Ben Gurion University

NegevSat

Platform for Satellite Software

Application Design Document

Shimon Lahihani

Idan Mor

Hod Amran

Yaniv Radomsky

May 2015

Contents

1.Use Cases

1.1 User Profiles:

- Ground Station – This actor uses the system to send mission file which contains single or several tasks to the system.

- An example of a task is to change the mode of the satellite from operational mode into safe mode or restart the satellite.

- Another usage of this actor is to receive data about the satellite's inner status in order to verify that everything is ok with the satellite.

- Temperature sensor – This actor is not human and external to the system, the temperature sensor measures the temperature of the satellite and the system uses this information in order to switch from operational mode into safe mode when the temperature is above or below normal.

- Power supply sensor – This actor is not human and external to the system, the power supply sensor measures the power level of the power accumulator and the system uses this information in order to switch from operational mode into safe mode when the power level is below normal.

- Magneto meter sensor – This actor is not human and external to the system, mostly used for *attitude* sensing of the satellite, the magneto meter senses the magnetic field of the earth and the system gets this information in order to sense its attitude.

- Timer – This actor is not human and external to the system, the timer triggers the system to perform the tasks, this system is real time system and therefore most of the actions are performed autonomously – the timer triggers the events.

1.2 Use case Diagram:

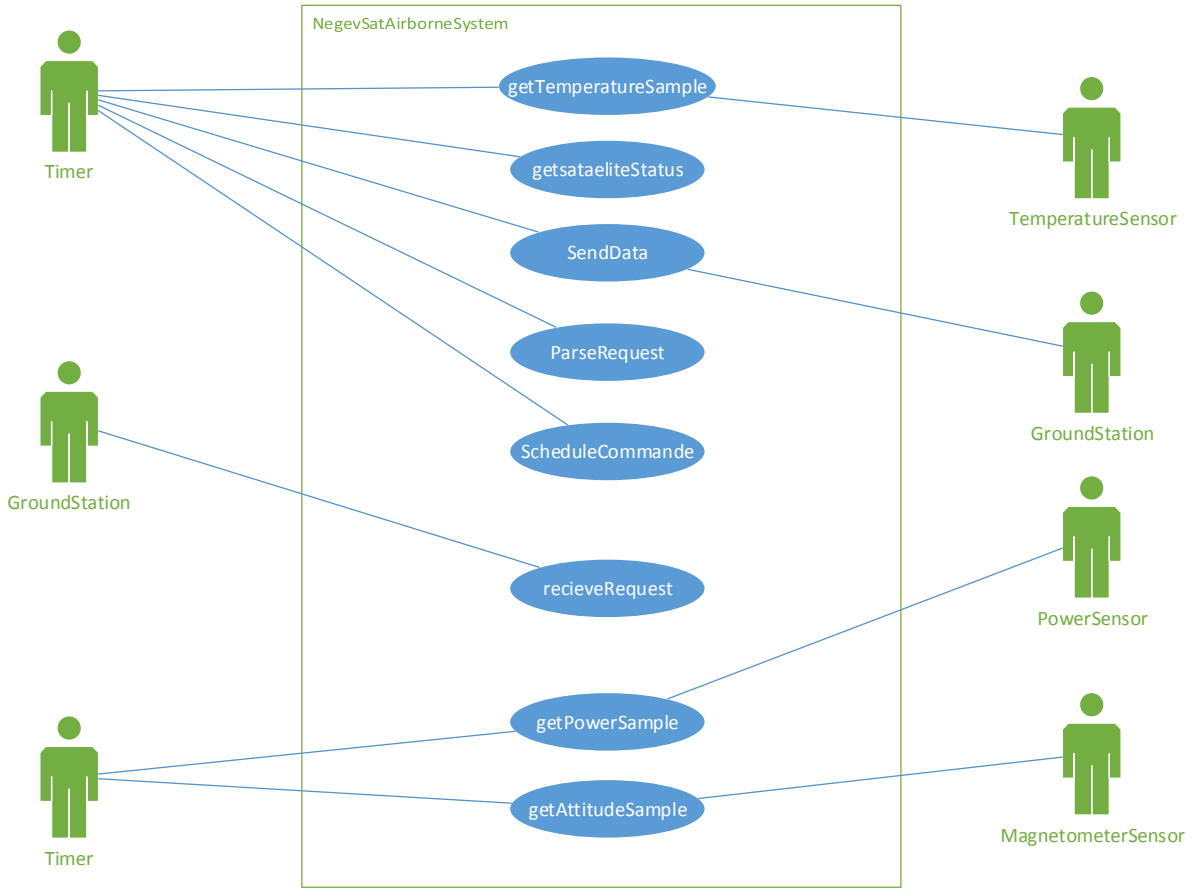


Figure 1.1.1 – Usage Scenarios

1.3 Usage Scenarios:

1.3.1 Get Attitude Sample–

Primary actor: Timer
Description: Get the current attitude measure from magnetometer.
Pre-conditions: The system is not in safe mode.
Post-conditions: The measurement data is stored in the system's memory.

Main success scenario:

1. The timer triggers the system to perform a task - get magnetometer measure.
2. The LifeCycleTask asks the magneto meter to perform a measurement.
3. The magnetometer executes a measurement and sends the data back to the LifeCycleTask.
4. LifeCycleTask adds the sample to the packet.

Alternative flows:

- At any time, the system switches from operational mode into safe mode – the operation will get interrupted and aborted.

- If the magnetometer failed to perform an attitude measure the system will count the failures and if the counter reaches a specific number the system will raise an event for failure.

1.3.2 Get power Sample–

Primary actor: Timer
Description: Get the current power supply measure.
Pre-conditions: none.
Post-conditions: The measurement data is stored in the system's memory.

Main success scenario:

1. The timer triggers the system to perform a task - get power measure.
2. The LifeCycleTask asks the power supply sensor to perform a measurement.
3. The power supply sensor executes a measurement and sends the data back to the LifeCycleTask.
4. LifeCycleTask adds the sample to the packet.

Alternative flows:

- If the power supply sensor failed to perform a measure the system will count the failures and if the counter reaches a specific number the system will raise an event for failure.

1.3.3 Get temperature Sample–

Primary actor: Timer
Description: Get the current temperature measure. **Pre-conditions:** The system is not in safe mode.
Post-conditions: The measurement data is stored in the system's memory.
Main success scenario:

1. The timer triggers the system to perform a task - get temperature measure.
2. The LifeCycleTask asks the temperature sensor to perform a measurement.
3. The temperature sensor executes a measurement and sends the data back to the LifeCycleTask.
4. LifeCycleTask adds the sample to the packet.

Alternative flows:

- At any time, the system switches from operational mode into safe mode – the operation will get interrupted and aborted.
- If the temperature sensor failed to perform a measure the system will count the failures and if the counter reaches a specific number the system will raise an event for failure.

1.3.4 Get satellite status–

Primary actor: Timer.
Description: Get the current satellite measures.
Pre-conditions: none.
Post-conditions: The measurement data is stored in the system's memory.
Main success scenario:

1. The timer triggers the system to perform a task – get satellite status.
2. If CYCLE_LIMIT=0 (mean need to create new packet) call packets factory to create new packets.
3. Perform use case: 1.3.3 – get temperature sample.
4. Perform use case: 1.3.1 – get magneto meter sample.
5. Perform use case: 1.3.2 – get power sample.

6. Increase the cycle counter and check if the counter has reached to CYCLE_LIMIT which was defined statically.
    - 6.1 If the counter reached CYCLE_LIMIT
        - 6.1.1 Transform the packet into bytes using the CMDParser.
        - 6.1.2 Create an empty packet for later storage using the CMDParser.
        - 6.1.3 Enqueue the bytes of the packet into the send queue.
        - 6.1.4 Set the counter to 0.

