

Listing A.21: Telemetry Data Storage

```
package storage.implementation;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import storage.DataObject;
import storage.interfaces.ICommonDataBase;
import storage.interfaces.IStorageData;

/**
 * @author Brian Schmidt Hermansen
 */
public class StorageData implements IStorageData, ICommonDataBase {

    private Connection conn = null;
    private Statement stmt = null;
    private ResultSet rs = null;
    private String driver = "com.mysql.jdbc.Driver";
    private String host = "jdbc:mysql://localhost/test";
    private String login = "eclipse";
    private String pswd = "brian";

    public Connection startConnection() {
        try {
            Class.forName(driver);
        } catch (ClassNotFoundException e) {
            System.out.println("Unable to load Driver Class");
            return null;
        }
        try {
            conn = DriverManager.getConnection(host, login, pswd);
            return conn;
        } catch (SQLException se) {
            System.out.println("SQL␣Exception:␣" + se.getMessage());
            se.printStackTrace(System.out);
            return null;
        }
    }

    public Connection closeConnection(Connection conn) {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException se) {
            }
        }
        this.conn = conn;
        return null;
    }

    public void existsTable(Connection conn, String tableName) {
        try {
            this.conn = conn;
            stmt = conn.createStatement(java.sql.ResultSet.
                    TYPE_FORWARD_ONLY,
                    java.sql.ResultSet.CONCUR_UPDATABLE);
            stmt.executeUpdate("CREATE␣TABLE␣IF␣NOT␣EXISTS␣"
                    + tableName + "␣("
                    + "priKey␣INT␣NOT␣NULL␣AUTO_INCREMENT␣PRIMARY␣KEY,␣"
                    + "data␣OBJECT)");
        } catch (SQLException se) {
        }
    }

    public void destroyTable(Connection conn, String tableName) {
        try {
            this.conn = conn;
            stmt = conn.createStatement();
            stmt.executeUpdate("DROP␣TABLE␣IF␣EXISTS␣" + tableName);
        } catch (SQLException se) {
        }
    }

    public void insertData(Connection conn, String tableName,
            DataObject daob) {
        if (conn == null) {
            conn = startConnection();
        }
        try {
            stmt = conn.createStatement(java.sql.ResultSet.
                    TYPE_FORWARD_ONLY,
                    java.sql.ResultSet.CONCUR_UPDATABLE);
            rs = stmt.executeQuery("SELECT␣priKey,␣data␣"
                    + "FROM␣" + tableName);
            rs.moveToInsertRow();
            rs.updateObject("data", daob);
            rs.insertRow();
            rs.last();
        } catch (SQLException se) {
        }
        this.conn = conn;
    }
    public ResultSet list(Connection conn, String tableName) {
        if (conn == null) {
            conn = startConnection();
        }
        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery("SELECT * FROM " + tableName);
        } catch (SQLException se) {
        }
        this.conn = conn;
        return rs;
    }

    public DataObject retriveData(Connection conn, String tableName,
        int UID) {
        rs = list(conn, tableName);
        DataObject daob = null;
        try {
            if (rs != null) {
                while (rs.next()) {
                    // <type> info = rs.get<type>( <column name> );
                    daob = (DataObject) rs.getObject("data");
                    if (daob.getIdentification() == UID)
                        return daob;
                }
            }
        } catch (SQLException se) {
        }
        return null;
    }
}
```

