Patient Simulator

PSIMU2

Project documentation

Requirement specification
Analysis and design
Implementation
Test

”Teknisk IT” – spring 2004

Project by PSIMU2:

Henrik Jensen
20022464

Almir Mesanovic
20023894

Jan Lauritzen
20034439

Mads Pedersen
20034440

TABLE OF CONTENTS

TABLE OF FIGURES

Word list

Word/abbreviation	Description
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

This document has been constructed starting with the requirement specification that specifies all requirements of the 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.

The document is primarily intended for system developers. Furthermore it should be widely used in the following project in the course “TI-DRTS”.

1 Requirement Specification

1.1 Introduction

1.1.1 Purpose

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.

1.1.2 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/)

1.2 Overall description

1.2.1 System Description

The purpose of this project is to implement a prototype of a simulation system 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.

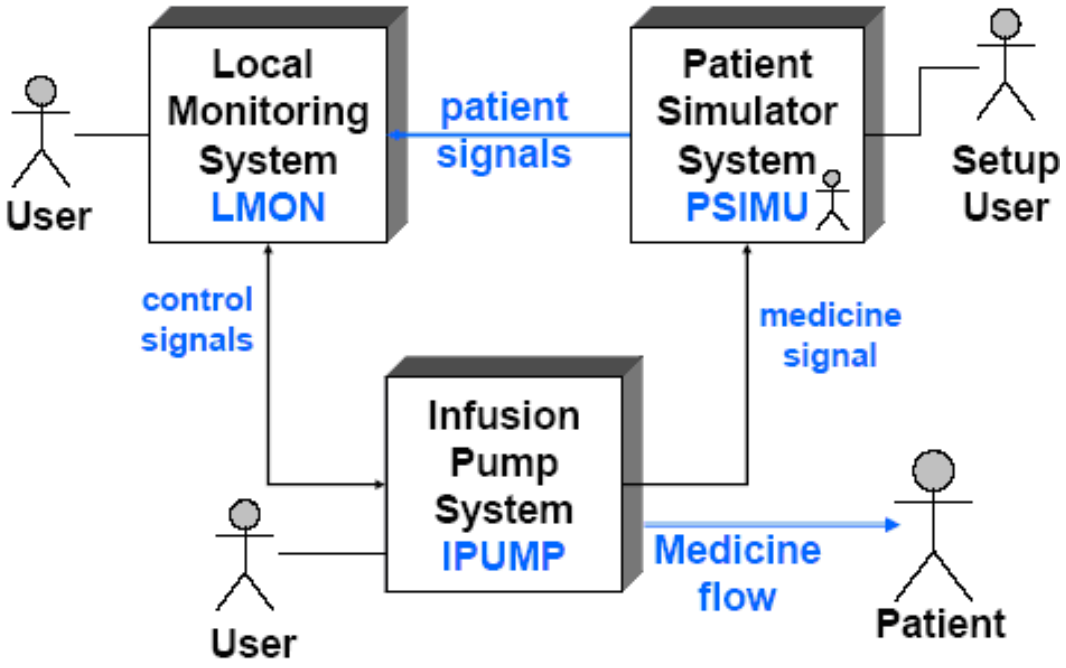


Figure 1: System Overview Diagram

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.

The system will be constructed using an embedded hardware platform for real-time systems (SBC686) giving the users access to software features using a touch-screen display as input and output. The software will be based upon on the RTK operating system from OnTime.

A two channel digital-to-analogue converter will be used for the analogue output signals and an 8-bit digital port for the pulse emitter.

The connection between the infusion pump and the simulation system will be established using a RS232 communication link.

To prepare future integration in a distributed environment, the simulator system will be prepared to allow for this type of expansion.

1.2.1.1 Actor-Context Diagram

The following diagram (Figure 2) shows the patient simulation subsystem as an independent unit, describing external systems and users as actors, using the UML convention. The purpose of this diagram is to identify system boundaries and external participants, both hardware and humans.

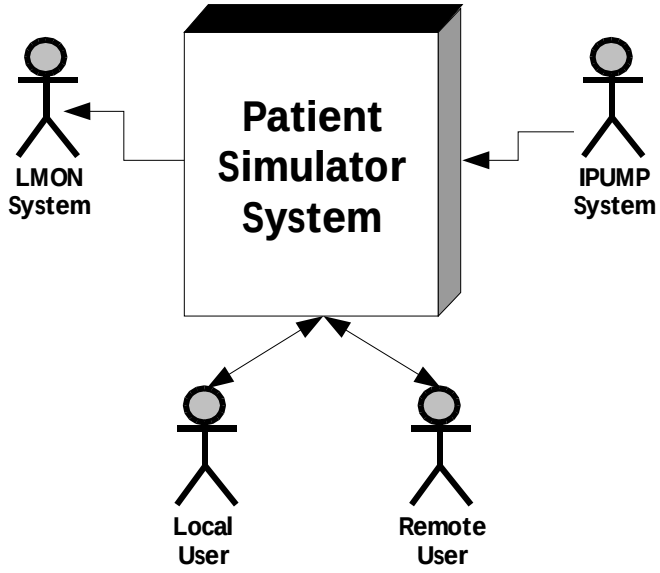


Figure 2: Actor-Context Diagram

1.2.1.2 Actor Descriptions

Actor name	**Monitor**
Type [primary/secondary]	Primary
Description	A patient monitor is responsible to sample from analogue output channels (ECG and EDR) with whatever frequency the monitor finds suitable. PSIMU responsibility towards monitor is that output data should be within the boundaries described in the interface specification.
Number of concurrent actors	One can be expected, although the electrical output can be used by as many receivers as physical possible.

Actor name	**Local operator**
Type [primary/secondary]	Primary
Description	Local operator can select different patient signals and change the rate at which these are outputted. Furthermore the local operator can see the outputted signals on the display.
Number of concurrent actors	1

Actor name	**Remote operator**
Type [primary/secondary]	Primary
Description	Remote operator can remotely control the simulator system by changing patient signals etc.
Number of concurrent actors	1

Actor name	**Pump**
Type [primary/secondary]	Secondary
Description	The Infusion Pump System can send information to the simulator system, forcing a regulation of the outputted patient signals.
Number of concurrent actors	1

1.2.2 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 1.3.

1.2.2.1 Use Case Diagram

The following use case diagram illustrates all the use cases that the patient simulator is composed of:

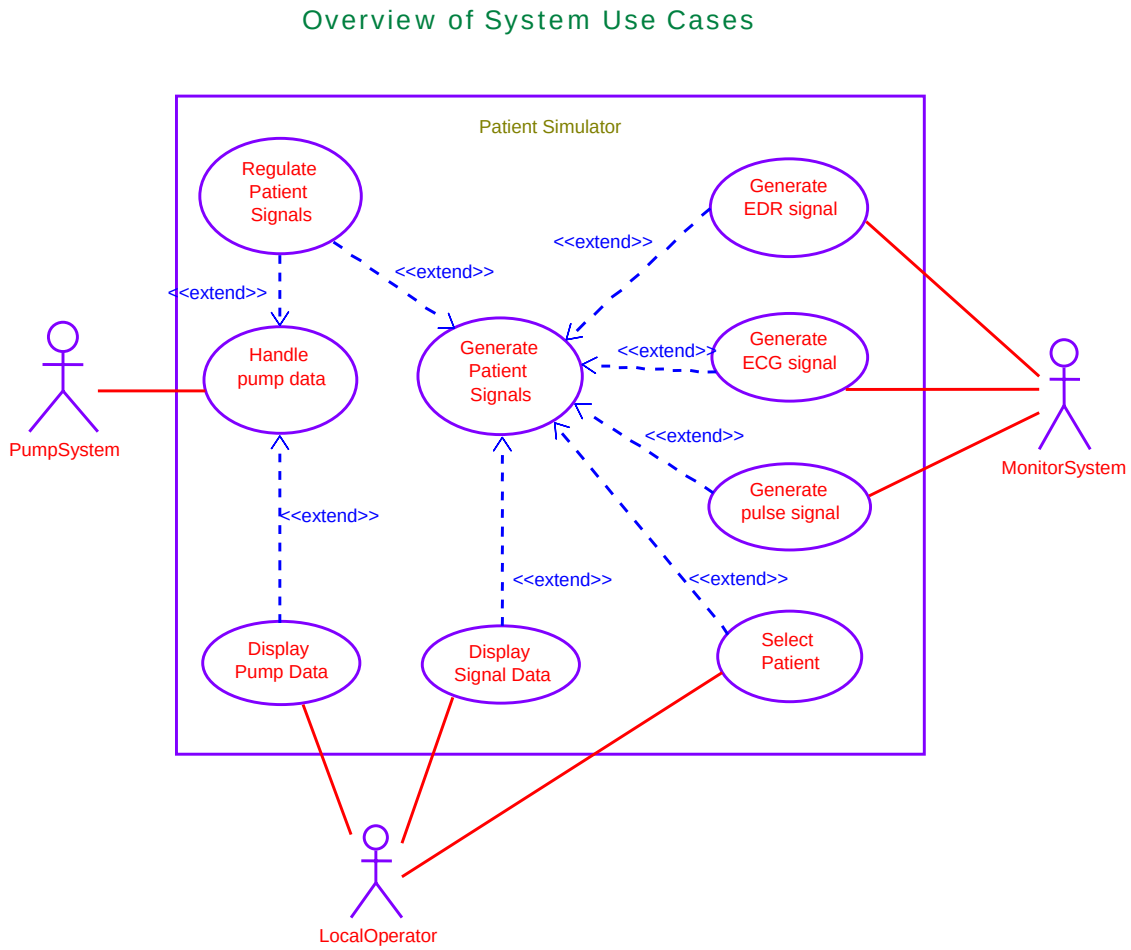


Figure 3: Use Case Diagram

1.2.3 System Constraints

The patient simulator prototype 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.

1.2.4 Future of the System

It can at present be predicted that the system will be integrated in a larger distributed system and as such will be using other means of communication.
Additional external equipment for patient stimulation and simulation of other patient life signals can at this stage also be thought of.

1.2.5 User Characteristics

The system identifies two direct users of the patient simulator system, described with the actors, local user and remote user. The remote user is only included to show the possible expansion of the local user's activity area to a remote location,

e.g. another room or building. 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.

1.2.6 Development Process Requirements

The fictitious customer from the Project Description (World Wide Medico) has the following requirements to the development process:

- The project will consist of a minimum of two deliveries, based upon the defined use cases.
- The project group will supply an architectural model which covers the defined requirements.
- The first delivery should be designed, implemented and tested using object oriented technology.

1.2.7 Customer Deliveries

After approval of this document, the two deliveries will consist of the following components:

- 1st iteration (26th of March 2004):
  - Requirements specification
  - System architecture and design documentation for the use cases described in 1.9

- 2nd iteration (24th of May 2004):
  - System design documentation for the entire project
  - Software design documentation for the entire project
  - Executable software
  - Software code documentation for all use cases
  - Development report describing project development

Architecture, design and instructional documentation will be delivered in paper format and software documentation will be supplied on a CD-ROM media.

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 make up the Patient Simulator.

1.3.1 Use Case 1: Generate Patient Signals

Goal	The purpose of this super use case is to generate patient signals for the Monitorsystem.
Initiation	From inside the system
Actors and stake holders	Monitor
No of concurrent instances	1
Frequency	This continuous process runs for the entire lifetime of the application.
Non functional requirements	N/A.
References	None
Preconditions	None
Post conditions by success	N/A.
Post conditions by failure	None.
Main scenario	1. On startup, a default patient is selected 2. Available signals for the specified patient are emitted to the respective hardware
Extensions	[*Extension point*: Signal] Allows the extension of available patient signals [*Extension point*: Display] Allows the emission of display events [*Extension point*: Select] Allows the selection of what patient data to use [*Extension point*: Regulate] Allows for the extensions to manipulate the output

1.3.2 Use Case 2: Generate ECG Signal

Goal	To deliver a continuous pre-recorded PhysioBank compliant ECG data signal on an analoque-to-digital converter.
Initiation	This use case is initiated from inside the system.
Actors and stake holders	Monitor
No of concurrent	1

instances	
Frequency	This continuous process runs for the entire lifetime of the application.
Non functional requirements	N/A
References	This use case extends the **Generate Patient Signal** use case on the *signal* port.
Preconditions	Analogue output channel initialized, and data file are present.
Post conditions by success	A continuous analogue signal is emitted.
Post conditions by failure	A warning message is shown on the display. Analogue output is interrupted.
Main scenario	1. Read value from data file. *[Failure: Reading from data file]* *[Failure: No more data in file]* 2. Write value to analogue output. 3. Wait until next value will be output 4. Repeat from step 1 *ad infinitum*
Extensions	*[Failure: reading from data file]* a. A warning message is shown to the local user. b. Output is interrupted *[Failure: No more data]* a. The file is restarted

1.3.3 Use Case 3: Select Patient

Goal	This use case allows for the local operator actor to select which patient data should be displayed.
Initiation	Initiated by the local operator
Actors and stake holders	Local operator, Monitor
No of concurrent instances	1
Frequency	Estimated to 100 times per day
Non functional requirements	
References	This use case extends the **Generate Patient Signal** use case on the *select* port.
Preconditions	User data for 5 patients has been installed on the target system. This data should comply with the PhysioBank specifications. A graphical button has been drawn on the display for each available patient.

