Patient Simulator

DPSIMU2

Project documentation

Requirement specification
Analysis and design
Implementation
Test

”Teknisk IT” – autumn 2004 – 10.12.2004

Project by group DPSIMU2:

Henrik Jensen
20022464

Almir Mesanovic
20023894

Jan Lauritzen
20034439

Mads Pedersen
20034440

TABLE OF CONTENTS

WORD LIST **4**

READING GUIDE **4**

1 REQUIREMENT SPECIFICATION **5**

1.1 INTRODUCTION 5
1.2 OVERALL DESCRIPTION 6
1.3 SPECIFIC REQUIREMENTS (USE CASES) 11
1.4 EXTERNAL INTERFACE REQUIREMENTS 16
1.5 PERFORMANCE REQUIREMENTS 17
1.6 SYSTEM QUALITIES 17
1.7 DESIGN CONSTRAINTS 17
1.8 PART DELIVERIES 17

2 SYSTEM ARCHITECTURE **18**

2.1 INTRODUCTION 18
2.2 SYSTEM OVERVIEW 20
2.3 SYSTEM INTERFACES 20
2.4 USE CASE VIEW 21
2.5 LOGICAL VIEW 22
2.6 PROCESS/TASK VIEW 31
2.7 DEPLOYMENT VIEW 33
2.8 IMPLEMENTATION VIEW 35
2.9 GENERAL DESIGN DECISIONS 37
2.10 COMPILATION AND LINKING 38
2.11 INSTALLATION AND EXECUTING 40

3 IMPLEMENTATION **41**

3.1 INTRODUCTION 41
3.2 DESIGN PATTERNS 41
3.3 DISTRIBUTED COMMUNICATION 48
3.4 CODE 48

4 SYSTEM TESTING **49**

4.1 INTRODUCTION 49
4.2 CODE WALKTHROUGHS 49
4.3 INTEGRATION TESTING 49
4.4 VERIFICATION 49
4.5 FINAL GRAPHICAL USER INTERFACES 51

5 REFERENCES **53**

TABLE OF FIGURES

Figure 1: Basic system overview 6
Figure 2: Distributed system overview 7
Figure 3: Actor-Context Diagram 7
Figure 4: Use Case Diagram 9
Figure 5: GUI prototype 16
Figure 6: Quality Factors 17
Figure 7: System context 20
Figure 8: Overall package diagram 22
Figure 9: The platform package 23
Figure 10: The communication package 24
Figure 11: The CSIMU package 26
Figure 12: The PSIMU package 27
Figure 13: Sequence for creating connections on PSIMU 28
Figure 14: Sequence diagram for handling PSIMU connections on CSIMU 29
Figure 15: Sequence to set patient values 29
Figure 16: Sequence diagram of setting heatbeat factor 30
Figure 17: Sequence for sending and receiving log messages 30
Figure 18: Task overview 31
Figure 19: Process communication 32
Figure 20: Simulator deployment 33
Figure 21: Deployment of CSIMU test 34
Figure 22: Deployment of PSIMU test 34
Figure 23: Patient simulator component diagram 35
Figure 24: The Central Simulator Control 35
Figure 25: Reactor pattern 42
Figure 26: Acceptor/Connector pattern 43
Figure 27: Leader/Followers pattern 45
Figure 28: State of threads in the ThreadPool 46
Figure 29: GUI on PSIMU 52
Figure 30: GUI on CSIMU 52

Word list

Word/abbreviation	Description
DPSIMU	Distributed Patient Simulator System
PSIMU	Patient Simulator System
LMON	Local Monitor System
IPUMP	Infusion Pump System
ECG	Electro Cardio Gram
EDR	ECG-Derived Respiration
PhysioBank Archives	PhysioBank is a large archive of digital recordings of physiological signals (http://www.physionet.org/physiobank/)

Reading Guide

The distributed system is an extension to the stand-alone system developed in the course "TI-IRTS". The documentation for the stand-alone system could relevantly be read at first [Ref. 5].

This document has been constructed starting with the requirement specification that specifies all requirements of the distributed Patient Simulator System.
Following after this is the overall system architecture and design followed by the implementation details. The closing item is about testing the system.

1 Requirement Specification

1.1 Introduction

1.1.1 Purpose

The basic purpose of a patient simulator system is:

The main purpose of the Patient Simulator System is to simulate different patient signals (ECG, EDR and pulse). These signals are monitored by a local monitoring system. The patient signals can be regulated according to medicine infused by an infusion pump.

For control purpose, the system is equipped with a monitor to display the patient signals and information from the infusion pump system. From this monitor it is possible to select different patient signals.

The simulating data are fetched from the PhysioBank Archives.

The distributed system extends the system in the following way:

The patient simulator is now a client to a central server. The purpose of the central server is remotely to be able to control each patient simulator client. The content of this control is to select simulating values (by selecting a specific patient simulation) and to select the heartbeat factor. Furthermore the patient simulator can send log messages to the central server. These log messages are shown on the server screen.

This document extends the previous design documentation [Ref. 5], for the stand-alone version of the patient simulator and as such, details from the previous design are only present were necessary.

1.1.2 References

- Project documentation for the stand-alone PSIMU2 system from the course “TI-IRTS” [Ref. 5]

1.2 Overall description

1.2.1 Basic system description

The purpose of this project is to implement a prototype of a simulation system (called PSIMU) for a human patient that is capable of simulating patient life signals.

An external monitoring system will be able to connect to the output from the simulator that reproduces real-time signal data on analogue and digital output ports.
This system will be able to simulate analogue Electro Cardio Gram (ECG), ECG-Derived Respiration (EDR) signals and a digital pulse. Furthermore it can receive input from an external infusion pump about which, and how much, medicine is being infused, and from this information regulate the outputs.

The simulator will be able to handle data from the PhysioBank databases.

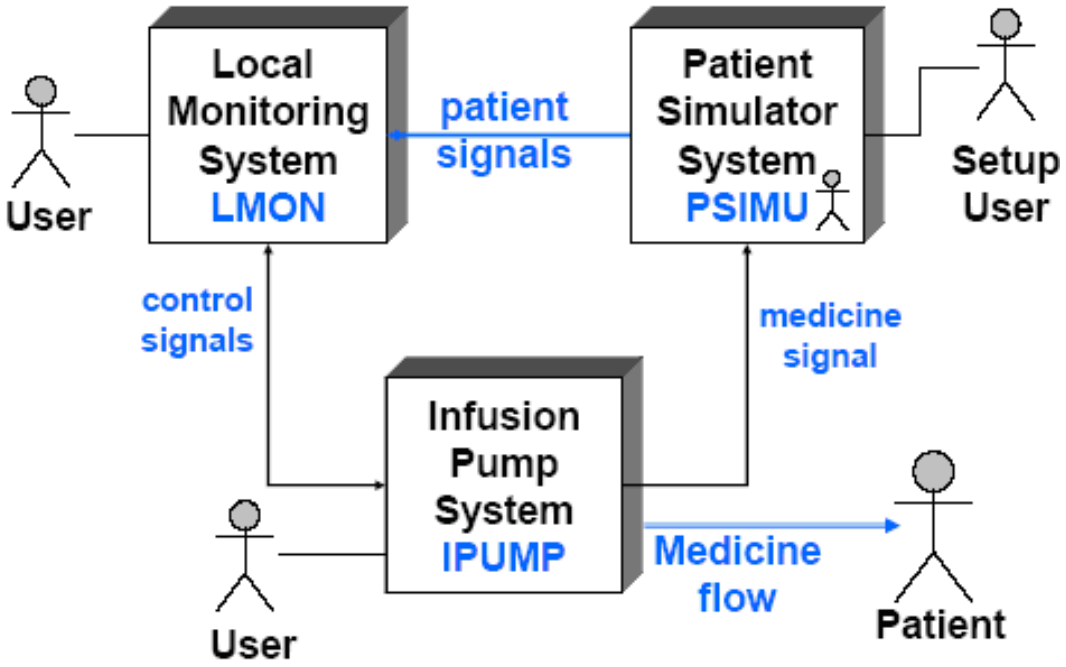

Figure 1: Basic system overview

The patient simulator is capable of interfacing to external systems for stimuli, such as a medicine infusion pump, adjusting its output of life signals accordingly. This can be seen on Figure 1.

1.2.2 Distributed system description

The above mentioned PSIMUs are connected to a central simulator control (called CSIMU) shown on Figure 2.
The CSIMU is able to control the connected PSIMUs in the context of selecting simulating values and the heartbeat-rate.

These can be regulated by a remote user on the CSIMU.

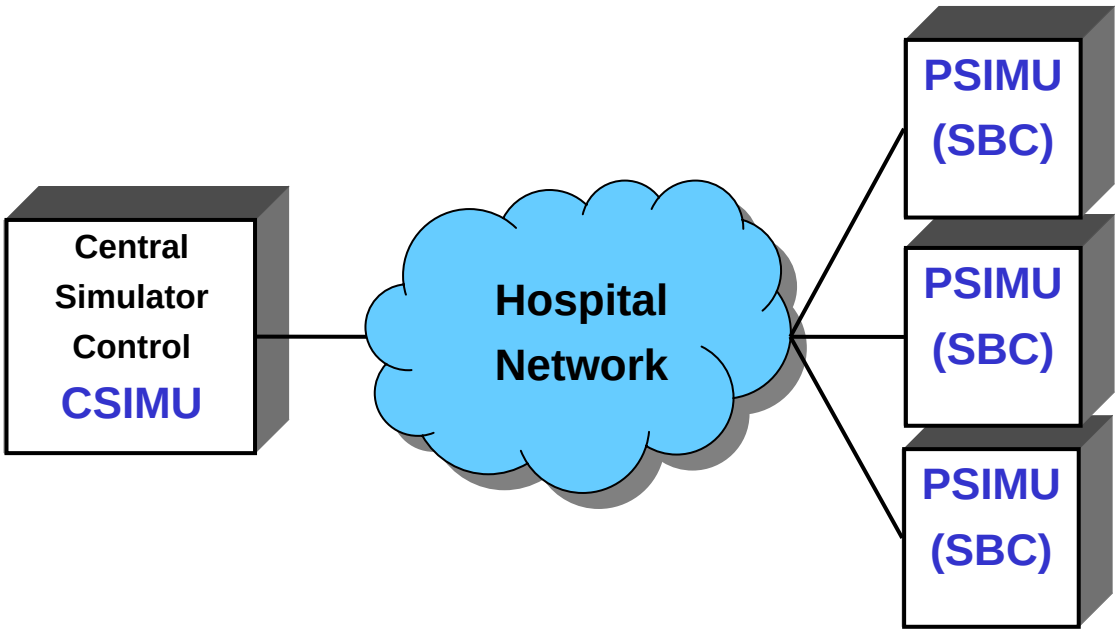

Figure 2: Distributed system overview

1.2.2.1 Actor-Context Diagram

The following diagram (Figure 3) shows the actors and their context of both the PSIMU and CSIMU systems.

Notice that here only the distributed parts of the systems are shown. The detailed documentation from the IRTS project is still valid.

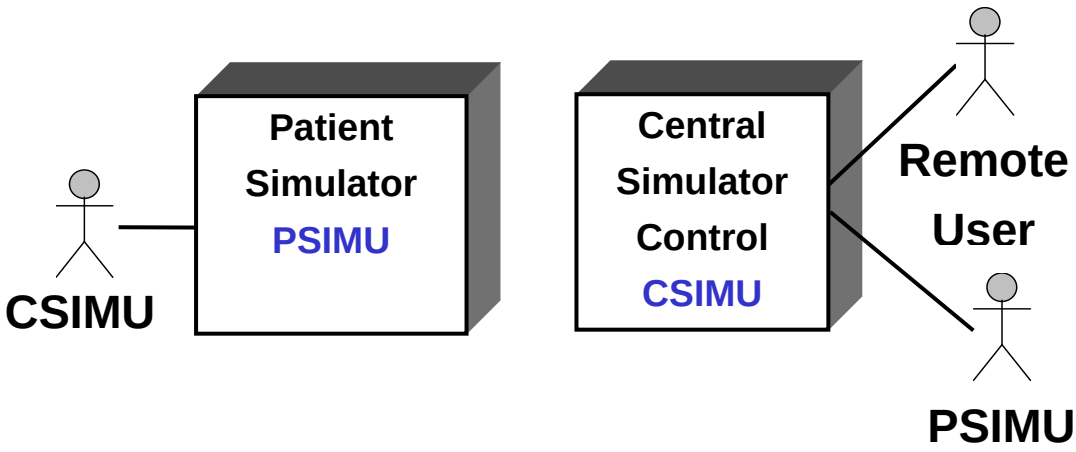

Figure 3: Actor-Context Diagram

1.2.2.2 Actor Descriptions

Actor name	**CSIMU**
Type [primary/secondary]	Primary
Description	A CSIMU can be connected to a PSIMU allowing CSIMU to control PSIMU in context of setting the simulator values and the heartbeat factor. The PSIMU can the other way send log messages to be displayed on CSIMU.
Number of concurrent actors	There is at most one central simulator control (CSIMU) connected to a patient simulator (PSIMU).

Actor name	**PSIMU**
Type [primary/secondary]	Primary
Description	A number of PSIMUs can be connected to a CSIMU allowing the CSIMU to control each PSIMU and allowing each PSIMU to send log messages to CSIMU.
Number of concurrent actors	Several

Actor name	**Remote User**
Type [primary/secondary]	Primary
Description	The remote user is capable of selecting the simulator values (by selecting a specific patient record) and to set the heartbeat factor. Furthermore the remote user can watch the displayed log messages.
Number of concurrent actors	1

1.2.3 System Functions

The system functions will be described here after using the use case technique. An overview of their association with actors is provided by the following use case diagrams, with their details described in section 1.31.2.7.

