Requirements Specification Document

Team D

July 24, 2013

Table 1: Team

Name	ID Number
Stefanie Lavoie	1951750
Pinsonn Laverdure	9684352
Ghislain Ledoux	6376320
Rigil Malubay	6262732
Philippe Milot	9164111
Christopher Mukherjee	6291929

Contents

1 Introduction

This document will describe in detail the architecture specifications for the Vessel Monitoring System which is to be developed in the context of the COMP354 course. This design document include a detailed version of the architecture design pattern, a 4+1 Architectural View diagram (as well as a description of the rationale of this design), Module Interface Specifications for each subsystem, a complete description of the design of the system and of all its subsystems, full dynamic design scenarios of two substantial use cases, and a revised cost estimation and project schedule.

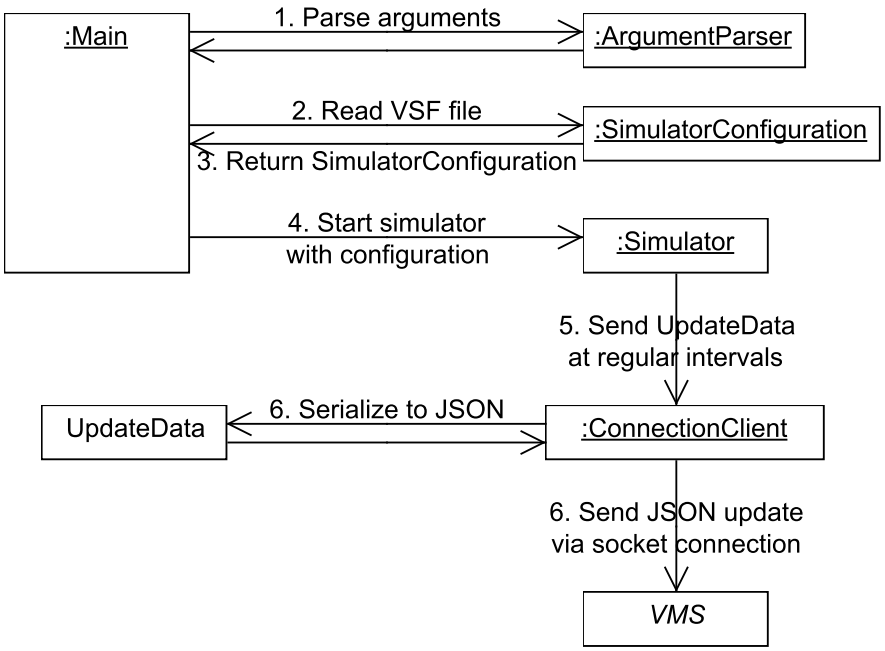
2 Architectural Design

2.1 Architecture Diagram

Previously in the program, we had the main method calling the information on a timer and sending the information to the GUI. This approach caused the entire system (including all subsystems) to execute synchronously, making the VMS dependent on the implementation details of the vessel simulator. We reworked the architecture of the program and now it is separated into several subsystems. All of these subsystems now work and involve independently from each other.

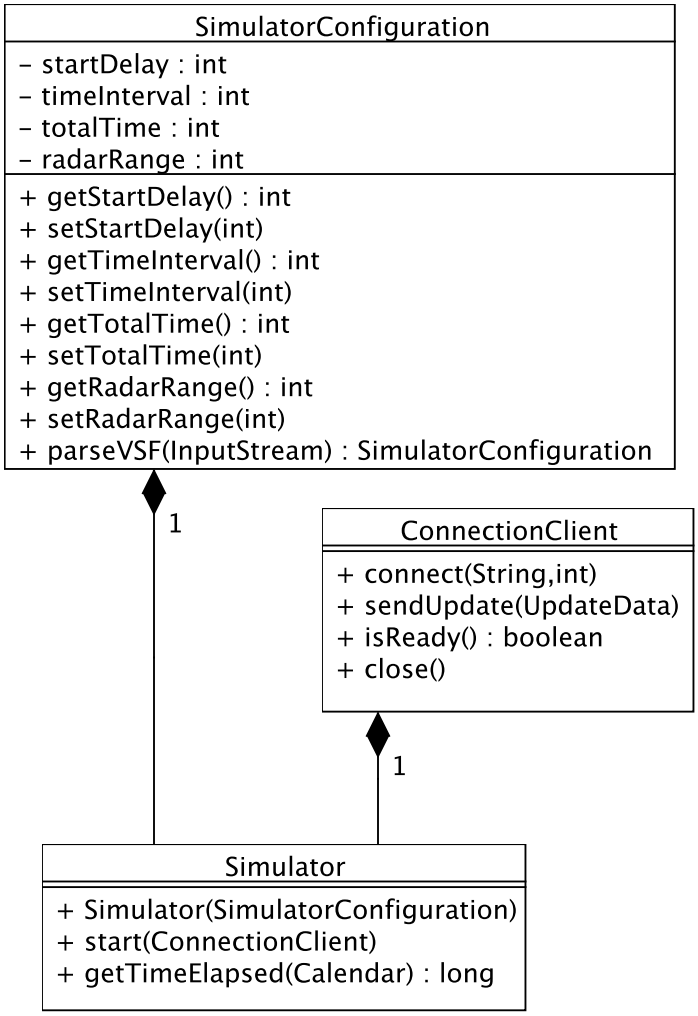
2.1.1 Logical View

Figure 1: Communication Diagram of the Simulator Subsystem



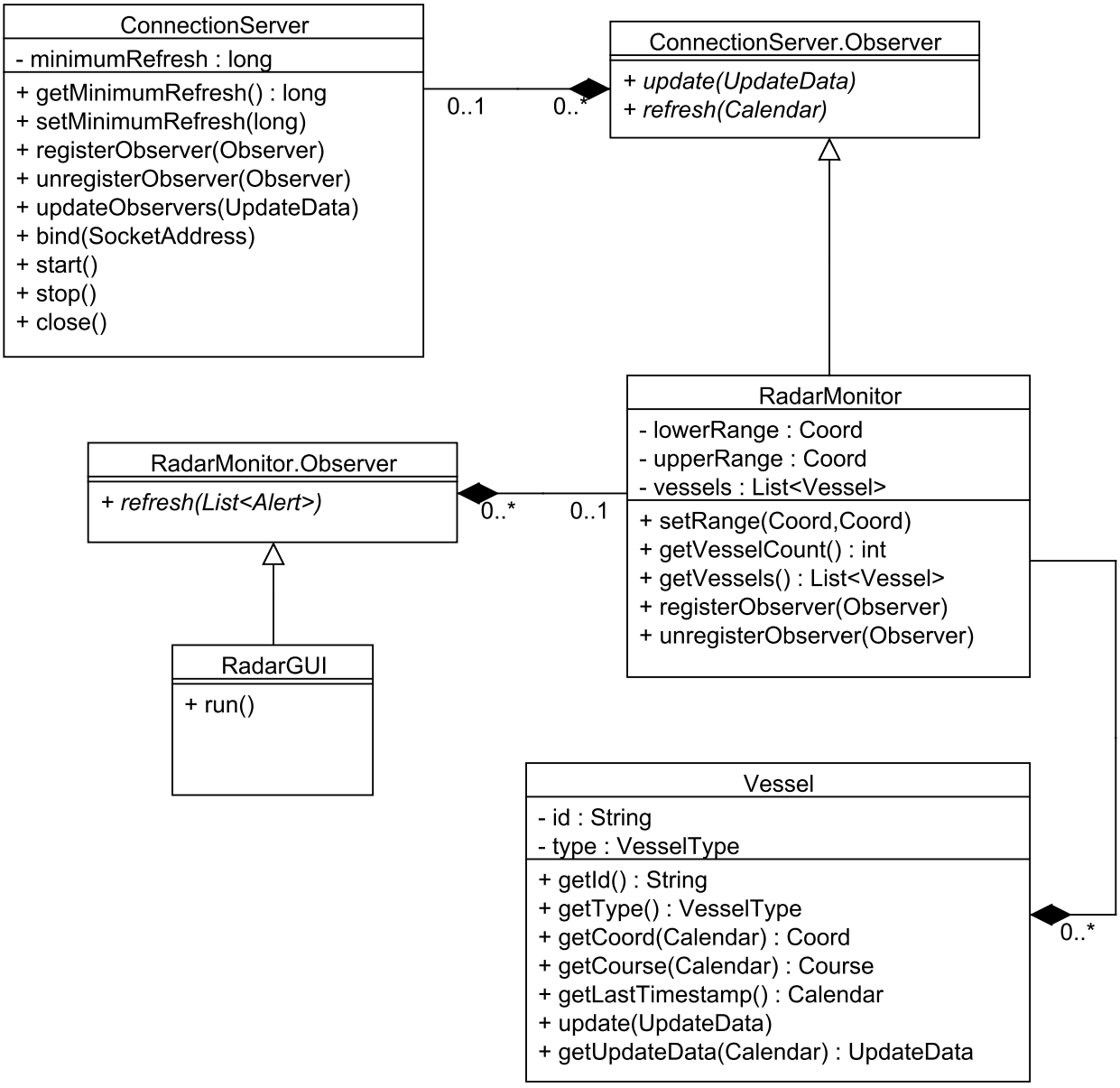
When the program is started, this subsystem will start and parse the arguments that are given to it. Following that it will read from the vsf file from which it will create a simulatorConfiguration instance. With the configuration created from the vsf file, it will start the simulator. At regular intervals, the simulator will generate an UpdateData from the VSF information and send it to the VMS server using the ConnectionClient class.