A.4.5 CommandBase.java

Listing A.22: Command List Data Storage

```
package storage.implementation;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

import storage.interfaces.ICommandBase;
import storage.interfaces.ICommonDataBase;

import commandList.CommandObject;
import commandList.Pass;

/**
 * @author Brian Schmidt Hermansen
 */
public class CommandBase implements ICommandBase, ICommonDataBase {

    private Connection conn = null;
    private Statement stmt = null;
    private ResultSet rs = null;
    private String sql_op = "";
    private String driver = "com.mysql.jdbc.Driver";
    private String host = "jdbc:mysql://localhost/test";
    private String login = "eclipse";
    private String pswd = "brian";

    public Connection startConnection() {
        try {
            Class.forName(driver);
        } catch (ClassNotFoundException e) {
            System.out.println("Unable to load Driver Class");
            return null;
        }
        try {
            conn = DriverManager.getConnection(host, login, pswd);
            return conn;
        } catch (SQLException se) {
            System.out.println("SQL Exception: " + se.getMessage());
            se.printStackTrace(System.out);
            return null;
        }
    }

    public Connection closeConnection(Connection conn) {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException se) {
            }
        }
        this.conn = conn;
        return null;
    }

    public void existsTableCommand(Connection conn, String tableName) {
        try {
            this.conn = conn;
            stmt = conn.createStatement(java.sql.ResultSet.
                    TYPE_FORWARD_ONLY,
                    java.sql.ResultSet.CONCUR_UPDATABLE);
            stmt.executeUpdate("CREATE TABLE IF NOT EXISTS "
                    + tableName + " ("
                    + "priKey INT NOT NULL AUTO_INCREMENT PRIMARY KEY, "
                    + "command OBJECT");
        } catch (SQLException se) {
        }
    }

    public void existsTablePass(Connection conn, String tableName) {
        try {
            this.conn = conn;
            stmt = conn.createStatement(java.sql.ResultSet.
                    TYPE_FORWARD_ONLY,
                    java.sql.ResultSet.CONCUR_UPDATABLE);
            stmt.executeUpdate("CREATE TABLE IF NOT EXISTS "
                    + tableName + " ("
                    + "priKey INT NOT NULL AUTO_INCREMENT PRIMARY KEY, "
                    + "pass OBJECT");
        } catch (SQLException se) {
        }
    }

    public void destroyTable(Connection conn, String tableName) {
        try {
            this.conn = conn;
            stmt = conn.createStatement();
            stmt.executeUpdate("DROP TABLE IF EXISTS " + tableName);
        } catch (SQLException se) {
            // Ignore
        }
    }

    public void insertCommand(Connection conn, String tableName,
            CommandObject co) {
        if (conn == null) {
            conn = startConnection();
        }
        try {
            stmt = conn.createStatement(java.sql.ResultSet.
                    TYPE_FORWARD_ONLY,
                    java.sql.ResultSet.CONCUR_UPDATABLE);
            rs = stmt.executeQuery("SELECT priKey, command "
                    + "FROM " + tableName);
            rs.moveToInsertRow();

            rs.updateObject("command", co);
            rs.insertRow();
            rs.last();
        } catch (SQLException se) {
        }
        this.conn = conn;
    }

    public void updateCommand(Connection conn, String tableName,
            CommandObject co) {
        if (conn == null) {
            conn = startConnection();
        }
        try {
            stmt = conn.createStatement();
            sql_op = "UPDATE " + tableName + " SET command = '" +
            co + "'" +
            " WHERE priKey = '" + co.getIdentification() + "'";
            stmt.executeUpdate(sql_op);
        } catch (SQLException se) {
        }
        this.conn = conn;
    }

    public ResultSet list(Connection conn, String tableName) {
        if (conn == null) {
            conn = startConnection();
        }
        try {
            stmt = conn.createStatement();
            rs = stmt.executeQuery("SELECT * FROM " + tableName);
        } catch (SQLException se) {
        }
        this.conn = conn;
        return rs;
    }

    public void insertPass(Connection conn, String tableName,
            Pass pass) {
        if (conn == null) {
            conn = startConnection();
        }
        try {
            stmt = conn.createStatement(java.sql.ResultSet.
                    TYPE_FORWARD_ONLY,
                    java.sql.ResultSet.CONCUR_UPDATABLE);
            rs = stmt.executeQuery("SELECT priKey, pass "
                  + "FROM " + tableName);
            rs.moveToInsertRow();
            rs.updateObject("pass", pass);
            rs.insertRow();
            rs.last();
        } catch (SQLException se) {
        }
        this.conn = conn;
    }

    public void updatePass(Connection conn, String tableName,
            Pass pass, int passLocation) {
        if (conn == null) {
            conn = startConnection();
        }
        try {
            stmt = conn.createStatement();
            sql_op = "UPDATE " + tableName + " SET pass = '" +
            pass + "'" +
            " WHERE priKey = '" + passLocation + "'";
            stmt.executeUpdate(sql_op);
        } catch (SQLException se) {
        }
        this.conn = conn;
    }

    public ResultSet retriveCommand(Connection conn, String tableName) {
        if (conn == null) {
            conn = startConnection();
        }
        try {
            stmt = conn.createStatement();

            rs = stmt.executeQuery("SELECT * FROM " + tableName);
        } catch (SQLException se) {
        }
        this.conn = conn;
        return rs;
    }
}
```

A.4.6 StorageAccess.java

Listing A.23: Telemetry Data Access

```
package storage.access;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Vector;

import storage.DataObject;
import storage.implementation.StorageData;

/**
 * @author Brian Schmidt Hermansen
 */
public class StorageAccess {
    private Connection conn = null;
    private StorageData db = null;
    private String tableName = "";
    private ResultSet rs = null;
    public Vector<DataObject> vl = new Vector<DataObject>();
    public Vector<String> vtl = new Vector<String>();

    public StorageAccess(Connection conn, String tableName) {
        db = new StorageData();
        if (conn == null) {
            conn = db.startConnection();
        }
        db.existsTable(conn, tableName);
        this.conn = conn;
        this.tableName = tableName;
    }

    public void close() {
        conn = db.closeConnection(conn);
    }

    public void createLists() {
        rs = db.list(conn, tableName);
        convertRsToVector(rs);
    }

    public Connection getConnection() {
        return conn;
    }

    public void insertData(DataObject daob) {
        db.insertData(conn, tableName, daob);
    }

    private void convertRsToVector(ResultSet rs) {
        DataObject daob = null;
        try {
            if (rs != null) {
                while (rs.next()) {
                    // <type> info = rs.get<type>( <column name> );
                    daob = (DataObject)rs.getObject("data");
                    vl.addElement(daob);
                    vtl.addElement(daob.toString());
                }
            }
        } catch (SQLException se) {
        }
    }
}
```

A.4.7 CommandAccess.java

Listing A.24: Command List Data Access

```
package storage.access;

import java.sql.Connection;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.Vector;
```

```java
import storage.implementation.CommandBase;

import commandList.CommandObject;
import commandList.Pass;

/**
 * @author Brian Schmidt Hermansen
 */
public class CommandAccess {

    private Connection conn = null;
    private CommandBase db = null;
    private String commandTable = "";
    private String passTable = "";
    private ResultSet rs = null;
    private ResultSet rs2 = null;
    public Vector<CommandObject> cl = new Vector<CommandObject>();
    public Vector<String> vtl = new Vector<String>();
    public Vector<Pass> pl = new Vector<Pass>();

    public CommandAccess(String commandTable, String passTable) {
        db = new CommandBase();
        if (conn == null) {
            conn = db.startConnection();
        }
        db.existsTableCommand(conn, commandTable);
        db.existsTablePass(conn, passTable);
        this.commandTable = commandTable;
        this.passTable = passTable;
    }

    public void destroyTables() {
        db.destroyTable(conn, commandTable);
        db.destroyTable(conn, passTable);
    }

    public void close() {
        conn = db.closeConnection(conn);
    }

    public void createLists() {
        rs = db.list(conn, commandTable);
        rs2 = db.list(conn, passTable);
        convertRsToVector(rs);
        convertRsToPassVector(rs2);
    }

    public Connection getConnection() {
        return conn;
    }

    public void insertCommand(CommandObject co) {
        db.insertCommand(conn, commandTable, co);
    }

    public void insertPass(Pass pass) {
        db.insertPass(conn, commandTable, pass);
    }

    public void updateCommand(CommandObject co) {
        db.updateCommand(conn, commandTable, co);
    }

    public void updatePass(Pass pass, int passLocation) {
        db.updatePass(conn, commandTable, pass, passLocation);
    }

    private void convertRsToVector(ResultSet rs) {
        CommandObject co = null;
        try {
            if (rs != null) {
                while (rs.next()) {
                    co = (CommandObject)rs.getObject("command");
                    cl.addElement(co);
                    vtl.addElement(co.toString());
                }
            }
        } catch (SQLException se) {
        }
    }

    private void convertRsToPassVector(ResultSet rs2) {
        Pass pass = null;
        try {
            if (rs2 != null) {
                while (rs2.next()) {
                    pass = (Pass)rs2.getObject("pass");
                    pl.addElement(pass);
                }
            }
        } catch (SQLException se) {
        }
    }
}
```