1.2.3.1 Use Case Diagram

The following use case diagram (Figure 4) illustrates all the use cases that the distributed simulator system is composed of:

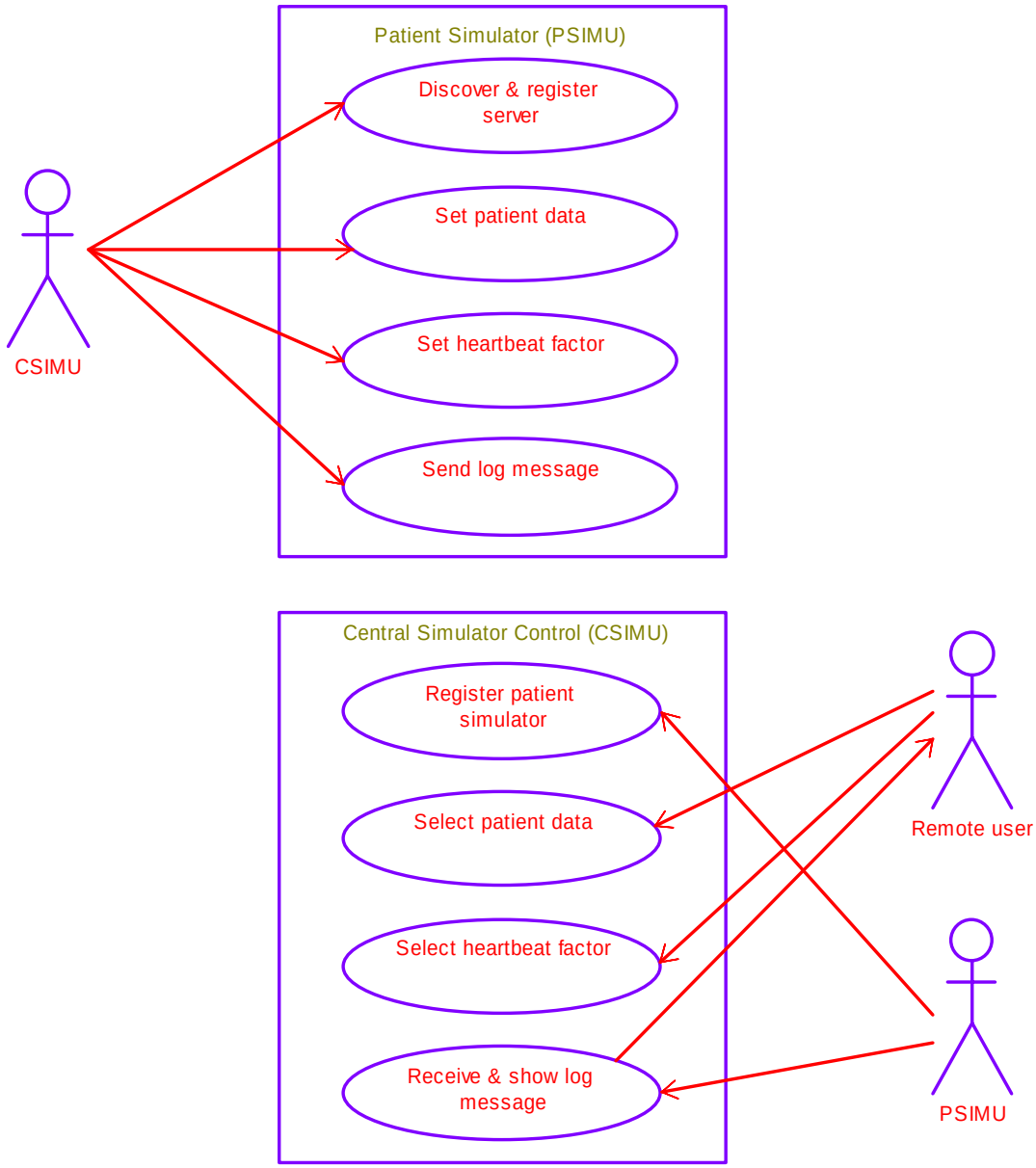

Figure 4: Use Case Diagram

1.2.4 System Constraints

Each patient simulator prototype (PSIMU) will, in the scope of this project, only allow for the possibility of one external infusion pump system and one monitor system.
The system should not be considered a 100% replica of a human body, and therefore should not be used for final testing and approval of other medical equipment.

1.2.5 Future of the System

At present, simulator data is located on each PSIMU. In the future, this data could be maintained only on the server (CSIMU) and hence be sent from CSIMU to each PSIMU.

1.2.6 User Characteristics

There are two kinds of users to the distributed patient simulator system.

1. Remote user: Operates the CSIMU (controlling the connected PSIMUs)
2. Local user: Can operate a single PSIMU (just like in the IRTS project)

The skills required by both users should be considered equivalent. Another possibility is for the remote user to be an external control system.
For this project the user will be regarded as a person who intends to test some bedside medical equipment and who requires the simulation of a patient. This could be a medical instructor coaching medical staff in the usage of hospital equipment or an engineer during test of newly developed equipment prototypes.
It is expected that the user interaction with the system is episodic and only few corrections to the output data will be enforced. A typical situation would be a single instructional session where two or three types of data would be selected for simulation.

1.2.7 Customer Deliveries

There is a single delivery of the DPSIMU project on Friday December 10th 2004.

Architecture, design and instructional documentation will be delivered in paper format and software documentation will be supplied on a CD-ROM media, including all material from the IRTS project.

1.2.8 Prerequisites

During the development process of this project, it will be required that the necessary hardware and software will be available for the developers. These are specified below in section 1.4.

1.3 Specific requirements (Use Cases)

This section contains the detailed description of the use cases that extend the basic PSIMU system from the IRTS course.

1.3.1 Use Case 1: Discover and register server

Goal	Locate and connect to the CSIMU, allowing the PSIMU to send log messages later to CSIMU and to let CSIMU control PSIMU
Initiation	From inside the system
Actors and stake holders	CSIMU
No of concurrent instances	1
Frequency	This is performed when the PSIMU system initializes itself and if the connection to the CSIMU is lost.
Non functional requirements	None
References	None
Preconditions	None
Post conditions by success	PSIMU will be able to log messages to CSIMU and this PSIMU is registered on CSIMU allowing CSIMU to control this PSIMU.
Post conditions by failure	PSIMU will not be able to log messages to CSIMU and this PSIMU is not registered on CSIMU so CSIMU cannot control this PSIMU.
Main scenario	1. Locate CSIMU 2. Connect to CSIMU 3. If connected, show it on the local user interface
Extensions	N/A

1.3.2 Use Case 2: Set patient data

Goal	Based upon the information received from CSIMU, set the current simulator values to the patient selected.
Initiation	CSIMU
Actors and stake holders	CSIMU
No of concurrent instances	1
Frequency	Estimated to 100 times per day
Non functional	None

requirements	
References	None
Preconditions	The required data has been installed on the PSIMU
Post conditions by success	A patient has been selected and its data is being output on the system hardware.
Post conditions by failure	The system has not selected a new patient, meaning that it still outputs the last selected patient data
Main scenario	1. The PSIMU receives a set patient request 2. The PSIMU changes to the selected patient record 3. A log message is sent to CSIMU informing about a new patient record
Extensions	N/A

1.3.3 Use Case 3: Set heartbeat factor

Goal	Based upon the information received from CSIMU, set the current heartbeat factor.
Initiation	CSIMU
Actors and stake holders	CSIMU
No of concurrent instances	1
Frequency	Estimated to 100 times per day
Non functional requirements	None
References	None
Preconditions	None
Post conditions by success	The heartbeat factor has been selected and the data is now output with the new heartbeat factor.
Post conditions by failure	The system has not changed the heartbeat factor that was last set.
Main scenario	1. The PSIMU receives a set heartbeat factor request 2. The PSIMU changes to the selected heartbeat factor 3. A log message is sent to CSIMU information about a new heartbeat factor
Extensions	N/A

1.3.4 Use Case 4: Send event message

Goal	An event message should be sent to CSIMU
Initiation	From inside the system
Actors and stake holders	CSIMU
No of concurrent instances	1
Frequency	Estimated to 100 times per day
Non functional requirements	
References	None
Preconditions	A connection to CSIMU has been established.
Post conditions by success	An event message has been successfully sent to CSIMU.
Post conditions by failure	The event message has not been sent to CSIMU.
Main scenario	1. PSIMU sends an event message to CSIMU
Extensions	N/A

1.3.5 Use Case 5: Register patient simulator

Goal	A PSIMU is registered and CSIMU is ready to receive control signals from it and send event messages to it.
Initiation	PSIMU
Actors and stake holders	PSIMU
No of concurrent instances	1
Frequency	-
Non functional requirements	None
References	None
Preconditions	None
Post conditions by success	PSIMU is successfully registered and connection is established.
Post conditions by failure	The connection is failed.
Main scenario	1. Accept the request for connection. 2. Connection to the requesting PSIMU established
Extensions	N/A

1.3.6 Use Case 6: Select patient data

Goal	The patient is selected and the new information is sent to the selected PSIMU.
Initiation	Remote user.
Actors and stake holders	Remote user, PSIMU
No of concurrent instances	1
Frequency	Estimated to 100 times per day
Non functional requirements	None
References	None
Preconditions	None
Post conditions by success	Information about selected patient has been sent to the selected PSIMU.
Post conditions by failure	Information about selected patient has not been sent to PSIMU.
Main scenario	1. Remote user has selected a patient. 2. Information about the selected patient has been sent to the PSIMU.
Extensions	N/A

1.3.7 Use Case 7: Select heartbeat factor

Goal	Heartbeat factor information is selected and sent to the selected PSIMU.
Initiation	Remote user
Actors and stake holders	Remote user, PSIMU
No of concurrent instances	1
Frequency	Estimated to 100 times per day
Non functional requirements	None
References	None
Preconditions	None
Post conditions by success	The heartbeat factor has been selected and sent to the selected PSIMU.
Post conditions by failure	The heartbeat factor has not been selected and sent.
Main scenario	1. A new heartbeat factor has been selected. 2. Information is sent to the selected PSIMU.
Extensions	N/A

1.3.8 Use Case 8: Receive and show event message

Goal	Event message received and showed on the screen.
Initiation	PSIMU
Actors and stake holders	PSIMU, Remote user
No of concurrent instances	-
Frequency	Estimated to 100 times per day
Non functional requirements	
References	None.
Preconditions	A connection to PSIMU has been established.
Post conditions by success	An event message has been successfully received. An event message has been successfully displayed on the screen.
Post conditions by failure	The event message has not been received from PSIMU. The event message cannot be displayed on the screen.
Main scenario	1. CSIMU receives an event message from PSIMU. 2. The event message is displayed on the screen.
Extensions	N/A

1.4 External Interface Requirements

1.4.1 User Interfaces

The user interface to CSIMU consists of a single dedicated screen. Users can be expected to have basic knowledge on interpretation of output values.

The purpose of the CSIMU screen is to control each of the connected PSIMUs.
For every PSIMU unit that is connected to CSIMU, it must be possible to view log events and to control the output of that PSIMU, i.e. select a new patient data file and to set heartbeat factor. These controls are fitted in a box for every connected PSIMU.

A prototype of the graphical user interface is shown in Figure 5.

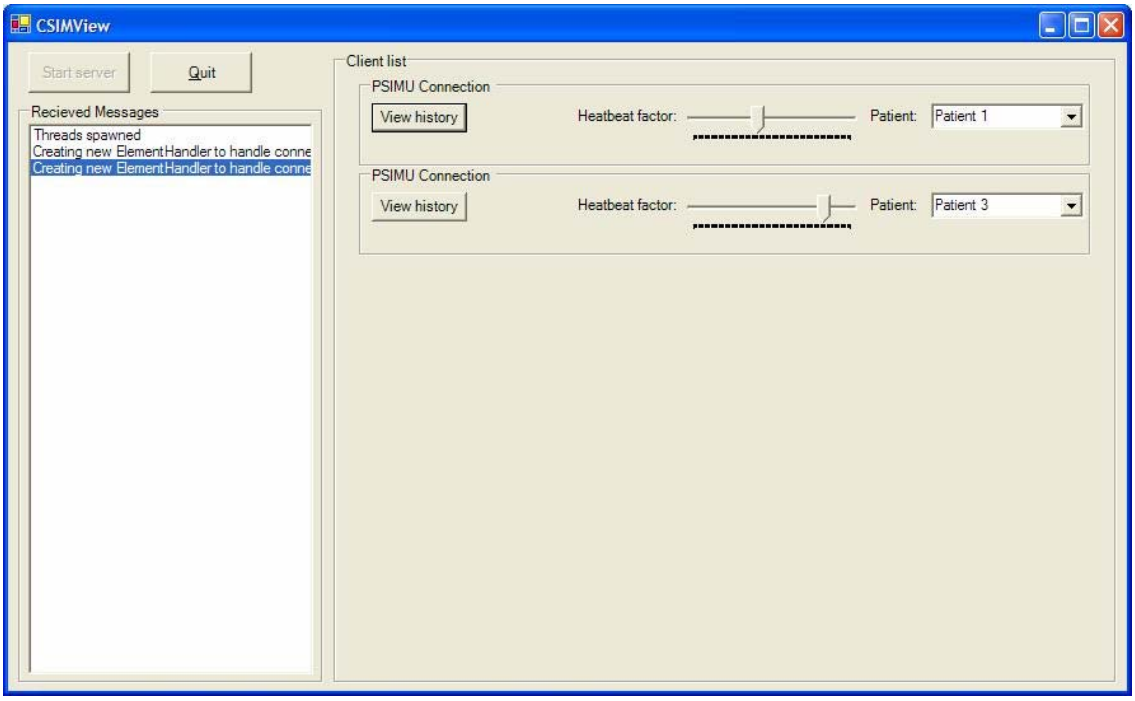

Figure 5: GUI prototype

1.4.2 Hardware Interfaces

The client application is specified to execute on a SBC embedded computer.
Output: To provide analogue output, a PV2019 I/O card is used.
For digital output, an IO686 I/O card is used.
Input: RS232 is used for receiving medicine data from IPUMP.

The server application (CSIMU) is specified to execute on a PC with Windows XP operating system.