Figure 2: Class Diagram of the Simulator Subsystem



These are the classes involved in the configuration process of the creation of the simulator. As stated for Figure 1, simulatorConfiguration class will take on a vsf file for it to parse, with that it will ensure that the configuration is valid. The ConnectionClient abstracts away the details of sending an UpdateData over the network to the VMS. The Simulator aggregates these two classes in order to send UpdateData instances to the VMS at regular intervals.

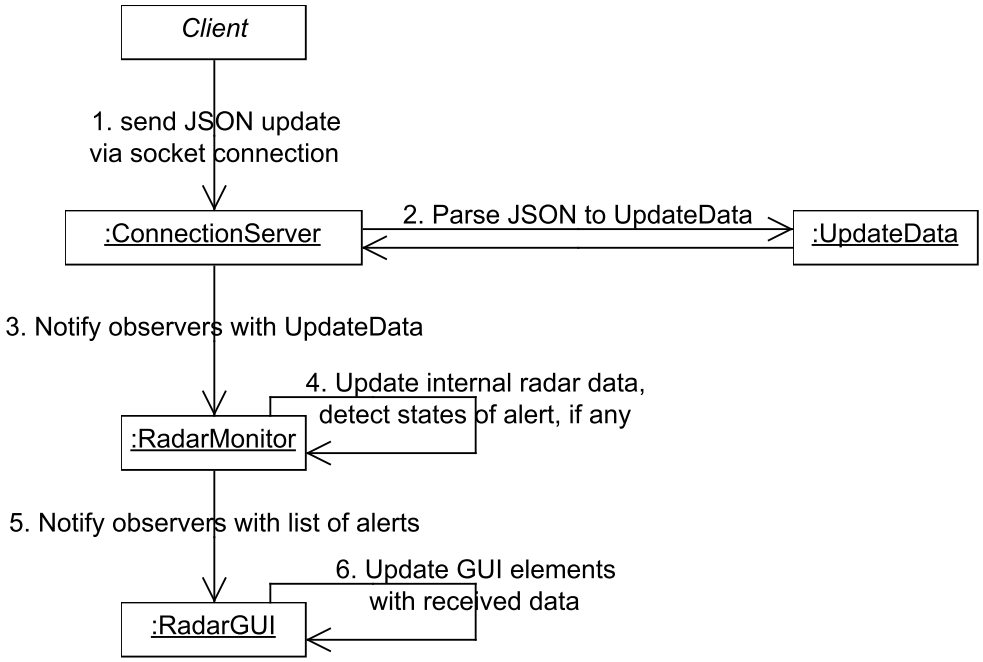
Figure 3: Class Diagram of the Vessel Monitoring System (VMS) Subsystem



These are the classes involved to update the data for the graphical interface. The RadarGUI class is where the user can see the output and is updated by a notification from RadarMonitor. The RadarMonitor keeps track of the state of vessels within its range. It issues alerts to its observers whenever they occur. It receives UpdateData from clients via notifications from the ConnectionServer class.

The ConnectionServer class abstracts away the implementation details of receiving an UpdateData from a client over the network; it does not care whether the client is a Simulator program, or a real vessel (both use the same network protocol and data format).

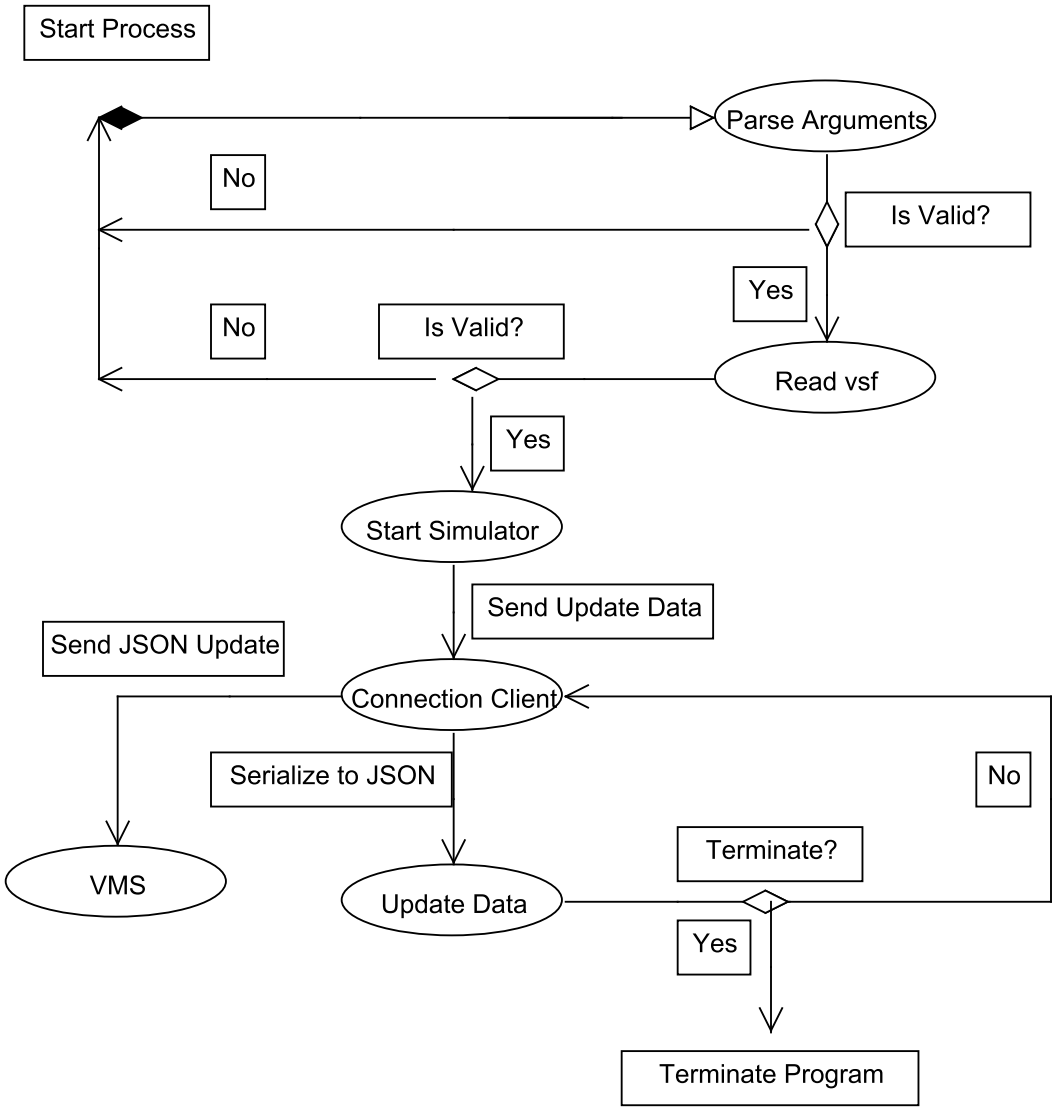
Figure 4: Communication Diagram of the Vessel Monitoring System (VMS) Subsystem