A.4.8 DataObject.java

Listing A.25: Object for storing Telemetry

```
package storage;

import java.util.Calendar;

/**
 * @author Brian Schmidt Hermansen
 */
public class DataObject {

    private int identification = 0;
    private String data;
    private Calendar date;

    public DataObject(int identification, String data) {
        this.identification = identification;
        this.data = data;
        date = Calendar.getInstance();
    }

    public DataObject(int identification, String data, Calendar date) {
        this.identification = identification;
        this.data = data;
        this.date = date;
    }
    public String toString() {
        return identification + " " + data + " " + date.toString();
    }

    public int getIdentification() {
        return identification;
    }

    public String getData() {
        return data;
    }

    public Calendar getDate() {
        return date;
    }
}
```

A.5 Pass Control

A.5.1 MCManager.java

Listing A.26: Mission Control Manager

```java
package passControl.mcManager;

import passControl.passSchedule.PassScheduleHandler;

import commandList.CommandList;

/**
 * @author Brian Schmidt Hermansen
 */
public class MCManager {

    private static PassScheduleHandler handler;
    public static CommandList cmdl;

    public MCManager() throws Exception {
        cmdl = new CommandList();
        cmdl = Test.createCMDL3();
        handler = new PassScheduleHandler(this);
        handler.start();
    }

    public static void main(String[] args) {
        try {
            System.out.println("MCManager␣is␣operating!");
            MCManager MCMgr = new MCManager();
            //InsertGui iGUI = new InsertGUI(cmdl);
            //MainGUI mGUI = new MainGUI(cmdl);
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

A.5.2 PassScheduleHandler.java

Listing A.27: Pass Schedule Handler

```java
package passControl.passSchedule;

import java.rmi.NotBoundException;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Calendar;

import passControl.mcManager.MCManager;
import passControl.protocol.Protocol;

import commandList.Pass;
import common.protocol.echo.EchoDProtocol;
import common.session.InvalidSessionException;
import common.session.NoTrackingSession;
import common.sessionEnv.IMcc;

/**
 * @author Brian Schmidt Hermansen
 */
public class PassScheduleHandler extends Thread {

    public MCManager mcManager;
    private final static Integer second = 1000;
    private NoTrackingSession session;
    private PassSchedule passSchedule;
    private boolean passedSession = false;
    private boolean createdSession = false;
    String serverAddress = "130.225.79.188";
    int RMIServerPort = 4545;
    int gsProtocolPort = 5656;

    public PassScheduleHandler(MCManager mcManager) {
        passSchedule = new PassSchedule(mcManager.cmdl);
        this.mcManager = mcManager;
    }

    public synchronized void run() {
        try {
            while (true) {
                wait(10 * second);
                System.out
                        .println("Checking␣pass␣Queue␣-"
                        + "␣Done␣every␣10␣seconds");
                if (mcManager.cmdl.passScheduleChanged)
                    passSchedule.nextPass = mcManager.cmdl.getNextPass();
                System.out.println("Next␣pass␣found");
                if (passSchedule.nextPass != null) {
                    if (passSchedule.nextPass.getTime()
                            .firstElement().getTimeInMillis()
                            < System.currentTimeMillis()+(60* second)) {
                        if (!mcManager.cmdl.checkPass(passSchedule.
                                nextPass).lastElement()) {
                            if (!passedSession) {
                                if (!createdSession) {
                                    SessionCreater();
                                    System.out.println
                                        ("Session␣Created");
                                }
                                System.out.println("Trying␣to␣"
                                +"pass␣Session");
                                PassSession();
                            }
                            if (passedSession) {
                                System.out
                                        .println("Transmission"
                                        + "␣getting␣ready");

                                while (true) {
                                    Pass tempPass =
                                            passSchedule.nextPass;
                                Protocol sp = new Protocol(
                                        tempPass, mcManager.cmdl);
                                Calendar start = tempPass.getTime()
                                        .firstElement();
                                wait(start.getTimeInMillis()
                                    − System.currentTimeMillis());
                                wait(5 ∗ second);
                                System.out.println("Transmitting"
                                + "␣data...");
                                sp.run();
                                System.out.println("Done"
                                + "␣transmitting");
                                passedSession = false;
                                break;
                            }
                            passSchedule.donePass = true;
                            passSchedule.updatePassSchedule();
                            System.out.println
                                ("Transmission␣terminated");
                        }
                    } else {
                        System.out.println("Next␣pass␣is␣empty,␣" +
                                "trying␣to␣assign␣commands");
                        mcManager.cmdl.autoInsertInPass(
                                passSchedule.nextPass);
                        if (mcManager.cmdl.checkPass(passSchedule.
                                nextPass).lastElement()) {
                            System.out.println("Still␣empty");
                        }
                    }
                } else {
                    System.out.println("Delay,␣sending␣"
                    + "session␣60␣seconds␣before␣start");
                }
            } else {
                System.out.println("No␣pass␣scheduled");
            }
        }
    } catch (Exception e) {
    }
}