1.3.5 Send data–

Primary actor: Timer.
Description: Send the data which was stored in the satellite to the ground station when possible.
Pre-conditions: The satellite is in PassAndDownload state (when the satellite faces the ground station and get "Pass" commend from ground station).
Post-conditions: The data which was sent by the satellite arrived to the ground station.
Main success scenario:

1. While there are packets in the send queue do:
    1.1 Dequeue a packet from the send queue.
    1.2 Send the packet using the CommunicationHandler.

1.3.6 Receive request–

Primary actor: Ground station.
Description: Receive a packet from ground station and store it in the receive queue.
Pre-conditions: The satellite is in Operational or SafeMode state, and the satellite faces the ground station.
Post-conditions: The data which was sent by the ground station arrived to the satellite and stored into the received queue.
Main success scenario:

1. The ground station sends packet to the system.
2. ReceiveTask "listens" to the communication port and receives the bytes.
3. While there are bytes on the port do:
3.1 Verify the correction code of the bytes (checksum).
3.2 If the bytes are correct do:
    3.2.1 Enqueue the bytes into the receive queue
3.3 Else drop the bytes

Alternative flows:

- If port is closed or bytes are corrupted or any communication error drops the bytes.

1.3.7 Parse request–

Primary actor: Timer.
Description: Take bytes which arrived to the system and translate it into a work or works.
Pre-conditions: none.
Post-conditions: The bytes translated into works (commands from ground station) and stored into the work queue.

Main success scenario:
1. The timer triggers the MPTask (message parser task) to perform parsing operation.
2. While MPTask dequeues bytes from the receive queue do:

2.1 The Validator builds a packet (Binary)
2.2 The Validator validates the packet with the data protocol (using Binary schema)
2.3 If the packet is valid do:
    2.3.1 CMDParser (command parser) parses the packet into works.
    2.3.2 CMDParser enqueues the works into the work queue.
    2.3.3 CMDParser sorts the works by the time of execution.
2.4 Else drop the bytes

Alternative flows:
- If port is closed or bytes are corrupted or any communication error drops the bytes.

1.3.8 Schedule commands–

Primary actor: Timer.
Description: Schedule commands which arrived from the ground station into the execution work queue when the time of a work arrives.
Pre-conditions: none.
Post-conditions: The works which should be executed (work.time <= current time) are stored in the execution work queue.
Main success scenario:
1. Timer triggers the system to scan for executable commands.
2. While there are works in the work queue (contains all pending works) do:
    2.1 CMDTask (command task) dequeues work from the work queue
    2.2 if the works time is less or equals to the current time do:
        2.2.1 Enqueue the work into the execution work queue.

2. System Architecture

2.1 Components Diagram:

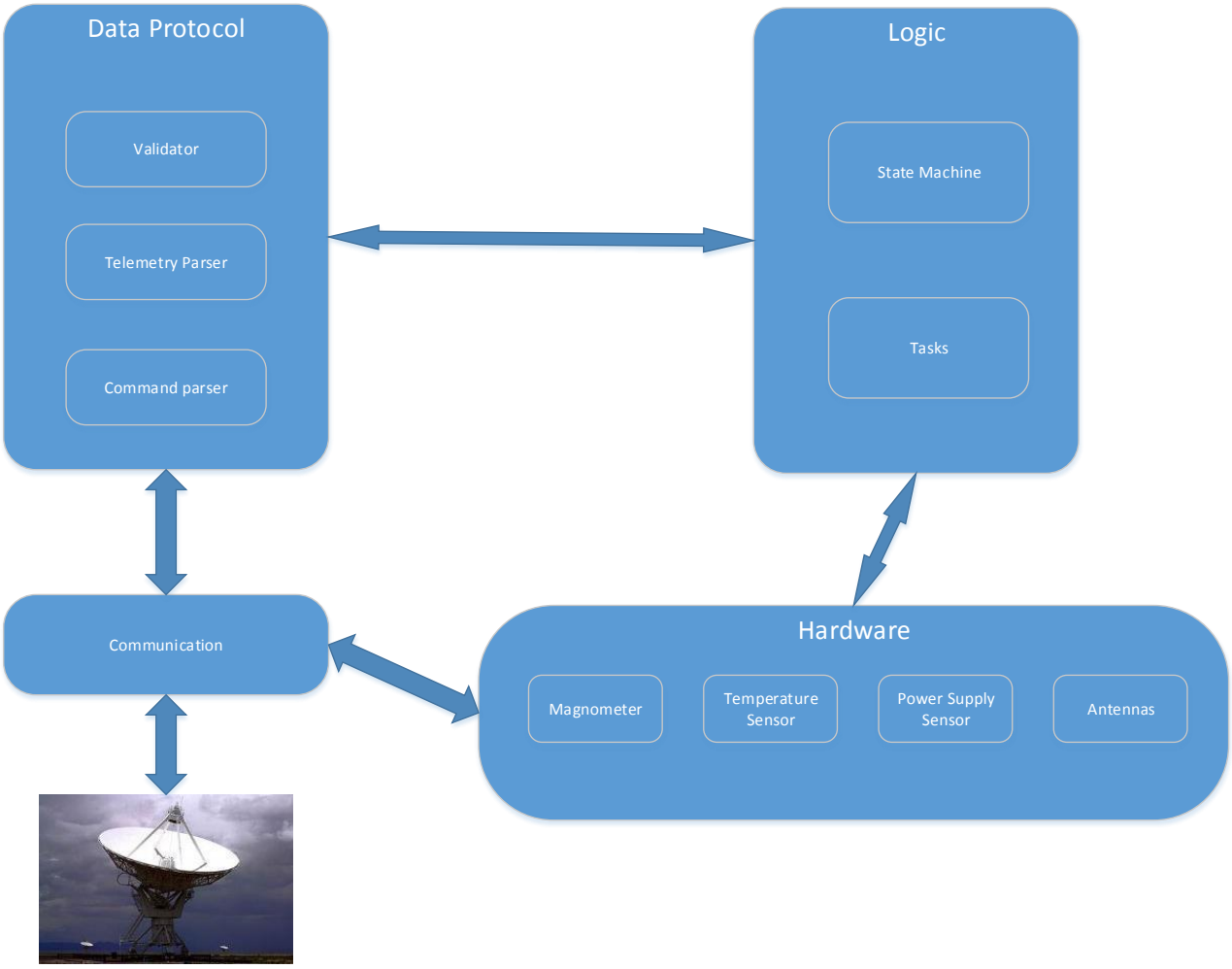


Figure 2.1.1 – System Components

2.2 Components Description:

1. **Communication** – This component is responsible for sending/receiving data from/to the ground station and verifies the bytes are correct using correction/detection code (checksum).

2. **Data Protocol** – Validate the data in order to fit the data protocol (using Binary schema), translate data into readable format for the logic layer or create the formatted packet when trying to send data to the ground station.

This component is divided into 3 parts:

- Telemetry Parser – Responsible for taking the data which was processed by the system and to generate a formatted packet (XML packet) in accordance to the data protocol.
- Command Parser – Responsible for translating the received data into works (commands from the ground station).
- Validator – This unit is responsible for validation of the data which arrived from the ground station, it is implemented by validating the Binary file which was created at earlier stage using the Binary schema (BinSD file).