Post conditions by success	A patient has been selected and its data is being output on the system hardware.
Post conditions by failure	N/A
Main scenario	1. The local operator activates a patient by pressing a graphical button on the touch screen. 2. The system changes to the specified patient and the data for this patient is now emitted on the output hardware
Extensions	N/A

1.3.4 Use Case 4: Handle Pump Data

Goal	The purpose of this use case is to receive data from an external infusion pump system connected to the RS232 serial communication port (COM2), for controlling system behaviour related to medicine infusion
Initiation	Periodical transmission from pump system
Actors and stake holders	Pump System
No of concurrent instances	1
Frequency	Once every second
Non functional requirements	
References	TI-RTS Project Interface Specification (version 16.02.2004) for PDU format
Preconditions	An external infusion pump system is connected to the PSIMU system and is operating
Post conditions by success	Data from the pump has been received correctly
Post conditions by failure	
Main scenario	1. The system listens on the COM2 port for incoming PDUs 2. The system receives a communication event on the COM2 port and extracts the data content *[Failure: Data checksum]* 3. The PDU is stored for usage by the system
Extensions	*[Failure: Data checksum]*: The system discards the package; last received PDU is the current valid reading

1.3.5 Use Case 5: Display Patient Signals

Goal	Display data to the screen that is being output to the system hardware.
Initiation	Initiated by the local operator
Actors and stake holders	Local operator
No of concurrent instances	1
Frequency	Up to 1000 times per second
Non functional requirements	
References	This use case extends the **Generate Patient Signal** use case on the *display* port.
Preconditions	The generate patient signal use case is active and producing output; A graphical chart widget for displaying analogue data has been initialized (used for EDR and ECG signals); A graphical widget for displaying numerical data has been initialized (used for pulse signals);
Post conditions by success	The output data is correctly displayed on the screen.
Post conditions by failure	N/A.
Main scenario	*Analogue data scenario:* 1. An analogue signal sample (EDR or ECG) is generated by the *Generate Patient Signal* use case 2. The sample is displayed on the screen using the chart widget. A distinct colour line is maintained for each analogue output channel. *Digital data scenario:* 1. A digital signal sample (pulse) is generated by the *Generate Patient Signal* use case 2. The sample is displayed on the screen using the numerical widget.
Extensions	N/A

1.3.6 Use Case 6: Generate EDR Signal

Goal	To deliver a continuous simulated EDR data on an analogue-to-digital converter based on a prerecorded PhysioBank compliant ECG signal.
Initiation	This use case is initiated from inside the system.
Actors and stake	Monitor

holders	
No of concurrent instances	1
Frequency	This continuous process runs for the entire lifetime of the application.
Non functional requirements	N/A
References	This use case extends the **Generate Patient Signal** use case on the *signal* port.
Preconditions	Analogue output channel initialized, and data file are present.
Post conditions by success	A continuous analogue signal is emitted.
Post conditions by failure	A warning message is shown on the display.
Main scenario	1. Read value from data file *[Failure: Reading from data file]* *[Failure: No more data in file]* 2. Write value to analogue output 3. Wait until next value should be outputted 4. Repeat from step 1 *ad infinitum*
Extensions	*[Failure: Reading from data file]* a. A warning message is shown to the local user b. Output is interrupted *[Failure: No more data]* a. The file is restarted

1.3.7 Use Case 7: Regulate Patient Signals

Goal	The purpose of this use case is to apply simulated deviations to the output data, based on the external stimuli supplied by external systems, mainly the pump system or by an internal regulation
References	This use case extends the **Handle Pump Data** use case
Main scenario	1. A valid PDU has been received 2. An algorithm for regulating the output signals, based on the PDU data is executed and the result stored
Extensions	None

1.3.8 Use Case 8: Generate Pulse Signal

Goal	Output a digital 8-bit signal simulating a pulse in the range 0-250 beats/min
Initiation	System initiated
Actors and stake holders	Monitor
No of concurrent instances	1
Frequency	Once every second
Non functional requirements	N/A
References	This use case extends the **Generate Patient Signal** use case on the *signal* port.
Preconditions	N/A.
Post conditions by success	Digital pulse signal correctly outputted
Post conditions by failure	Error message shown on local display
Main scenario	1. Read value from data file *[Failure: Reading from data file]* *[Failure: No more data]* 2. Write value to digital output 3. Wait for next value 4. Repeat step from step 1 *ad infinitum*
Extensions	*[Failure: Reading from data file]* a. Warning message is shown to the local user *[Failure: No more data]* a. The file is restarted

1.3.9 Use Case 9: Display Pump Data

Goal	This use case displays the data received from the connected pump system on the display. That is medicine name and accumulated volume.
Actors	Local operator
References	This use case extends the **Handle Pump Data** use case
Main scenario	1. A PDU has been received successfully by the *Handle Pump Data* use case 2. The name of the medicine is displayed on the screen 3. The volume of the medicine is displayed on the screen
Extensions	-

1.4 External Interface Requirements

1.4.1 User Interfaces

The user interface to PSIMU consists of a single dedicated screen. Users can be expected to have basic knowledge on interpretation of output values. The display constantly shows the state of all output channels. Additionally it is possible to change the behaviour of output, and thereby simulate a patient's different states (heart failure and so forth).

A prototype of the graphical user interface is shown in Figure 4.

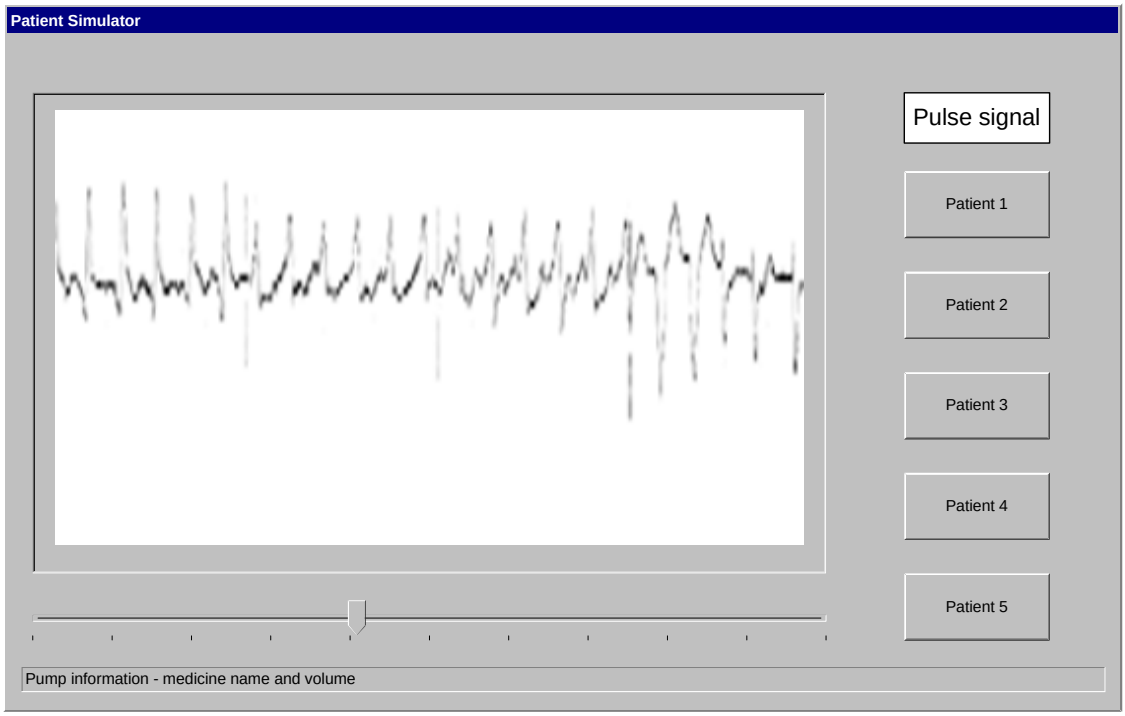


Figure 4: GUI prototype

It must here be possible to view the two analogue outputs (ECG and EDR signals) on two different graphs and the digital pulse as a number. The buttons are used to change patient data. The slider can be used for changing the interval with which the signals are generated. Finally the pump information is shown with medicine name and volume.

1.4.2 Hardware Interfaces

The system is specified to execute on a SBC embedded computer. This computer is dedicated to this system only.

Output: To provide analogue output, a PV2019 is used. For digital output, an IO686 is used.

Input: A RS232 is used for receiving medicine data from IPUMP.

1.4.3 Communication Interfaces

The hardware communication to receiving medicine data from IPUMP is RS232. The protocol is specified in the interface specification.

1.4.4 Software Interfaces

The system uses On Time's RTKernel (RTK) as operating system, as it supports the hardware platform. With RTK comes a set of APIs that will be used, including RT Peg for graphical user interface.

1.5 Performance Requirements

Since the hardware platform is dedicated to this system, there is no need to preserve resources for other processes.

The system is required to provide a continuous signal on analogue output channels and a pulse signal on a digital channel every second. These signals are based on the PhysioBank specifications used in their PhysioToolkit software package. These use typical sample rates of 125 to 360 samples per second.

Additionally it is necessary to be able to read a single PDU every second.

1.6 System Qualities

The quality factors of the system are put in Figure 5.

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 reuseablility are very important issues, since the system is expected to be extended later on. These priorities should manifest through modelling.

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

Figure 5: Quality Factors

1.7 Design Constraints

Each one of the iterations should be developed according to the ROPES development process.

1.8 Other Requirements

1.8.1 Authority Requirements

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

1.9 Part deliveries

There are two deliveries, the first with 'basic functionality' as headline, the next is a complete system ready for use.

The first iteration will contain architecture components relating to all external systems so as to have a working prototype. It will be composed of the use cases:

1: Generate Patient Signals
2: Generate ECG Signal
3: Select Patient
4: Handle Pump Data

The second iteration will complete the prototype to a fully functional patient simulator by adding the following use cases:

5: Display Signal Data
6: Generate EDR Signal
7: Regulate Patient Signals
8: Generate Pulse Signal
9: Display Pump Data

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 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.

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 1.2.1 of that document.

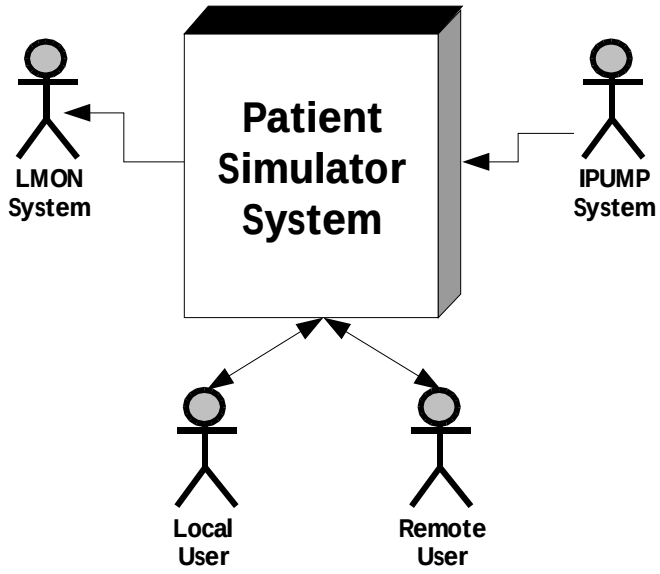


Figure 6: System context

2.2.2 System introduction

See Requirement Specification, section 1.2.1

2.3 System Interfaces

2.3.1 Interface to human actors

This subsection describes the Man Machine Interface (MMI) for the human actors interfacing with the system.
The local user can communicate with the system with a mouse.
In future versions, it is planed to use a touch screen.

2.3.2 Interface to external system actors

- PSIMU system generates continuous 12 bit analogue signals and a digital signal to LMON system
- A fixed 64 bytes ASCII PDU is transmitted from IPUMP system to PSIMU system.
- This PDU is transmitted every second when the pump is started

PDU format:

- 4 bytes start frame (##?*)

- 46 bytes medicine name
- 12 bytes volume infused (since started)
- 2 bytes CRC checksum

2.3.3 Interface to hardware actors

- A PV2019 card is used for transmitting analogue and an IO686 is used for transmitting digital data to LMON system.
- An RS232 card is used to communicate with the IPUMP.
- A flash RAM (disk on chip) is used in the SBC686 as a media.
- A mouse and a screen are used to communicate with the local user.

2.3.4 Interface to external software actors

There are no external software actors

2.4 Use Case View

2.4.1 Overview of architecture significant Use cases

The second and current iteration of the project is composed of the use cases shown in Figure 7 and Figure 8. The first iteration is illustrated to show the historical progress of the design.

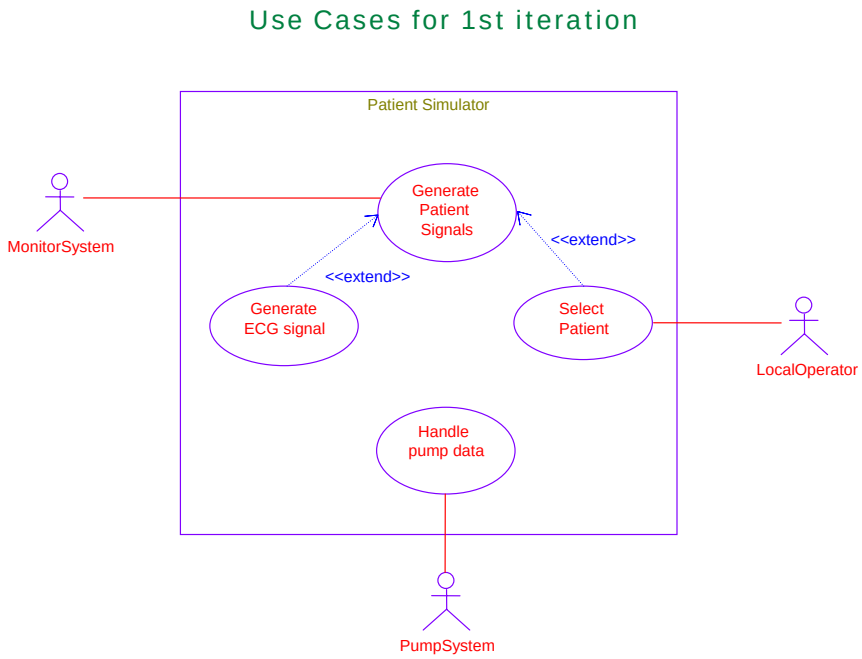


Figure 7: Use cases for 1st iteration

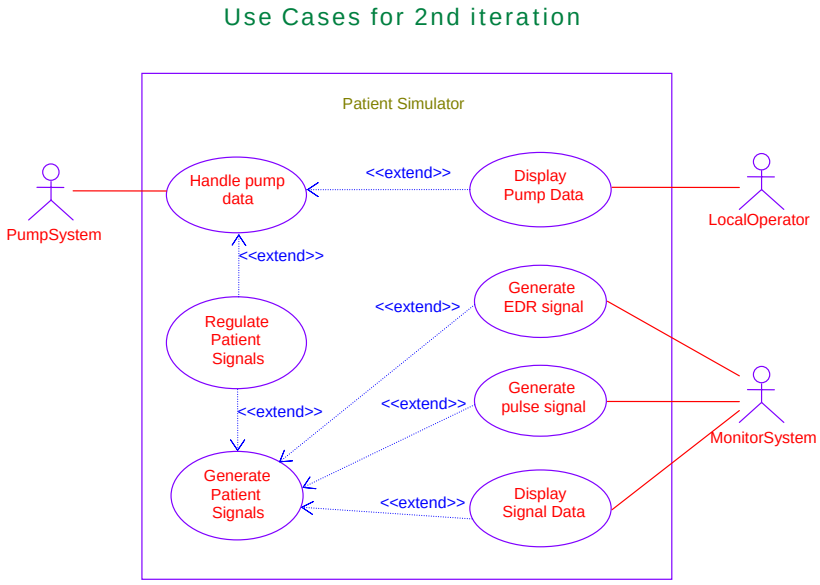


Figure 8: Use cases for 2nd iteration

2.4.2 Use case 1: Generate Patient Signals

2.4.2.1 Use case goal

The purpose of this super use case is to generate patient signals for the Monitorsystem.

2.4.2.2 Use case scenarios

Sunshine scenario:

⇒ On startup, a default patient is selected

⇒ Available signals for the specified patient are emited to the respective hardware

See extended use cases for alternative scenarios.

2.4.3 Use case 2: Generate ECG Signal

2.4.3.1 Use case goal

To deliver a continuous pre-recorded PhysioBank compliant ECG data signal on an analoque-to-digital converter.

2.4.3.2 Use case scenarios

Sunshine scenario:

⇒ Read value from data file.

⇒ Write value to analogue output.

⇒ Wait until next value should be outputted

⇒ Repeat from step 1 *ad infinitum*

Read Failure scenario:

⇒ Read value from data file fails.

⇒ A warning message is shown to the local user.