When the program is started, the ConnectionServer instance will bind to a specific address and listen for JSON updates from any connected clients. When an UpdateData is received and parsed successfully, it will notify its observers that the data has been changed. The main observer of ConnectionServer is the RadarMonitor instance. Inside the radar monitor it will search for any data that requires attention (e.g. alerts). If any need attention, it will notify the observers (RadarGUI) with the list of alerts, and they will be displayed to the user.

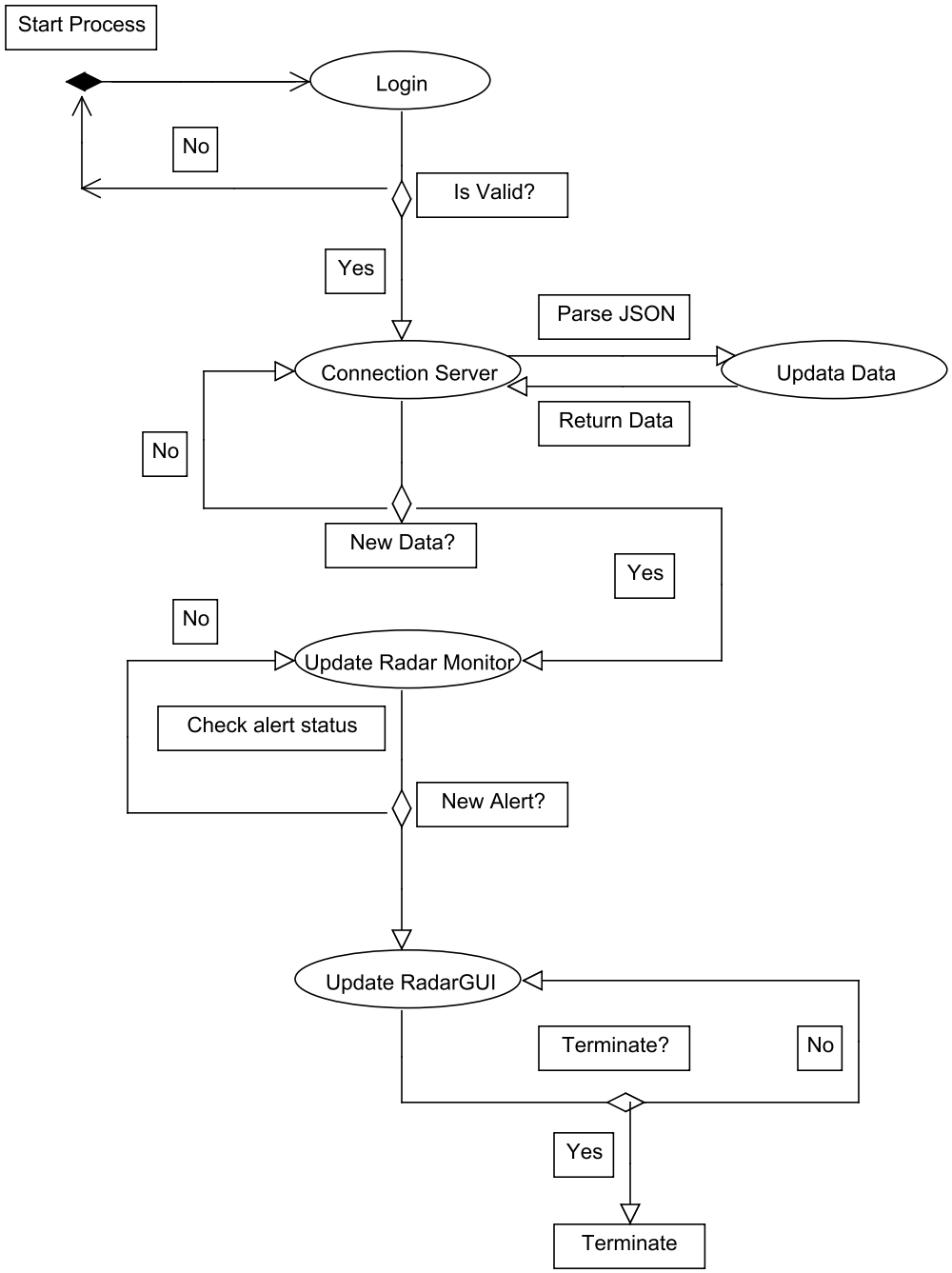
2.1.2 Process View

Figure 5: Activity Diagram for the Simulator Subsystem



The simulator has two main functionality: validating input from user and initializing/updating the data. The first step is to validate the parameters given from the user, it will check if the host, port and the .vsf file. If anything fails in step one the program will exit and wait for the user to input the next command line. Then step two happens, it will start the simulator, use the data received, and send it to the connection client. Connection client will serialize the data to update data and will keep looping around till the program terminates.

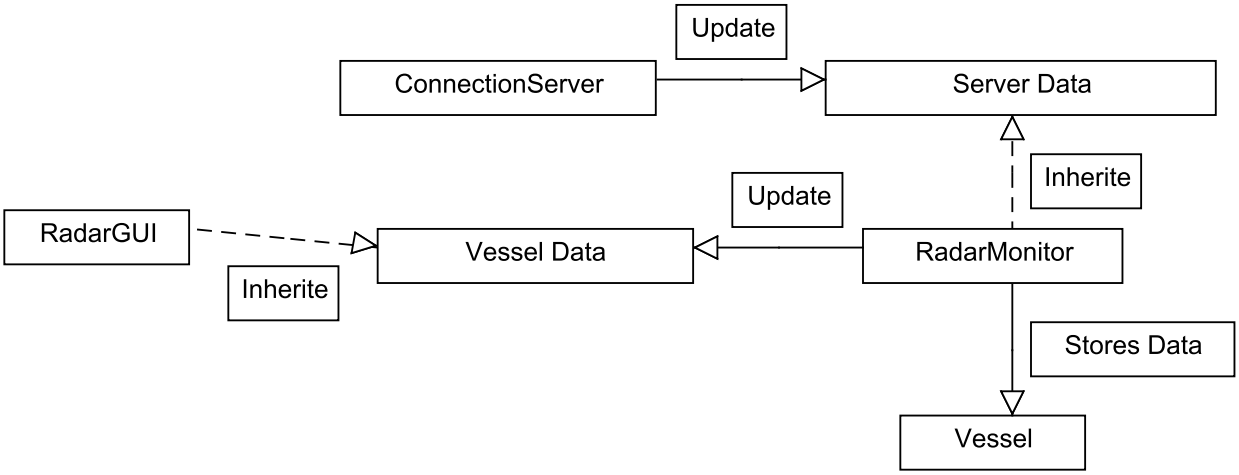
Figure 6: Activity Diagram for the Vessel Monitoring System (VMS)



The Vessel Monitoring System has two main functionality: connecting to server and updating the graphical user interface with the data. The first step is to connect to then server, in this step the user will be connected via socket to the server. If it fails it will simply return to the start. Then step two happens, the connection server will request data from update data. If new data was found it will send it to the radar monitor. Then when the radar monitor receives the new data it will check if any alert were triggered and update the graphical user interface.