3. **Logic** – the "heart" of the project, this is the most important unit – responsible for executing the lifecycle of the satellite and take care of "online" requests from the ground station, this unit is divided into two parts:

- **Tasks** – All the runnable tasks of the system, the active objects (equivalent to threads) which are created at the earliest stage of the system and perform all the actions of the system using the passive objects – a more detailed explanation about tasks can be found in section 5.2.
- **State** machine – each state behaves differently in the aspect of hardware devices and software behavior.

This unit includes all the tasks and events of the system and controls the behavior of the system in accordance to the state of the machine and to the events which arrive at any time (see sections 5.2, 4.2, 4.3 and 4.4 for more details).

4. **Hardware** – the Hardware satellite Component that used to measure the satellite status and to connect to the ground station.

It contains the follow devices:

- **Temperature sensor** - measures the temperature of the satellite.
- **Power supply sensor** - measures the power level of the power.
- **Magneto meter sensor** - measures the *attitude* of the satellite, the magneto meter senses the magnetic field of the earth and the system gets this information in order to sense its attitude.

- **Antennas** - used to receive and transmit data to ground station.

3. Data Model

3.1 Description& Relationships Data Objects:

A. for parsing receiving data:

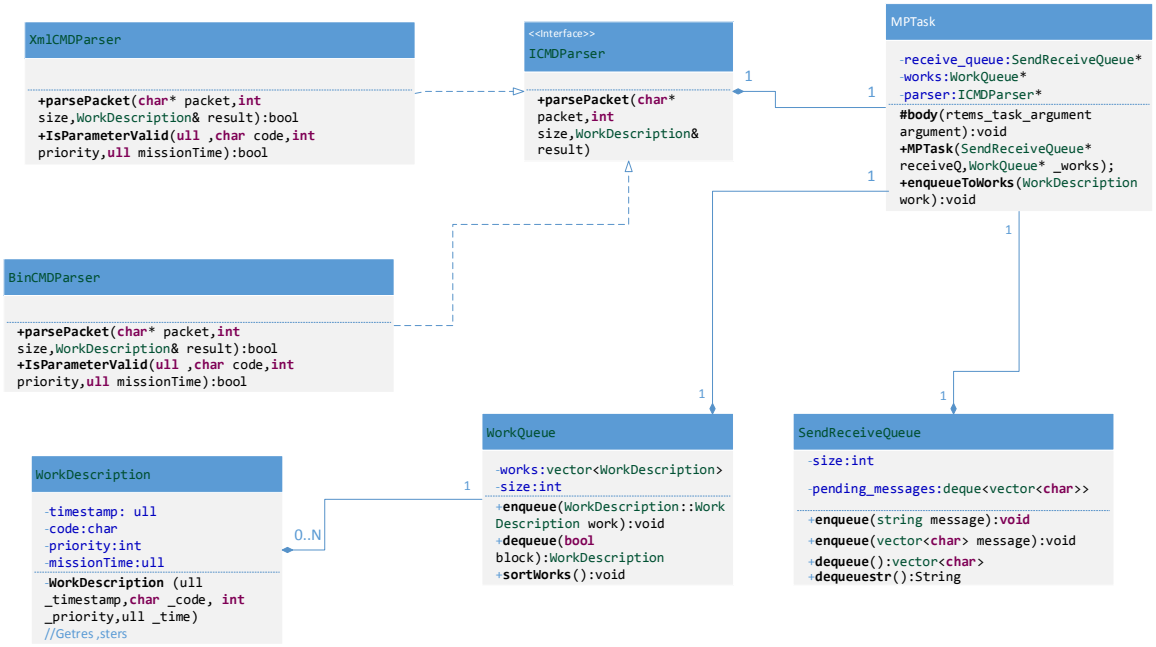


Figure 3.1.a – Data Objects Diagram(parsing receiving data)

3.1.a.1. **ICMDParser**: the interface of parsing packet that comes from ground station.

3.1.a.2. **BinCMDParser**: parser that implement the ICMDParser interface, it parse the packet according to binary encoding.

3.1.a.3. **XmlCMDParser**: parser that implement the ICMDParser interface, it parse the packet according to Xml encoding, this class is optional and can replace the *BinCMDParser* if we want that the communication will be in xml encoding.

3.1.a.4. **MPTask**: Also known as message parsing task – consumes the data which was stored in the received data queue, parses the packet into works (using the *ICMDParser*) and puts the works in the work queue.

3.1.a.5. **SendReceiveQueue**: The queue which stores the data which should be sent or received.

3.1.a.6. **WorkQueue**: The queue which stores the works which arrived to the satellite from the ground station, this queue should be sorted by the time of the execution of the work. Also used to store the works which are ready to be executed by the satellite.

3.1.a.7. **WorkDescription**: Describes the work that the satellite is suppose-to do, includes *code* - which indicates the type of work, *priority* – the priority of the work, *timestamp* –the time that the work received ,missionTime- the time that the mission need to be execute.

3.1.b data Structures for sending data:

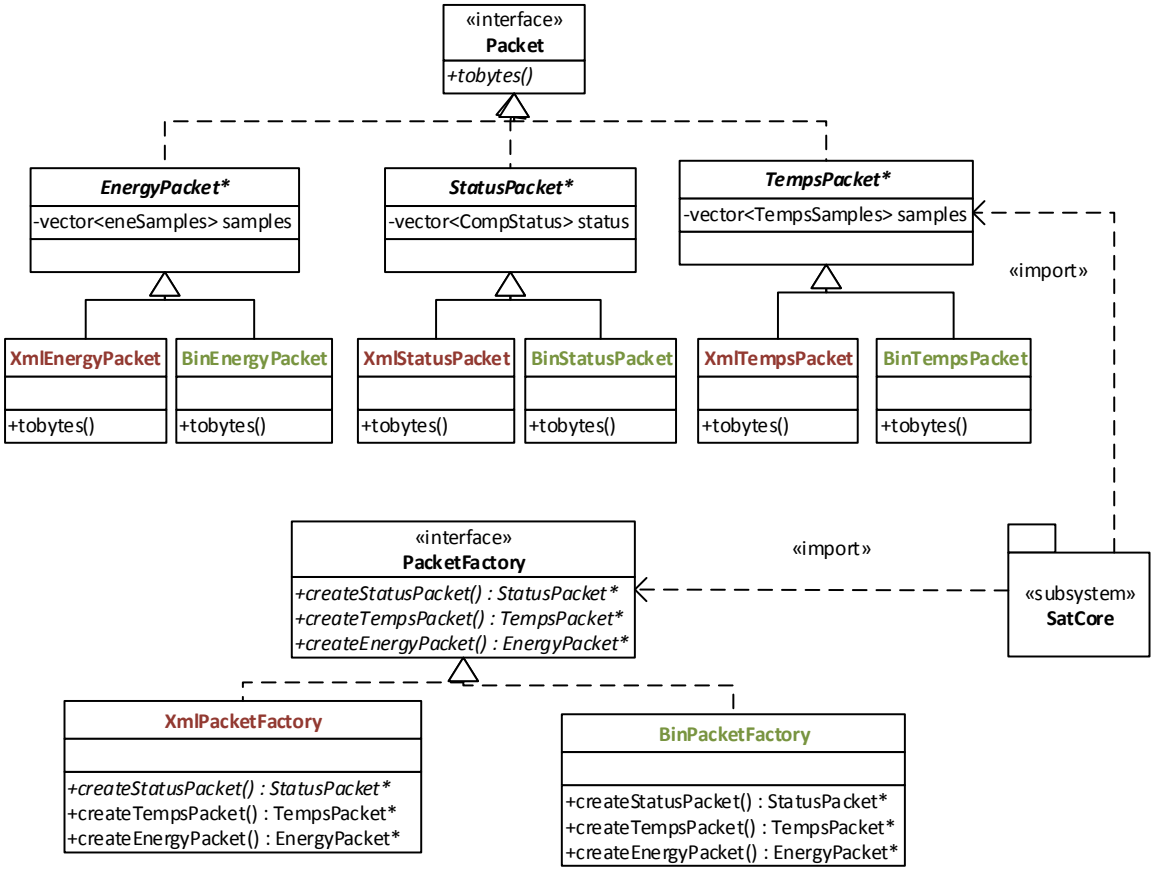


Figure 3.1.b – Data Objects Diagram(data Structures for sending data)

3.1.b.1. **Packet**: the interface of packet that will be sent to ground station, it contains the function *tobytes()* for converting the packets to bytes , that will be send to ground station.