⇒ Output is interrupted

End of data scenario:

⇒ Read value from data file

⇒ The data was the last sample in the file

⇒ Rollover and restart the file

2.4.4 Use case 3: Select Patient

2.4.4.1 Use case goal

This use case allows for the local operator actor to select which patient data should be displayed.

2.4.4.2 Use case scenarios

Sunshine scenario:

⇒ The local operator activates a patient by pressing a graphical button on the touch screen.
⇒ The system changes to the specified patient and the data for this patient is now emited on the output hardware.

2.4.5 Use case 4: Handle Pump Data

2.4.5.1 Use case goal

The purpose of this use case is to receive data from an external infusion pump system connected to the RS232 serial communication port (COM2), for controlling system behaviour related to medicine infusion

2.4.5.2 Use case scenarios

Sunshine scenario:

⇒ The system listens on the COM2 port for PDU
⇒ The system receives a communication event on the COM2 port and extracts the data content
⇒ The PDU is stored for usage by the system

Bad PDU scenario:

⇒ The system listens on the COM2 port for PDU
⇒ The system receives a communication event on the COM2 port and extracts the data content containing bad data
⇒ The PDU is ignored

2.4.6 Use case 5: Display Signal Data

2.4.6.1 Use case goal

Display data to the screen that is being output to the system hardware.

2.4.6.2 Use case scenarios

Analogue data scenario:

An analogue signal sample (EDR or ECG) is generated by the Generate Patient Signal use case
The sample is displayed on the screen using the chart widget. A distinct colour line is maintained for each analogue output channel.

Digital data scenario:

A digital signal sample (pulse) is generated by the Generate Patient Signal use case.
The sample is displayed on the screen using the numerical widget.

2.4.7 Use case 6: Generate EDR Signal

2.4.7.1 Use case goal

To deliver a continuous simulated EDR data on an analogue-to-digital converter based on a prerecorded PhysioBank compliant ECG signal.

2.4.7.2 Use case scenarios

The scenarios for this use case are identical to use case *Generate ECG Signal*.

2.4.8 Use case 7: Regulate Patient Signals

2.4.8.1 Use case goal

The purpose of this use case is to apply simulated deviations to the output data, based on the external stimuli supplied by external systems, mainly the pump system or by an internal regulation

2.4.8.2 Use case scenarios

Sunshine scenario

⇒ A valid PDU has been received

⇒ An algorithm for regulating the output signals, based on the PDU data is executed and the result stored

2.4.9 Use case 8: Generate Pulse Signal

2.4.9.1 Use case goal

Output a digital 8-bit signal simulating a pulse in the range 0-250 beats/min

2.4.9.2 Use case scenarios

The scenarios for this use case are identical to use case *Generate ECG Signal*.

2.4.10 Use case 9: Display Pump Data

2.4.10.1 Use case goal

This use case displays the data received from the connected pump system on the display. That is medicine name and accumulated volume.

2.4.10.2 Use case scenarios

Sunshine scenario:

⇒ A PDU has been received successfully by the *Handle Pump Data* use case

⇒ The name of the medicine is displayed on the screen

⇒ The volume of the medicine is displayed on the screen

2.5 Logical View

2.5.1 Overview

The overall architecture of the system is the “Five-Layer Architectural Pattern” [Ref. 1]. Figure 9 shows this overall package diagram.

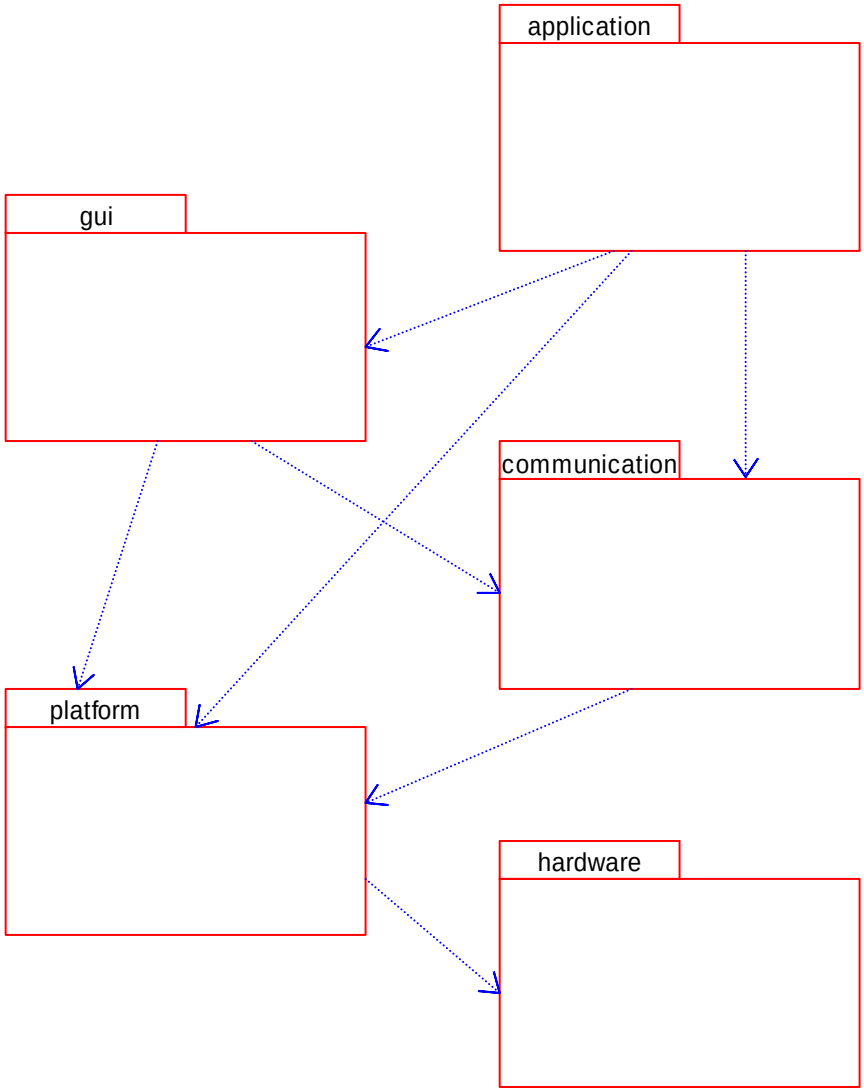
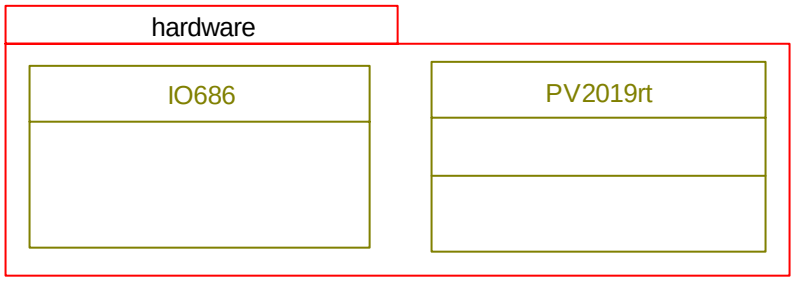


Figure 9: Overall package diagram

2.5.2 Architecturally significant design packages

2.5.2.1 Package: Hardware

This package contains all the direct hardware access logic. For this project this means the C code that accesses the PV2019 and IO686 boards. This is manufacture supplied code and because it is not object-oriented, it cannot be modelled with object or classes. Therefore two classes are used to wrap the functionality of the C code.



Class: IO686

Provides access to a connected IO686 board, which is able to output digital values.

Class: PV2019rt

Interface to the PV2019 I/O card. Notice that it is not developed during this project; it is a handed out class from IHA.

2.5.2.2 Package: Platform

The platform package shown in Figure 10 contains basic abstractions over platform specific functionality. This means that other packages do not need to access the underlying OS directly.

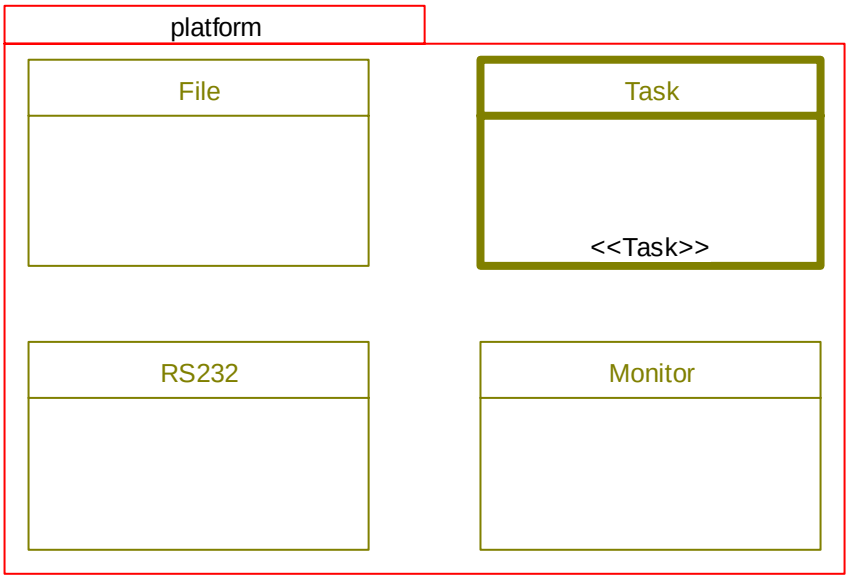


Figure 10: The platform package

Class: Task

A thread abstraction. It handles creation of a thread under RTKernel.

Class: File

A generic class to provide OO access to files.

Class: RS232

This class implements the basic functions for communication with the RS232 port under the RTKernel platform.

2.5.2.3 Package: Communication

The communication package handles all interfacing between the application business logic and the external actors.

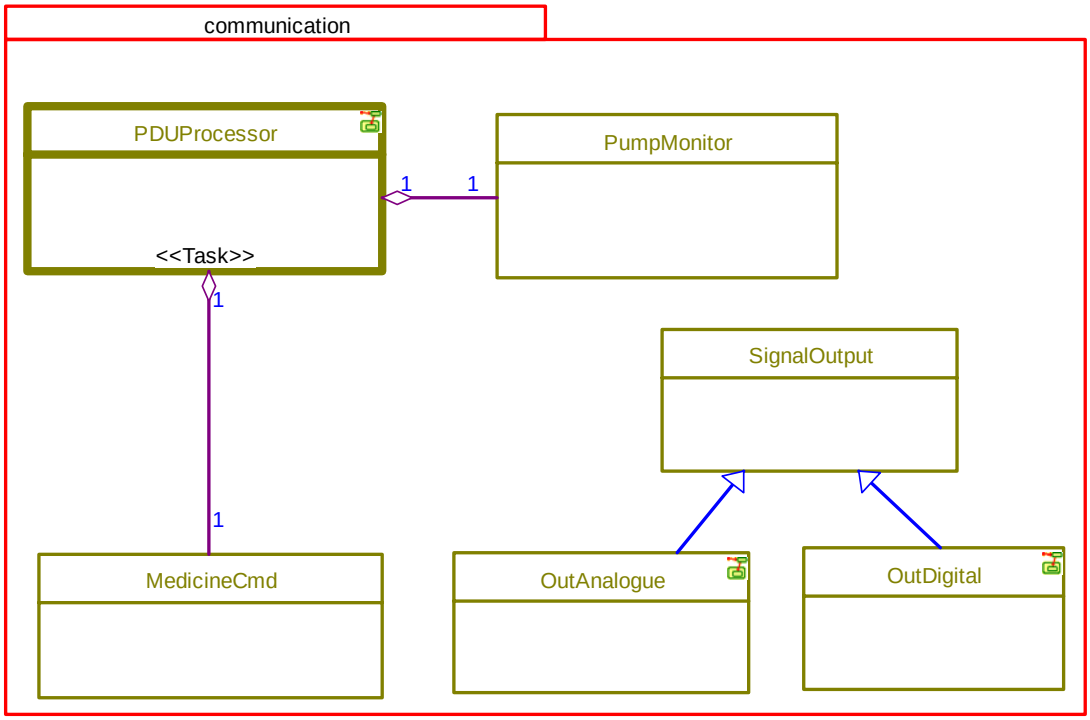


Figure 11: The communication package

Class: SignalOutput

An abstract class used to define the methods used for output of data.

Class: OutAnalogue

Super class for analogue outputs. Has a reference to the PV2019 card.

Class: OutDigital

Super class for digital outputs. Has a reference to the IO686 card.

Class: PDUProcessor

Polls the pump monitor for incoming messages (PDUs) and invokes the returned command.

This class implements the Invoker class from the GoF [Ref. 2] Command pattern.

Class: MedicineCmd

Abstract class that defines the basic interface for the incoming protocol data units from the infusion pump. This class is an implementation of the Command class described in the GOF Command pattern.

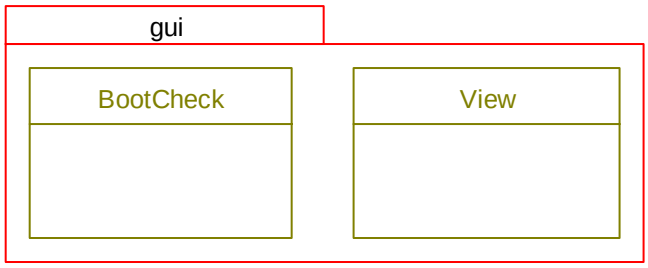
Class: PumpMonitor

This class acts as a monitor for the infusion pump.

It handles incoming protocol data units (PDUs) from the data pump. On receiving a PDU from the pump, it is verified for consistency and translates it into the corresponding MedicineCmd. This class also handles instantiation of the communication port.

2.5.2.4 Package: GUI

This package handles all interaction between the display, hardware and the business logic in the application package.



Class: View

This class is responsible for updating the display and handling user interaction with the application through the user interface. It uses ConfigReposity, from the application package, to do the actual handling of user input.

Class: BootCheck

Description: Provides a user interface that is shown while system checks and initializations are preformed.

2.5.2.5 Package: Application

The application package contains all the business logic for the patient simulator.