1.4.3 Communication Interfaces

PSIMUs are communicating with the CSIMU through a LAN Ethernet connection.

1.4.4 Software Interfaces

The client application uses On Time's RTKernel (RTK) as operating system. With RTK comes a set of APIs that will be used, including RTPeg for graphical user interface.

The server application is developed to run on a Windows XP operating system.

1.5 Performance Requirements

1.6 System Qualities

The quality factors of the system are shown in Figure 6.
It is not essential that the data simulated are reliable, since even if output channels should differ a little from the data in the archive, it will still be good enough for the system reading the output.
Usability is not an important issue either, since the user interface is not the primary function of the system.
Extensibility and reusability are very important issues, since the system is extended from the PSIMU project. These priorities should manifest through modelling (by using design patterns).

Quality Factor	**Estimation** 1 = not critical, 5 = very critical
Stability	3
Reliability	3
Usability	3
Extensibility	5
Reusable	4

Figure 6: Quality Factors

1.7 Design Constraints

The DPSIMU2 system will be developed according to the ROPES development process.

1.7.1 Authority Requirements

Since this system never interacts with actual patients, there are no authority requirements.

1.8 Part deliveries

There are no part deliveries. Only the fully functional system is delivered.

2 System Architecture

2.1 Introduction

2.1.1 Purpose and Scope

This section describes the analysis and design of the patient simulator formalized by the previous requirement specification:

The purpose of this project is to implement a prototype of a simulation system (called PSIMU) for a human patient that is capable of simulating patient life signals.

An external monitoring system will be able to connect to the output from the simulator that reproduces real-time signal data on analogue and digital output ports.
This system will be able to simulate analogue Electro Cardio Gram (ECG), ECG-Derived Respiration (EDR) signals and a digital pulse. Furthermore it can receive input from an external infusion pump about which, and how much, medicine is being infused, and from this information regulate the outputs.

The simulator will be able to handle data from the PhysioBank databases.

And the distributed extension of this project:

The above mentioned PSIMUs are connected to a central simulator control (called CSIMU) shown on Figure 2.
The CSIMU is able to control the connected PSIMUs in the context of selecting simulating values and the heartbeat-rate.

These can be regulated by a remote user on the CSIMU.

2.1.2 References

- Project description for Patient Simulator System (PSIMU)
- Project description for Local Monitor System (LMON)
- Project description for Infusion Pump System (IPUMP)
- TI-RTS Project Interface Specification (version 16.02.2004)
- PhysioBank Archives (http://www.physionet.org/physiobank/)

2.1.3 Definitions and acronyms

See word list in requirement specification document.

2.1.4 Document structure and reading guide

This section is organized following the “4+1 view”, described by Phillippe Kruchten, which divides the presentation of the project analysis and design in 4 major groups: Logical, Process, Deployment and Implementation. These groups are glued together with a Use case view which will be described initially.

2.1.5 Document role in an iterative development process

As this project is being developed using an interactive process, the associated documentation will also be inserted and updated accordingly. This means that this section reflects the current state of the analysis and design in the current development iteration.

2.2 System Overview

2.2.1 System context

The system context in relation to its external actors can be viewed in the following diagram from requirement specification. A more detailed description hereof can be found in section 1.2 of that document.

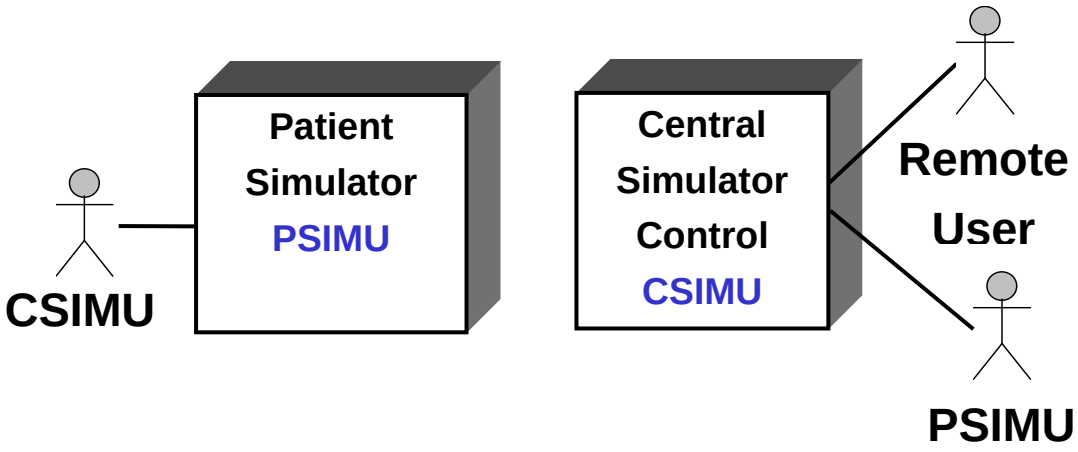

Figure 7: System context

2.2.2 System introduction

See Requirement Specification, section 1.2.1 .

2.3 System Interfaces

2.3.1 Interface to human actors

On PSIMU the GUI from IRTS is used. On CSIMU a GUI is available which allows the remote user to maintain an overview of connected PSIMU's, and controls these.

2.3.2 Interface to external system actors

- All interfaces from the IRTS project are maintained

2.3.3 Interface to hardware actors

- All hardware interfaces from the IRTS project are maintained

2.3.4 Interface to external software actors

- PSIMU creates a socket to CSIMU, and CSIMU is able to control PSIMU through this socket

2.4 Use Case View

2.4.1 Overview of architecture significant Use cases

All use cases from the requirement specification are to be implemented in the DPSIMU system.

The use cases from the IRTS project are not included here as they have already been documented before [Ref. 5].

2.5 Logical View

2.5.1 Overview

Figure 8 shows this overall package diagram. Notice that the platform and communication packages are used both by PSIMU and CSIMU, while hardware is only used by PSIMU.

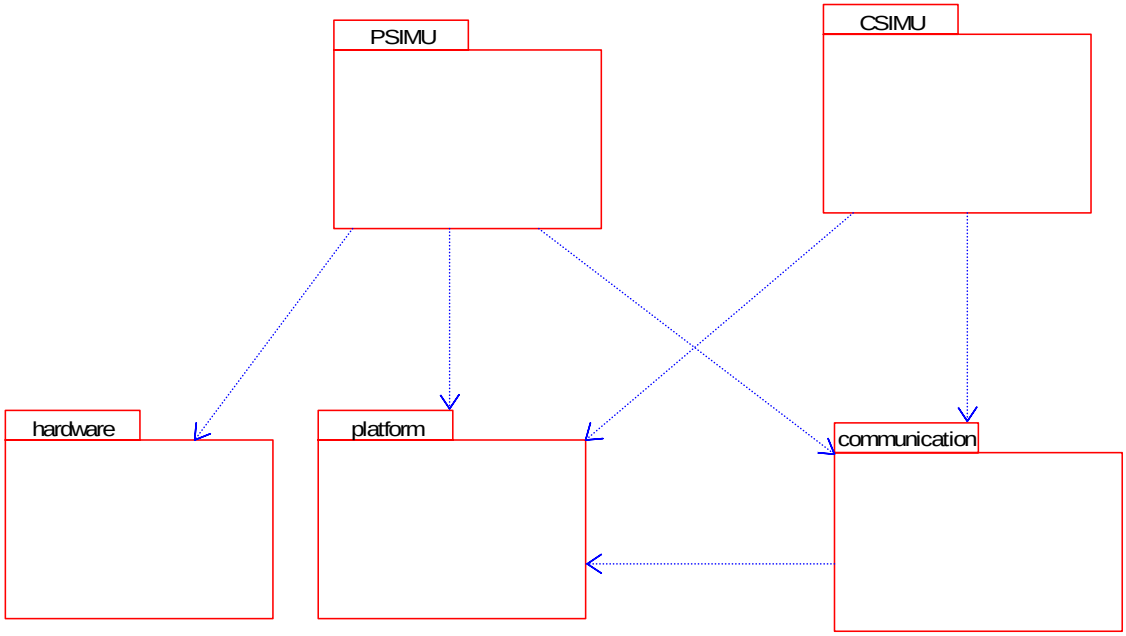

Figure 8: Overall package diagram

2.5.2 Architecturally significant design packages

2.5.2.1 Package: Platform

This package contains classes that rely on native OS functions. In IRTS this packages only contained an implementation for the RT-Kernel, but now an implementation for WIN32 is also provided, since CSIMU is dependant on it.

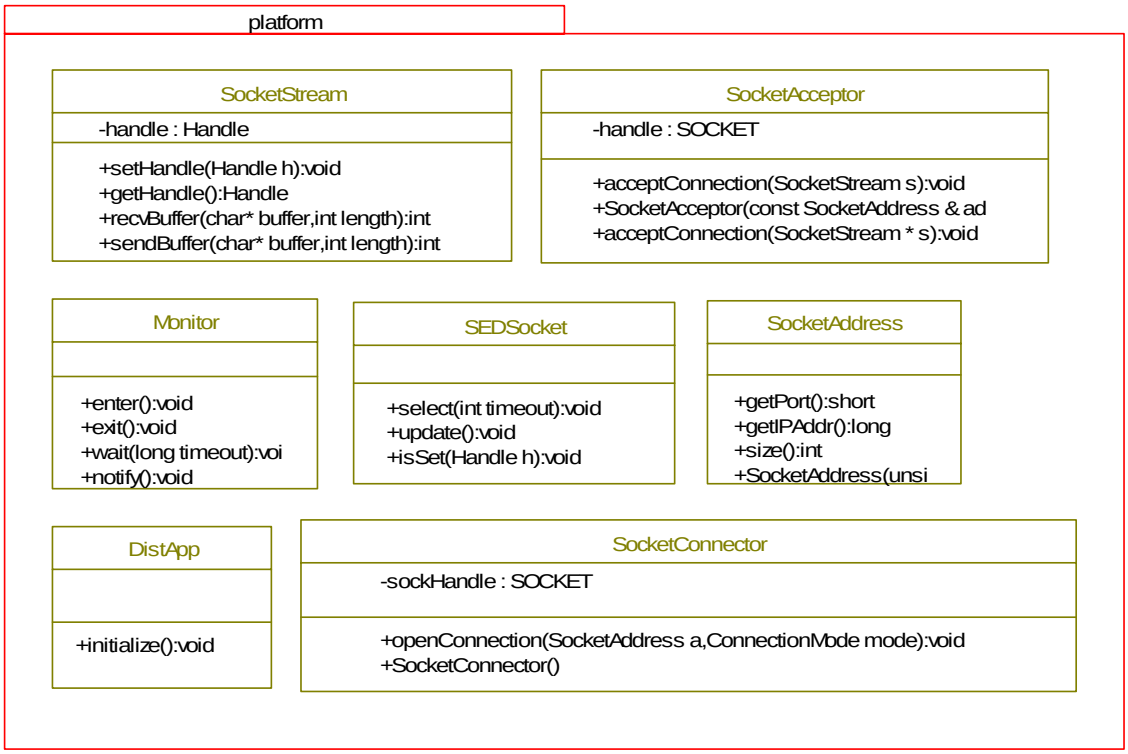

Figure 9: The platform package

Class: SocketStream

Represents the underlying stream of a socket and therefore handles sending and receiving data.

Class: SocketAcceptor

Handles listening on a port of a network interface, and creating a SocketConnector for each client that connects.

Class: Monitor

Provides an implementation of a Monitor class, allowing the user to create synchronized code.

Class: SEDSocket

This class implements the abstract SED class from the communication package. It is able to wait for action on multiple sockets. It is used for both wait for data, and waiting for asynchronous connection requests.

Class: DistApp

Since both platforms need some initialization code, this class has been created to handle platform initialization.

Class: SocketAddress

This class represents an ip/port address, which can be used for listening and connecting.

Class: SocketConnector

This class is can be used on both client and server. On the client, it is able to create a socket connection based on a SocketAddress. On the server it is used to handle accepted connections.

2.5.2.2 Package: Communication

This package contains generic classes which is non-application specific, and is an application framework.

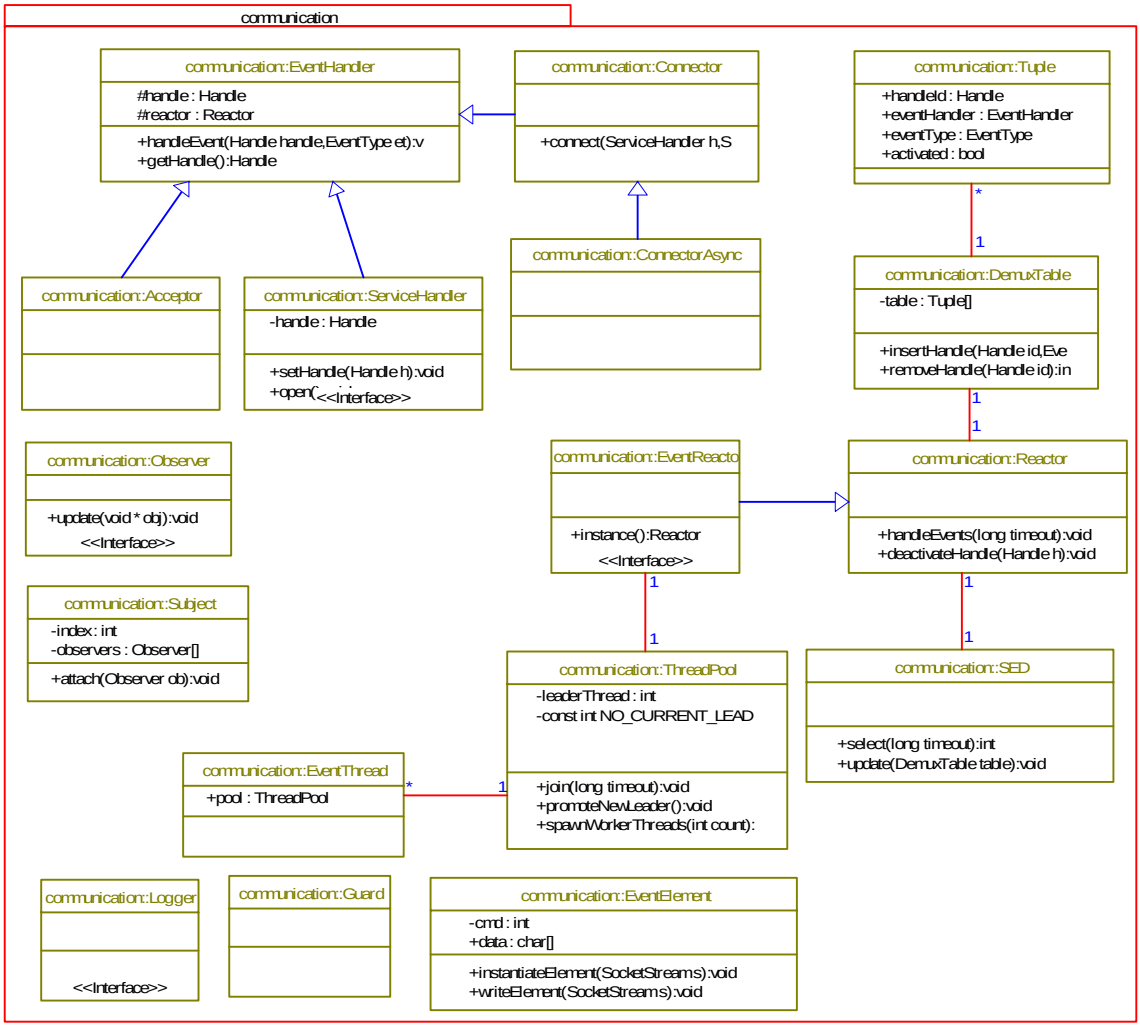

Figure 10: The communication package

Class: Acceptor
Implemented according to the Acceptor in the Acceptor/Connector pattern. Concrete implementations should subclass from this class.