private void SessionCreater() {
    Calendar stime = passSchedule.nextPass.getTime().get(0);
    Calendar etime = passSchedule.nextPass.getTime().get(1);
    try {
        session = new NoTrackingSession(CreateSessionName(stime,
                etime), stime, etime, EchoDProtocol.class,
                gsProtocolPort);
    } catch (InvalidSessionException e) {
        e.printStackTrace();
    }
    createdSession = true;
}

private String CreateSessionName(Calendar startTime,
        Calendar endTime) {
    return "session:" + startTime.get(Calendar.MINUTE) + ":"
            + endTime.get(Calendar.MINUTE);
}

private void PassSession() {
    IMcc GSmgr;
    Registry registry;
    try {
            if (System.getSecurityManager() == null) {
                System.setSecurityManager(new RMISecurityManager());
            }
            registry = LocateRegistry.getRegistry(serverAddress,
                        RMIServerPort);
            System.out.println("Sending session to " + serverAddress
                    + ":" + Integer.toString(RMIServerPort));
            GSmgr = (IMcc) (registry.lookup("GSManager"));
            GSmgr.addSession(session);
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (NotBoundException e) {
            e.printStackTrace();
        } catch (Exception e) {
            System.out.println("Error: " + e.toString());
        }
        passedSession = true;
    }
}
```

A.5.3 PassSchedule.java

Listing A.28: Pass Schedule

```
package passControl.passSchedule;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.TimeZone;

import commandList.CommandList;
import commandList.Pass;

/**
 * @author Brian Schmidt Hermansen
 */
public class PassSchedule {

    public Pass nextPass;
    public boolean donePass = false;
    private CommandList cmdl;

    public PassSchedule(CommandList cmdl) {
         this.cmdl = cmdl;
         nextPass = cmdl.getNextPass();
    }

    public boolean insertPass(Pass pass) throws PassScheduleException {
        int i;
        Calendar startTime;
        Calendar endTime;
        Calendar tmpST;
        Calendar tmpET;
        Calendar currDate;
        Pass tmpPass;
        int count;
        String message = "";
        int interval;
        try {
            message = "";
            startTime = pass.getTime().get(0);
            endTime = pass.getTime().get(1);
            interval = 5; // the minutes
            if (startTime.after(endTime)){
                message = "The session start time is after end time";
                throw new PassScheduleException(message);
            }
            count = cmdl.listPass().size();
            currDate = new GregorianCalendar(TimeZone.getDefault());
            currDate.set(Calendar.MONTH, (currDate.get(Calendar.MONTH)
                    + 1));
            currDate.set(Calendar.MINUTE,
                    (currDate.get(Calendar.MINUTE) + interval));
            if (currDate.after(startTime)) {
                message = "The start time of new pass is outdated";
                throw new PassScheduleException(message);
            }
            synchronized (cmdl.listPass()) {
                for (i = 0; i < count; i++) {
                    tmpPass = cmdl.listPass().get(i);
                    tmpST = tmpPass.getTime().get(0);
                    tmpET = tmpPass.getTime().get(1);
                    if ((tmpET.after(startTime))) {
                        if (tmpST.before(startTime)) {
                            message = "The new pass is "
                            + "overlaping with an existing pass";
                            throw new PassScheduleException(message);
                        } else {
                            if (tmpST.before(endTime)) {
                                message = "The new pass is "
                                + "overlaping with an existing pass";
                                throw new PassScheduleException(message);
                            } else {
                                cmdl.insertPass(i, pass);
                                return true;
                            }
                        }
                    }
                }
                cmdl.insertPass(i, pass);
                return true;
            }
        } catch (Exception e) {
            throw new PassScheduleException("Insert new pass failed: "
                    + message, e);
        }
    }

    public void updatePassSchedule() {
        if (donePass) {
            cmdl.removePass(nextPass);
            nextPass = cmdl.getNextPass();
            donePass = false;
        }
    }
}
```

A.5.4 PassScheduleException.java

Listing A.29: Pass Schedule Exception

```
package passControl.passSchedule;

/**
 * @author Yu Du, Brian Schmidt Hermansen
 */
public class PassScheduleException extends ChainedException{

    private static final long serialVersionUID = −7328064973302193220L;

    public PassScheduleException(String message){
        super(message);
    }

    public PassScheduleException(String message, Throwable cause){
        super(message, cause);
    }
}
```

A.5.5 ChainedException.java

Listing A.30: Chained Exception

```
package passControl.passSchedule;

/**
 * @author Yu Do, Brian Schmidt Hermansen
 */
public class ChainedException extends Exception{

    static final long serialVersionUID = 1435892340570L;
    private Throwable cause = null;

    public ChainedException() {
        super();
    }

    public ChainedException(String message){
        super(message);
    }

    public ChainedException(String message, Throwable cause){
        super(message);
        this.cause = cause;
    }

    public void printStackTrace(){
        super.printStackTrace();
        if (cause != null){
            System.out.println("Caused␣by:␣");
            cause.printStackTrace();
        }
    }

    public void printStackTrace(java.io.PrintStream ps){
        super.printStackTrace(ps);
        if (cause != null){
            ps.println("Caused␣by:␣");
            cause.printStackTrace(ps);
        }
    }

    public void printStackTrace(java.io.PrintWriter pw) {
        super.printStackTrace(pw);
        if (cause != null){
            pw.println("Caused␣by:␣");
            cause.printStackTrace(pw);
        }
    }
}
```

A.5.6 Protocol.java

Listing A.31: Protocol on Mission Control side