3.1.b.2 **EnergyPacket**: Abstract class, represent an packet that contain energy information of the battery , implement the packet interface.

3.1.b.3. **StatusPacket**: Abstract class, represent an packet that contain information on payloads that is in the satellite, implement the packet interface.

3.1.b.4. **TempsPacket**: Abstract class, represent a packet that contain temperature information of the satellite, implement the packet interface.

3.1.b.5 **XmlEnergyPacket**: this class inherit from class *EnergyPacket* ,implement the *tobytes()* function in XML encoding.

3.1.b.6 **XmlStatusPacket**: this class inherit from class *StatusPacket*,implement the *tobytes()* function in XML encoding.

3.1.b.7 **XmlTempsPacket**: this class inherit from class *TempsPacket*,implement the *tobytes()* function in XML encoding.

3.1.b.8 **BinEnergyPacket**: this class inherit from class *EnergyPacket* ,implement the *tobytes()* function in Bin encoding.

3.1.b.9 **BinStatusPacket**: this class inherit from class *StatusPacket*,implement the *tobytes()* function in Bin encoding.

3.1.b.10 **BinTempsPacket**: this class inherit from class *TempsPacket*,implement the *tobytes()* function in Bin encoding.

3.1.b.11 **PacketFactory**: the interface of packet Factory for creating different type of packets.

3.1.b.12 **XmlPacketFactory**: factory that implement *PacketFactory ,* crate packets in Xml encoding.

3.1.b.13 **BinPacketFactory**: factory that implement *PacketFactory ,* crate packets in Bin encoding.

3.2 Data Base:

This system doesn't have a database – therefore there are no tables to describe. The persistent data is stored as files with timestamps.