Class: Connector
An abstract class that defines an event handler, that is able to connect to a server.

Class: ConnectorAsync
Implements connector using the asynchronous method. It does so by sending a connect request, and waiting for EventReactor to call back.

Class: DemuxTable
Holds a list of Tuple objects in a list.

Class: Tuple
An object that holds a handle and a reference to the EventHandler, which should be notified on events.

Class: EventElement
Defines the data format for the communication channels. It has methods to read/write from/to a socket stream

Class: EventHandler
Interface that defines an event handler, for the reactor pattern.

Class: Reactor
An abstract class, which defines a reactor. A reactor uses a Demux table to store handles and EventHandlers.

Class: EventReactor
Implements the reactor class, using a SEDSocket instance to wait for events.
On events it will dispatch one event only.

Class: Logger
This is an abstract class that provides an interface for logging system events.

2.5.2.3 Package: CSIMU

The server part of the system. It has a network interface part, and a graphical part which enables the user to interact.

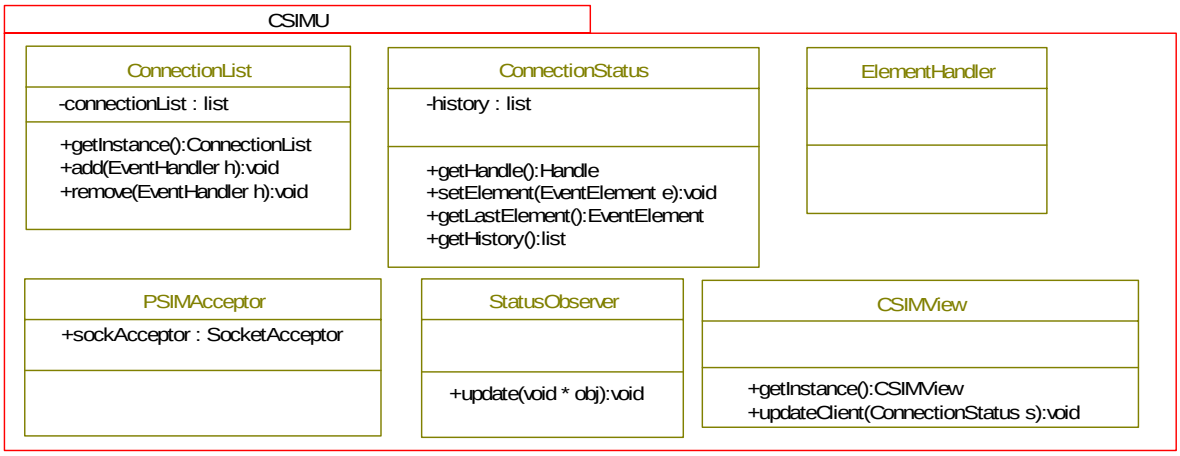

Figure 11: The CSIMU package

Class: ConnectionList

Contains a list with all current *ConnectionStatus* objects. It is implemented as a singleton. It is used mainly by GUI, to display a list of connected PSIMU clients.
It extends the Subject class, from the observer pattern, which enables observers to be notified whenever something happens on a connection.

Class: ConnectionStatus

Serves as a data container for all data related to a single connected, this includes the socket and handle received elements.
Whenever an EventElement by the *EventHandler*, it is sent to it's associated connection status, where it is stored.
ConnectionStatus will store a specified number of elements, and thereby maintains a history of received elements.

Class: ElementHandler

A concrete implementation of *ServiceHandler*, which is used for handle incoming data from on a single socket. It is responsible for reading data and saving it in the associated ConnectionStatus.

Class: StatusObserver

Implements the *Observer* interface, from the Observer pattern. It observes events on the *ConnectionList*, and *updates CSIMView* on each update.

Class: CSIMView

Implements entire GUI of CSIMU, using the .NET framework. More specific the Windows Forms part of .NET.
It is implemented using Managed C++, which is an extensision of C++. It enables ordinary C++ programs to contain .NET code.

It is notified by *StatusObserver* with *ConnectionStatus* objects, and *CSIMView* stores these, and uses them to send messages on a connection, whenever the user whishes to do so.

2.5.2.4 Package: PSIMU

This package contains the classes that implement the PSIMU patient simulator client on the SBC686 platform. It contains all the necessary classes for providing a user interface, handling patient simulation and establishing and maintaining a communication channel with a remote server.
This package is a combination of the *application* and *GUI* packages from the stand-alone version. Also several classes from the previous *communication* package have been moved into this package.

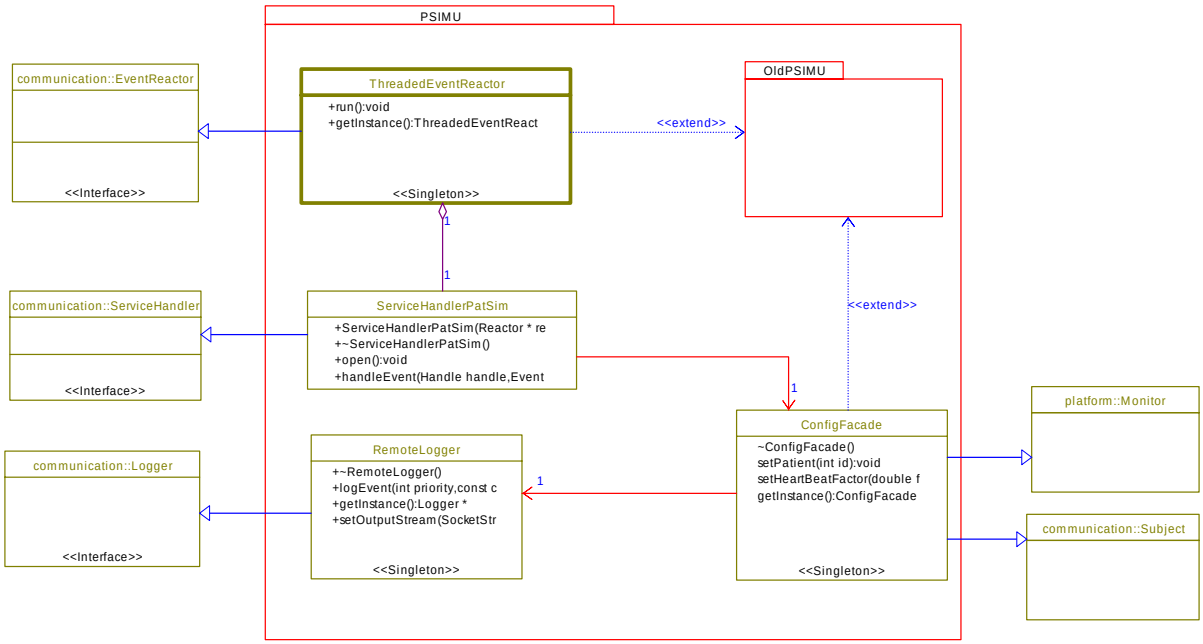

Figure 12: The PSIMU package

In Figure 12, the previous design is symbolized by the *OldPSIMU* package where the *ConfigFacade* class replaces what was previously known as the *ConfigRepository*.

Class: ConfigFacade

This class provides the central internal communication of system events between the software components. Similar as it earlier version (ConfigRepository), this class is implemented as a *Subject* in the GOF Observer pattern [Ref. 2], so as to notify any registered listeners of system events. This class inherits the semaphore capabilities of the *Monitor* class to protect the events propagation mechanism of the *Subject* class.

Class: ThreadedEventReactor

This new singleton component, in the patient simulator, extends the capabilities of the *EventReactor* class from the *communication* package. Basically it provides a thread mechanism, allowing the instances of this class to be spawned off, and returning control to its creator. This avoids blocking caused by the `select()` call used in the EventReactor class.

Class: ServiceHandlerPatSim

This class is responsible for handling incoming events from the ***central server*** and re-dispatching them onto the *ConfigFacade* class. It implements the functionality of *ServiceHandler*, and thus the *EventHandler* from the *communication* package.
It implements a mechanism for receiving *EventElements* and parsing these into the relevant system calls found in the *ConfigFacade* class.

Class: RemoteLogger

This class handles communication of the events occurring inside the patient simulator client to the ***central server***. This is done by translating the information received from the *ConfigFacade* class into en *EventElement* and sending it over the communication channel that it shares with the *ServiceHandlerPatSim* class.

2.5.3 Use case realizations

2.5.3.1 Use Case 1: Discover and register server

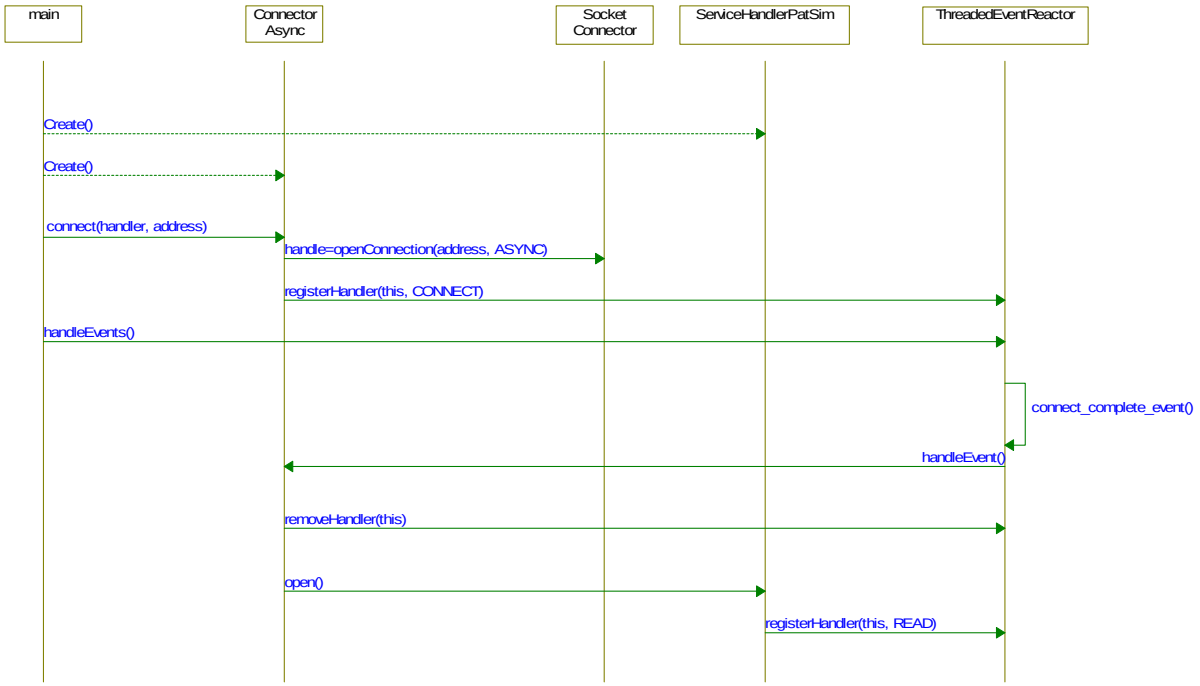

Figure 13: Sequence for creating connections on PSIMU

Figure 13 shows the design of creating new connection on PSIMU. It is designed using the asynchronous part of the Connector pattern.