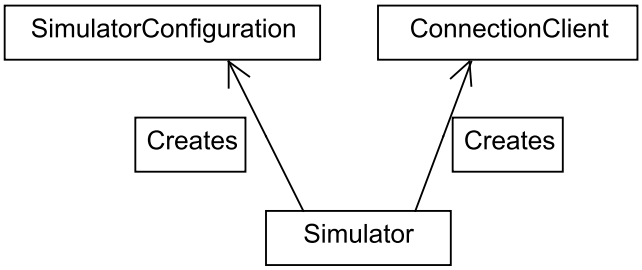
2.1.3 Development View

Figure 7: Component Diagram of the Vessel Monitoring System (VMS)



In the Vessel Monitoring System, the RadarMonitor is the core of this subsystem. The information of the server is recorded in RadarMonitor. It also updates the information that is sent to the user interface.

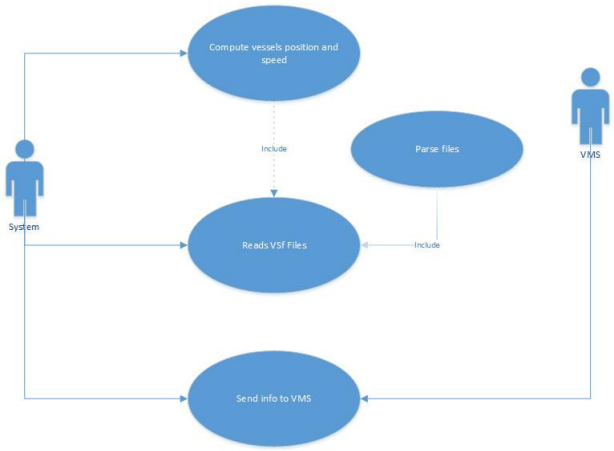
Figure 8: Component Diagram of the Simulator



In the Simulator subsystem, the simulator itself is the core. From the simulator, it goes ahead and create an instance of SimulatorConfiguration and ConnectionClient. With those in hand, it will be able to start the simulation.

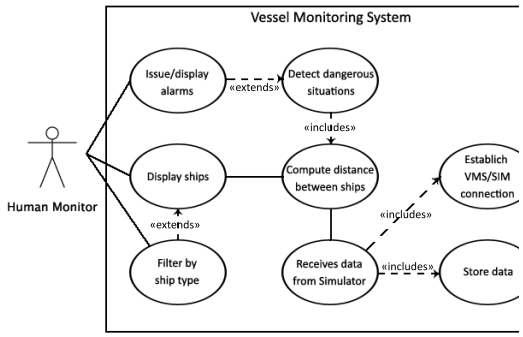
2.1.4 Scenarios

Figure 9: Use Case Diagram for the Simulator



The Simulator in general when started, the system will parse the settings and read the .vsf file. From there it will send it to the server where the Vessel Monitoring System can retrieve it.

Figure 10: Use Case Diagram for the Vessel Monitoring System (VMS)



The Vessel Monitoring System system is where the user can interact with the data. He can either look at the data or a subset of the data. The information is supplied by the Radar Simulator and sent to the VMS. Then with that data the VMS checks if any alerts are needed to be outputed to the user.

2.2 Subsystem Interfaces Specifications

2.2.1 Simulator subsystem

1. Parse command-line invocation from user:

   `"./Simulator --host 192.168.0.1 --port 1024 --input filename.vsf"`

   Note: `--host`, `--port` and `--input` can be replaced with `-h`, `-p`, `-i` respectively.
   The arguments must be entered in this order.

2. In the main method, it will be ensured that the command-line invocation is written as it is above, otherwise the program will exit.

3. When all cleared, `"SimulatorConfiguration.parseVSF(InputStream in)"` will return the configuration instance of the simulator.

4. Any files that are opened at this time will be closed to ensure no unnecessary streams are open.

5. `"ConnectionClient.connect(String host, int port)"` will take the validated host and port from the command-line and create a new socket for the user.

6. `"Simulator.start(ConnectionClient client)"` will take the client instance created in step 5 and use it to start the simulator.

2.2.2 VMS subsystem

1. Create a ConnectionServer instance

2. Create a RadarMonitor instance
3. Register RadarMonitor as an observer of ConnectionServer using: `"ConnectionServer.registerObserver(Observer o)"`
4. Run `"ConnectionServer.start()"`
5. Create vms.gui.Login instance, show the window
6. After user has logged in, create vms.gui.RadarDisplay instance
7. Register RadarDisplay instance as observer of RadarMonitor
8. Show RadarDisplay window

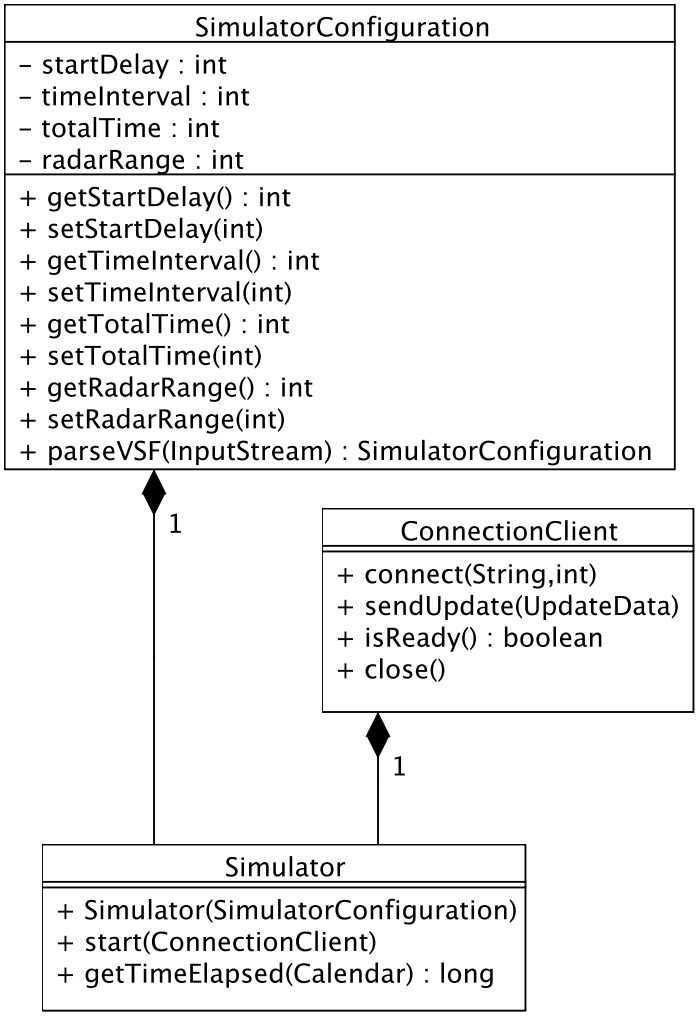
3 Detailed Design

In this section, we will describe in detail each subsystem of the software.