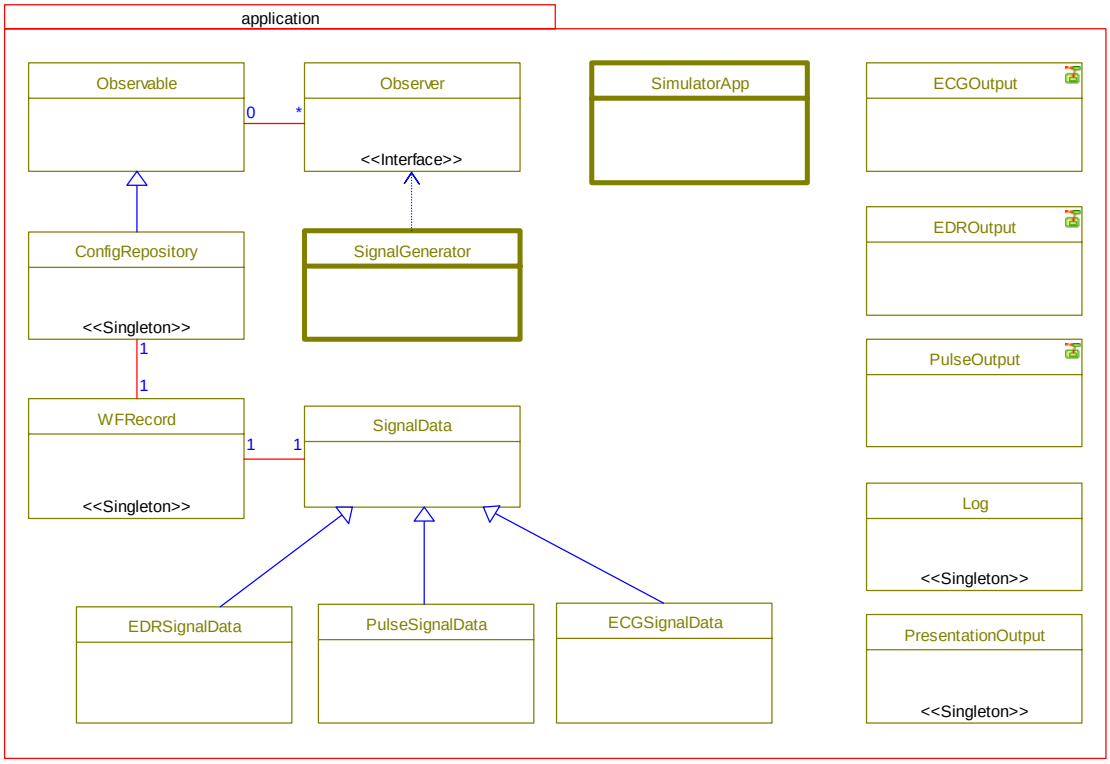


Figure 12: The application package

Class: SimulatorApp

Responsible for creating and starting all tasks in the patient simulator system. This is the main thread of the application.

Class: ECGOutput

Responsible for outputting ECG data on an analogue channel using the PV2019 I/O card.

Class: EDROutput

Responsible for outputting EDR data on an analogue channel using the PV2019 I/O card.

Class: PulseOutput

Responsible for outputting pulse data, on a digital channel using the IO686 I/O card.

Class: Log

A class that provides static access to log functionality. It uses RtPeg's message queue, to send messages to the user interface.

Class: PresentationOutput

This class is capable of sending values to the UI that is meant to be shown, e.g. EDR or ECG.

Class: Observable
Implemented according to the observer, as sugested by GoF. The subject should inherit from this class.

Class: Observer
An interface that Observable uses according to the GoF Observer pattern.

Class: ConfigRepository
Responsible for both storing data changes, and notify other of the change, therefore is has the role of subject in the Observer pattern. Callers to this class could come from pump input or from user interface.

Class: WFRecord
Reads data from the files that belong to the selected patient. Provides a method to change patient and read the next value.

Class: SignalGenerator
The task that executes the output methods in a continous loop.

Class: SignalData
An abstraction used for all data used to simulate signals.

Class: ECGSignalData
Uses WFRecord to provide ECG signals.

Class: EDRSignalData
Uses WFRecord to provide EDR signals.

Class: PulseSignalData
Calculates pulse signals accordingly.

2.5.3 Use case realizations

2.5.3.1 Use case 1: Generate patient Signals

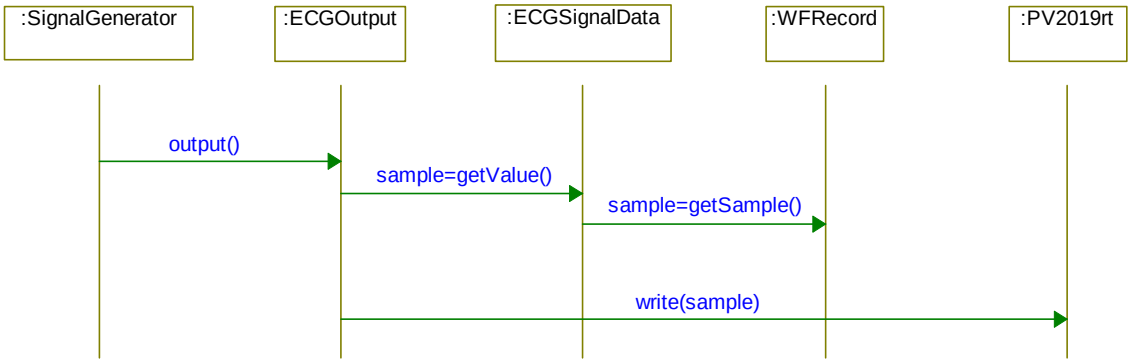


Figure 13: Sequence diagram showing generation of signals

Figure 13 shows the sequence that result in an output on the ECG output channel. The same sequence occurs with EDR and pulse signal. The sequence is executed only by the *SignalGenerator* thread. *SignalGenerator* contains a list of *SignalOutput* objects, each can be an instance of either *ECGOutput*, *EDROutput* or *PulseOutput* (since they are all subclasses).

Each of these output instances has a reference to its corresponding *SignalData* subclass, i.e. an *ECGOutput* instance has a reference to an *ECGSignalData* instance.

Each *SignalData* subclass is able to provide data which is used to to simulate the patient.

ECGSignalData and *EDRSignalData* use the *WFRecord* singleton to retrieve data from the PhysioBank archive.

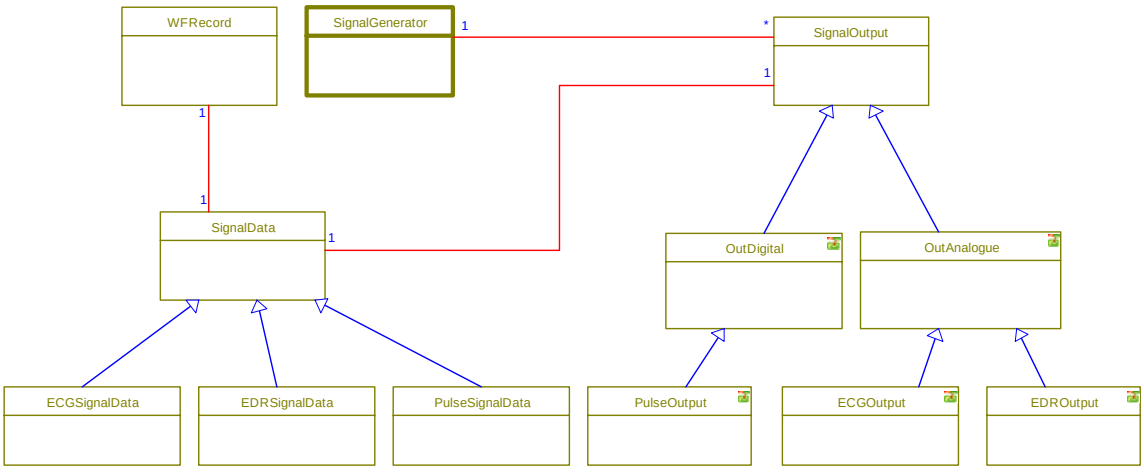


Figure 14: Classes used to generate signals

Figure 14 shows the relations between the classes involved. Notice how SignalGenerator has 0..* *SignalOutput* objects, and each of the *SignalOutput*s has one *SignalData*.

2.5.3.2 Use case 2: Generate ECG signal

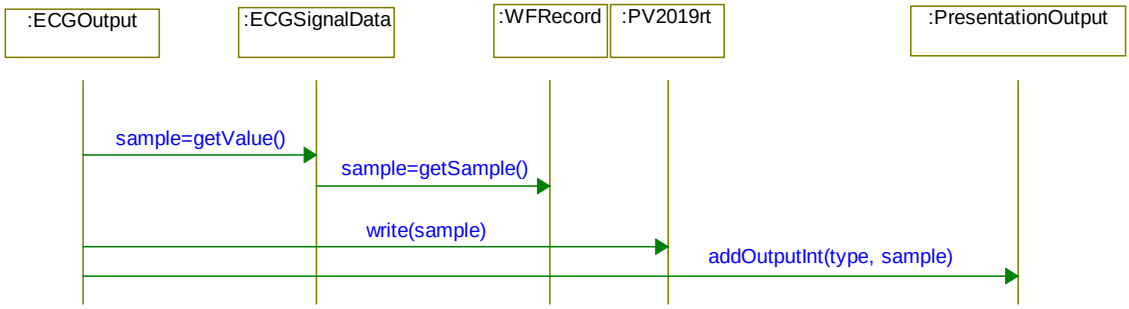


Figure 15: Sequence to generate ECG signal

The sequence in Figure 15 shows how an ECG value is generated and sent to the physical output. *ECGOutput* uses *ECGSignalOutput* as its data provider.
After the value has been written to the PV2019, it is sent to *PresentationOutput* to be shown on the user interface.

2.5.3.3 Use case 3: Select Patient

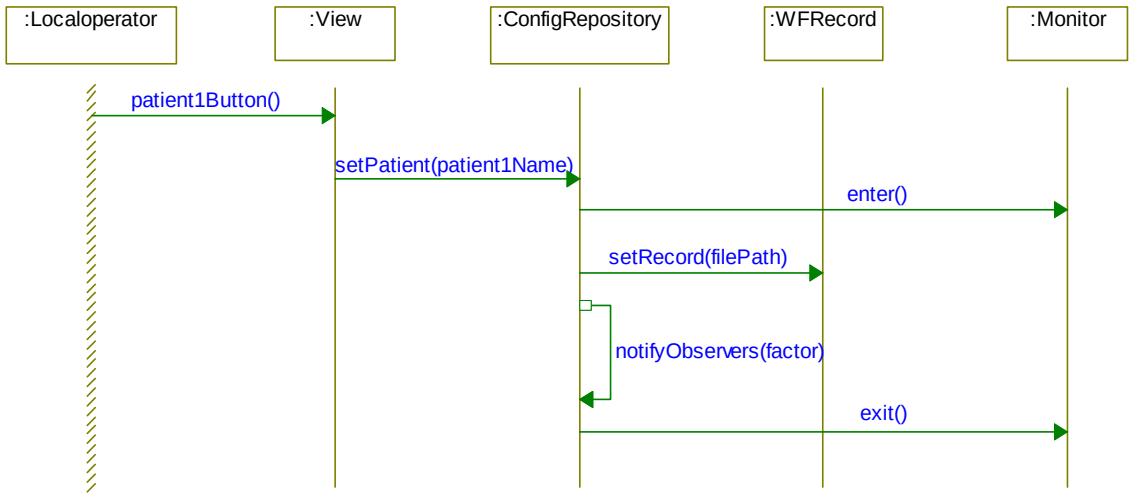


Figure 16: Sequence diagram of Select Patient

Figure 16 shows how to change the data set on which the current output is based. To change patient data, the *View* class invokes the *changePatient* method in the *ConfigRepository* singleton.
ConfigRepository then sets the new value in *WFRecord*, so that the next time a *SignalOutput* fetches a value, will get it from the new patient.
Each patient has a basic heartbeat value, so *ConfigRepository* notifies it's observers with this new value.
Monitor is used to make changing patient becoming an atomic operation, which is nessesary, since in the future other than *View* will be able to call *setPatient*.

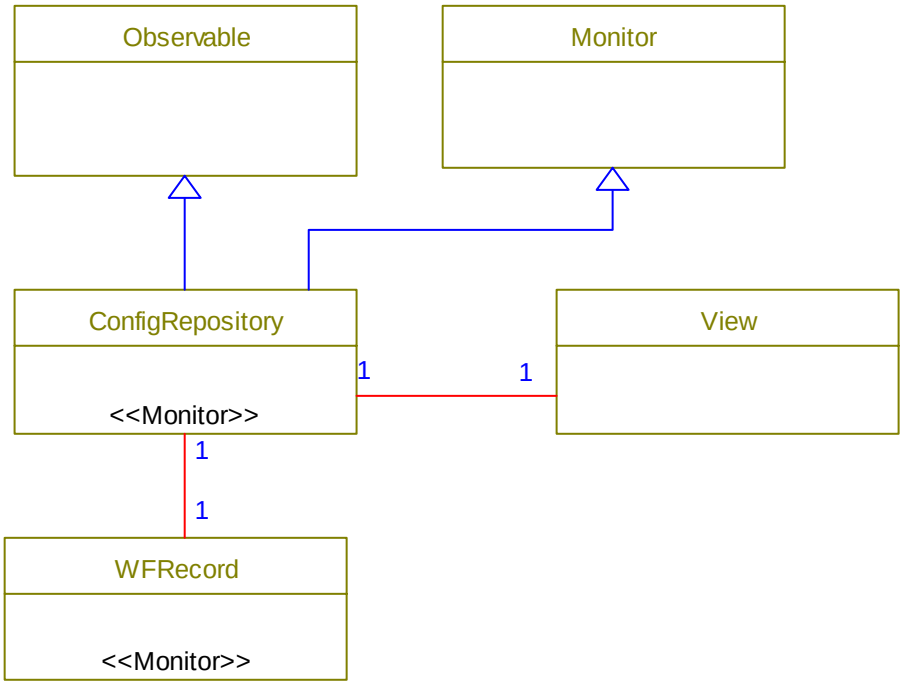


Figure 17: Classes used to select patient

Figure 17 shows the relation between the classes uses to realize the Select Patient use case.

2.5.3.4 Use case 4: Handle Pump Data

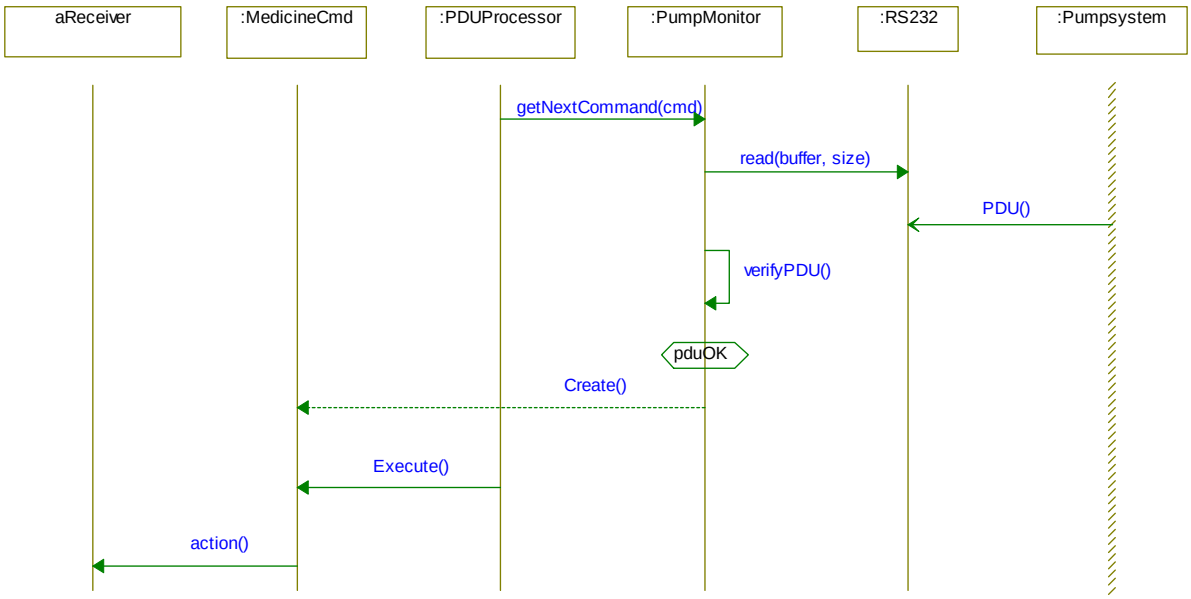


Figure 18: Sequence for handling pump data

PDUProcessor is an active class which is responsible for listening for pump data and delivering it. Figure 18 shows what happens in a single loop in *PDUProcessor*. First *getNextCommand* which blocks until the pump system send a correct PDU.