2.5.3.2 Use case 5: Register patient simulator

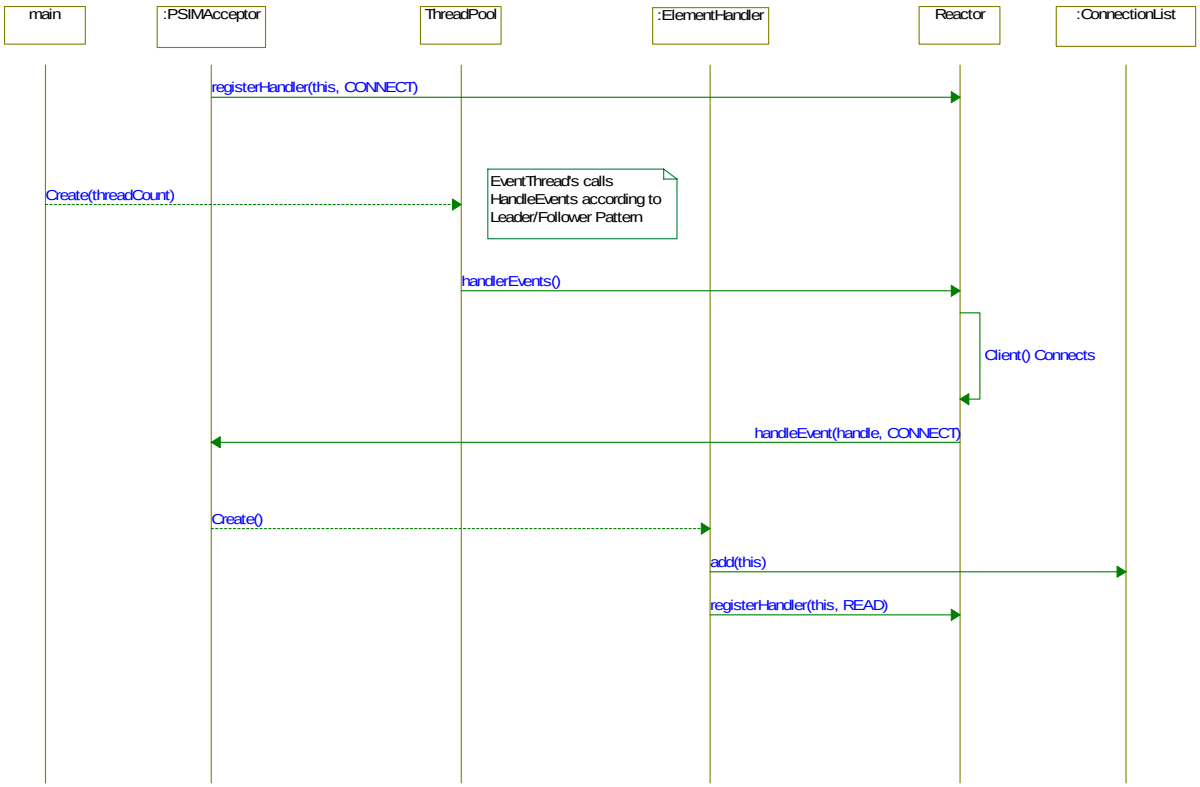

Figure 14: Sequence diagram for handling PSIMU connections on CSIMU

Figure 14 shows how accepting new PSIMU connections is handled. It is designed according to the Acceptor part of the Acceptor/Connector pattern. The only modification is that the EventHandler now adds itself to a list of active connections. This is done to provide outside access to connections, which would not have been possible otherwise.

2.5.3.3 Use case 2+6: Select/set patient

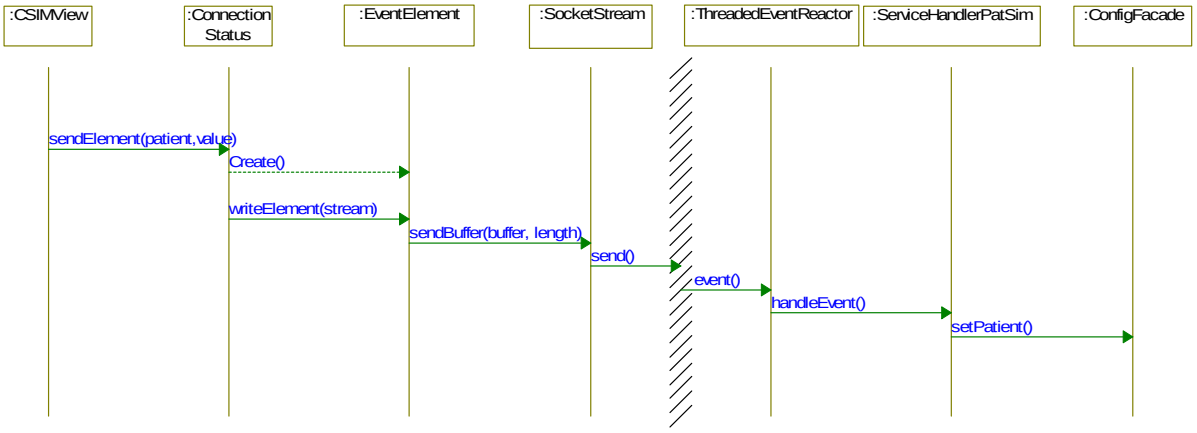

Figure 15: Sequence to set patient values

The sequence in Figure 15 shows how a CSIMU application is able to select a patient on a PSIMU. On PSIMU the ConfigFacade is used, in the same way as the

PSIMU's own GUI would. The *CSIMView* is able to get a reference to the *ConnectionStatus* through the *ConnectionList*.

2.5.3.4 Use case 3+7: Select/set heatbeat factor

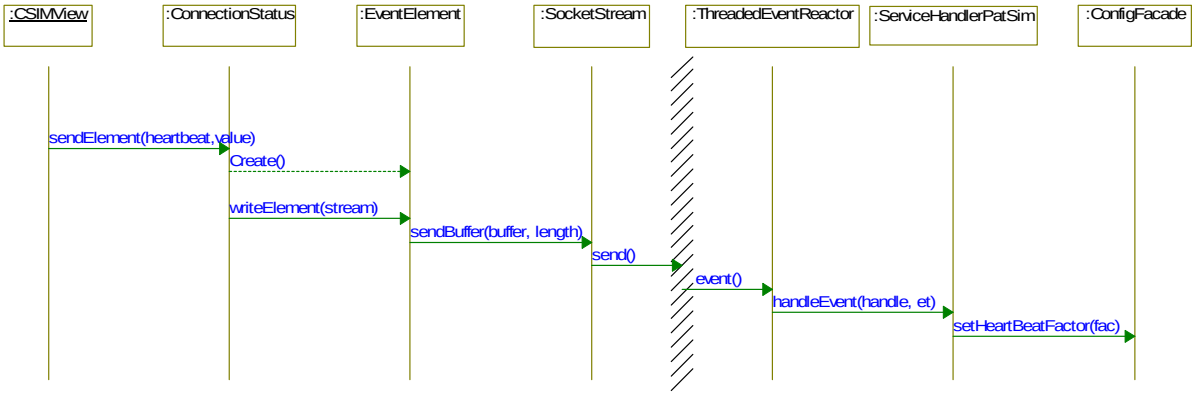

Figure 16: Sequence diagram of setting heatbeat factor

Figure 16 shows how CSIMU's GUI, is able to set the heatbeat factor. The sequence is exactly the same as usecase 2 + 6.

2.5.3.5 Use case 4+8: Send event message and Receive and show event message

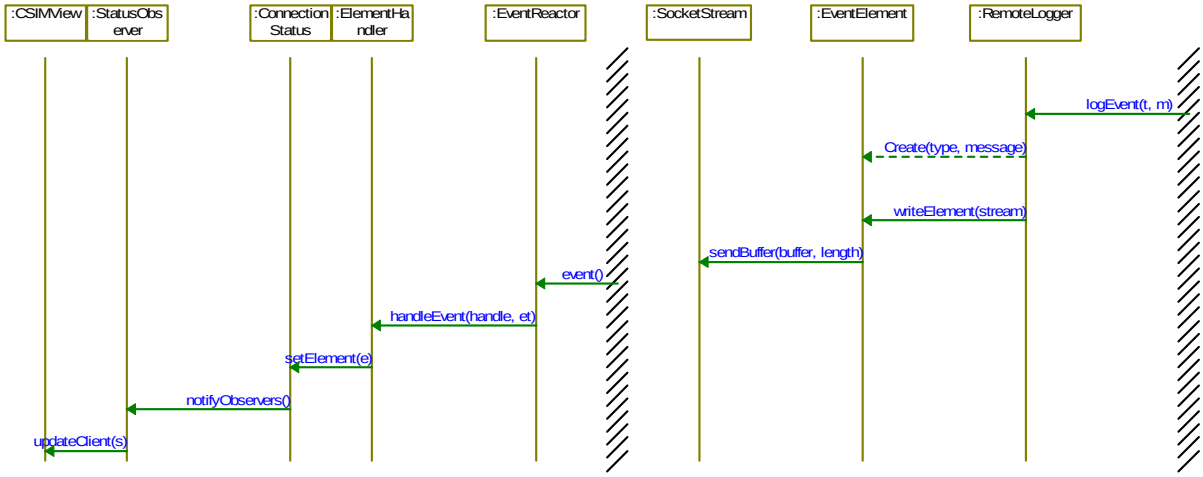

Figure 17: Sequence for sending and receiving log messages

Figure 17 shows the sequence involved in sending log messages from PSIMU to CSIMU. The right hand border represents the original system, which can be any class, including *ConfigFacade*, in the system. Notice that this sequence is initiated from the PSIMU, where the other use cases are initiated from CSIMU.

2.6 Process/task View

2.6.1 Process/task overview

In the course of extending the patient simulator with distributed capabilities, an additional task has been created to handle incoming events from the **Central Simulator Control** server. This task has given rise to the creation of a new task group based upon the classes used for the extension.
The design issue of the stand-alone version of the simulator has been omitted, as all communication with the existing system goes through the *ConfigFacade* class, and thus relies on the synchronization mechanisms implemented within this class.
Figure 18 illustrates the integration of new task (*ThreadedEventReactor*) into the existing application.

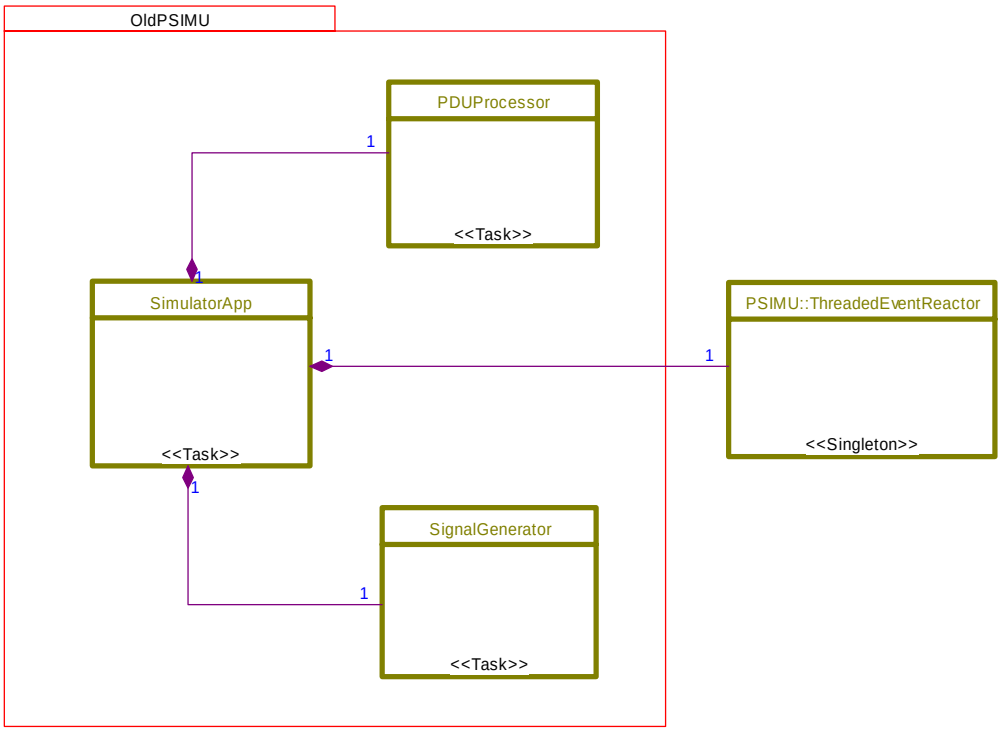

Figure 18: Task overview

The central server has some concurrency issues related to is client handling, which will be handled in the description of the **Leader**/**Followers** architecture pattern in section 3.2.3.

2.6.2 Process/task implementation

The base class for the active classes is the abstract class *Thread* from the *platform* package. It implements threading using the RTKernel command – *RTKRTL-CreateThread*. This means that all the required logic for implementing an active class is limited to extending this class with a *run* method containing the business logic for the thread.

As the *SimulatorApp* thread is also responsible for updating the screen through its ownership of the PEG *PresentationManager,* is should have the same priority as the *SignalGenerator* as events can happen 500 times per second and many calculations are performed in a duty cycle. The *PDUProcessor* has a much lower duty cycle and is set to a lower priority.

The number of events expected to occur on the *ThreadedEventReactor* should be considered very low, and as such the priority of the thread has been given the same priority as the *PDUProcessor*, the lowest.

The revised priorities are shown in the following table:

Task	Priority
SimulatorApp (main thread)	2
SignalGenerator	3
PDUProcessor	1
ThreadedEventReactor	1

2.6.3 Process/task communication and synchronization

For communication between the tasks in the current implementation, the system uses procedure calls and RTPeg's IPC messages. An explanation of how they inter-communicate is described in the design documentation for the stand-alone version [Ref. 5].

