Embedded Real-Time Systems (TI-IRTS)

Project Report for Sapien 190 (PSIMU)

Spring 2010 – (Version 0.9)

iha.dk

Abstract

This project report describes the specification, design and implementation of the Sapien 190 patient simulator, simulating human physiological behavior…. (200 – 300 words)

- Presentation of the problem
- Aim of the project
- Materials and methods
- Most important results
- Conclusion

Peter Høgh Mikkelsen (20087291)
Anders Block Arnfast (20085515)
Kim Bjerge (20097553) **PAsien**

Table of contents

1. Preface (Peter)

The prerequisite for writing this project report and its accompanying "Architecture Description" document are to pass the graduate course "Embedded Real-Time Systems" (TIIRTS) at the Engineering College of Aarhus. The bases for the project are the exercises described in the handed-in journal: "Exercises 1-5".

The subjects involved: Object Oriented Analysis & Design and patterns for embedded real-time systems are key areas at the Distributed Real-Time Systems specialization at the Engineering College of Aarhus.

The exercise is about designing a product, but the scope of the course is software architecture and design, so this will be the focus of this report, rather than the actual product.

2. Introduction (Peter)

What is a Real-Time Embedded System, what characterizes it and how do we create a robust, maintainable- and efficient software design for it?
An embedded system is traditionally characterized as a custom build system made for a special purpose, with limited power, memory and processing resources.
A Real-Time System is characterized by interaction with the outside environment and having performance deadlines to meet.
An example of a real-time embedded system is a potato sorter: Designed specifically for the food processing industry, it weighs the potatoes while in motion on the wavier and sorts them at the drop-out. This requires special hardware as it must fit in a watertight enclosure where a cooling fan is banned. It also requires real-time processing of weight input to control the sorter mechanism, thus it is an embedded real-time system.

Traditionally these kinds of systems have been built on an 8-bit microcontroller platform with code written in assembler or C.
As the system matures, the code base evolves. The code becomes hard to maintain and to extend and competition pressures the development time; the software calls for a change in paradigm.

Several companies[1] have moved their real-time embedded software development to use design patterns and the methods from Object Oriented Analysis and Design. They have hereby gained better products that are easier to maintain and extend. This is the motivation for this course as it is explained next.

Finn/Danfoss

2.1. Teaching OOA/D at IHA

OOA/D is used extensively throughout the under-graduate studies of the Information Technology Engineering at IHA, it is however primarily taught in a non-time critical Windows PC context, rather than in an embedded real time context, which is the scope of this graduate course.

The course is designed to alternate between lecture and practical exercises. The lectures have been split into two sections: "Real-Time Design Patterns" primarily referring to the book by Bruce Powel Douglas[2] and "Design Patterns – GoF" referring to the book by Gamma et al[3].

The learning objectives for the course are:

- *Analyze and describe* requirements for an embedded real-time system
- *Design and construct* an architecture for an embedded real-time system
- *Judge and use* design patterns in development of an embedded real-time system
- *Develop* a product documentation for an embedded real-time system using UML

These skills are taught in the lectures, and their application is taught in the exercises, in particular in this final hand-in.

2.2. Applying Patterns in Real-Time Embedded Systems

Design patterns originate back to the Small-Talk days of Xerox Parc, where the MVC pattern was first invented by Trygve Reenskaug in 1979[4]. I was not until the release of the famous GOF book, that design patterns became widely-spread.

A design pattern is a named software design that describes the solution to a problem in such a way that you can use it over and over again [BPD, page xiv]. Bruce Powel Douglas was with his book in 2002 the first to take design patterns into the real-time embedded domain.

Design patterns can be put into several categories:

- *Creational Patterns – "Abstracts the Instantiation Process" [GOF p. 83]*
- *Structural Patterns – "Concerned with how classes and objects are composed to form larger structures [GOF p. 137]*
- *Behavioral Patterns – "Concerned with algorithms and the assignment of responsibilities of objects" [GOF p. 221]*
- *Architectural Patterns – "Address the largest-scale system design concerns: architectural design" [BPD p. 137]*

Architectural patterns is a Bruce Powel Douglas concept and covers both structural- and behavioural patterns, but in an embedded hardware context. These patterns involve subjects such as memory allocation, memory sharing, concurrency and layered architectures.

2.3. Problem Statement

The basis for this report is the implementation of a Patient Simulator System. The product implementation is used for application and reflection of design choices.

The problem statement for this report is:
"How do we design an embedded real-time system that fulfils the requirements of the Project Description[5] and incorporates design patterns and OOA/D methodology?
What architecture- and design patterns should be incorporated to create a design that is easy to extend and maintain and yet has acceptable real-time performance on the embedded platform?"

The following section will give a short introduction to the Patient Simulator System and set the scope for the implementation according to the problem statement.

2.4. Presentation of the Patient Simulator System

In this exercise we could choose between two products to implement, a Patient Simulator System (PSIMU) or a Local Monitor System (LMON).
The Patient Simulator System project was chosen because it raised a lot of interesting real-time design questions: How do we transform discrete measurements into a real-time output that acts on external real-time inputs? How can we design a system that is very flexible when it comes to patient scenarios?

- How do we transform discrete measurements[6] into a real-time output that acts on external real-time inputs?
- How can we design a system that is very flexible when it comes to patient scenarios?

The purpose of the PSIMU is to simulate human physiological signals that can be measured by a patient monitor system. The simulated signals are based on real recordings taken from the PhysioBank database. The signals are processed by mathematical models that take inputs from example an injection pump.