When *PumpMonitor* receives a PDU, it creates a *MedicineCmd* object based on the data. Finally *PDUProcessor* executes the command.

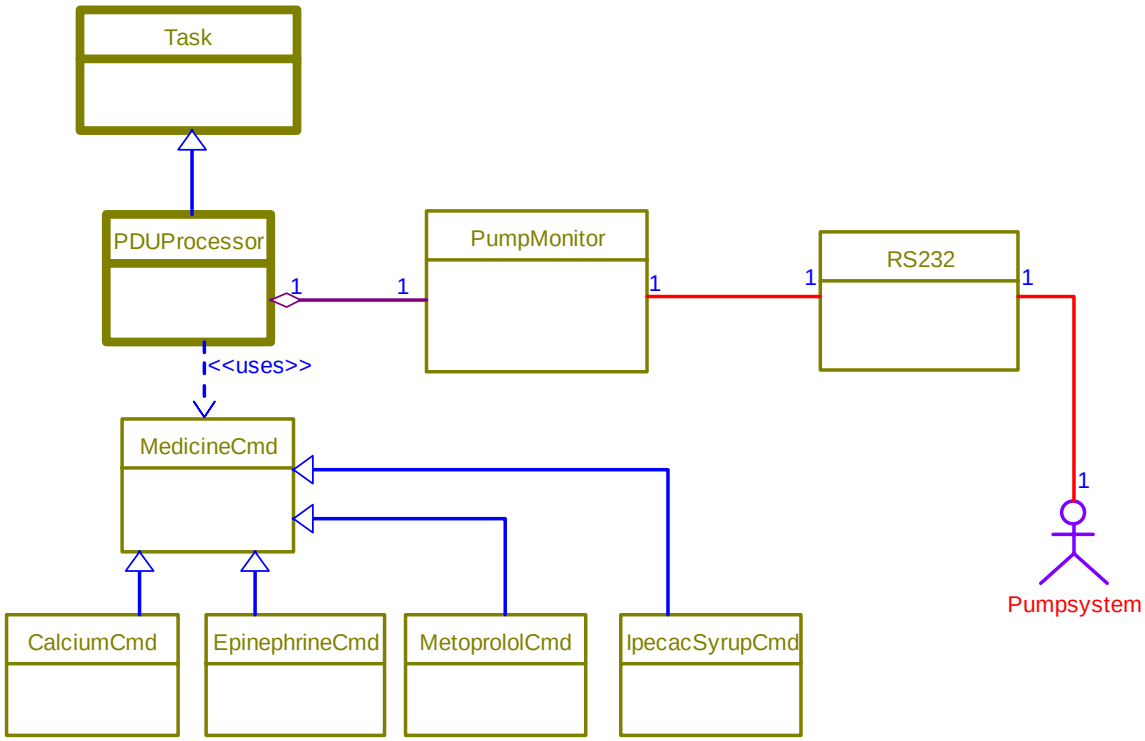


Figure 19: Classes that handle pump data

Figure 19 shows the relations between the classes involved.

2.5.3.5 Use case 5: Display Patient Signals

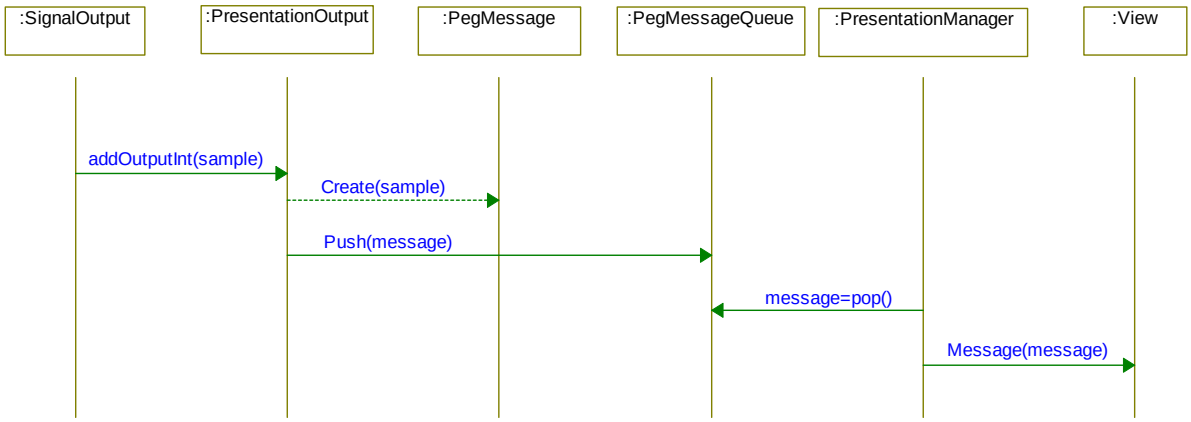


Figure 20: Sequence for displaying signals

Figure 20 shows how a *SignalOutput* instance uses *PresentationOutput* to send a *PegMesssage* to the *PegMessageQueue*. The *PegMessageQueue* is the queue in which all GUI events are put. The GUI has a *PresentationManager* that is responsible for emptying that queue and send it to a registered handler, in this system *View*'s *Message* method.

When *Message* detects the sent *PegMessage*, it extract the value and type and updates the GUI accordingly.

2.5.3.6 Use case 6: Generate EDR Signal

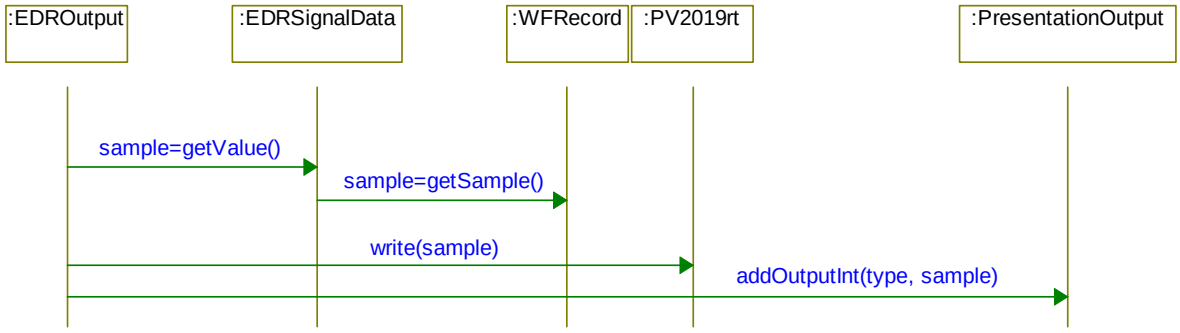


Figure 21: Sequence to generate EDR signal

The sequence in Figure 21 shows how an EDR value is generated and sent to the physical output. *EDRSignalData* is used to retrieve the EDR signals. When the sample has been generated, it is written to PV2019 and *PresentationOutput*.

2.5.3.7 Use case 7: Regulate Patient Signal

Each subclass of *MedicineCmd* implements it's own *Execute* method. Each of these implementations calls *ConfigReposity* with a heartbeat factor based on the type of medicine. *ConfigRepository* uses the Observer pattern to notify the necessary objects, so that the following signals are based on that factor.

2.5.3.8 Use case 8: Generate Pulse Signal

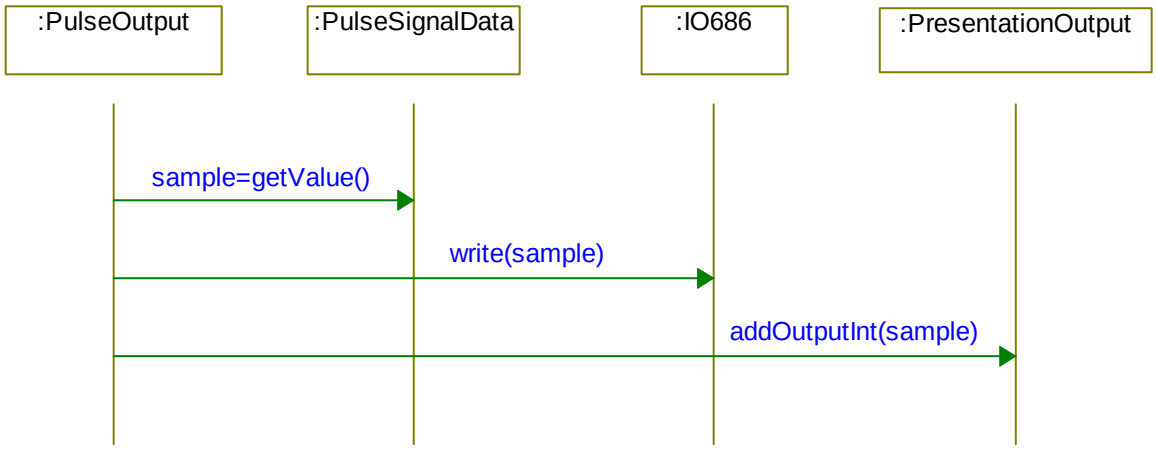


Figure 22: Sequence for generating pulse signals

Figure 22 shows how *PulseOutput* uses *PulseSignalData* to create a pulse sample. That sample is then written to the IO686 board and the display is updated through *PresentationOutput*.

2.5.3.9 Use case 9: Diplay Pump Data

When data is received, it sends the new pump data through *PresentationOutput* to *View* and will accordingly be displayed.

2.6 Process/task View

2.6.1 Process/task overview

The section describes the concurrency design of the project. As can be seen from Figure 23, there are 3 active tasks controlling their associated objects. The figure also shows the direction of the data flow between them, where the data can be byte streams, messages or procedure calls. The *SimulatorApp* is active due to the fact that it contains a PEG *PresentationManager*.

This figure only shows the generalizations classes and the flow of data between them – the following sections contain detailed instantiated model.

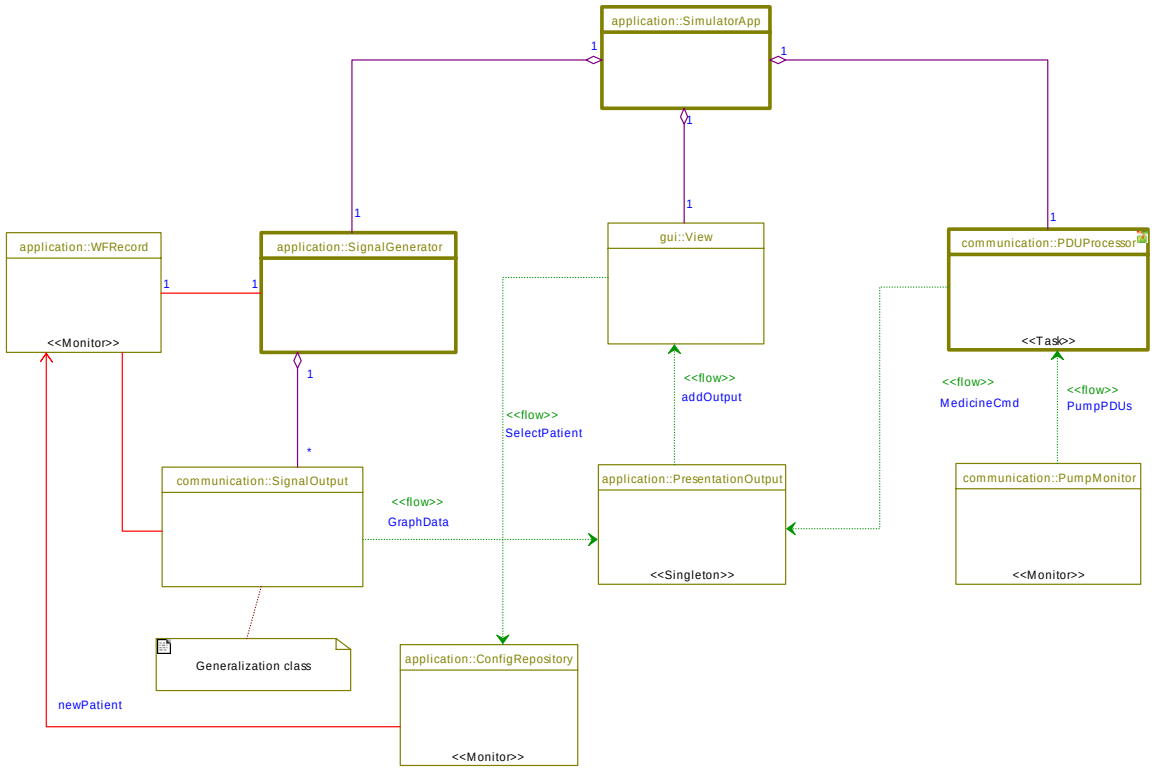


Figure 23: Active class communication

2.6.2 Process/task implementation

The base class for the active classes is the abstract class *Task* (not illustrated). It implements threading using the RTKernel command – *RTKRTLCreateThread*. 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. The class is described in more detail in section 3.3 about the implementation.

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 priorities are seen from Figure 24.

Task	Priority
SimulatorApp (main thread)	5
SignalGenerator	5
PDUProcessor	4

Figure 24: Task priorities

2.6.3 Process/task communication and synchronization

For communication between the tasks in the current implementation, we use procedure calls and RTPeg's IPC messages.

The central unit in the system is the *ConfigRepository*, that is, it is used for communication between the active tasks. Therefore it extends the *Monitor* class (not illustrated) that implements synchronization protection through RTKernels resource semaphore.

The *SignalGenerator* implements its own *round-robin* scheduler for retrieving data from the *WFRecord* object and dispatching it to all the signal output objects. As this happens sequentially, no further synchronization is required.

The only place that simultaneous data flow may occur is from the *ConfigRepository* to the *View* object. This is handled by the message queue in the PEG *PresentationManager* that contains a synchronized FIFO buffer.

2.6.4 Process group: User interaction

This group is composed of the classes related to the MMI (Man Machine Interface).

2.6.4.1 Process communication

All communication between the user (local operator) and the system is done through the *ConfigRepository* and *View* classes using IPC messaging.

2.6.4.2 Description

Figure 25 shows the flow of data in the user interface. Input from the screen (patient selection) is directed to the instance of the *ConfigRepository* singleton that in turns informs the *WFRecord* singleton. The *WFRecord* object protects the *next* and *getSampleValue* ports with a semaphore, so as not to allow patient change in the middle of a sample reading. Information flow to the screen uses PEG messaging as described above.