4. Behavioral Analysis

4.1 Sequence Diagrams:

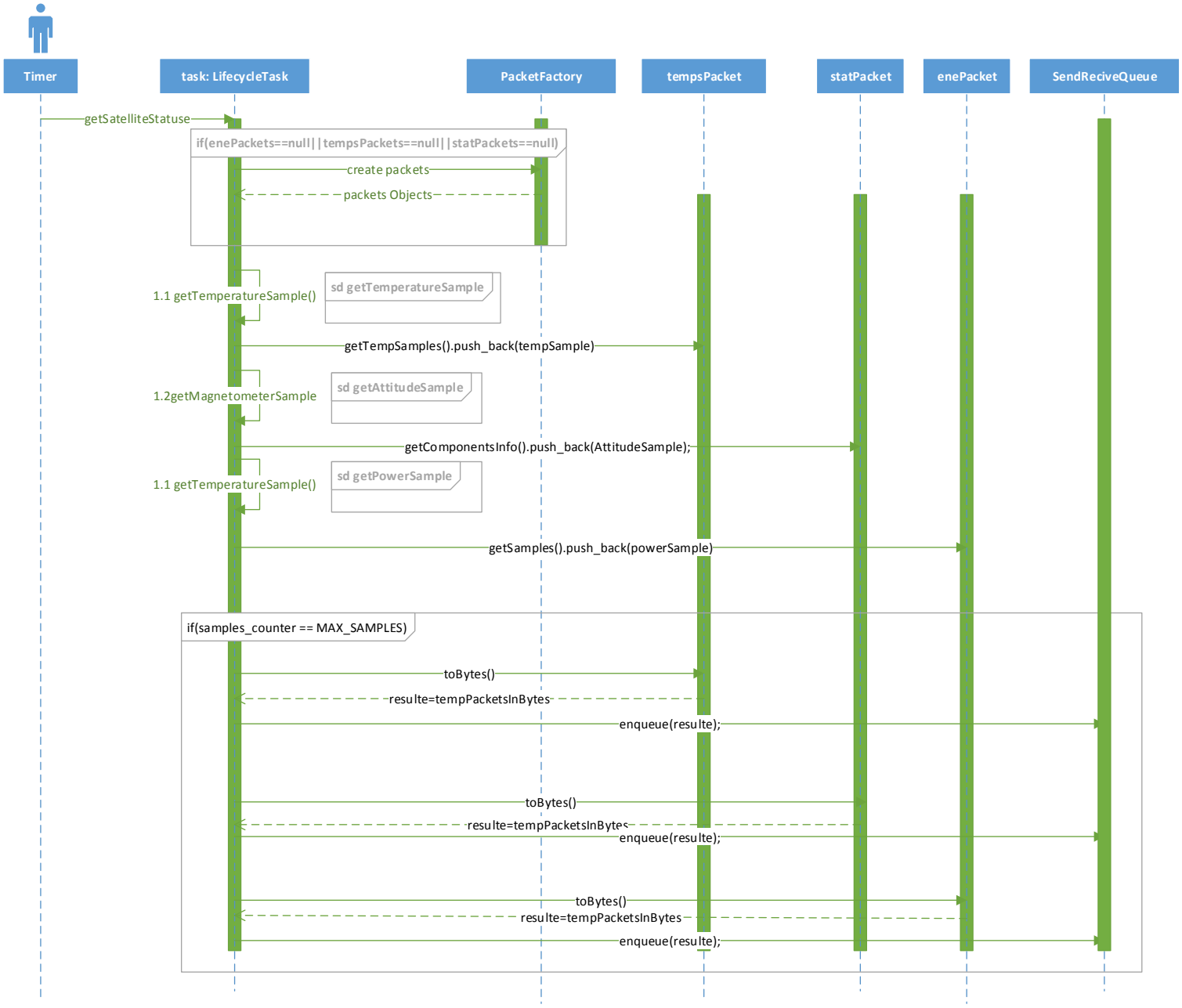


Figure 4.1.1– getSatelliteStatuse Sequence Diagram

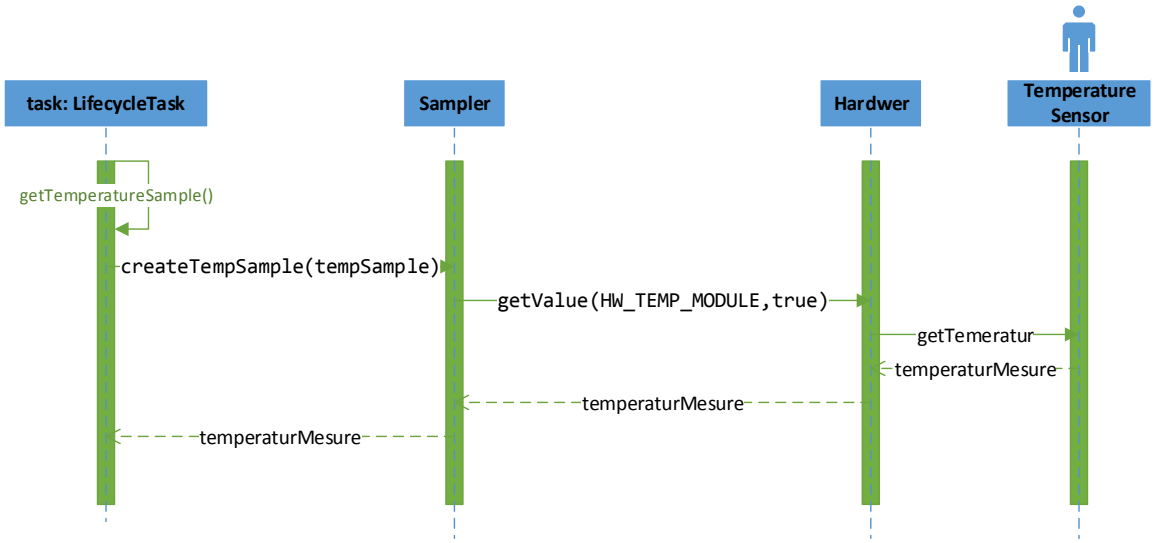


Figure 4.1.2– getTemperatureSample Sequence Diagram

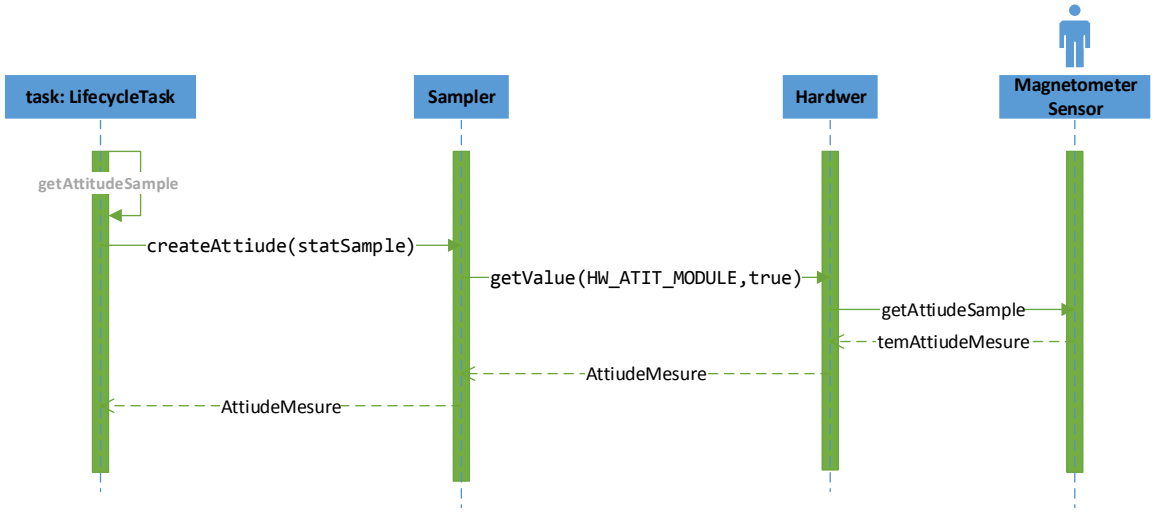


Figure 4.1.3– getAttiudeSample Sequence Diagram

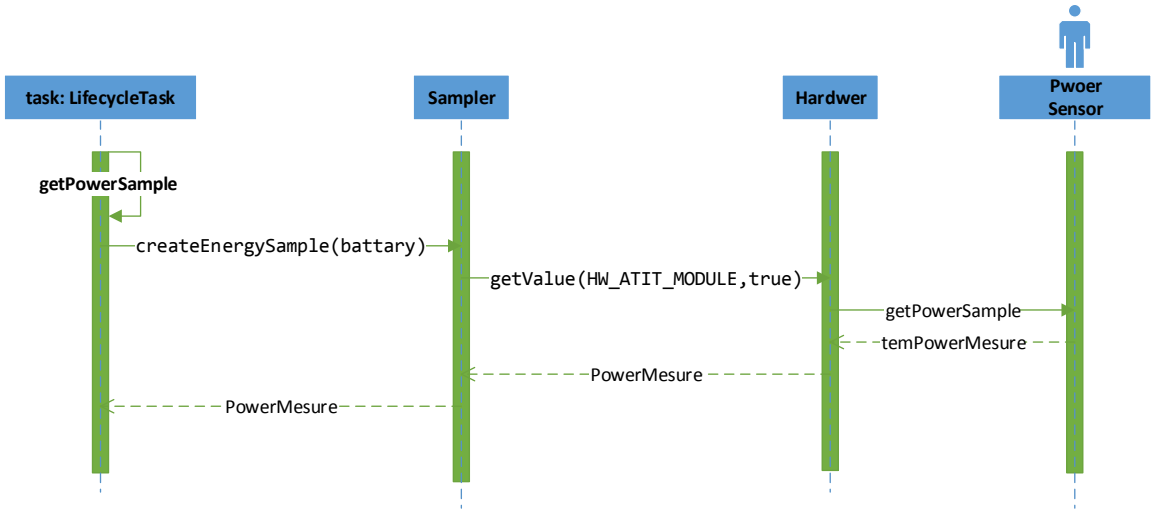


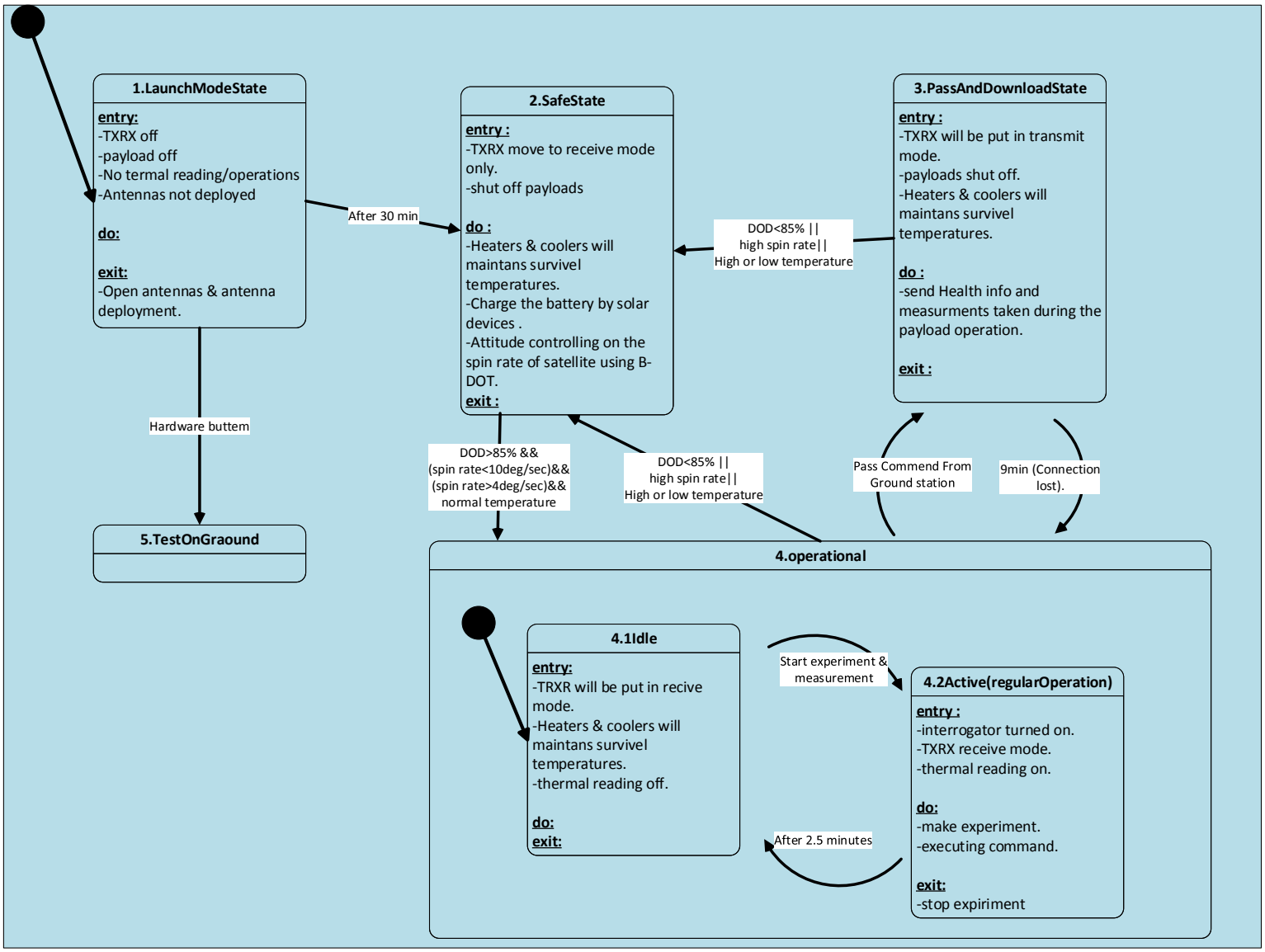
Figure 4.1.4– getPowerSample Sequence Diagram