3.1 Subsystems

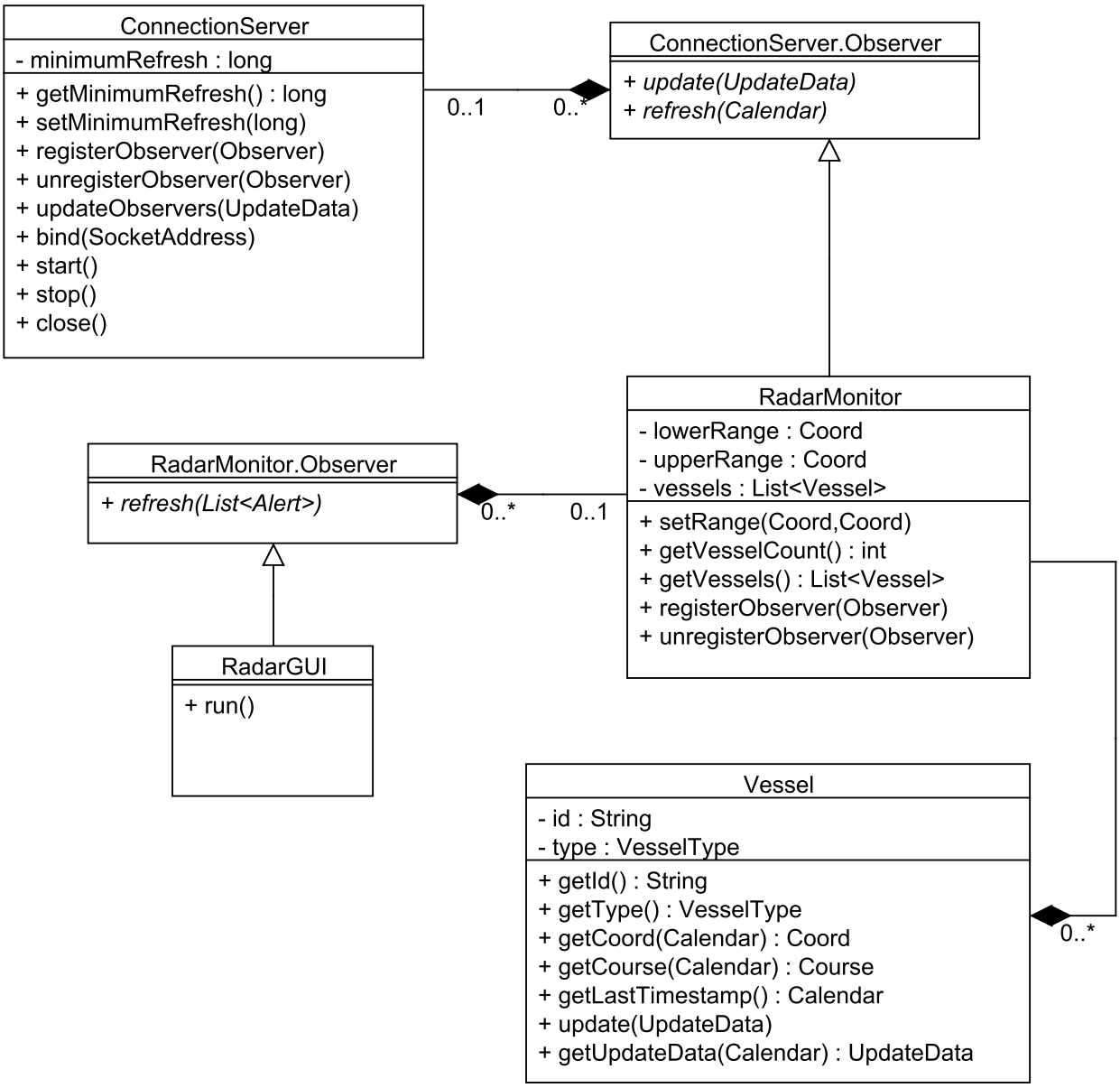
3.1.1 Detailed Design Diagram

Figure 11: Simulator Class Diagram



Simulator Subsystem The Simulator subsystem is the part of the software that will simulate the actual physical radar and the data it sends to the Vessel Monitoring System (VMS). A .vsf file, which lists the positions and courses of given vessels, as well as various configuration parameters for the VMS, is parsed and interpreted as vessel data by the simulator, which then establishes socket communication with the VMS. The simulator then proceeds to send updated data to the VMS at set intervals, until the pre-defined total time limit expires.

Figure 12: VMS Class Diagram



Vessel Monitoring System (VMS) Subsystem The Vessel Monitoring System (VMS) subsystem is the part of the software that will receive vessel data from the radar (represented by the Simulator) and interpret this data so that it can be viewed by the user via a graphical interface, and emit high- or low-risk alerts if any two vessels are within a certain range of each other. The VMS accepts socket connections from any number of Simulator instances.

3.1.2 Unit Descriptions

Simulator Subsystem

Class SimulatorConfiguration The main purpose of this class is to parse the .vsf file and to keep track of the data obtained, as well as registering any modifications to this data.

Attributes:

1. private int _StartDelay
2. private int _TimeInterval
3. private int _TotalTime
4. private int _RadarRange
5. List<Vessel> _Vessels

Functions:

1. private SimulatorConfiguration()
2. public int getStartDelay()
3. public void setStartDelay(int)
4. public int getTimeInterval()
5. public void setTimeInterval(int)
6. public int getTotalTime()
7. public void setTotalTime(int)
8. public int getRadarRange()
9. public void setRadarRange(int)
10. public List<Vessel> getVessels()
11. public void addVessel(Vessel)
12. public static SimulatorConfiguration parseVSF(InputStream)

Class ConnectionClient This class is responsible for establishing and managing the socket connection to the VMS, as well as sending all update data via this connection. The sent data is converted to JSON and encoded as Netstring.

Attributes:

1. private final int TIMEOUT
2. private Socket _Socket

Functions:

1. public ConnectionClient()
2. public void connect(String, int)
3. public void sendUpdate(UpdateData)
4. public boolean isReady()
5. public void close()

Class Simulator This class performs and determines the frequency of data updates and communications to the VMS. Once a connection to the VMS is established, the Simulator creates a thread for a set duration, during which the data from SimulatorConfiguration is updated and sent to VMS through the ConnectionClient at set intervals.

Attributes:

1. SimulatorConfiguration _Configuration

Functions:

1. public Simulator(SimulatorConfiguration)
2. public void start(ConnectionClient)
3. public long getTimeElapsed(Calendar)

Vessel Monitoring System (VMS) Subsystem

Class ConnectionServer The purpose of this class is to accept connections established by one or more radars, and receive the transmitted data. In addition, observers may be registered so that they will be notified of every incoming data update.

Attributes:

1. private static long DEFAULT_REFRESH
2. private boolean _Continue
3. private long _RefreshTime
4. private Selector _Selector
5. private ServerSocketChannel _Channel
6. private List<Observer> _Observers

Functions:

1. public ConnectionServer()
2. public void setMinimumRefresh(long)
3. public long getMinimumRefresh()
4. public void registerObserver(Observer)
5. public void unregisterObserver(Observer)
6. public void refreshObservers(Calendar)
7. public void updateObservers(UpdateData)
8. public void bind(SocketAddress)
9. public void setRadarRange(int)
10. public List<Vessel> getVessels()
11. public void addVessel(Vessel)
12. public static SimulatorConfiguration parseVSF(InputStream)

Class RadarMonitor (implements ConnectionServer.Observer) This class stores and updates any vessel data received by the ConnectionServer.