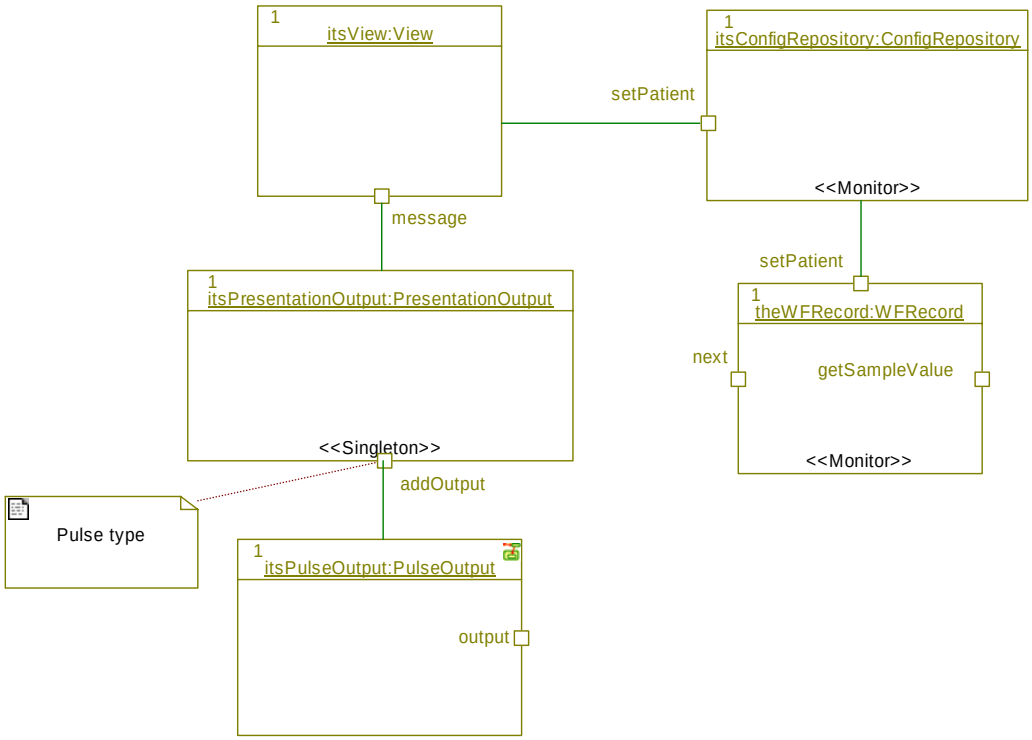


Figure 25: MMI process communication

2.6.5 Process group: Pump communication

This group handles the interaction with the pump system.

2.6.5.1 Process communication

Communication between the pump and the system is done through the *PDUProcessor* class and its associated *PumpMonitor* class using a procedure call (*getNextMessage*). After processing the PDU, it is relayed to the *PresentatonOutput* singleton that uses PEG messages to display the data on screen.

2.6.5.2 Description

Figure 26 illustrates the flow of data from the *PumpMonitor* to the *PresentationOutput*. The *getNextMessage* port in the *PumpMonitor* is a blocking procedure that awaits notification from the serial port when a new data package has arrived. This happens once every second if a pump is connected to the system.

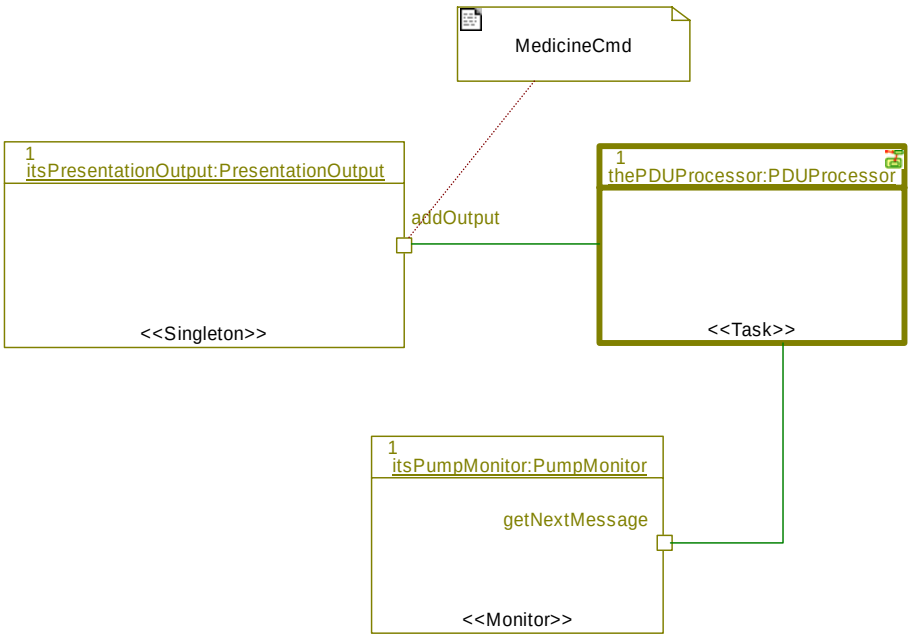


Figure 26: Pump communication process

2.6.6 Process group: Signal Generator

This group handles the output of digital and analogue data signals to the hardware.

2.6.6.1 Process communication

The *SignalGenerator* is the central process in the system with the heaviest duty cycle. It has consequently been implemented with the highest priority. Future development to the design should bear this consideration in mind.

2.6.6.2 Description

Figure 27 illustrates the instantiated concurrency model of the signal generator. On the figure is only shown the communication in the *PulseOutput* cycle, but the implementation of the ECG and EDR cycles are identical, although with different duty cycles.

The signal generator has been implemented as a *round-robin* scheduler to allow for only one fetch of data from the *WFRecord*, thus maintaining synchronization of the data output to the I/O ports. This also simplifies the addition of future output ports as the plug-in nature of the scheduler will maintain the overall duty cycle required for outputting reliable signals[1].

[1] In the extend that the CPU on the hardware can cope with the required calculations.

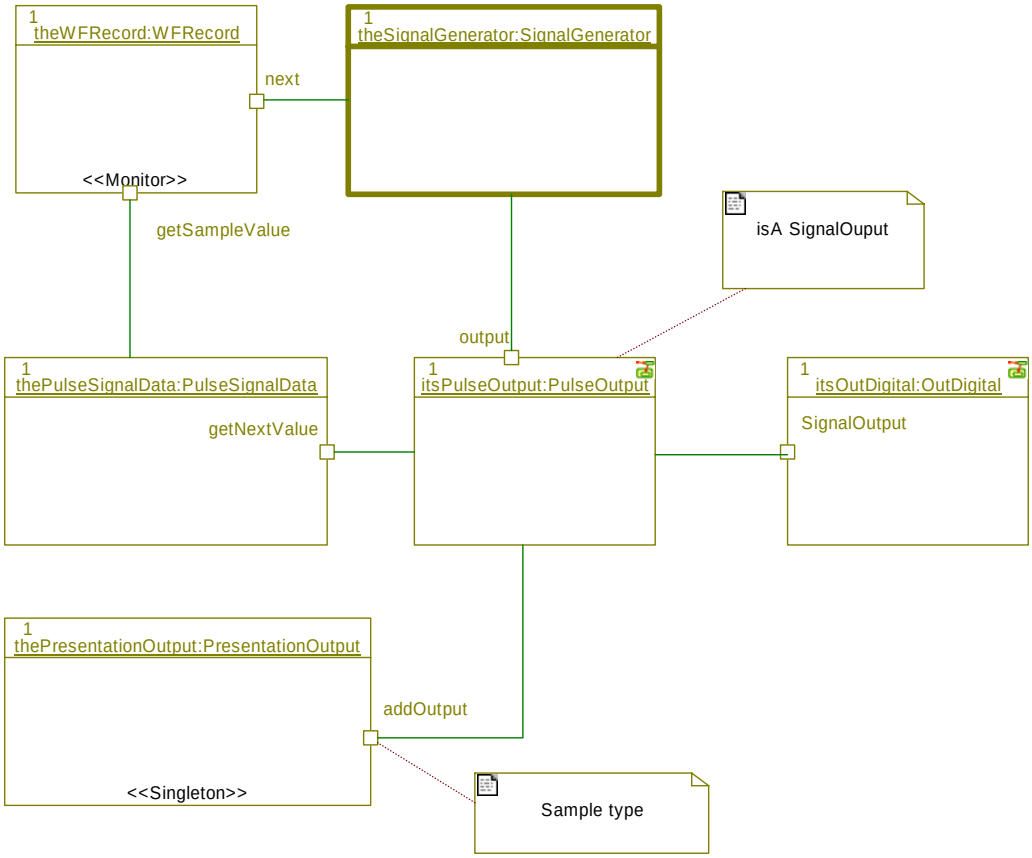


Figure 27: SignalGenerator concurrency implementation

2.7 Deployment View

2.7.1 System configurations overview

The system can be run either in the real simulation situation or in a test situation.

2.7.2 System configurations

2.7.2.1 Configuration 1: Simulator

Figure 28 shows the patient simulator system in the real simulating situation. Here LMON shows the outputted data from PSIMU and the IPUMP sends medicine data to PSIMU.

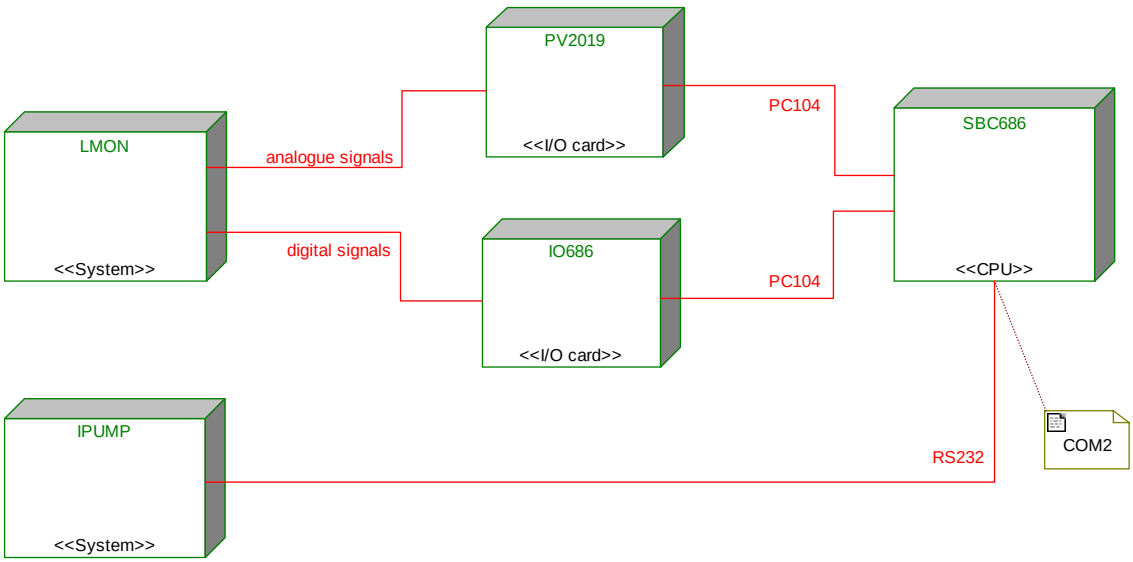


Figure 28: Simulator deployment

2.7.2.2 Configuration 2: Test

Figure 29 shows the patient simulator system in a test situation. The following changes have been made:

- LMON is here replaced by:
  - An oscilloscope to show the analogue signals
  - A digital LED console to show the digital pulse output
- The IPUMP is replaced by a pump simulator that simulates the PDU sent on the RS232 channel.

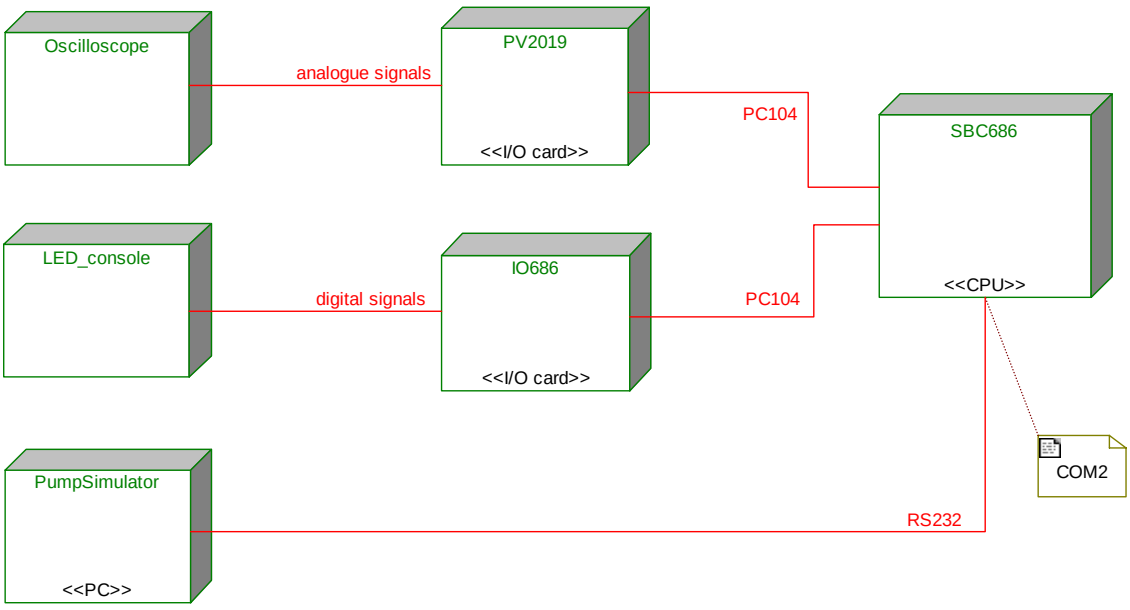


Figure 29: Test configuration deployment

2.7.3 Node descriptions

2.7.3.1 SBC686

Primary computer unit of the PSIMU system.

2.7.3.2 PV2019

PV2019 I/O card responsible for outputting analogue data.

2.7.3.3 IO686

The IO686 I/O card is responsible for outputting digital data.

2.7.3.4 LMON

Patient monitoring system to show outputted data from the PSIMU system. Notice that the analogue and digital signals are not directly received by the LMON, but it is shown so because LMON is not part of this project.

2.7.3.5 IPUMP

Infusion pump system that sends medicine information to PSIMU on a RS232 channel. Notice that this is not a part of the PSIMU project.

2.7.3.6 Oscilloscope

In the test situation, an oscilloscope is used instead of the LMON. This makes it easier to verify the outputted analogue signals.

2.7.3.7 LED_console

In the test situation, a digital LED console is used instead of the LMON. This makes it easier to verify the outputted digital signals.

2.7.3.8 PumpSimulator

Developed pump simulator that simulates medicine information (in a PDU) on a RS232 channel.

2.8 Implementation View

2.8.1 Overview

Figure 30 shows the components that assemble the system.