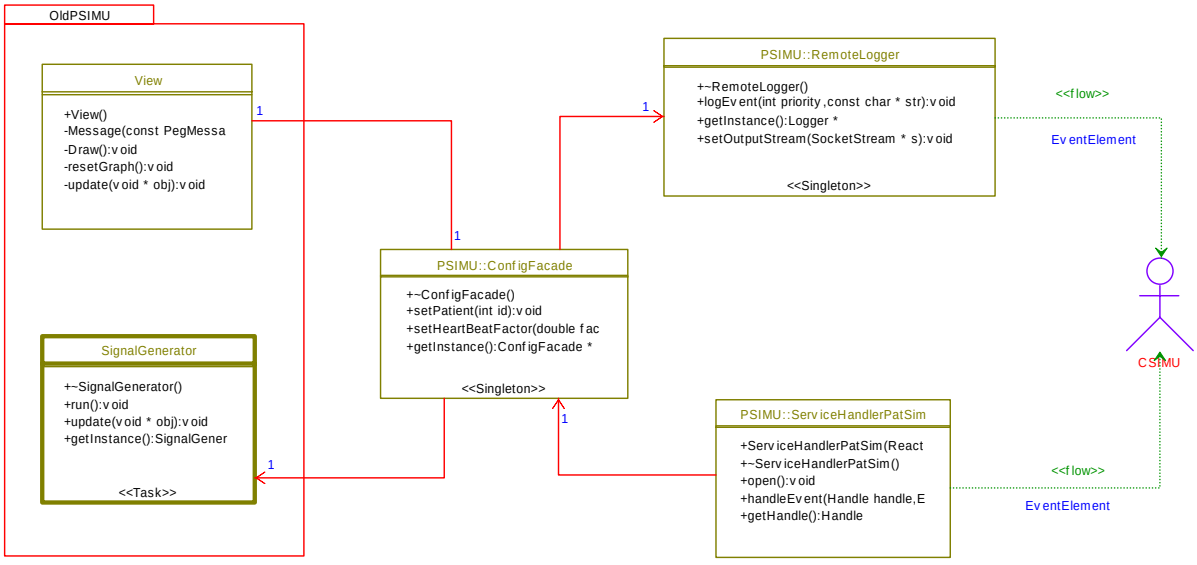

Figure 19: Process communication

Figure 19 shows the centralized nature of the *ConfigFacade* class in the inter-process communication between the classes.

2.7 Deployment View

2.7.1 System configurations overview

The system can be run in either the real simulation situation with a client and a server or in a test of server or client.

2.7.2 System configurations

2.7.2.1 Configuration 1: Simulator

Figure 20 shows the patient simulator system in the real simulating situation.
The clients (PSIMUs) connect to the central simulation control (CSIMU).
On the figure, two clients (PSIMUs) are shown communicating with the server (CSIMU). There can be 0..* PSIMUs connected to a single CSIMU.

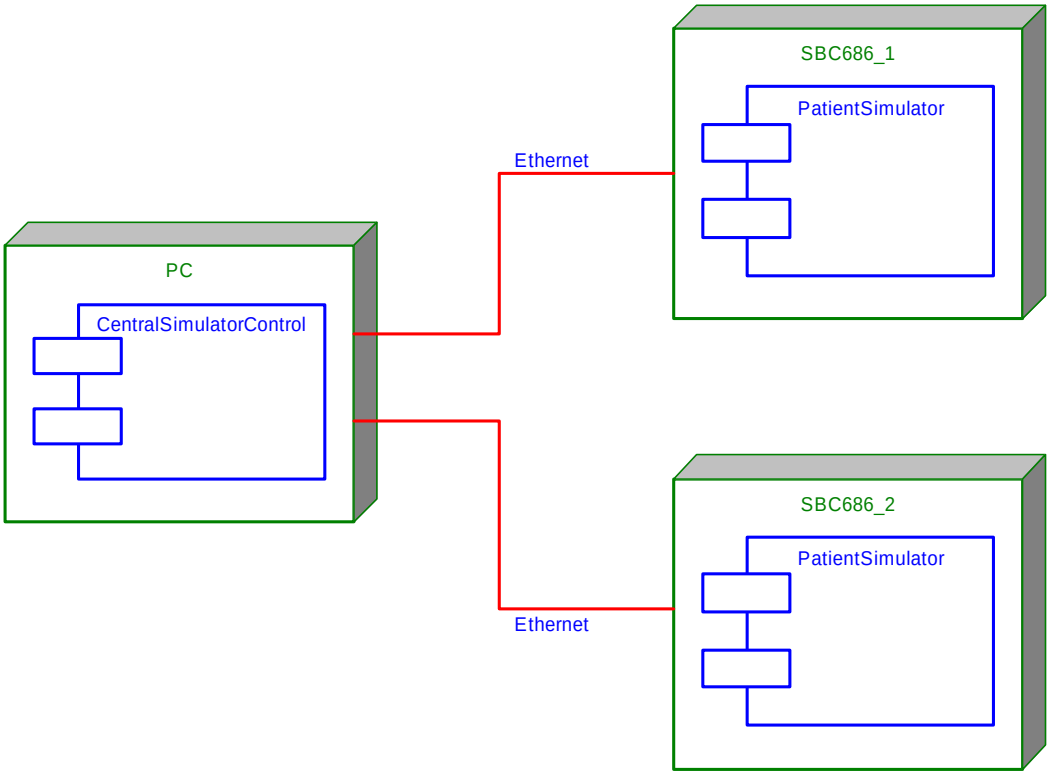

Figure 20: Simulator deployment

2.7.2.2 Configuration 2: Test of CSIMU

Figure 21 shows the patient simulator system when CSIMU is being tested.
The following changes have been made:

- The client application has been stubbed into a test application playing the role as a PSIMU towards the CSIMU.
- Only the communication functionalities are implemented into the test application.
- The PSIMU application is placed on the same computer as CSIMU.
- The Ethernet connection has therefore been omitted.

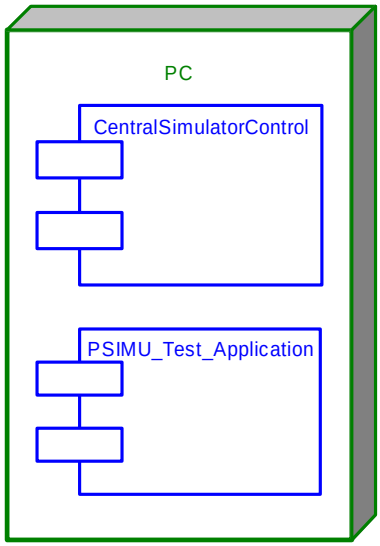

Figure 21: Deployment of CSIMU test

2.7.2.3 Configuration 2: Test of PSIMU

Figure 22 shows the patient simulator system when PSIMU is being tested.
The following changes have been made:

- The CSIMU application has been stubbed into a test application playing the role as a CSIMU towards the CSIMU.
- Only the communication functionalities are implemented into the test application.

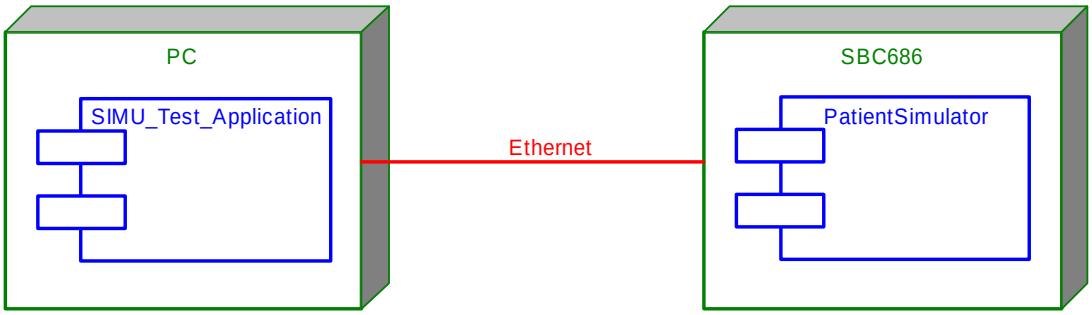

Figure 22: Deployment of PSIMU test

2.7.3 Node descriptions

2.7.3.1 SBC686

Hosts the PSIMU application that is able to connect to a CSIMU making it capable of controlling the PSIMU.

2.7.3.2 PC

Hosts the CSIMU application that controls a number of connected PSIMUs.

2.8 Implementation View

2.8.1 Overview

The distributed extensions to the Patient Simulator have not required addition of any new components to the implementation on the embedded platform. Figure 23 shows the components found on the SBC686 platform.

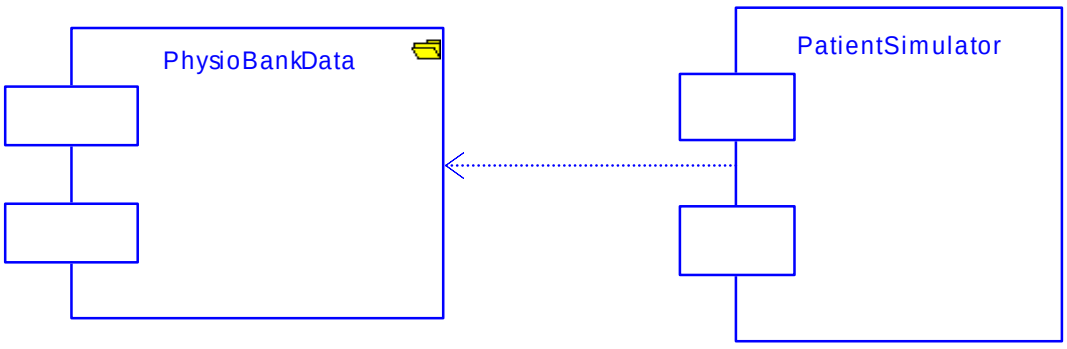

Figure 23: Patient simulator component diagram

The components added to the project in the Central Simulator Control can be seen in Figure 24.

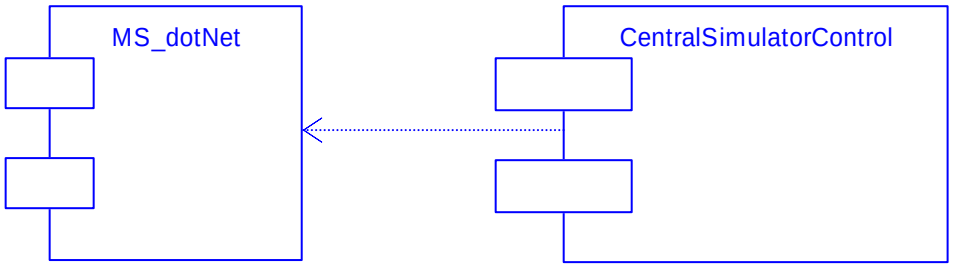

Figure 24: The Central Simulator Control

2.8.2 Component descriptions

2.8.2.1 PatientSimulator

This component is the executable file precompiled for the RT Kernel and post processed for installation on the SBC686 hardware platform.

2.8.2.2 PhysioBankData

This folder, on the SBC686, contains the actual medical data used for generating the output in the simulator. These files can be downloaded from the PhysioBank web site.

2.8.2.3 CentralSimulatorControl

The Central Simulator Control is the precompiled application for the MS Windows operating system. It can be compiled into two versions:

- A console version where all output if shown on the system screen and the user controls the Patient Simulator through a command prompt, dependant only on the underlying operating system. This version is only applicable for debugging the system as many features are omitted;
- A graphical user application with a single screen overview of all the Patient Simulators connected. This is the version supplied to the final user of the system and has a dependency on the .NET framework from Microsoft.

2.8.2.4 MS_dotNET

This is the .NET framework component required to be installed on the server machine if the GUI version of the server is to be applied.

2.9 General design decisions

CSIMU design decisions are based upon flexibility and reusability, according to section 1.6.

2.9.1 Architectural goals and constraints

It is a requirement that the PSIMU system is developed for the kernel RTKernel v. 4.07 from On-Time.
PSIMU system is developed for SBC686.
CSIMU system is developed for Windows XP.

2.9.2 Architectural patterns

Design patterns described for PSIMU system can be found in IRTS project documentation [Ref. 5].

The class *ConfigFacade* acts as the central controller (hence the Façade pattern is used) in this system between the model classes and the GUI.

The *Reactor design pattern* [Ref. 3] allows event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.

The *Acceptor-Connector design pattern* [Ref. 3] decouples the connection and initialization of cooperating peer services in the networked system from the processing performed by the peer services after they are connected and initialized.

The *Leader/Followers design pattern* [Ref. 3] provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in the order to detect, demultiplex, dispatch, and process service requests that occur on the event sources.

To read more about these architectural patterns used in project, see section 3.2.

2.9.3 General user interface design rules

General user interface design rules for PSIMU system can be found in IRTS project documentation [Ref. 5].
The user interface on CSIMU is a graphical user interface (GUI), where the user navigates with the mouse.

The remote user must have an opportunity to do the following:

- Select connected PSIMU unit.
- Select the outputted patient data from connected PSIMUs
- Select the heartbeat rate from connected PSIMUs

A prototype of the GUI can be found in the Requirement Specification, section 1.4.1.

2.9.4 Implementation languages and tools

The chosen implementation language is C++.
CSIMU graphical user interface is implemented in Managed C++.
The PSIMU graphical user interface is implemented in RT-Peg Window Builder.
The necessary tools are:

- Microsoft Visual Studio .NET – is the environment for compiling the code
- RT-Peg Window Builder is the environment for building the graphical user interface.
- Rhapsody 5.0 – used for UML documentation

2.9.5 Implementation libraries

PSIMU implementation libraries:

- Standard C++ libraries (libc)
- RT-Kernel libraries (v. 4.07)
- RT-PEG libraries
- RT-Files

CSIMU implementation libraries:

- .NET
- Standard C++ libraries (libc)
- Standard Windows libraries

2.10 Compilation and Linking

2.10.1 Compilation hardware

The system consists of two different applications, PSIMU and CSIMU. PSIMU will be compiled for an RTKernel platform. CSIMU can be compiled for Microsoft Windows XP.

2.10.2 Compilation software

Both CSIMU and PSIMU are compiled using Microsoft's nmake system. To compile PSIMU, the Ontime RTOS 4.0 environment is required.
For compilation, version 7.10 of nmake and version 13.10.3077 of the command line compiler have been tested.
The compilation process is command line based, with appropriate make files placed in the source tree, so any development environment capable of calling an external process for compilation and linking can be used, as well as a normal command line prompt.
The GUI is developed using Microsoft .NET SDK version 1.1, so it is required to be able to compile the GUI. If not present, a command-line version of CSIMU can be used.

The GUI is developed using the Managed C++ language, which is an extension of ANSI C++. It has extras syntax which enables the developer to create and use .NET code, and at the same time use non managed C++. So there are .NET technologies available like garbage collection and a virtual machine in CSIMView, and no change in the rest of the CSIMU application. Note that CSIMU is still compiled into a single executable, where similar languages would require splitting the application into an executable and a DLL.

2.10.3 Compilation and linking process

To compile and link, setup the development environment to be able to locate the include files and external libraries found in the before mentioned tools.