```
package passControl.protocol;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.util.Date;

import storage.DataObject;
import storage.access.StorageAccess;

import commandList.CommandList;
import commandList.CommandObject;
import commandList.Pass;
import commandList.Status;
import common.protocol.echo.tokens.BasicToken;
import common.protocol.echo.tokens.CommandToken;
import common.protocol.echo.tokens.EndToken;
import common.protocol.echo.tokens.ReceivedToken;
import common.protocol.echo.tokens.RequestToken;
import common.protocol.echo.tokens.SatelliteToken;
import common.protocol.echo.tokens.TeledataToken;

/**
 * @author Brian Schmidt Hermansen
 */
public class Protocol extends Thread {

    private Pass pass;
    private CommandList cmdl;
    private DataObject daob;
    private String tableName = "data";
    private StorageAccess SA = new StorageAccess(null, tableName);
    String gsAddress = "dtusat2.oersted.dtu.dk";
    Integer gsProtocolPort = 5656;

    public Protocol(Pass pass, CommandList cmdl) {
        this.pass = pass;
        this.cmdl = cmdl;
        // this.satellite = satellite;
    }

    private CommandToken generateCommand(int i) {
        CommandObject co = cmdl.getCommandObject(pass.getID(i));
        CommandToken ct = new CommandToken();
        String command;
        command = co.getCommand();
        ct.command = command;
        ct.UID = co.getIdentification();
        ct.timeStamp = new Date();
        return ct;
    }

    public synchronized void run() {
        Object o;
        boolean handshake = false;
        try {
            Socket socket = new Socket(gsAddress, gsProtocolPort);
            System.out.println("Connected!");
            final ObjectOutputStream oos = new ObjectOutputStream(socket
                    .getOutputStream());
            oos.flush();
            final ObjectInputStream ois = new ObjectInputStream(socket
                    .getInputStream());

            try {
                System.out.println("Waiting␣on␣data");
                o = (BasicToken) ois.readObject();
                if (o instanceof RequestToken)
                    handshake = true;
                while (handshake) {
                    int j = 0; //At command number j
                    CommandToken ct = new CommandToken();
                    boolean moreCommands = true;
                    ct = generateCommand(j);
                    j++;
                    oos.writeObject(ct);
                    oos.flush();
                    System.out.println("Sending␣the␣first␣Command");
                    while (moreCommands) {
                        o = (BasicToken) ois.readObject();
                        if (o instanceof ReceivedToken) {
                            ReceivedToken rt = (ReceivedToken)o;
                            System.out.println("GS␣received␣command␣" +
                                    "token:␣" + rt.UID);
                        } else if (o instanceof RequestToken) {
                            System.out.println
                                ("Sending␣the␣next␣Command");
                            if (j < pass.numberOfCommands()) {
                                ct = generateCommand(j);
                                ++j;
                                oos.writeObject(ct);
                                oos.flush();
                                System.out.println("Next␣Command␣sent");
                            } else {
                                moreCommands = false;
                                oos.writeObject(new EndToken());
                                System.out.println("Sending␣EndToken");
                                oos.flush();
                            }
                        } else {
                            System.out.println
                                ("Critical␣location...␣"
                                + "basic␣token␣but␣not␣expected␣token");
                        }
                    }
                    while (!moreCommands) {
                        o = ois.readObject();
                        if (o instanceof SatelliteToken) {
                            SatelliteToken st = (SatelliteToken)o;
                            System.out.println(st.UID + "␣At␣satellite");
                        } else if (o instanceof TeledataToken) {
                            TeledataToken tt = (TeledataToken)o;
                            cmdl.modifyCommand(cmdl.getCommandObject
                                (tt.UID), Status.DONE);
                            System.out.println("Receiving␣teledata:␣"
                                    + tt.answer);
                            daob = new DataObject(tt.UID, tt.answer);
                            SA.insertData(daob);
                        } else if (o instanceof EndToken) {
                            System.out.println("All␣data␣"
                            + "accounted␣for");
                            break;
                        } else {
                            System.out.println("Wrong␣type␣of"
                                    + "␣token␣Received");
                        }
                    }
                    handshake = false;
                }
                ois.close();
                oos.close();
                socket.close();
                System.out.println("System␣disconnected");
            } catch (ClassNotFoundException e) {
                System.out.println(e);
            }
        } catch (IOException e) {
            System.out.println(e);
        }
    }
}
```

A.6 Test tools

A.6.1 DummyTest.java

Listing A.32: Protocol Testing

```
package mccClient;

import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
import java.rmi.NotBoundException;
import java.rmi.RMISecurityManager;
import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.util.Calendar;

import common.protocol.echo.EchoDProtocol;
import common.protocol.echo.tokens.CommandToken;
import common.protocol.echo.tokens.EndToken;
import common.protocol.echo.tokens.ReceivedToken;
import common.protocol.echo.tokens.RequestToken;
import common.protocol.echo.tokens.SatelliteToken;
import common.protocol.echo.tokens.TeledataToken;
import common.session.AutomatedSession;
import common.session.NoTrackingSession;
import common.sessionEnv.IMcc;

/**
 * @author Brian Schmidt Hermansen
 */
public class DummyTest {

    private static int count = 1;

    public static void main(String[] args) {
        IMcc GSmgr;
        Registry registry;
        AutomatedSession session;
        Calendar start;
        Calendar end;
        int delay = 30;
        int duration = 90;

        for (int i = 0; i < args.length; i++) {
            if (i == 0)
                delay = Integer.parseInt(args[0]);
            if (i == 1)
                duration = Integer.parseInt(args[1]);
        }

        String gsAddress = "dtusat2.oersted.dtu.dk";
        Integer gsRMIPort = 4545;
        if (gsRMIPort == null)
```