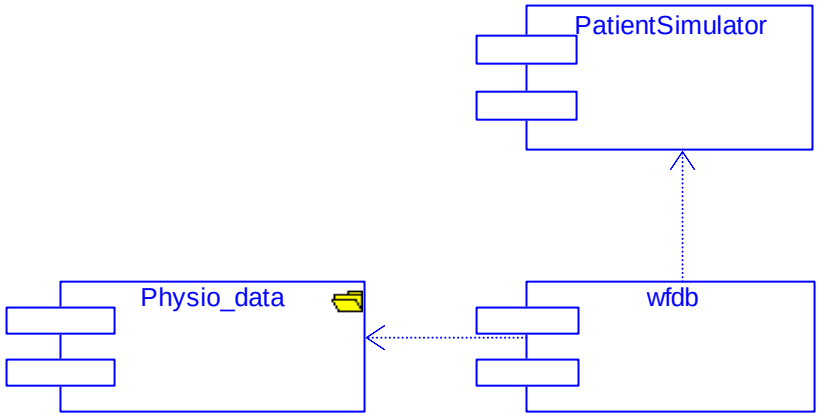


Figure 30: Component diagram

2.8.2 Component descriptions

2.8.2.1 PatienSimulator

This component is the executable file. It depends on the wfdb libery to provide the data to simulate.

2.8.2.2 wfdb

The wfdb is a library made by PhysioBank which is able to interpret the data files which PhysioBank also provide.

2.8.2.3 Physio_data

These are the actual data that the system use as base for simulation. The files are placed on the local storage of the platform.

2.9 Data View

2.9.1 Data model

The PSIMU system uses data from PhysioBank, which is a collection of databases with signals from real patients.
Ref. http://www.physionet.org/physiobank/physiobank-intro.shtml

The PSIMU handles five different set of patient data files.
Each patient data set consists of three files:

- [XXX].hea
- [XXX].dat
- [XXX].atr

where XXX is a number uniquely to this data set downloaded from the PhysioBank database.

The system outputs three signals:

- ECG (continuous analogue signal)
  - Can be fetched directly from some software from PhysioBank (called WFDB) that is able to read the binary data files.
- EDR (continuous analogue signal)
  - Is derived from ECG. The PhysioBank software is also capable of calculating this “on the fly” from the ECG signals. This is what will be done in this system.
- Pulse (digital signal)
  - Can also be derived from the PhysioBank software. This is quite complex, though, so this system will normally output the pulse 60 (i.e. 60 heartbeats per second). This number is changed if the heartbeat factor is regulated upon either pump information or from the GUI.

2.9.2 Implementation of persistence

Persistent data is stored on a FAT file system on the SBC686 hard drive.

2.10 General design decisions

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

2.10.1 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.

2.10.2 Architectural patterns

MVC (Model View Controller) design pattern is the base for the PSIMU system.
This pattern is very flexible and it consists of three kinds of objects:

- Model: The application object
- View: The screen presentation
- Controller: Defines the way the system is connected to outside world

The class *ConfigRepository* acts as the central controller in this system between the model classes and the GUI (implemented in the *View* class).

The *Singleton design pattern* [Ref. 2] ensures that a class has only one instance, and provides a global point of access to it. This pattern is a part of PSIMU system.

The *Observer design pattern* [Ref. 2] is used for sending information around the system. This is used in the situation when the socalled heartbeat factor is changed from either on basis of received pump data resulting in a heartbeat change or when the factor is changed from the slider on the GUI.

The *Command design pattern* [Ref. 2] is used in the case of receiving pump data. Each different medicine is modelled as a Command that has its own *Execute* method that affects the heartbeat factor.

2.10.3 General user interface design rules

The user interface is a graphical user interface (GUI), where the user navigates with the mouse.
GUI must be user friendly.
The local user must have an opportunity to do the following:

- Select output data for the patient
- Change the heartbeat factor (i.e. how fast data signals are outputted)

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

2.10.4 Exception and error handling

During initialization, some checks are performed. If one of these checks fails, an error message is shown on the GUI. Runtime errors are not handled in this system.

2.10.5 Implementation languages and tools

The chosen implementation language is C++.
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

2.10.6 Implementation libraries

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

2.11 Size and performance

There must at maximum be installed 15 MB of patient data files.

The system is supposed to output data in correct intervals. E.g. the sample rate for the patient data files are normally 360 samples per second. The system can simulate a heartbeat up to the double of this samplerate, i.e. 720 samples per second. Two outputs (ECG and EDR) should both be outputted with this samplerate.

2.12 Quality

There will be no requirements upon operating performance, i.e. mean-time between failure (MTBF) etc.

The system is intended to evolve into a distributed system in the future.

2.13 Compilation and Linking

2.13.1 Compilation hardware

For compiling the software, any platform compatible with Microsoft Windows XP operating system and capable of executing Microsoft's nmake application and command line compiler version will suffice.

2.13.2 Compilation software

For compilation, version 7.10 of nmake and version 13.10.3077 of the command line compiler have been tested.

The project 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.

To compile successfully, the RTKernel development tools and the Microsoft Development tools are required.

2.13.3 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 all necessary module libraries and the simulator application.

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

2.14 Installation and Executing

2.14.1 Installation

2.14.1.1 Software

The personal computer must be connected to the COM port on SBC686 computer with a 0-modem cable.
After the connection is established, the program is downloaded onto SBC disk on chip.

2.14.1.2 Data

The simulator handles five different patient data files. The data files should be copied into:

```
C:\PATSIM\PATIENT[X]
```

where 'X' is the patient number from 1 to 5.
To each patient, there are three data files, consisting of:

- [XXX].hea
- [XXX].dat
- [XXX].atr

where XXX is a number uniquely to this data set downloaded from the PhysioBank database (see 2.9).

2.14.2 Executing hardware

The hardware components are limited to the following:

- SBC686 computer
- PV2019 analogue to digital extension board
- IO686 I/O extension board (this system only uses two digital outputs)
- 20 Mb Disk-on-chip hard drive

2.14.3 Executing software

The simulation software will start automatically when pushing on the power button. It is executed directly from the SBC686.

2.14.4 Execution control (start, stop and restart)

The system is started upon loading the executable, and there is no way possible to stop the software other than turning it off.

2.14.5 Error messages

Some checks are performed during system initialization to make sure that I/O cards and data files are present.
Any errors during this check are reported on the splash screen.

3 Implementation

3.1 Introduction

This section describes various implementation issues that have come up during the implementation and have been found worth mentioning in the context of future development.

3.2 Design patters

In this project, three design patterns from the GoF book [Ref. 2] have been used. In the following, it will be shown briefly how these are implemented in specific C++ code.

3.2.1 Singleton design pattern

As mentioned in section 2.10.2, the purpose of the Singleton design pattern is to ensure that there is only one instance of the class, and provide a global access to it.

This is for instance implemented in the *ConfigRepository* class, which acts as the one and only controller between model and view, as noted in section 2.10.2 as well.

The GoF structure is shown in Figure 31.

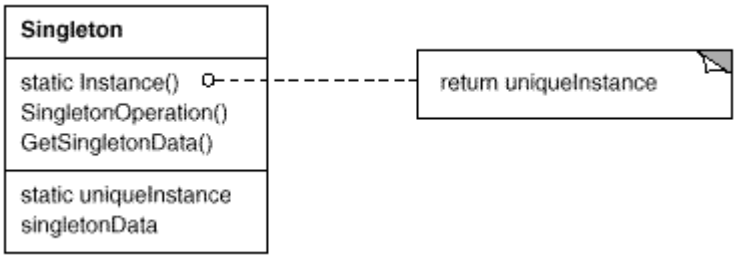


Figure 31: GoF Singleton design pattern

The specific code of interest in *ConfigRepository* is:

```
protected :
  ConfigRepository();
…
private :
  static ConfigRepository* instance;
…
static ConfigRepository* getInstance()
{
  if (instance == 0) {
    instance = new ConfigRepository();
  }
  return instance;
}
```

What should be noticed here is the protected constructor. This is done to make sure that no outer class can make a direct instance of the *ConfigRepository*.

The *getInstance* method creates a new (the only one) instance of the class and keeps it for next time the method is accessed.

3.2.2 Observer design pattern

The purpose of the Observer design pattern is to establish a general way of updating dependent classes about a change in one object. In this system it is used to sending the heartbeat factor to the classes that depends on it.
The GoF structure is shown in Figure 32.

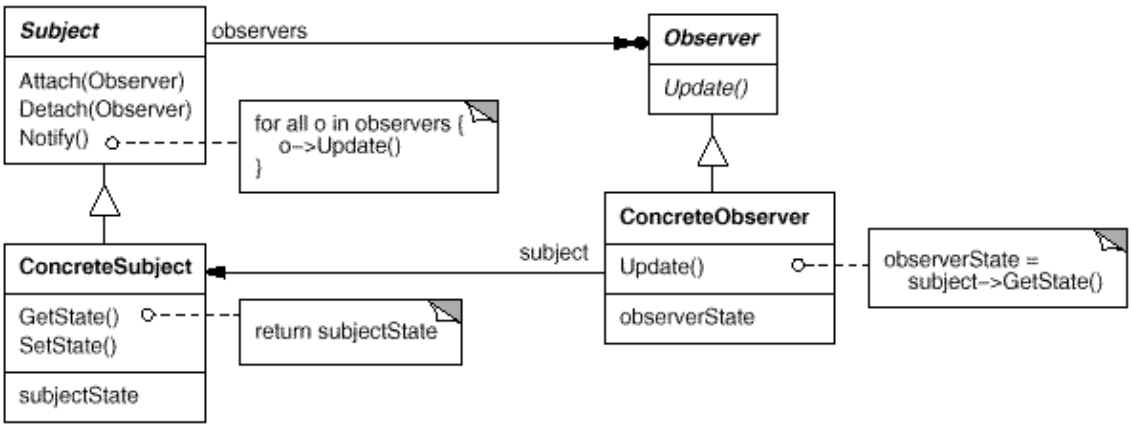


Figure 32: GoF Observer design pattern

The specific use of this pattern can be seen from Figure 33.

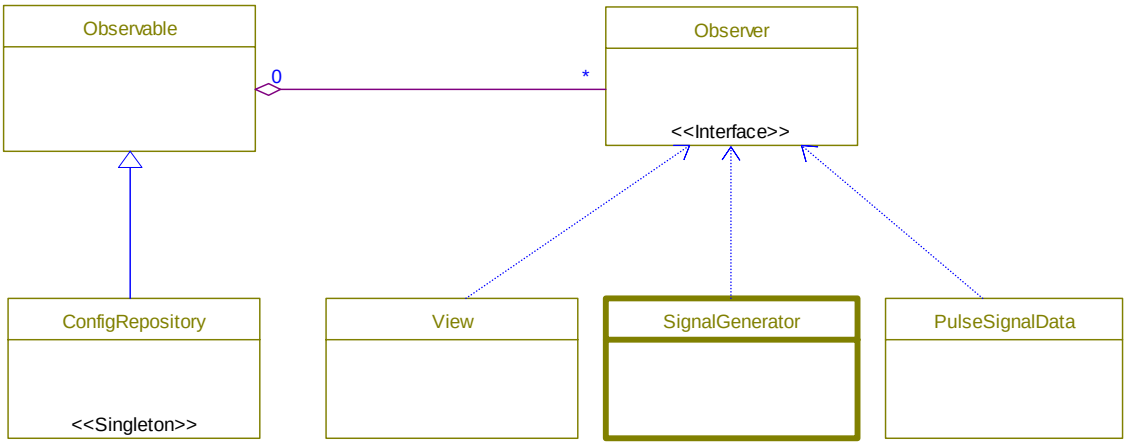


Figure 33: Use of Observer design pattern

The *Subject* class has been renamed to *Observable* because it is a more natural name.
In the *Observable* class the *notifyObservers* method sends the requested information to all registered observers:

```
void Observable::notifyObservers(void* obj)
{
  for(int i=0; i<index; i++)
  {
```

```
      observers[i]->notify(obj);
    }
  }
```

The *ConfigRepository* class inherits from *Observable*. Every time the *ConfigRepository* class needs to update its observers, it calls the *notifyObservers* method.

Classes that want updates from *ConfigRepository* can inherit from the abstract class *Observer*:

```
class Observer {
public :
  virtual void notify(void* obj) = 0;
};
```

One of the dependents of the heartbeat factor is the *View* class, because the slider on the screen should reflect the actual factor. The notification is taken care of in the *notify* method that must be implemented because it is "pure virtual" in the *Observer* class:

```
class View : public PegWindow, public Observer
…
void View::notify(void* obj)
{
  double* pFactor   = (double*) obj;
  double  newFactor = (*pFactor) * 100;
  mSlider->SetCurrentValue((int)newFactor, true);
}
```

3.2.3 Command design pattern

The purpose of using the command design pattern is to be able to easily add new commands with different behaviour (this is implemented in the *Execute* method). The GoF structure is shown in Figure 34.

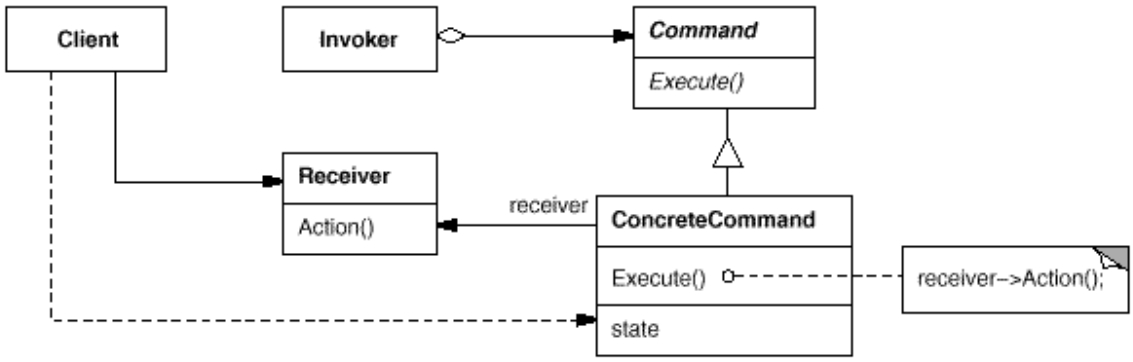


Figure 34: GoF Command design pattern

The superclass *Command* is named *MedicineCmd* that has its pure virtual *Execute* method:

```
class MedicineCmd
{
public:
  …

  virtual void Execute() = 0;
  …
};
```

Each medicine type that the infusion pump sends is modelled as such a *MedicineCmd*, as shown for exampe:

```
class CalciumCmd : public MedicineCmd
{
public:
  …
  void Execute() { ConfigRepository::getInstance()
                                          ->setHeartBeatFactor(0.7); };
};
```

3.3 Thread implementation

For hiding the RT-Kernel specific code, a generic *Task* class has been made that encapsulates the RT-Kernel code. It is with this *Task* class possible to control the priority of the thread:

```
Task::Task(unsigned int aPriority): priority(aPriority) {
}
…
void Task::start() {
  TaskHandle = RTKRTLCreateThread(taskRun, priority, 4096,
TF_MATH_CONTEXT, this, "Task 1");
}
```

All task classes must implement the method *run*, because it is declared pure virtual in the abstract *Task* class.

For further details about this class, see section 2.6.

3.4 Monitor implementation

A method that needs to be executed atomically can use the *Monitor* class by letting the class inherit from the *Monitor* class.

The *Monitor* uses two methods for locking and unlocking:

```
Monitor::Monitor(char* name) {
  sem = RTKCreateSemaphore(ST_BINARY, 0, name);
  RTKSignal(sem);
}
…
void Monitor::enter() {
  RTKWait(sem);
}
…
void Monitor::exit() {
  RTKSignal(sem);
}
```

For further details about this, see section 2.6.