Attributes:

1. private Coord lowerRange
2. private Coord upperRange
3. private ArrayList<Vessel> _Vessels
4. private List<Observer> _Observers

Functions:

1. public void setRange(Coord, Coord)
2. public int getVesselCount()
3. public List<Vessel> getVessels()
4. public void registerObserver(Observer)
5. public void unregisterObserver(Observer)
6. public void update(UpdateData)
7. public void refresh(Calendar)

Class RadarDisplay (implements RadarMonitor.Observer) This class displays the data maintained by the RadarMonitor, using GUI components.

Attributes: (none)

Functions:

1. public RadarDisplay()
2. public void refresh(List<Alert>)

Class Vessel This class represents a vessel and all of its relevant characteristics.

Attributes:

1. private VesselType type
2. private String id
3. private Course course
4. private Coord coords
5. private Calendar lastTimestamp

Functions:

1. public Vessel(String, VesselType)
2. public String getId()
3. public VesselType getType()
4. public Coord getCoord(Calendar)
5. public Course getCourse(Calendar)
6. public Calendar getLastTimestamp()
7. public void update(Coord, Course, Calendar)
8. public void update(UpdateData)
9. public UpdateData getUpdateData(Calendar)

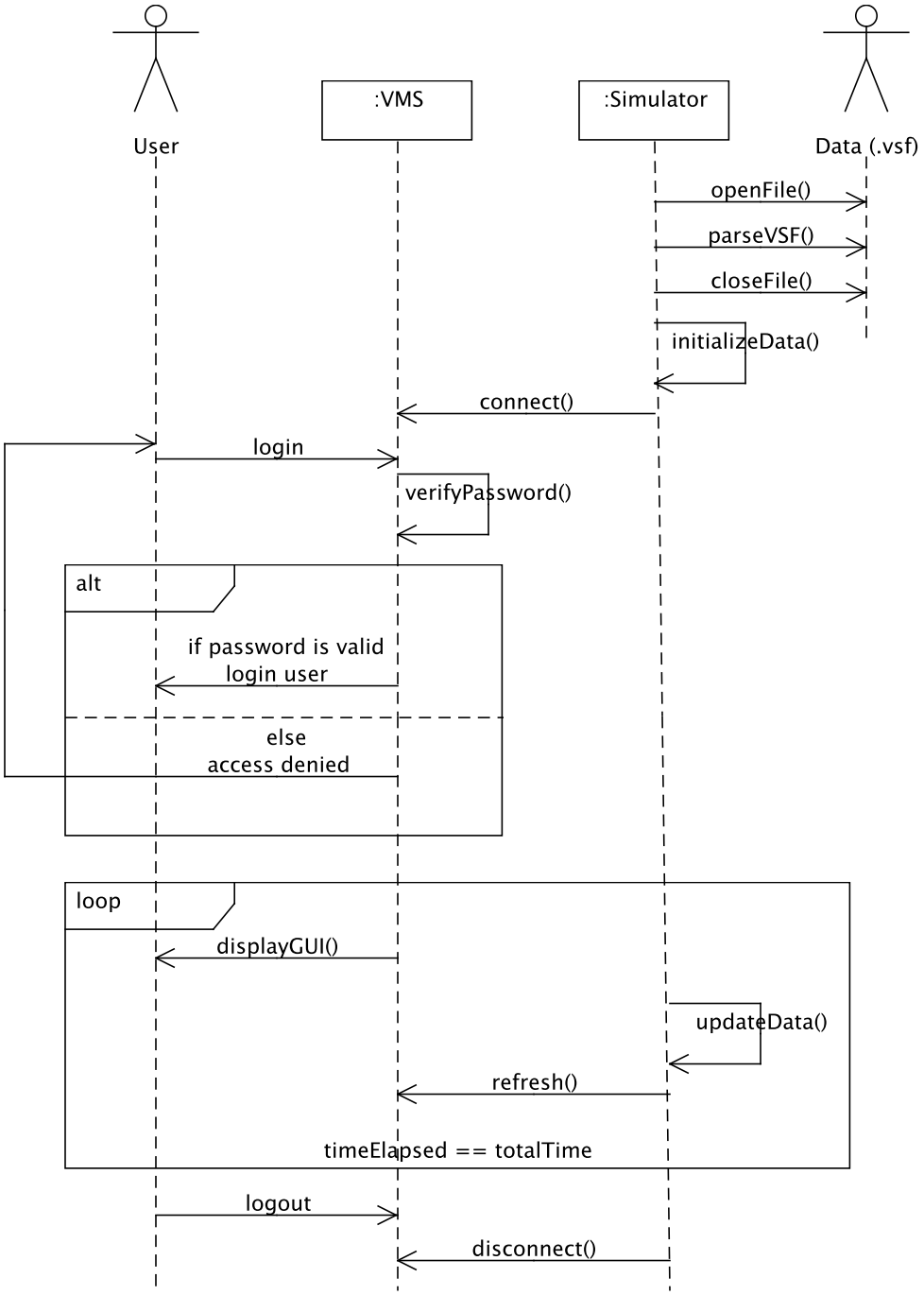
4 Dynamic Design Scenarios

In this section, we introduce two possible scenarios (use cases), which show interactions between the software and external users. In addition, we show how components of the software interact, within each subsystem.

4.1 Use Case Scenarios

Here, we present the interactions between the software and external users.

Figure 13: Sequence Diagram - User



Scenario 1: User This scenario represents the interactions between the software and a user with normal access rights ("User").

Operational Contracts for Scenario 1:

Operation: openFile() Pre-conditions: valid file name and path were provided.
Post-conditions: stream to .vsf file is created.

Operation: parseVSF() Pre-conditions: .vsf file was correctly opened, and stream was created.
Post-conditions: text from .vsf file is read by the system.

Operation: closeFile() Pre-conditions: .vsf file was opened.
Post-conditions: none.

Operation: initializeData() Pre-conditions: all data parsed from .vsf file were of correct value type and range.
Post-conditions: text from .vsf file is converted to vessel and configuration data.