```
          gsRMIPort = 1099;
      Integer gsProtocolPort = 5656;

      try {

          //System.setSecurityManager(null);

          if (System.getSecurityManager() == null) {
              System.setSecurityManager(new RMISecurityManager());
          }

          System.out.println("Looking up registry at " + gsAddress
                  + ":" + Integer.toString(gsRMIPort));
          registry = LocateRegistry.getRegistry(gsAddress, gsRMIPort);
          System.out.println("Looking up " + gsAddress + ":"
                  + Integer.toString(gsRMIPort));
          GSmgr = (IMcc) (registry.lookup("GSManager"));

          start = Calendar.getInstance();
          start.add(Calendar.SECOND, delay);
          end = (Calendar) start.clone();
          end.add(Calendar.SECOND, duration);

          session = new NoTrackingSession(
                  "DummyTest",
                  start, end,
                  EchoDProtocol.class,
                  gsProtocolPort);

          System.out.println("Adding session ...");
          GSmgr.addSession(session);
          System.out.println("Waiting for start of session ...");
          Thread.sleep(start.getTimeInMillis()
              - System.currentTimeMillis());
          System.out.println("Session starting, trying to connect...");
          Thread.sleep(5000); // Allow server to be ready

          try {
              Socket socket = new Socket(gsAddress, gsProtocolPort);
              boolean done = false;
              boolean moreCommands = true;

              System.out.println("Connected!");

              final ObjectOutputStream oos =
                      new ObjectOutputStream(socket.getOutputStream());
              oos.flush();
              final ObjectInputStream ois =
                      new ObjectInputStream(socket
                      .getInputStream());

              Object o = ois.readObject();
              System.out.println(o);
              if (!(o instanceof RequestToken)) {
                  System.out.println("Unexpected first object: " + o);
                  return;
              }
              oos.writeObject(generateCommand());
              oos.flush();
              while (!done) {
                  while (moreCommands) {
                      System.out.println("Reading next token from GS");
                      o = ois.readObject();
                      if (o instanceof ReceivedToken) {
                          ReceivedToken rt = (ReceivedToken) o;
```

```
                                    Thread.sleep(1000);
                                    System.out.println("GS received command " +
                                            "token: " + rt.UID);
                               } else if (o instanceof RequestToken) {
                                    System.out.println("GS requesting next " +
                                            "command.");
                                    if (count <= 5) {
                                         System.out.println("Count is: " + count);
                                         oos.writeObject(generateCommand());
                                         oos.flush();
                                         System.out.println("Next Command sent");
                                    } else {
                                         System.out.println("Sending EndToken");
                                         oos.writeObject(new EndToken());
                                         oos.flush();
                                         System.out.println("EndToken sent");
                                         moreCommands = false;
                                    }
                               } else {
                                    System.out.println("Unknown or false " +
                                            "Tokentype received: " + o);
                               }
                          }
                          while (!moreCommands) {
                               o = ois.readObject();
                               if (o instanceof SatelliteToken) {
                                    SatelliteToken st = (SatelliteToken)o;
                                    // responce to userdisplay
                                    System.out.println("Command " + st.UID
                                            +" at satellite");
                               } else if (o instanceof TeledataToken) {
                                    TeledataToken tt = (TeledataToken)o;
                                    // responce to userdisplay
                                    System.out.println("Received teledata: "
                                            + tt.answer);
                                    // storedata
                               } else if (o instanceof EndToken) {
                                    // close down connection
                                    System.out.println("All data accounted for");
                                    break;
                               } else {
                                    // no such token
                                    System.out.println("What happened?");
                               }
                          }
                          done = true;
                     }
                try {
                          socket.close();
                          System.out.println("Connection closed by client");
                     } catch (IOException e) {
                          System.out.println(e.toString());
                     }
               } catch (IOException e) {
                     System.out.println(e.toString());
               }
          } catch (RemoteException e) {
               e.printStackTrace();
          } catch (NotBoundException e) {
               e.printStackTrace();
          } catch (Exception e) {
               System.out.println("Exception: " + e);
          }
     }
```

```
    private static CommandToken generateCommand() {
        CommandToken ct = new CommandToken();
        ct.UID = count;
        ct.command = "Task: " + count;
        ++count;
        return ct;
    }
}
```

A.6.2 Test.java

Listing A.33: Tests

```
package passControl.mcManager;

import java.util.Calendar;
import java.util.Vector;

import storage.access.CommandAccess;

import commandList.CommandList;
import commandList.CommandObject;
import commandList.Pass;
import commandList.Priority;

/**
 * @author Brian Schmidt Hermansen
 */
public class Test {

    public static CommandList cmdl;

    public static CommandList createCMDL() {
        int delay = 40;
        int duration = 90;
        int passDelay = 90;
        int commandDelay = 5;
        Calendar start = Calendar.getInstance();
        Calendar stop;
        Calendar start2;
        Calendar stop2;
        Calendar tmp;
        Calendar tmp2;
        start.add(Calendar.SECOND, delay);
        stop = (Calendar)start.clone();
        stop.add(Calendar.SECOND, duration);
        Pass pass = new Pass(start, stop);
        cmdl.insertPass(pass);
        tmp =(Calendar)start.clone();
        tmp.add(Calendar.SECOND, commandDelay);
        cmdl.insertCommand(Priority.LOW, "Simple Command:1:p1", tmp);
        cmdl.insertCommand(Priority.LOW, "Simple Command:2:p1", tmp);
        cmdl.insertCommand(Priority.LOW, "Simple Command:3:p1", tmp);
        cmdl.insertCommand(Priority.NORMAL, "Simple Command:4:p1", tmp);
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(0));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(1));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(2));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(3));
        //cmdl.insertCommandInPass(pass, cmdl.getCommandObject(4));
        start2 = (Calendar)stop.clone();
        start2.add(Calendar.SECOND, passDelay);
        stop2 = (Calendar)start2.clone();
        stop2.add(Calendar.SECOND, duration);
        Pass pass2 = new Pass(start2, stop2);
        cmdl.insertPass(pass2);
        tmp2 = (Calendar)start2.clone();
        tmp2.add(Calendar.SECOND, commandDelay);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:1:p2", tmp2);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:2:p2", tmp2);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:3:p2", tmp2);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:4:p2", tmp2);
        cmdl.insertCommand(Priority.URGENT, "Urgent_Command:1");
        cmdl.insertCommand(Priority.URGENT, "Urgent_Command:2");
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(5));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(6));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(7));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(8));
        return cmdl;
    }