4.2 Sequence Diagrams:

This system is event based system therefore events affect the system's behavior drastically. There are many events in this system, the main events will be described in this section:

Event	Description	Action
Pass	Triggered when the Ground station send this command (real time)	The system moves to PassAndDownloadState State.
Low/high temperature	Triggered when the temperature samples were low.	The system moves to safe mode and quits doing the non-crucial actions (see States chapter)
(DOD) Depth of discharge low- Low power	Triggered when the power samples were low.	The system moves to safe mode and quits doing the non-crucial actions (see States chapter)
High spin rate	Triggered when the spin rate samples were low from *4 degrees/second* or high than 10 *egrees/second*.	The system moves to safe mode and quits doing the non-crucial actions (see States chapter)
Timer Elapsed	When a timer for being in a state elapsed.	The system moves to the requested state
Test Button pressed	Ask the system to move to Test On ground state in order to test the system while it's not in space.	The system moves to TestOnGraound state – units are shut down except the requested units (see States chapter).
Start experiment & measurement	Triggered when there is a job to be done.	The system moves to Active mode
Done experiment	Triggered when there isn't a job to be done.	The system moves to Idle mode.

4.3 States Diagram:



4.4 States Description:

4.4.1 **LaunchModeState** – The initial state of the satellite, when the satellite is lunched the system Initialized to this state.

4.4.2 **TestOnGraound** - This is a unique state which is used when the satellite is on the ground in order to test the satellite and to develop the system. All hardware devices are turned off and there is an option to test each unit separately by turning one unit at a time and perform tests on it. There are some hardware devices which can be harmful for humans to be nearby – this state allows the developers to test the system without activating those devices even if it's necessary when the satellite is in space (decouple the units dependencies to test them). The system can move to this state only by manual command.

4.4.3 **OperationalState** - This state is the main state of the system, the system will be in this state most of its lifetime. In this state the system will perform its lifecycle tasks (see tasks under objects description in next chapter).
This state is divided into 2 sub states:

- **Idle** – Executing the regular lifecycle tasks, doing measurement and save them to check proper life of satellite. The first state when entering to OperationalState, works on idle if there is no commend or experiment to execute, and when there an commend in work queue go to Active state.
- **Active(regularOperation)** – Executing the same tasks and commands as **Idle** but also execute commend or experiment and when done go to Idle state.

4.4.4 **SafeState** – This is the rarest state of the system, the system will move to this state when the system is in critical condition and after launch mode, here are the conditions to move to safe state:

- The power battery is very low
- The temperature is extremely low
- The spin rate of the satellite is High.
- One of the hardware units crashed
- Manually asked from the ground station

The system will execute the following actions in safe state:

- Turn off all unneeded units that are not crucial for its survivability (camera, part of the thermal control, part of the sampling process)
- Turn the satellite towards the sun (to charge battery)
- Turn on heaters if needed.
- Perform crucial samples on the needed units
- Attitude controlling on the spin rate of satellite using B-DOT.

5. Object Oriented Analysis

5.1 Class Diagram:

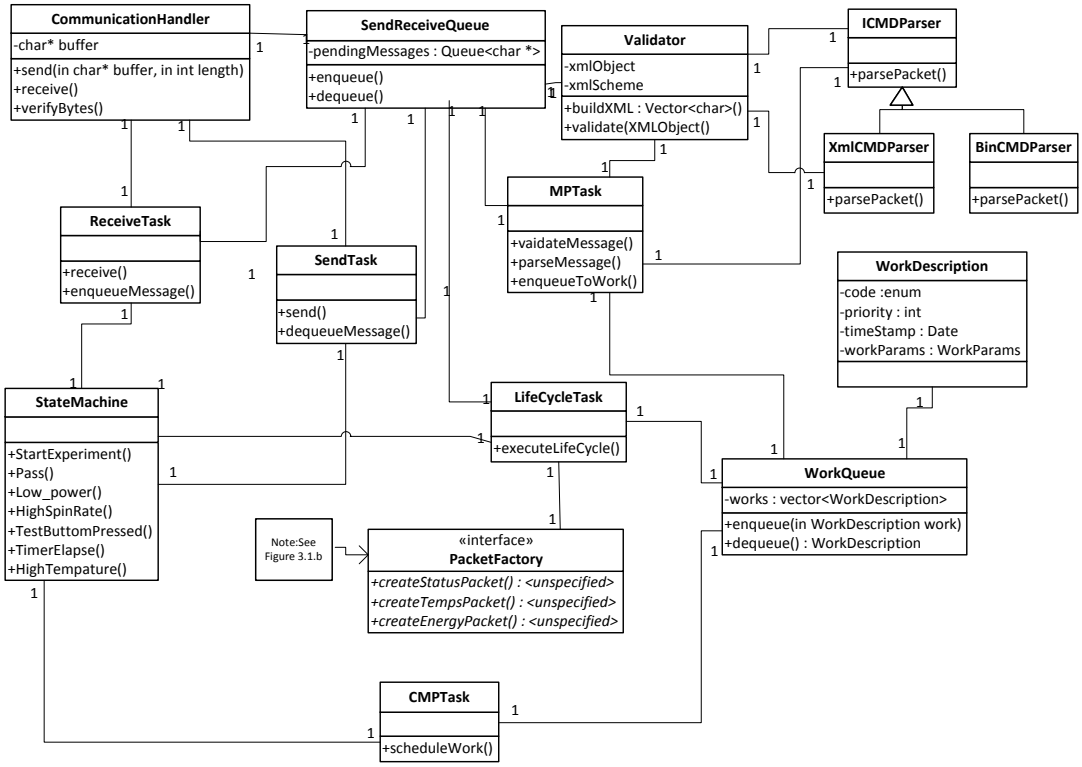


Figure 5.1.1 – Class Diagram

5.2 Class Description:

5.2.1 **Communication Handler** – This class is responsible for sending and receiving data from/to the satellite using UART protocol – this unit deals with bytes that arrive from the serial cable which is connected to the ground station.

a. **send**:

1. This method implements the UART protocol and sends the bytes from the satellite to the ground station.

2. Pre-conditions: serial cable is connected and the port is open in both sides

3. Post-conditions: packet was delivered on the other side (ground station)

b. **receive**:

1. This method implements the UART protocol and receives bytes to the satellite from the ground station.

2. Pre-conditions: serial cable is connected and the port is open in both sides.

3. Post-conditions: packet was received to the system

c. **verifyBytes**:

1. This method verifies that checksum is correct.

5.2.2 **SendReceiveQueue** – This class is responsible for storing the incoming/outgoing messages which supposed to be send/receive to/from the satellite, works as a queue in producer-consumer pattern.

a. **enqueue**:

1. Enqueues the data into the queue (producer uses this method)

2. Pre-conditions: The queue is not full

3. Post-conditions: data is stored inside the queue

b. **dequeue**:

1. Dequeues the data from the queue (consumer uses this method)

2. Pre-conditions: The queue is not empty

3. Post-conditions: data has been taken from the queue

5.2.3 **Validator** – This class is responsible for generating the xml packet which was received by the satellite only if the packet fits the data protocol between the ground station and the satellite – the main functionality is the validate the XML packet using the XML schema.

a. **buildXML**:

1. Generates xml file from the input bytes

2. Pre-conditions: Input bytes are correct and represent xml file

3. Post-conditions: XML file was created

b. **validate**:

1. Validates the correctness of the xml file by using the xml schema as a validation tool.

2. Pre-conditions: Syntax of XML file is correct, XML schema exists and correct (syntax)

3. Post-conditions: returns true if XML file fits the schema, false otherwise.

5.2.4 **ICMDParser** – Interface which represent the parser which the Satellite Core use and contain the method parsePacket this interface instantiate by XmlCMDParser or BinCMDParser which determine the receive protocol

5.2.5 **XmlCMDParser** – This is the parser which gets a packet (in XML form) which contains commands for the satellite and parses the packet into small commands (WorkDescription).

a. parsePacket(char *XmlPacket):

1. Parses the file into works (WorkDescription) – each command from ground becomes WorkDescription.

2. Pre-conditions: XML file fits the schema

3. Post-conditions: WorkDescritions[] is returned

5.2.6 **BinCMDParser** – This is the parser which gets a packet (in Binary form) which contains commands for the satellite and parses the packet into small commands (WorkDescription).
a. parsePacket(char *BinPacket):

1. Parses the file into works (WorkDescription) – each command from ground becomes WorkDescription.

2. Pre-conditions: Binary data fits the schema

3. Post-conditions: WorkDescritions[] is returned

5.2.7 **PacketFactory** this is interface is part of AbstractFactory Design Pattern
Its define 3 method which create the packets

5.2.8 **Packet** is the abstract datatypes which contains the packet data
a.toBytes()
1.Translate the packet into bytes

2. Pre-conditions: the Packet Object was initiate.

3. Post-conditions: The output bytes represent the packet depend on the actual object the packet is. xml packet will make xml type bytes and the binary packet will make binary packet depend on scheme

5.2.9 **WorkDescription** - Data object which contains all the required information describing a single work (in example: take a photo in a specific location), also contains WorkParams object which stores the parameters for the specific work.
5.2.10 **WorkQueue** – The queue which contains WorkDescriptions, the works which are pending to be executed by the satellite (came from ground station), this class is also used to store the works which are ready to be executed (their time has come and now should be executed).
a. sort():

1. Sorts the works (WorkDescription objects) by their time of execution from the closest time of execution to the farest.

2. Pre-conditions: There is at least one work in the queue

3. Post-conditions: works are sorted as described

5.2.11 **StateMachine** – This is the heart of the project, the state machine is implemented using the state design pattern (external MIT code).
Contains the states which the satellite can be in a specific time (see sections 4.3, 4.4) the system functionality depends on the state of the state machine and behaves differently.
This class holds all the Tasks which are running simultaneously and affects their execution in dependence of the system's state.

All methods of this class are the events which were described in the events section 4.2.
5.2.12 **Tasks** – This system is a real time system and therefore it is tasks based system (also event based), the tasks are created and executed in early stage of the system and "lives" all the time (all tasks work simultaneously) – each task has its responsibility and priority - RTEMS operating system handles their scheduling.

Tasks types:
5.2.9.1 **ReceiveTask** – This task is responsible for constantly "listening" to the communication channel and check if there is a data which arrives to the channel, reads the data (using the CommunicationHandler) and if the checksum code of the data holds, places it inside the receive queue.
5.2.9.2 **MPTask** – Also known as message parsing task – consumes the data which was stored in the received data queue, creates xml formatted packet (using the Validator) and validates the packet fits the data protocol and finally parses the packet into works (using the CMDParser) and puts the works in the work queue.
5.2.9.3 **CMDTask** – The command task, constantly checks the work queue for works (only works which were arrived from the ground station) which should be done "at this moment" – each work has the time that it should be done, this task checks if there are works which their time arrived it puts them into a special queue which the lifecycle task will consume the works from.
5.2.9.4 **SendTask** – This task takes the packets which are stored in the send queue (contains formatted packets) and sends the packets to the ground station (using the CommunicationHandler).
This task is performed only when the satellite is facing the ground station.
5.2.9.5 **LifeCycleTask** – This is the top priority task which is essential for the satellite health and correct activity, in this project this task is mainly implemented using stubs but eventually this task will be the main activity of the satellite, this task do the following logic:
1. Samples the control units (the crucial units like the power sensor) – implemented using stubs.

2. Performs control unit algorithmic

3. Executes control commands

4. Performs logic actions: checks the events and samples which were stored and consider moving to other state (like moving to safe mode if power

sample is really low).

5. Command execution: consume a command from the ready works queue (the works which supposed to be executed at this time) and execute the work (like taking a photo on earth)

6. Stores all results from samples and work executions on the send queue using the Packet.toBytes() (creates packet or appends to existing packet).

7. Monitoring: asserts the samples data are in the correct range (divides into crucial and non-crucial samples) and dispatches an event if the samples exceeds the normal range

8. Thermal control: turn on heat if needed (also stub)

5.3 Package Diagram:

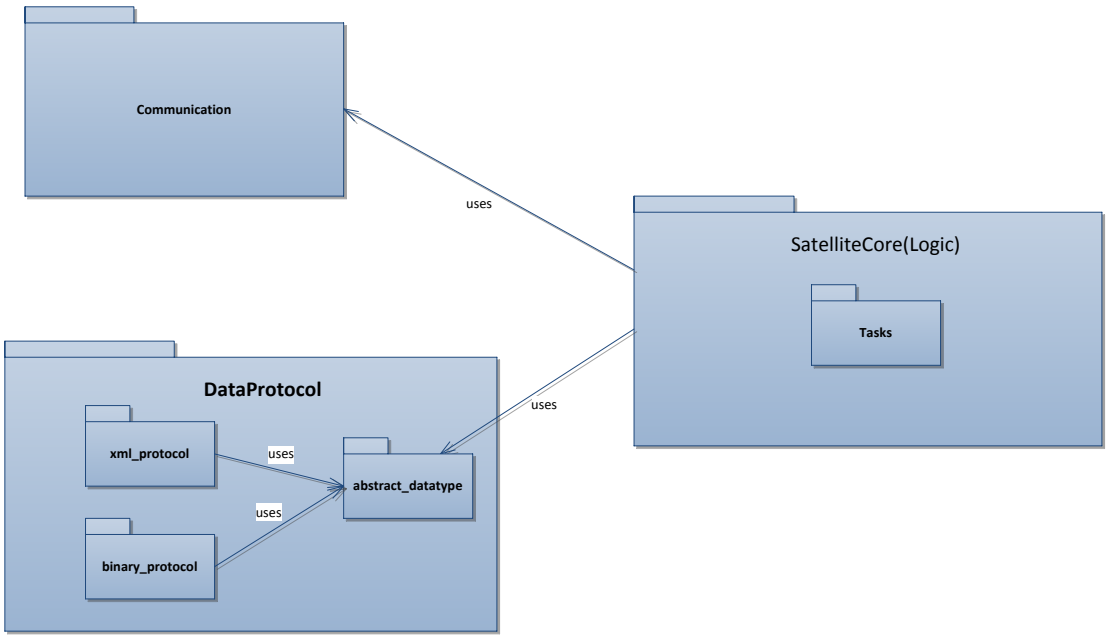


Figure 5.3.1 – Package Diagram

5.3.2 Packages Description

5.3.2.1 **Communication**:

1. **Purpose**: To store all the classes which responsible for communicating with the ground station

2. **Content**: CommunicationHandler, SendReceiveQueue

5.3.2.2 **DataProtocol**:

1. **Purpose**: Contains all the classes which responsible for dealing with the data protocol between the ground station and the system – encode and decode the messages.