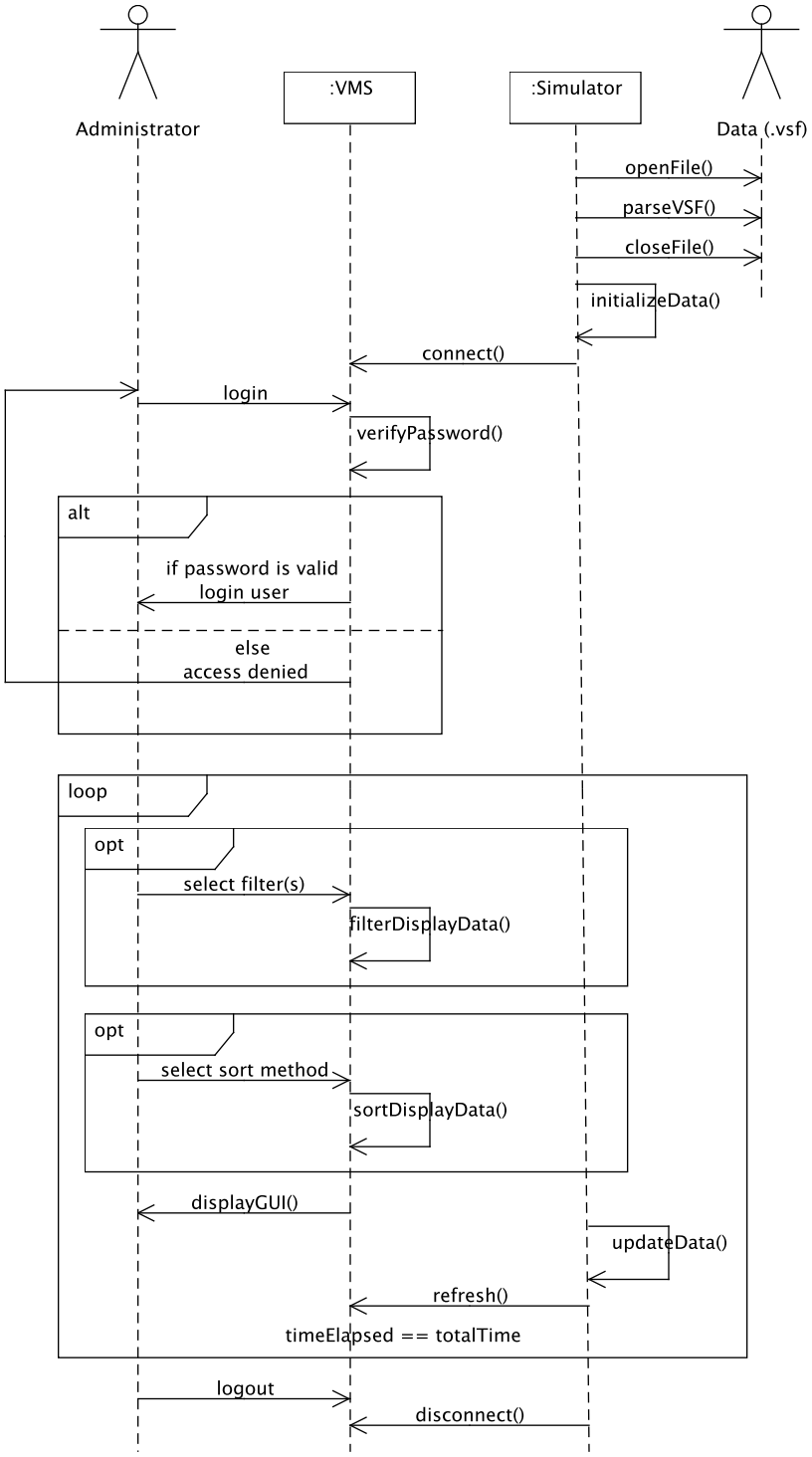
Operation: verifyPassword() Pre-conditions: user has entered a password.
Post-conditions:
- If password is recognized as valid: GUI display is created.
- If password is not recognized as valid: none.

Operation: displayGUI() Pre-conditions: none.
Post-conditions: any vessel data from the system is sent to GUI for display.

Operation: updateData() Pre-conditions: none.
Post-conditions: system vessel data is modified based on calculated time factor.

Operation: refresh() Pre-conditions: none.
Post-conditions: GUI-displayed vessel data corresponds to system vessel data.

Figure 14: Sequence Diagram - Administrator



Scenario 2: Administrator This scenario represents the interactions between the software and a user with full access rights ("Administrator").

Operational Contracts for Scenario 2: Most of the contracts for this scenario are the same as for Scenario 1.

Additional contracts:

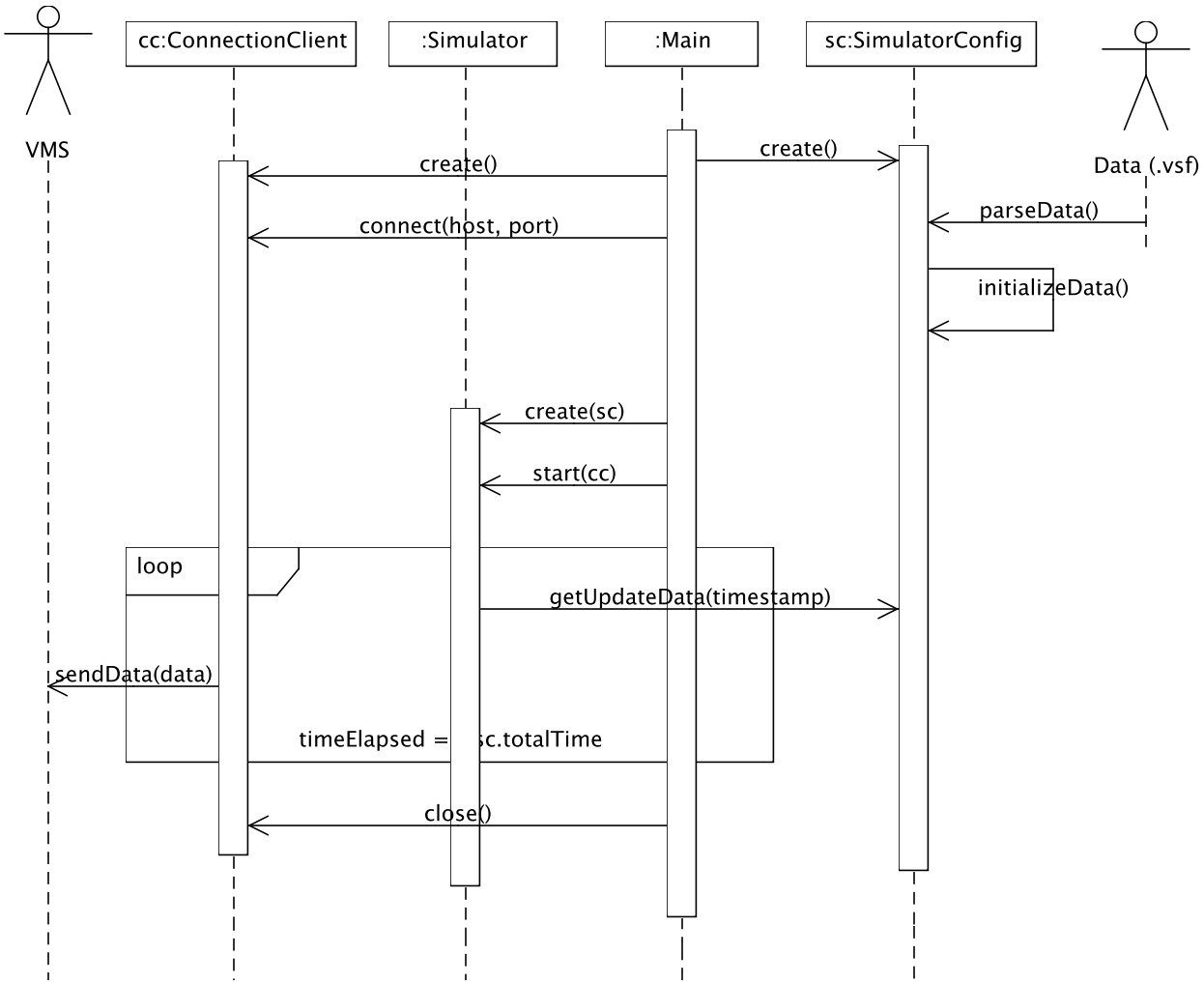
Operation: filterDisplayData() Pre-conditions: none.
Post-conditions: GUI display receives only vessel data allowed by the filter.

Operation: sortDisplayData() Pre-conditions: none.
Post-conditions: GUI displays vessel data based on order specified by the selected sorter.

4.2 Component Interactions

Here, we present in more detail the interactions between the system components, which were not shown in the above use case scenarios. We have now abstracted any interactions with external users.

Figure 15: Sequence Diagram - Simulator



Simulator Subsystem

Operational Contracts for Simulator Subsystem: Operation: create() (target: SimulatorConfiguration) Pre-conditions: none. Post-conditions: instance of SimulatorConfiguration is created.

Operation: create() (target: ConnectionClient) Pre-conditions: none. Post-conditions: instance of ConnectionClient is created.

Operation: connect(String host, int port) Pre-conditions: parameter values form a resolvable hostname + port number pair. Post-conditions: attempt is made to connect to the server.