3.5 Data implementation

As informed earlier, all data to the simulator systems is from the PhysioBank database. A piece of software from PhysioBank is used to extract concrete data value from the data files downloaded from PhysioBank. Concretely speaking, a *WFRecord* class handles the data files from where the values are extracted.

The code from PhysioBank is integrated into this system by porting it into a library called *wfdb*. For instance, the calculation of EDR values is not trivial. The algorithm for this is therefore just used without considerations about its correctness.

3.6 Considerations for the future

In the next course “Distributed Realtime Systems”, this system should be extended in a distributed manner. This has been incorporated into the design and the code by the *ConfigRepository* class. This class is a controller class between model and view (speaking in MVC terms). By making some remote control to this class will make it possible to control the system from one central place.
It might not contain all the functionality required for the next iteration but it is only here changes have to be made.

3.7 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 Test

4.1 Introduction

4.1.1 Purpose

This is a test specification for the PSIMU2

The test will carry out in tree levels of testing:

4.2. Unit test:
These tests include the test of the single functions that have been implemented in classes (modules).

4.3. Integration test:
These tests include the test of interfaces between classes (modules) and the test of the all system functionalities.

4.4. Accept test:
These tests include a test of the functional demands from the requirement specification document.

4.2 Unit test

4.2.1 Introduction

The unit test is described through several small test applications performed for vital functions of PSIMU2 system.

Due to that many of these tests are done at the SBC686, it is not possible to make screen shots to display the outcome of the test.

4.2.2 Tests Overview

This section gives an overview of the units that are tested.

4.2.2.1 Test of Analogue Signal

Application	Purpose	Expected result	CHECK
analogueapp.exe	Application is used to verify analogue output signal.	Analogue signals are outputted correctly.	√

4.2.2.2 Test of Digital Signal

Application	Purpose	Expected result	CHECK
digitalapp.exe	Application is used to verify digital output	Digital signals are outputted correctly.	√

	signal.		

4.2.2.3 RS232

Application	Purpose	Expected result	CHECK
rs232.exe	Application is used to check that data are being read from pump.	Data received correctly.	√

Application	Purpose	Expected result	CHECK
Pump Simulator	Application is used to check if PDU data are simulated, and to check if the valid PDU has been received.	PDU data are simulated and valid PDU is recieved.	√

4.2.2.4 Taskapp

Application	Purpose	Expected result	CHECK
taskapp.exe	Application is used to check if threading carries out correctly.	Threading carries out correct.	√

4.2.2.5 WFRecord

Application	Purpose	Expected result	CHECK
winapp.exe	Application is used to check if valid data are read from WFRecord to RTKernel.	The valid data are being read.	√

4.3 Integration test

4.3.1 Purpose

The purpose of this test is to test vital parts of PSIMU2 system.
This section includes the test of interfaces between the relevant components.

4.3.2 Test Overview

4.3.2.1 Communication between GUI and PV2019 A/D

Preconditions: The PSIMU has been successfully initiated. See Figure 35.
Notification: In this scenario, the patient is chosen from GUI by a local user. Oscilloscope is been connected to PV2019.

Step	Action/Input	Expected result	CHECK
1.	PV2019 card sends	ECG analogue signal is displayed on the	√

	analogue ECG signals	oscilloscope.	
2.	PV2019 card sends analogue EDR signals	EDR analogue signal is displayed on the oscilloscope.	√

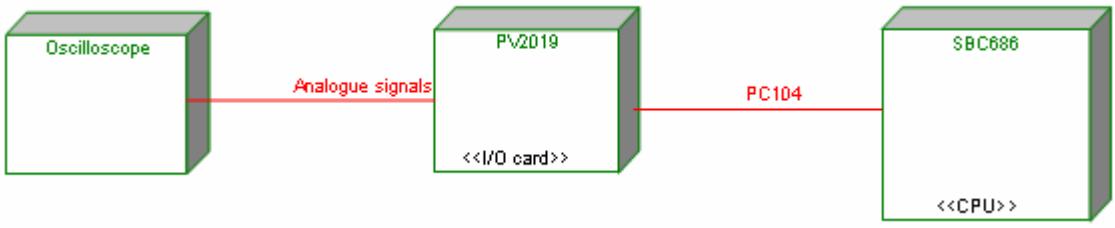


Figure 35: PV2019 test deployment

4.3.2.2 Communication between GUI and IO686

Preconditions: The PSIMU has been successfully initiated. See Figure 36.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Data sent through IO686 card	The Pulse displayed in GUI is shown on LED console.	√

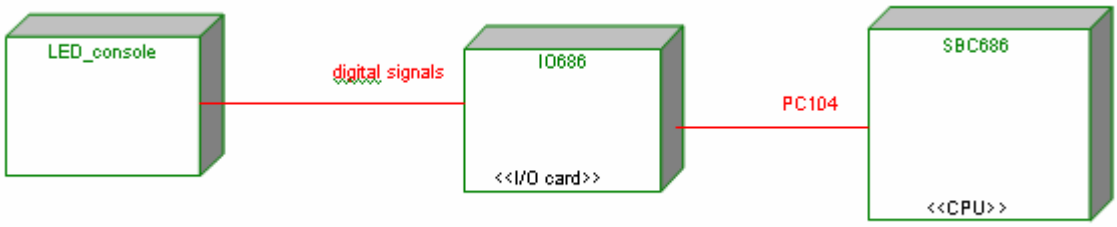


Figure 36: IO686 test deployment

4.3.2.3 Pump Simulator

Preconditions: The PSIMU has been successfully initiated. See Figure 37.
Notification: Pump Simulator application is initialized.
Pump Simulator simulate PDU data.
PDU data are validated.

Step	Action/Input	Expected result	CHECK
1.	PDU data sent through COM2.	GUI displays simulated PDU data.	√

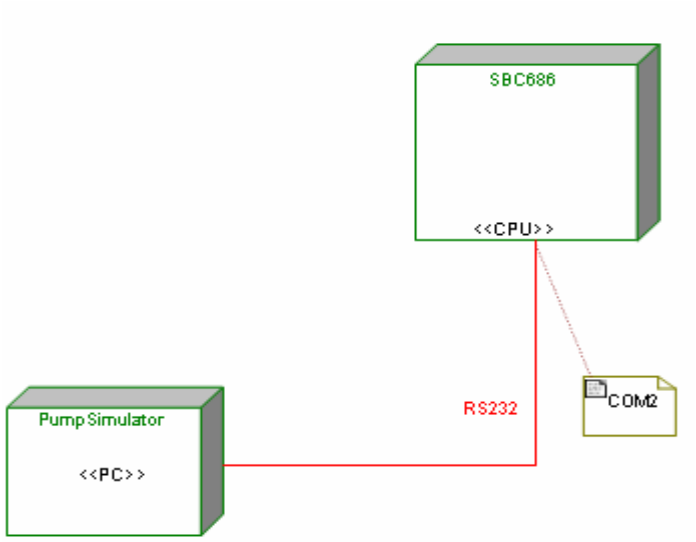


Figure 37: Pump Simulator deployment

4.4 Accept test

The test specification for PSIMU2.

4.4.1.1 Reference

This acceptance test takes an initial point in the requirement specification for the PSIMU2.

4.4.1.2 Scope

The scope of this section is to test the realizations of use cases in the use case diagram that has been described in the requirement specification document for PSIMU2.

4.4.1.3 Definitions

Accept test specification	The document that specifies the test of the functionalty demands from the requirement specification.
Accept test rapport	In the finishing point of the accept test, it will be a rapport.
Internal test object	Those objects/tests that are included in this accept test.
External test object	The object that employees to carry a test but it is not a part of the approval of the accept test. If there is a defect that has been fund in a test object, it will not cause the failure of the accept test.

4.4.2 Test specification

The test specification for PSIMU2.

4.4.2.1 Hardware and Software ID

Here follows the software and hardware, which is included in the acceptance test.

Software to be tested:

Software	Version	Release date	Remarks
ON-time RTK	4.07	-	none
PSIMU 2	1.0	24.05.2004	none

Hardware to be tested:

Hardware	Version	Release date	Remarks
PV2019 AD/DA	-	-	-
IO686	-	-	-
SBC686	-	-	-

4.4.2.2 Testing environment

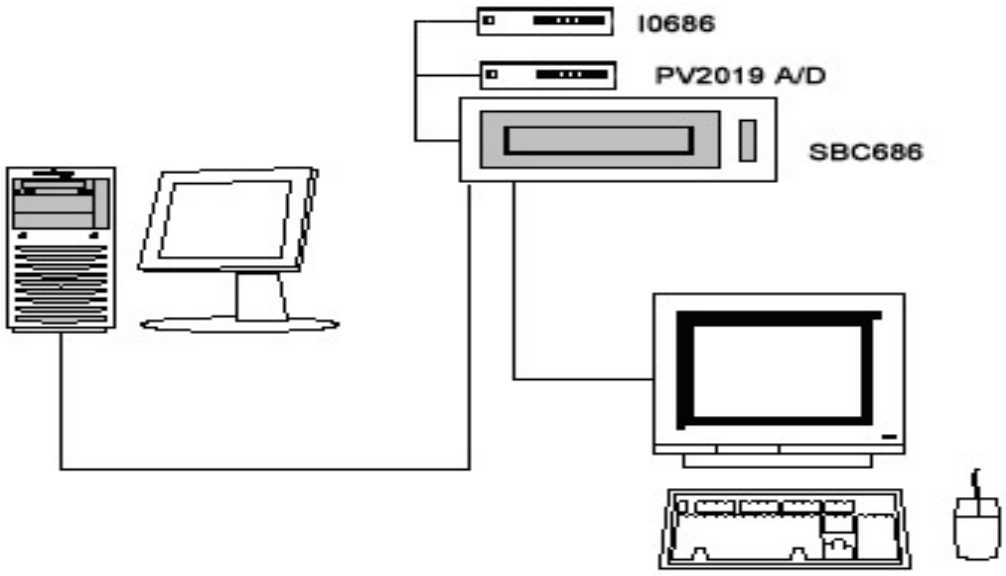


Figure 38: Test environment

Figure 38 shows the environment for the testing procedure. See also section 2.7.2.2.

The software environment is RTKernel.

The hardware environment is SBC686 with the PV2019 AD/DA and IO686 card.

A stationary PC has been used to upload the PSIMU2 testing program to SBC686.

4.4.2.3 Identification of test objects

External test objects:

All of the objects mentioned in the section 4.4.2.2 are external objects.

Internal test objects:

PSIMU2 system and patient data files.

4.4.2.4 Test object prepration

Stationary PC and the SBC686 must be connected and ON–TIME software must be installed in the SBC686.

4.4.3 Test procedure

4.4.3.1 Test of functional requirements

The test of the functional requirements is based in the use cases, which are described in the requirement specification.

If a test is marked with a '√', the actual test has been completed with success.

If a test is marked with a '÷', the actual test has not been completed with success.

4.4.3.1.1 USE CASE: GENERATE PATIENT SIGNALS

- Test case 1: Successful Generated Patient Signals

Preconditions: The system is initiated and respective hardware is connected.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Select default patient	The default patient is selected.	√
2.	Emit signal	Available signals for the specified patient are emitted.	√

4.4.3.1.2 USE CASE: GENERATE ECG SIGNAL

- Test case 1: Successful Generated ECG Signal

Preconditions: Analogue output channel initialized, and data file are present.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Read value	Reading value from data file.	√
2.	Write to output	Data values are written to analogue output.	√

- Test case 2: Failure

Preconditions: Analogue output channel initialized.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Date file not present	A warning message is shown to the local user.	÷
2.	No more data in file	The file is restarted.	√

4.4.3.1.3 USE CASE: SELECT PATIENT

- Test case 1: Successful Selected Patient

Preconditions: User data for 5 patients have been installed on the system.
Notification: Patient data should comply with the PhysioBank specifications.

Step	Action/Input	Expected result	CHECK
1.	Select patient	Selected patient is activated.	√
2.	Output data for selected patient	The data for selected patient is outputted on the system hardware.	√

4.4.3.1.4 USE CASE: HANDLE PUMP DATA

- Test Case 1: Successful Handle Pump Data

Preconditions: An external infusion pump system is connected to the PSIMU system and it is operating.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Receives an event	The system receives a communication event on the COM2 port.	√
2.	Extract the data	The the data are received correctly.	√
3.	Store data	The PDU is stored for usage by the system.	√

- Test Case 2: Failure handle pump data

Preconditions: An external infusion pump system is connected to the PSIMU system and is operating.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Check data checksum	The system discards the package; last received PDU is the current valid reading.	√

4.4.3.1.5 USE CASE: DISPLAY PATIENT SIGNALS

- Test Case 1: Successful Display Patient Signals

Preconditions: Patient signal is generated and producing output.
A graphical chart widget for displaying analogue data has been initialized (used for EDR and ECG signals);
A graphical widget for displaying numerical data has been initialized (Used for pulse signals).
Notification:

Step	Action/Input	Expected result	CHECK
1.	Display analogue signals	EDR and ECG analog data are displayed on the screen using the chart widget. A distinct colour line is maintained for each analogue output channel. ECG green and EDR red	√
2.	Display pulse signal	Pulse signal is displayed.	√

4.4.3.1.6 USE CASE: GENERATE EDR SIGNAL

- Test Case 1: Successful Generated EDR Signal

Preconditions: Analogue output channel initialized, and data file are present.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Read value	Reading value from data file.	√
2.	Write to output	Data values are written to analogue output.	√

- Test Case 2: Failure

Preconditions: Analogue output channel initialized.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Date file not present	A warning message is shown to the local user.	÷
2.	No more data in file	The file is restarted.	√

4.4.3.1.7 USE CASE: REGULATE PATIENT SIGNALS

- Test Case 1: Successful Regulated Patient Signals

Preconditions: A valid PDU has been received.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Regulate PDU data	An algorithm for regulating the output signals, based on the PDU data is executed.	√
2.	Stor PDU data	The result is stored	√

4.4.3.1.8 USE CASE: GENERATE PULSE SIGNAL

- Test Case 1: Successful Generated Pulse Signal

Preconditions: Digital output channel initialized.

Notification:

Step	Action/Input	Expected result	CHECK
1.	Read value	Reading value from data file.[2]	÷
2.	Write to output	Data values are written to digital output.	√

- Test Case 2: Failure

Preconditions: Digital output channel initialized.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Date file not present	A warning message is shown to the local user.	÷
2.	No more data in file	The file is restarted.	÷

4.4.3.1.9 USE CASE: DISPLAY PUMP DATA

- Test Case 1: Successful Display Pump Data

Preconditions: An external infusion pump system is connected to the PSIMU system and it is operating.
A graphical widget for displaying data has been initialized.
Notification:

Step	Action/Input	Expected result	CHECK
1.	Receive PDU	A PDU has been received successfully	√
2.	Display name of medicine	The name of the medicine is displayed on the screen	√
3.	Display volume of medicine	The volume of the medicine is displayed on the screen	√

4.4.4 Final graphical user interface

Figure 39 shows the graphical user interface setup.

[2] There is no data file. The value for pulse signal is being generated. This is not consistent with Use Case 8 "Generate Pulse Signal"

Figure 39: Graphical user interface

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