2. **Content**: Validator, WorkDescription, WorkQueue,

5.3.2.3 **Abstract_datatypes**:

1. **Purpose**: Contains all the classes which the Satellite Core is using this classes which not depending on the protocol(bin or xml) this making the Satellite Core protocol independent

2. **Content**: Packet,PacketFactory ,EnergyPacket, StatusPacket, TempsPacket, ICMDParser

5.3.2.4 **Xml_protocol**:

1. **Purpose**: Contains all the classes which implements the xml protocol.

2. **Content**: XmlPacketFactory , XmlEnergyPacket, XmlStatusPacket, XmlTempsPacket, XmlCMDParser

5.3.2.5 **Bin_protocol**:

1. **Purpose**: Contains all the classes which implements the xml protocol.

2. **Content**: XmlPacketFactory , XmlEnergyPacket, XmlStatusPacket, XmlTempsPacket, XmlCMDParser

5.3.2.6 **Logic**:
1. **Purpose**: Contains classes which deal with the logic of the system – states and execution of tasks and commands.

2. **Content**: Tasks package, StateMachine

5.3.2.7 **Tasks**:
1. Purpose: To store all tasks objects.

2. Content: RecieveTask, SendTask, LifeCycleTask, MPTask, CMDTask

5.4 Unit Testing:

The following classes will not be tested as units because of their simplicity and lack of functionality (data objects or very simple objects):
1. SendReceiveQueue
2. WorkDescription
3. WorkParams
4. WorkQueue
5. Tasks – All tasks are runnable objects which simply use the rest of the objects (one can think of tasks as drivers to the rest of the classes) therefore the unit tests for the rest of the classes will basically test the tasks functionality.

<u>CommunicationHandler</u>:

Function	Description	inputs	Expected output
Send(char* buffer, int length)	Send some msg and check it received	Buffer = "some msg" Length = msg length	Send Success &All bytes received
vector<char> Receive()	Receive a msg from and see the msg arrived Successfully	None	Received Success & the msg arrived entirely
VerifyBytes(vector<char>	Test the verify	Bytes – some valid msg	Function return true

bytes)	mechanism for valid msg		
VerifyBytes(vector<char> bytes)	Test the verify mechanism for invalid msg	Bytes – some invalid msg	Function return false

Validator:

Function	Description	inputs	Expected output
buildPacket (vector<char>) (XML)	Build the XML object from the bytes.	Valid bytes of an xml	Success, XML object defined successfully
buildPacket (vector<char>) (XML)	Try to build XML object but fail because the bytes are invalid.	Invalid bytes	Function return false
validate(XMLObject obj)	Validate the correct XML using the schema	Valid XML object	Success, method returned true.
validate(XML file)	Validate the incorrect XML using the schema	Invalid XML	Success, method returned false.

Packets Tests:

Test the Funtions .toByte() method (see Data Model section)

XmlEnergyPacket,XmlStatusPacket,XmlTempsPacket

Function	Description	inputs	Expected output
toBytes() (xml)	Serialize the Packet Object to an xml which represented in ascii	Pre Conditions-the object was init by the constructors	The output is in valid xml with the correct format which represented as a ascii bytes

BinEnergyPacket BinStatusPacket BinTempsPacket

Function	Description	inputs	Expected output
toBytes() (Binary)	Serialize the Packet Object to Binary data.	Pre Conditions-the object was init by the constructors.	The output is a valid Binary data with the appropriate format.

BinCMDParser XmlCMDParser

Function	Description	inputs	Expected output
parsePacket (char* packet,int size)	Parse valid bytes received to WorkDescription abstract data type	Packet- a valid packet Size = packet size	Success, the WorkDescription received has the right values
parsePacket (char* packet,int size)	Parse invalid bytes received to WorkDescription abstract data type	Packet- an invalid packet Size = packet size	Failed.

State machine testing:

Unit tests will only test the transitions between states on each event and system tests will check the overall functionality on each state (see chapter 7)

Function	Description	inputs	Expected output
StartExperiment()	Trigger START_EXPIRIMENT_EVENT event to the state machine and assert machine moved to Operational state	Pre Condition-State machine is in Idle state	Success, machine is in Regular Operation state
Pass()	Trigger PASS_EVENT event to the state machine and assert machine moved to PassAndDownload state	State machine is in RegularOperations or Idle state	Success, machine is in PassAndDownload state
low_power()	Trigger low_power event to the state machine and assert machine moved to Safe state	Any State	Success, machine is in Safe state
High_spin_rate()	Trigger High_spin_rate event to the state machine and assert machine moved to Safe state	Any State	Success, machine is in Safe state

TestButtomPressed()	Trigger TestButtomPressed event to the state machine and assert machine moved to TestOnGround state	Any State	Success, machine is in TestOnGround state
HighTempature()	Trigger HighTempature event to the state machine and assert machine moved to Safe state	Any State	Success, machine is in Safe State
Timer elapse()	Trigger TimerElapse event to the state machine in Launch state and assert the state is switch to Safe Stat	Launch state	Success, machine is in Safe State
Timer elapse()	Trigger TimerElapse event to the state machine in RegularOperation state and assert the state is switch to Idle State	RegularOperation state	Success, machine is in Idle State
Timer elapse()	Trigger TimerElapse event to the state machine in PassAndDownload state and assert the state is switch to Idle State	PassAndDownload state	Success, machine is in Idle State

6. User Interface Draft

Not relevant to our project – no user interface at all.

7. Testing

7.1 Testing Functional Requirements

1. Unit Testing: Each unit is tested independently using Min Unit as described in section 5.

2. Integration Testing: Integrate the units of the system in this order:

1. CommunicationHandler + SendReceiveQueue + Validator
2. CommunicationHandler + SendReceiveQueue + Validator + CMDParser
3. StateMachine + Tasks (Using mock communication)
4. CommunicationHandler + SendReceiveQueue + Validator + CMDParser + StateMachine + Tasks

3. **System Testing:** We developed an Ground Station Simulator which simulate a ground station commands (both Binary and xml format) and add to it some test commands**.**

Load the system using TSIM (emulator which emulates the LEON3 architecture and RTEMS operating system) and using the Ground Station Simulator in order to perform the system Tests

Testing all use cases when the system is running.

7.2 Testing Non Functional Requirements

1. **Testing memory limit:** TSIM (the emulator which described above) can be configured to limit the memory to certain number, the system will be tested when the memory limit of the emulator is set to 2MB SRAM – this will indicate that this amount of memory is enough.

2. **Testing cycle time:** Perform time measurement when the cycle begins and ends, which will ensure the time requirements for a cycle (300 ms).