Operation: create(SimulatorConfiguration sc) (target: Simulator) Pre-conditions: none. Post-conditions: instance of Simulator is created.

Operation: start(ConnectionClient cc) Pre-conditions: connection to server was established. Post-conditions: vessel data is sent to server at set intervals for a given duration, determined by configuration data.

Operation: getUpdateData(Calendar timestamp) Pre-conditions: parameter value represents a valid timestamp. Post-conditions: updated vessel data is returned.

Operation: sendUpdate(UpdateData data) Pre-conditions: none. Post-conditions: updated vessel data is sent to server.

Operation: close() Pre-conditions: none. Post-conditions: Socket is closed.

Vessel Monitoring System (VMS) Subsystem

Operational Contracts for VMS Subsystem: Operation: create() (target: ConnectionServer) Pre-conditions: none. Post-conditions: instance of ConnectionServer is created.

Operation: create() (target: RadarMonitor) Pre-conditions: none. Post-conditions: instance of RadarMonitor is created.

Operation: register(RadarMonitor rm) Pre-conditions: none. Post-conditions: passed RadarMonitor is registered as an observer to ConnectionClient.

Operation: start() Pre-conditions: Selector is bound. Post-conditions: incoming client connections can be accepted.

Operation: create() (target: Login) Pre-conditions: none. Post-conditions: instance of Login is created.

Operation: create() (target: RadarDisplay) Pre-conditions: none. Post-conditions: instance of RadarDisplay is created.

Operation: register(RadarDisplay rd) Pre-conditions: none. Post-conditions: passed RadarDisplay is registered as an observer to RadarMonitor.

Operation: _read(SocketChannel sc) Pre-conditions: - passed SocketChannel is open and connected. - data from client was formatted and encoded correctly. Post-conditions: data from client is interpreted as vessel data, and all observers are notified.

Operation: update(UpdateData data) Pre-conditions: none. Post-conditions: data from client replaces current vessel data.

Operation: refresh() Pre-conditions: none. Post-conditions: GUI-displayed vessel data corresponds to system vessel data.

Operation: close() Pre-conditions: none. Post-conditions: SocketChannel and Selector are closed.

5 Revised Cost Estimation

5.1 Cost Estimates

Simulator program: $150, cost to code and implement.

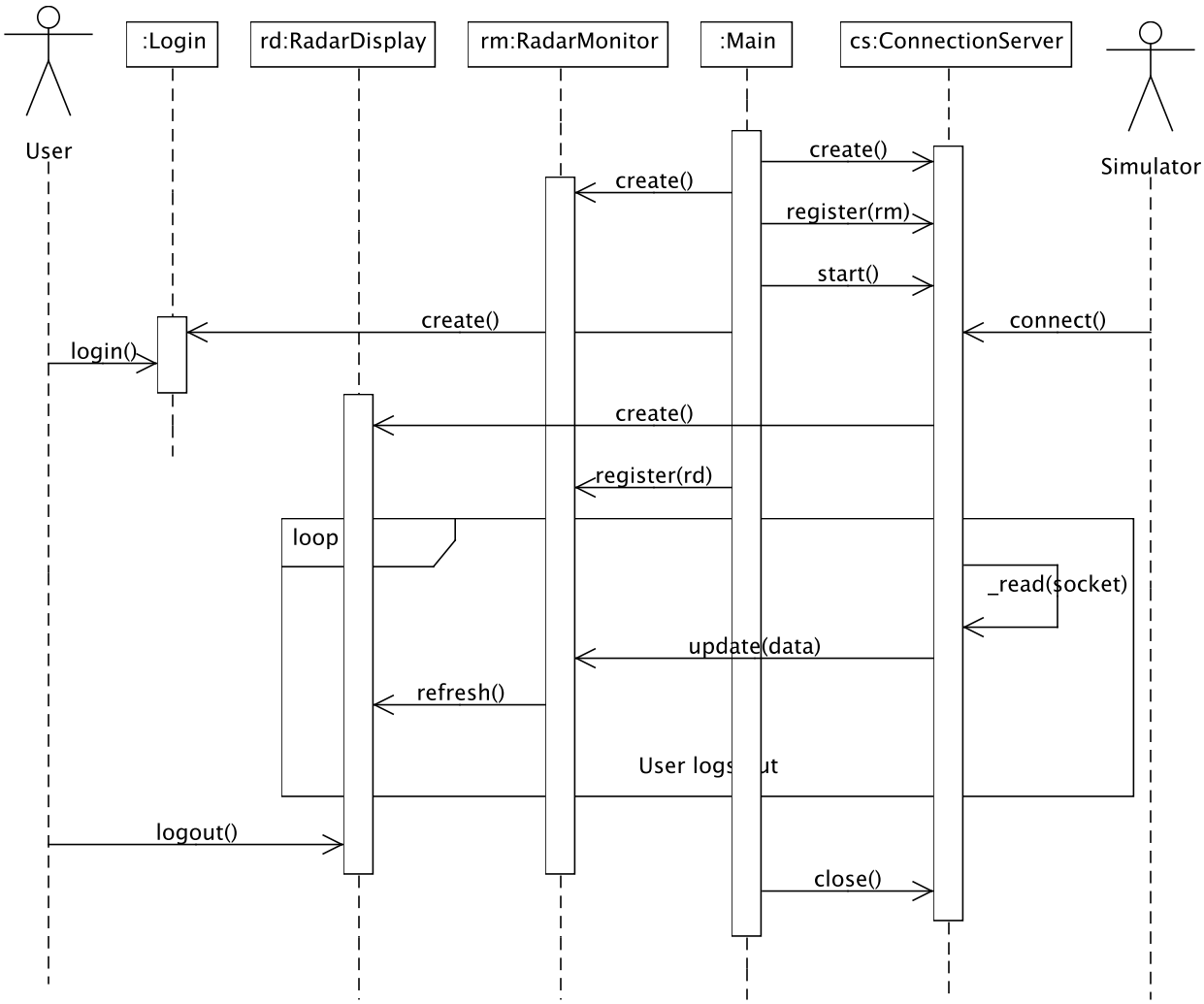
VMS program: $150, cost to code and implement.

Graphical User Interface: $100, cost to design, code, and implement.

Unit test suite: $25, cost to write test cases and perform testing.

Total Cost: $425

Figure 16: Sequence Diagram - VMS



Basis Costs have been calculated based on the amount of man hours spent working on project (at a low approximate labor cost of $5 an hour), as well as a reduced cost of tools used to produce code, test cases, documentation, and diagrams. Re-estimation of costs may occur when any of the above artifacts is complete, based on whether the artifact took less time or more time than we originally anticipated.

5.2 Project Schedule

Simulator program: 8 hours of coding. Due date: August 1st, 2013.

VMS program: 8 hours of coding. Due date: August 1st, 2013.

Graphical User Interface: 4 hours of coding. Due date: August 7th, 2013.

Unit test suite: 2 hour of testing. Due date: August 9th, 2013.

Final integration: 4 hours of implementation. Due date: August 12th, 2013.

Note The times listed to complete the Simulator program, VMS program, and Graphical User Interface are estimations of how much time the *remainder* of the coding will take, not accounting for the time that has already been spent on the programs as they currently are.

5.3 Risk

All due dates have been calculated with a buffer zone in mind to account for lack of availability depending on course load. Still, course load is difficult to predict, therefore there is a risk that some of the team members will not be able to produce their assigned artifact in time.

Also, writing test cases in parallel with the rest of the software may slightly delay production, although the reliability of the code will be improved.

All cost estimates and due dates are subject to change.

List of Figures