To compile and link the application: go to the project root and type ***nmake all***. This will compile and link both applications.

To upload the application through the serial port and using the RTKernel monitor tool, start the SBC686 and execute the ***grmon*** application and type ***nmake monitor*** on the development PC. This will locate the application and upload it.

Other useful commands found in the makefile are:

clean:	removes all binaries and old editor files.
backup:	creates a compressed backup file contains all the source code.
cvs:	updates the source files from the CVS repository.
doc:	generates doxygen documentation

2.11 Installation and Executing

2.11.1 Installation

This section will refer only to the CSIMU application, as the installation of the PSIMU software has been described in the design document for its stand-alone version [Ref. 5].

2.11.1.1 Software

Installation of the CSIMU software requires the .NET framework from Microsoft to be preinstalled on the host system before installation. The server software can then be installed under C:\Program Files\<directory>.
It is a prerequisite that the configuration of the network be configured beforehand.

2.11.2 Executing software

Locate the file with the Windows Explorer application and double click on it.

2.11.3 Execution control

Starting the server involves clicking on the “Start server” button and awaiting the connection of the clients.

When a client connects a control panel will appear, allowing manipulation of the client by selecting a patient or modifying the heart (see screenshot in Figure 30) rate. Any events taking place on the client will be propagated onto the server and the corresponding widgets will react to alterations taking place on the client.

Shutting down the application simply involves closing the application.

3 Implementation

3.1 Introduction

This section describes various implementation issues that have come up during the implementation.

3.2 Design patterns

In this project, primarily three new design patterns from the POSA2 book [Ref. 3] have been used. In the following, it will be shown briefly how these are implemented be means of a class diagram.
The Façade design pattern from GoF [Ref. 2] has also been used to make a unified interface to the PSIMU system.
Furthermore, some design patterns were introduced in the IRTS project that is the Singleton, Observer and Command patterns – all from the GoF book [Ref. 2].

3.2.1 Reactor pattern

3.2.1.1 Introduction

The purpose of Reactor pattern [Ref. 3, s. 179] is to allow event-driven applications to demultiplex and dispatch service requests that are delivered to an application from one or more clients.
The overall classes in the pattern are shown in Figure 25.

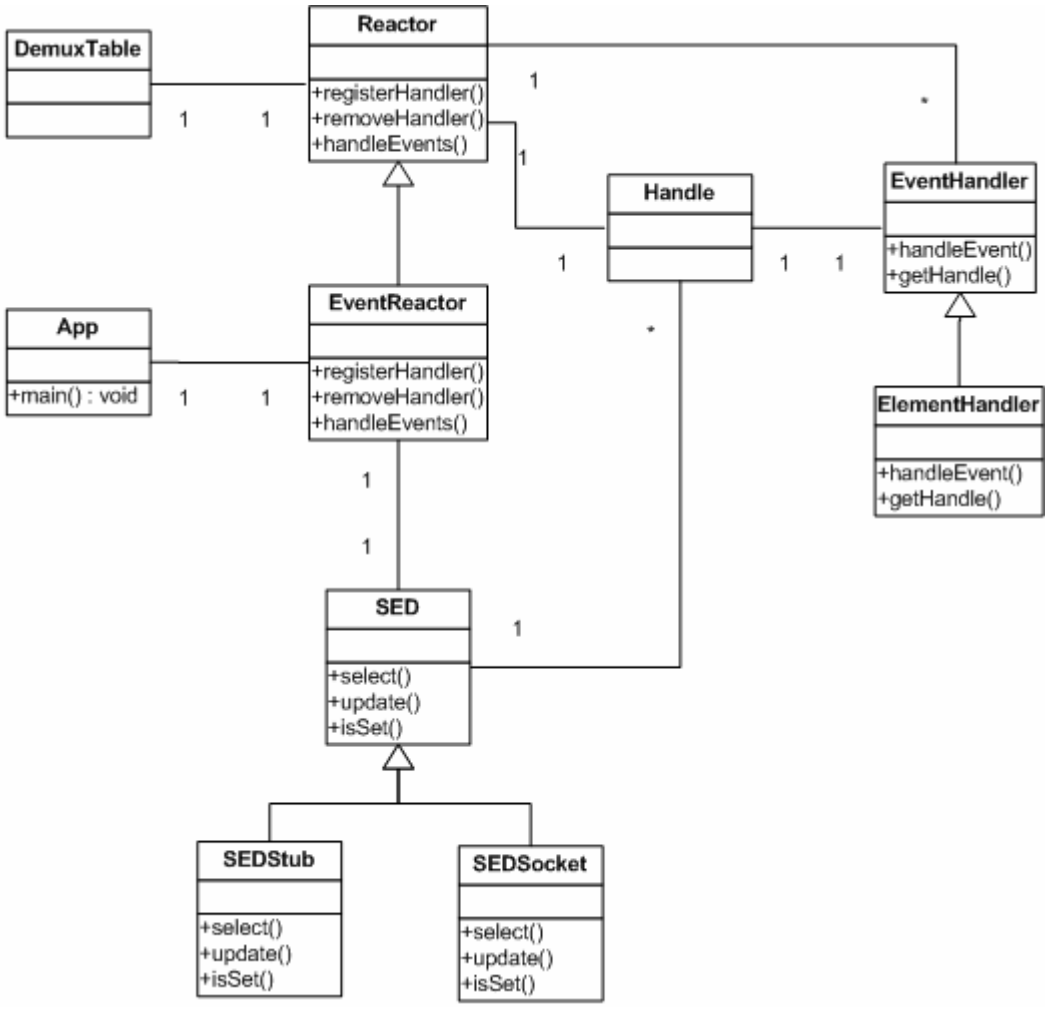

Figure 25: Reactor pattern

3.2.1.2 Description of classes

DemuxTable

The class contains an array of event handlers. This array is used when the SED class needs to demultiplex incoming events.

Reactor

The Reactor class registers different event handlers by placing them, together with event handler's types, inside of DemuxTable class array.

The Reactor class triggers SED class to listen for incoming events. When an event is received, the Reactor class is calling the appropriate EventHandler (here only the subclass ElementHandler is used).

ElementHandler

ElementHandler class is specialized from the EventHandler class.

The class is called from Reactor class, and it is responsible for picking up the data from the socket and handle data.

One of the strong sides of Reactor pattern is showed in this case. If we need to add more elements to our system, it is very easy. All we have to do is to add a new subclass of the EventHandler class.

EventHandler

Is an abstract class which defines an interface that all event handlers should obey.

SED

The SED class is responsible for demultiplexing of incoming events. The class binds handles together with file descriptors which are used to identify sockets with input. This identification happens with help of select() method. The class identifies what type of event the client has sent and gives the message to the Reactor about which event handler should be called to handle the event.

3.2.2 Acceptor/Connector pattern

3.2.2.1 Introduction

The purpose of Acceptor/Connector pattern [Ref. 3, s. 285] is to decouple the connection and initialization of cooperating peer services in a networked system from the processing performed by the peer services after they are connected and initialized. The overall classes in the pattern are shown in Figure 26.

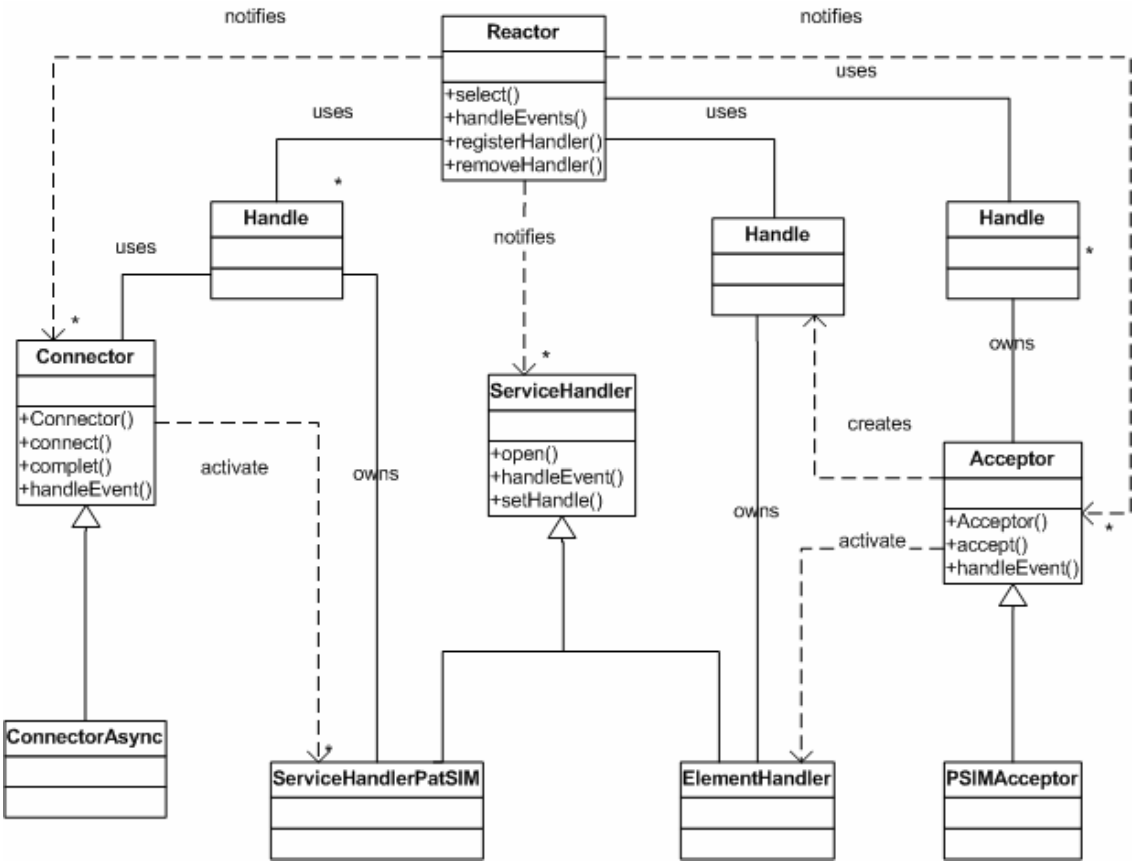

Figure 26: Acceptor/Connector pattern

3.2.2.2 Description of classes

Reactor

Reactor class is responsible for demultiplexing. If a connection event occurs, the Reactor class takes care of the handleEvent() from Connector class is called.

ServiceHandlerPatSIM, ElementHandler

Specializations of the ServiceHandler class. These classes are responsible for streaming of data between to peers when the connection is established.
Objects of those classes are created from Acceptor and Connector objects.

Acceptor

Abstract class which is responsible for the passive part of establishing of a connection. This happens when the class accepts the connection request.

Connector

Abstract class which is responsible for active part of establishing of a connection. This happens when the class sends a connection request.

ServiceHandler

Super class for the concrete service handler classes. The class handles streaming of data when the connection is established.

3.2.3 Leader/Followers pattern

3.2.3.1 Introduction

The purpose of Leader/Followers pattern [Ref. 3, s. 447] is to provide an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on the event sources.

As mentioned before, the Reactor pattern gives an effective solution to demultiplexing and expedition of service events. But the Reactor pattern is single threaded; so many clients cannot be serviced in parallel, reducing efficiency of the system.
For instance, if the server runs as single threaded, it cannot act on alarms sent from more then one client at the same time. In this case the requests (alarms) will be handled one by one, which is not satisfying for this system.
That is why it was necessary to introduce Leader/Followers which opens up for a multithreaded model allowing the server to process more than one event at a time.

The overall classes in the pattern are shown in Figure 25.

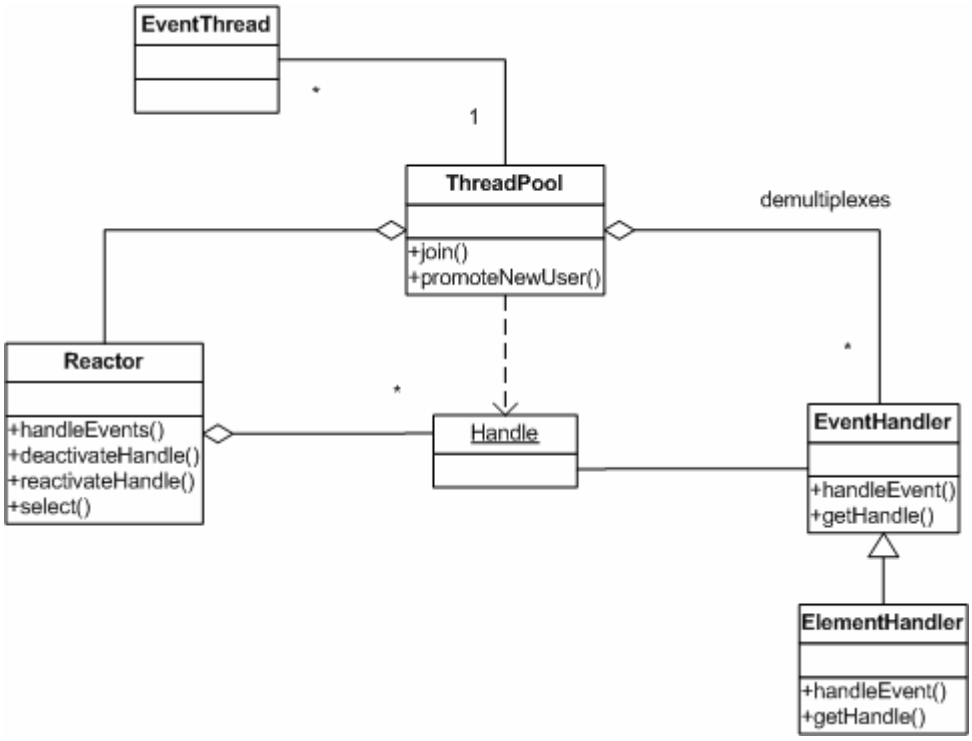

Figure 27: Leader/Followers pattern

3.2.3.2 Description of classes

Reactor

The Reactor class contains a number of Handle objects, which are used when the system is waiting for incoming events. All incoming events are detected by the select() method.