    public static CommandList createCMDL2() {
        int delay = 40;
        int duration = 90;
        Calendar start = Calendar.getInstance();
        start.add(Calendar.SECOND, delay);
        Calendar stop = (Calendar)start.clone();
        stop.add(Calendar.SECOND, duration);
        Pass pass = new Pass(start, stop);
        cmdl.insertPass(pass);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:1:p1");
        cmdl.insertCommand(Priority.LOW, "Simple_Command:2:p1");
        cmdl.insertCommand(Priority.LOW, "Simple_Command:3:p1");
        cmdl.insertCommand(Priority.NORMAL, "Simple_Command:4:p1");
        cmdl.insertCommand(Priority.NORMAL, "Simple_Command:5:p1");
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(0));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(1));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(2));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(3));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(4));

        Calendar start2 = Calendar.getInstance();
        start2.add(Calendar.SECOND, duration+duration);
        Calendar stop2 = (Calendar)start2.clone();
        stop2.add(Calendar.SECOND, duration);
        Pass pass2 = new Pass(start2, stop2);
        cmdl.insertPass(pass2);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:1:p1");
        cmdl.insertCommand(Priority.LOW, "Simple_Command:2:p1");
        cmdl.insertCommand(Priority.LOW, "Simple_Command:3:p1");
        cmdl.insertCommand(Priority.NORMAL, "Simple_Command:4:p1");
        cmdl.insertCommand(Priority.NORMAL, "Simple_Command:5:p1");
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(5));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(6));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(7));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(8));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(9));

        return cmdl;
    }

    public static CommandList createCMDL3() {
        int delay = 40;
        int duration = 90;
        Calendar start = Calendar.getInstance();
        start.add(Calendar.SECOND, delay);
        Calendar stop = (Calendar)start.clone();
        stop.add(Calendar.SECOND, duration);
        Pass pass = new Pass(start, stop);
        cmdl.insertPass(pass);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:1:p1", start);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:2:p1");
        cmdl.insertCommand(Priority.LOW, "Simple_Command:3:p1");
        cmdl.insertCommand(Priority.NORMAL, "Simple_Command:4:p1");
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(0));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(1));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(2));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(3));

        Calendar start2 = Calendar.getInstance();
        start2.add(Calendar.SECOND, duration+duration);
        Calendar stop2 = (Calendar)start2.clone();
        stop2.add(Calendar.SECOND, duration);
        Pass pass2 = new Pass(start2, stop2);
        cmdl.insertPass(pass2);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:1:p1");
        cmdl.insertCommand(Priority.LOW, "Simple_Command:2:p1");
        cmdl.insertCommand(Priority.LOW, "Simple_Command:3:p1");
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(4));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(5));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(6));

        // Creating two floating urgent commands
        cmdl.insertCommand(Priority.URGENT, "Urgent_1");
        cmdl.insertCommand(Priority.URGENT, "Urgent_2");

        Boolean tmp = cmdl.autoInsertInPass(pass);
        System.out.println(tmp);
        return cmdl;
    }

    public static CommandList createCMDL4() {
        int delay = 40;
        int duration = 90;
        Calendar start = Calendar.getInstance();
        start.add(Calendar.SECOND, delay);
        Calendar stop = (Calendar)start.clone();
        stop.add(Calendar.SECOND, duration);
        CommandAccess CA = new CommandAccess("commandList", "passList");

        Pass pass = new Pass(start, stop);
        cmdl.insertPass(pass);
        CA.insertPass(pass);
        cmdl.insertCommand(Priority.LOW, "Simple_Command:1:p1", start);
        CA.insertCommand(cmdl.getCommandObject(0));
        cmdl.insertCommand(Priority.LOW, "Simple_Command:2:p1");
        CA.insertCommand(cmdl.getCommandObject(1));
        cmdl.insertCommand(Priority.LOW, "Simple_Command:3:p1");
        CA.insertCommand(cmdl.getCommandObject(2));
        cmdl.insertCommand(Priority.NORMAL, "Simple_Command:4:p1");
        CA.insertCommand(cmdl.getCommandObject(3));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(0));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(1));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(2));
        cmdl.insertCommandInPass(pass, cmdl.getCommandObject(3));
        CA.updatePass(pass, cmdl.findPassLocation(pass));

        Calendar start2 = Calendar.getInstance();
        start2.add(Calendar.SECOND, duration+duration);
        Calendar stop2 = (Calendar)start2.clone();
        stop2.add(Calendar.SECOND, duration);
        Pass pass2 = new Pass(start2, stop2);
        cmdl.insertPass(pass2);
        CA.insertPass(pass2);
        cmdl.insertCommand(Priority.LOW, "Simple Command:1:p1");
        CA.insertCommand(cmdl.getCommandObject(4));
        cmdl.insertCommand(Priority.LOW, "Simple Command:2:p1");
        CA.insertCommand(cmdl.getCommandObject(5));
        cmdl.insertCommand(Priority.LOW, "Simple Command:3:p1");
        CA.insertCommand(cmdl.getCommandObject(6));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(4));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(5));
        cmdl.insertCommandInPass(pass2, cmdl.getCommandObject(6));
        CA.updatePass(pass2, cmdl.findPassLocation(pass2));

        cmdl.insertCommand(Priority.URGENT, "Urgent 1");
        CA.insertCommand(cmdl.getCommandObject(7));
        cmdl.insertCommand(Priority.URGENT, "Urgent 2");
        CA.insertCommand(cmdl.getCommandObject(8));