ThreadPool

The ThreadPool class contains a group of threads (objects).
These threads change between being leader and followers. A Leader waits for an incoming event and when the event occurs, it is processed by the thread that is requested and afterwards the next thread in the list is promoted to be the leader.
Hereby the old leader thread works on the event by multiplexing and expedition of the event to the proper event handler, while the new leader thread can handle new events.
After the old leader has finished the handling of event, it becomes a follower.

This is described in the state diagram in Figure 28.

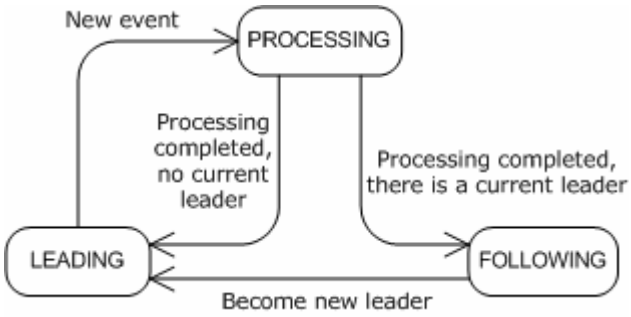

Figure 28: State of threads in the ThreadPool

ElementHandler

The ElementHandler class is a specialization of EventHandler class. The class is called from Reactor class, and it is responsible for picking up the data from the socket and thereby handles data.

When the event has been read, it deactivates itself in the reactor, and promotes a new leader. The reason for reading before promoting is to ensure that the next thread does not detect the same event.

The fact that it's deactivated while the event is being handled, prevent other threads from detecting events on the handle which is owned by the *EventHandler*.

After the event has been handled it will reactivate itself in the reactor, and thereby other leader-thread will be able to detect event for that *EventHandler*.

EventHandler

Abstract class which defines an interface to an event handler.

3.2.3.3 DPSIMU2 Leader/followers extension

There are several reasons for the ordinary pattern not being as efficient as it could be:

1. It does not allow more than one event from a single host to be handled concurrently.
2. The select mechanism in the socket API does not provide any interrupt methods, so implementation of the *reactivate* method would have to rely on timeouts. Therefore worst-case-scenario would be that only "1/timeout" events could be handled.

The implementation in this project has resulted in a simple change to accommodate this problem. Instead of calling *activate/reactivate*, a monitor is used.

The following table illustrate the changes made in the *EventHandler*, compared with the method shown in the POSA2 book [Ref. 3, s. 447]. The table shows a sequence, so the first line is execute first and so on:

POSA2 solution	**DPSIMU2 solution**
Read event from socket	Read event from socket
Deactivate 'this' in reactor	Enter monitor
Promote new leader	Promote new leader

Handle read event	Handle read event
Reactivate ‘this’ in reactor	Exit monitor

This change enables new leaders to detect events on the handle owned by the event handler. But there is still no chance of reading the same event twice, since promoting is still done after reading.

If the new leader detects an event while the old leader is this still processing the previous event, it will wait for the monitor lock.

Using this method, events will be processed in the sequence they arose, but if this is not a demand, the monitor can be omitted (which then requires event handling code to be thread-safe).

A disadvantage with this approach is that if a deadlock should occur while handling an event and the current leader detects an event on the handle owned by that event handler, then the leader will try to enter the monitor which it cannot. So no new leader is promoted, and the entire system deadlocks. In the original method a deadlock would only mean that one thread is locked, and no further events can be read on that connection. But if it happens frequently, it will also fail when there is no more thread in the *ThreadPool*.

3.2.3.4 Façade pattern

The purpose of the Façade design pattern [Ref. 2] is to provide a unified interface to the PSIMU system.

The class `ConfigFacade` is used for this purpose, as it is that way the PSIMU is controlled from the outside, that is both from the PSIMU itself (by clicking on the GUI) or from the CSIMU.

The Façade pattern is supplied with the Singleton pattern, as the `ConfigFacade` class is a Singleton. This way the *View* class (with the GUI implemented) and the *ServiceHandlerPatSim* class (that handles events sent from CSIMU) is able to interact with the same *ConfigFacade* instance, as the following code snippets show:

ServiceHandlerPatSim.cpp:

```
case EVENT_SET_PATIENT:
   {
      int pat = atoi(ee->getData());
      if ((pat > 0) && (pat <= 5)) {
        ConfigFacade::getInstance()->setPatient(pat);
      }
   }
```

View.cpp:

```
SIGNED View::Message(const PegMessage &mesg) {
  switch (mesg.wType) {
    case SIGNAL(ID_PATIENT1, PSF_CLICKED):
      ConfigFacade::getInstance()->setPatient(1);
      break;
    case SIGNAL(ID_PATIENT2, PSF_CLICKED):
      ConfigFacade::getInstance()->setPatient(2);
      break;
     ...
}
```

3.3 Distributed communication

The setup of the DPSIMU system implies some communication is needed between PSIMU and CSIMU. For this reason, an *EventElement* class has been introduced. This class represents an element that can be transmitted from PSIMU to CSIMU or the other way. The element is quite simple, as it consists of the following:

- Command (variable cmd) as an int (4 bytes)
- Data (variable data) as a char* (always 20 bytes)

This means that an *EventElement* is always 24 bytes long no matter what is sent.
The possible commands are specified in an *enum*:

```
enum {
  EVENT_DEBUG       = 0,
  EVENT_LOG         = 1,
  EVENT_SET_HB      = 2,
  EVENT_SET_PATIENT = 3
};
```

The method *writeElement(SocketStream* stream)* marshals data to the socket stream to the remote host:

```
void EventElement::writeElement(SocketStream* stream) {
   char chars[sizeof(EventElement)];
   memcpy(chars, (void*)this, sizeof(EventElement));
   stream->sendBuffer(chars, sizeof(EventElement));
}
```

The method *instantiatedElement(SocketStream* stream)* is used to unmarshals data sent from the remote host. The received data are then instantiated into the *EventElement* instance.

The *EventElement* is used several places:

- `EventHandler` on CSIMU that receives events sent from PSIMU
- `ServiceHandlerPatSim` on PSIMU that receives events sent from CSIMU
- `ConnectionStatus` on CSIMU that sends events to PSIMU
- `CSIMView` on CSIMU that sends events to PSIMU
- `RemoteLogger` on PSIMU that sends events to CSIMU

3.4 Code

All code of this system is to be found on the enclosed CD-ROM.
The code has been documented using a documentation tool, Doxygen, to make a browsable HTML edition. This documentation is also placed on the CD-ROM.

4 System testing

4.1 Introduction

The test of the DPSIMU2 system has been made in the following manners:

- Code walkthroughs
- Stubs to test integration of CSIMU and PSIMU
- Functional test verifying the DPSIMU2 system against the requirements specified in section 1.3.

4.2 Code walkthroughs

During the implementation phase, informal code walkthroughs [Ref. 4, p. 115] have been used. The purpose of these has been to check specific code functionalities. Both individual and group code walkthroughs have been performed.
The outcome of these tests has been very informative of possible problems in the implementation of the design.
As informal walkthroughs have been performed, no specific log has been made, because the needed changes have been performed right away.

4.3 Integration testing

A stub has been generated that can act as a PSIMU on the same computer as the CSIMU with the purpose to test the integration between CSIMU and PSIMU.
A simple communication stub has also been developed to test the communication from PSIMU to CSIMU.
The deployment diagrams for these test setups are shown in 2.7 (Figure 21 and Figure 22).
Furthermore the full implementations of PSIMU and CSIMU have been tested with success against each other (as can be seen in the following section).

4.4 Verification

Verification [Ref. 4, p. 118] is performed to test the developed system against the requirement specification in section 1.3.

4.4.1 Use case 1: Discover and register server

Step	Action/Input	Expected result	CHECK
1.	Locate CSIMU	The CSIMU is located	√
2.	Connect to CSIMU	A connection to CSIMU is established	√
3.	If connected, show it on the local user interface	Indication of a successful connection shown on GUI	√

4.4.2 Use case 2: Set patient data

Step	Action/Input	Expected result	CHECK
1.	The PSIMU receives a set patient request	The request is received on PSIMU	√
2.	The PSIMU changes to the selected patient record	The PSIMU outputs the correct new patient signals	√
3.	A log message is sent to CSIMU informing about a new patient record	Event message sent to CSIMU	√

4.4.3 Use case 3: Set heartbeat factor

Step	Action/Input	Expected result	CHECK
1.	The PSIMU receives a set heartbeat factor request	The request is received on PSIMU	√
2.	The PSIMU changes to the selected heartbeat factor	The PSIMU outputs with the correct new speed	√
3.	A log message is sent to CSIMU informing about a new heartbeat factor	Event message sent to CSIMU	√

4.4.4 Use case 4: Send event message

Step	Action/Input	Expected result	CHECK
1.	PSIMU sends an event message to CSIMU	A message sent from PSIMU to CSIMU and shown on CSIMU	√

4.4.5 Use case 5: Register patient simulator

Step	Action/Input	Expected result	CHECK
1.	Accept the request for connection	The CSIMU accepts the connection request	√
2.	Connection to the requesting PSIMU established	The connection is established successfully	√
3.	The CSIMU GUI is updated making it	CSIMU now shows the connected PSIMU on its GUI	√

	possible to control that specific connected PSIMU		

4.4.6 Use case 6: Select patient data

Step	Action/Input	Expected result	CHECK
1.	Remote user has selected a patient	A new patient data set is selected	√
2.	Information about the selected patient has been sent to the PSIMU	A message has been sent to PSIMU about which patient data set that was selected	√

4.4.7 Use case 7: Select heartbeat factor

Step	Action/Input	Expected result	CHECK
1.	A new heartbeat factor has been selected	A new factor is selected	√
2.	Information is sent to the selected PSIMU	A message has been sent to PSIMU about the heart beat factor that was selected	√

4.4.8 Use case 8: Receive and show event message

Step	Action/Input	Expected result	CHECK
1.	CSIMU receives an event message from PSIMU	The event message is received from PSIMU	√
2.	The event message is displayed on the screen	The message is displayed on the CSIMU GUI	√

4.5 Final graphical user interfaces

4.5.1 PSIMU

The GUI on PSIMU is not much changed from the IRTS project. Figure 29 shows the final GUI on PSIMU. As can be seen, a label with “Connected” is added compared with the IRTS GUI for PSIMU. This label shows that a successful connection has been established to CSIMU. Notice here that PSIMU can be controlled from both the PSIMU itself (as always) and from CSIMU.

The black area is still used for the curves of ECG and EDR.

Figure 29: GUI on PSIMU

4.5.2 CSIMU

Figure 30 shows the GUI on CSIMU. As can be seen, a panel for each connected PSIMU is shown allowing separate control of each of the connected PSIMUs.

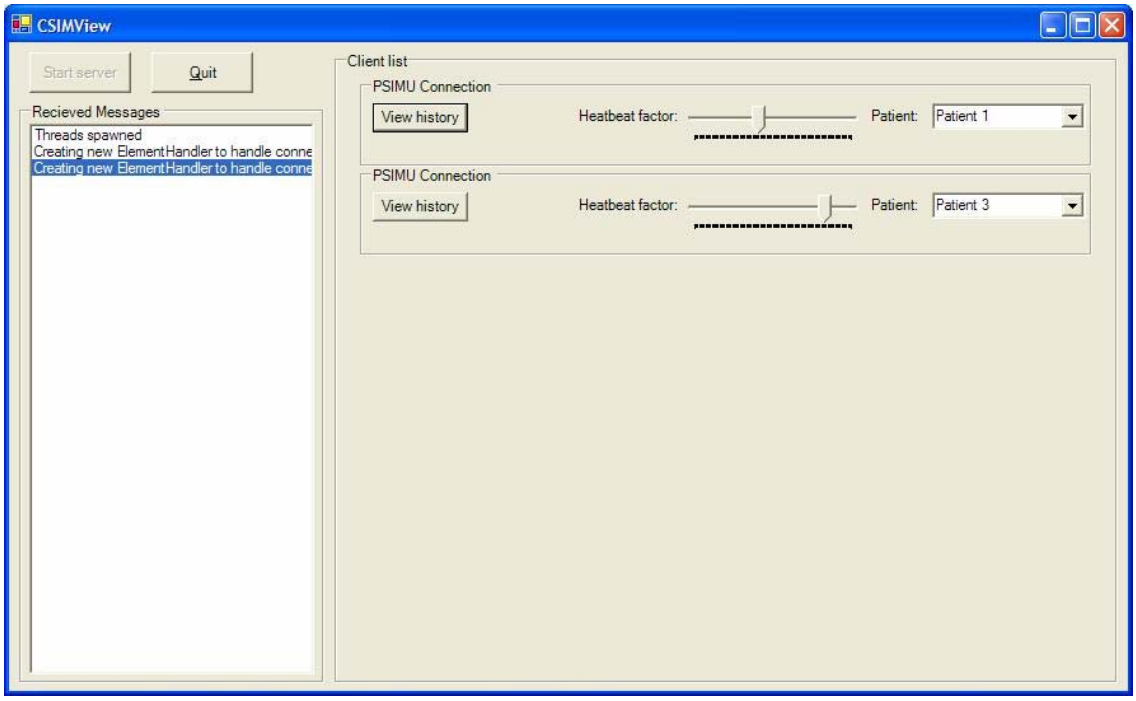

Figure 30: GUI on CSIMU

5 References

Ref. 1: B. P. Douglass: Doing Hard Time, Addison-Wesley, 1999, ISBN: 0201498375

Ref. 2: E. Gamma et al: Design Patterns, Addison-Wesley, 1994, ISBN: 0201633612

Ref. 3: Schmidt, Douglass: Pattern-oriented Software Architecture – Patterns for Concurrent and Networked Objects, 2004, ISBN: 0471606952

Ref. 4: Sean Beatty, Embedded System Programming: "Sensible SoftwareTesting", August 2000

Ref. 5: Project documentation for IRTS project