        Boolean tmp = cmdl.autoInsertInPass(pass);
        CA.updatePass(pass, cmdl.findPassLocation(pass));
        System.out.println(tmp);
        return cmdl;
    }

    public static CommandList createCMDL5() {
        CommandAccess CA = new CommandAccess("commandList", "passList");

        CA.createLists();
        Vector<Pass> pl = CA.pl;
        Vector<CommandObject> cl = CA.cl;
        for (int i = 0; i <= pl.size(); i++)
            cmdl.insertPass(pl.get(i));
        for (int i = 0; i <= cl.size(); i++)
            cmdl.insertCommand(cl.get(i));

        cmdl.updateCommandsFromPass();
        return cmdl;
    }

    public static void createCMDL6() {
        CommandAccess CA = new CommandAccess("commandList", "passList");

        CA.destroyTables();
    }
}
```

APPENDIX B

Testing

B.1 Test Cases of Unit Testing

B.1.1 Command List

B.1.1.1 Pass Schedule

Insert Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	pass start time earlier than current time.	The pass insert operation should fail.	Add new pass failed: The start time of the new pass is outdated	The insert failed.

Table B.1: Test 0 for inserting a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	pass start time earlier than the current pass and end time is inside.	The pass insert operation should fail.	Add new pass failed: The new pass is overlapping with an existing pass	The insert failed.

Table B.2: Test 1 for inserting a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	pass start time is inside the current pass and the end time is after.	The pass insert operation should fail.	Add new pass failed: The new pass is overlapping with an existing pass	The insert failed.

Table B.3: Test 2 for inserting a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	both the pass times are inside the current pass.	The pass insert operation should fail.	Add new pass failed: The new pass is overlapping with an existing pass	The insert failed.

Table B.4: Test 3 for inserting a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	pass start time earlier than the current pass and the end time is after.	The pass insert operation should fail.	Add new pass failed: The new pass is overlapping with an existing pass	The insert failed.

Table B.5: Test 4 for inserting a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	inserting a pass on an empty pass queue.	The pass insert operation should succeed.	Add new pass succeeded	The insert succeeded.

Table B.6: Test 5 for inserting a Pass

Remove Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	removing a pass on the queue with the pass present.	The pass remove operation should succeed.	Remove pass: true	The remove succeeded.

Table B.7: Test 0 for removing a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	removing a pass on the empty queue.	The pass remove operation should fail.	Remove pass: false	The remove failed.

Table B.8: Test 1 for removing a Pass

Get Next Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	inserting a new pass behind the current first pass.	The get new pass should return the current pass.	Get new pass: false	It failed to find a new pass ahead of the current pass.

Table B.9: Test 0 for getting the first Pass on queue

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	inserting a new pass infront the current first pass.	The get new pass should return the newly inserted pass.	Get new pass: true	It found a new pass ahead of the current pass.

Table B.10: Test 1 for getting the first Pass on queue

B.1.1.2 Crossover

Insert Command in Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	inserting a new command in an empty pass.	That the command is inserted	Insert Command: true	The command was inserted.

Table B.11: Test 0 for inserting a command in a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	inserting a command in a pass where it has already been queued on another pass.	The command will not be inserted in the pass	Insert Command: false	The command was not inserted.

Table B.12: Test 1 for inserting a command in a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	trying to insert a new command in a full pass.	That it fails since there is not any space left	Insert Command: false	The command was not inserted.

Table B.13: Test 2 for inserting a command in a Pass

Remove Command from Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	trying to remove an existing command from a pass.	Should just remove the command	Remove Command: true	The command was removed.

Table B.14: Test 0 for removing a command from a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	trying to remove a command from a pass where it does not exist.	That it fails since it is not present in the pass	Remove Command: false	The command was not removed.

Table B.15: Test 1 for removing a command from a Pass

Auto Insert in Pass

There is space for five commands in a pass and there is three low and one normal.

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	trying to auto-insert two new commands (both urgent priority) in a full pass.	That it removes the last low command and inserts both urgent	Auto-insert Command: true (means that there was one or more commands dequeued to make space)	The commands was inserted and one of the prior low priority commands was dequeued.

Table B.16: Test 0 for auto inserting commands in a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	trying to auto-insert two new commands (first one urgent priority and the second one normal priority) in a full pass.	That it removes the last low command and inserts both new commands	Auto-insert Command: true (means that there was one or more commands dequeued to make space)	The commands was inserted and one of the prior low priority commands was dequeued.

Table B.17: Test 1 for auto inserting commands in a Pass

Module Overview	Input	Expected Output	Real Output	Result
Pass Schedule	trying to auto-insert two new commands (both low priority) in a full pass.	That it fails since there is not any space left	Insert Command: false (as no command was dequeued to make space)	The the first low priority command was inserted and the last one was not.

Table B.18: Test 2 for auto inserting commands in a Pass

Bibliography

[1] Java Technology
http://java.sun.com/

[2] Yu Du: A satellite ground station control system
IMM-THESIS, 86, 2005

[3] European Space Agency: Packet utilisation standard
ESA PSS07101 Issue 1, May 1994

[4] MySQL Homepage
http://www.mysql.com/

[5] The TCP/IP Guide
http://www.tcpipguide.com/

[6] Software Testing
http://en.wikipedia.org/wiki/Software_testing

[7] Echo Protocol
http://en.wikipedia.org/wiki/ECHO_protocol