Based on the requirement specification and the problem statement, we have chosen to limit the project to implement use case 1. In doing so, we can keep focus on software design, rather than product finalization.

2.5. Report structure

Chapter 3 describes the implementation of the theory into the PSIMU project. It starts out by presenting the projects' context and the development process used. The remainder of the chapter follows the ROPES process, starting out in chapter 3.3 with the problem domain analysis and taking it into the software domain, where design patterns are applied in chapter 3.4. Chapters 3.5 + 3.6 describe the tool chain involved in development process. At the end of the report is a list of references.

Hard-/Soft Real-Time

Architectural Design

Mechanistic Design

Event and time driven

Soft vs. hard real-time

3. Project Description (Kim)

In this chapter we will describe the project in developing the Sapien 190 patient simulator (PISMU), simulating human physiological behaviour like pulse and ECG signals according to different patient records and scenarios. We will describe the methods and process in details covering analysis and design. Finally we will give a presentation on the achieved results and experience obtained.

3.1. Project context (Kim)

This project is part of the course Embedded Real-Time Systems (TIIRTS) in the graduated course on Technical IT. The learning objectives for this course are listed below:

- *Analyze and describe* requirements for an embedded real-time system
- *Design and construct* an architecture for an embedded real-time system
- *Judge and use* design patterns in development of an embedded real-time system
- *Develop* a product documentation for an embedded real-time system using UML

In development of the patient simulator we will focus on achieving the leaning objective of this course as described above. That means our focus for this report has been on analysis, design and documentation of the embedded patient simulator. More information on the product documentation can be found in the requirement specification reference [5] and architecture document reference [4]. In this project we have had focus on applying design patters introduced in the TIIRTS course for an embedded real-time system. In the report you will find a detailed discussion with arguments for why and how we have used different design patterns for the Sapien 190 patient simulator.

3.2. Project execution (Kim)

In the overall planning and execution of the Project we have been inspired of the ROPES[7] Development Process and Scrum. In this chapter we will describe how we have used parts ROPES and Scrum in planning and execution of the project.

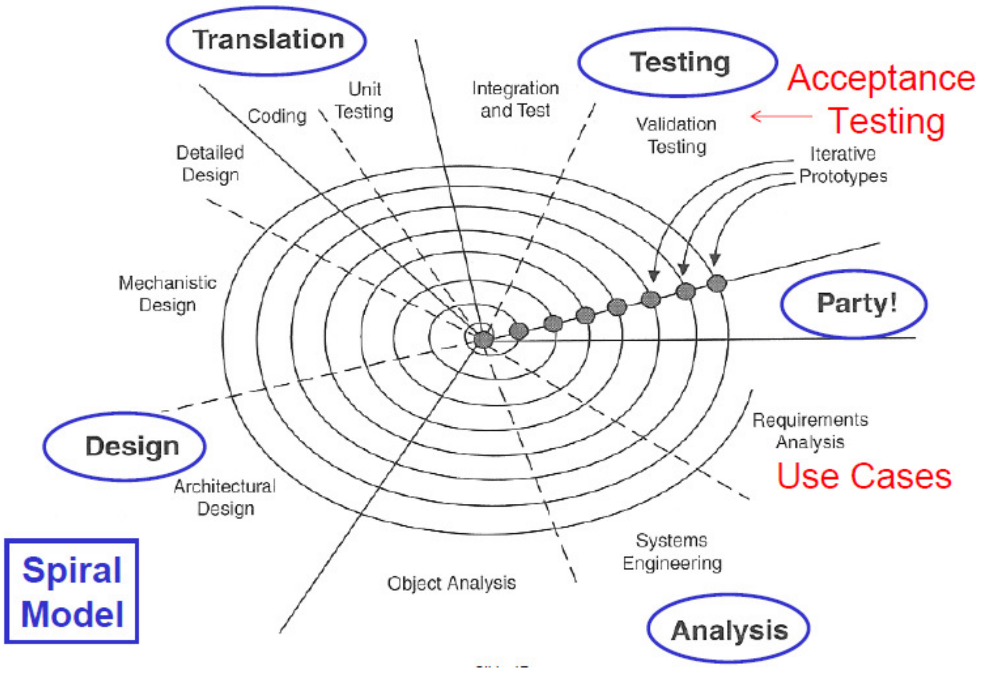

Figure 1 ROPES Spiral Micro-cycle (Detail)

The ROPES spiral micro-cycle model defines a number of phases to be executed for an iteration of a new prototype release. The purpose is to learn how the prototype performs being able to improve the design and implementation for every iteration. It also allows us to extend the functionality of the product in smaller steps and allows us always to have something to demonstrate. It starts with the basic requirements defined by use cases that have a significant impact on the architecture design of the product. After a number of iteration we will end up with the final product by implementing more and more use cases and functionality for each iteration. We have used the ROPES methodology for the steps we have followed for each prototype in the Sapien 190 project. We have produced a number of prototypes with different purpose to investigate the platform, technologies and finally the implementation of the functionality for the product.

In the analysis phase we have started with the requirement analysis by delivering a use case specification[8] for the Sapien 190 product to our "Customer = Teacher". Here we have identified a number of actors and use cases a shown in Figure 2.

[7] Rapid Object-oriented Process for Embedded Systems described in [2] chapter 3
[8] See use case specification for Sapien 190 in references [5]

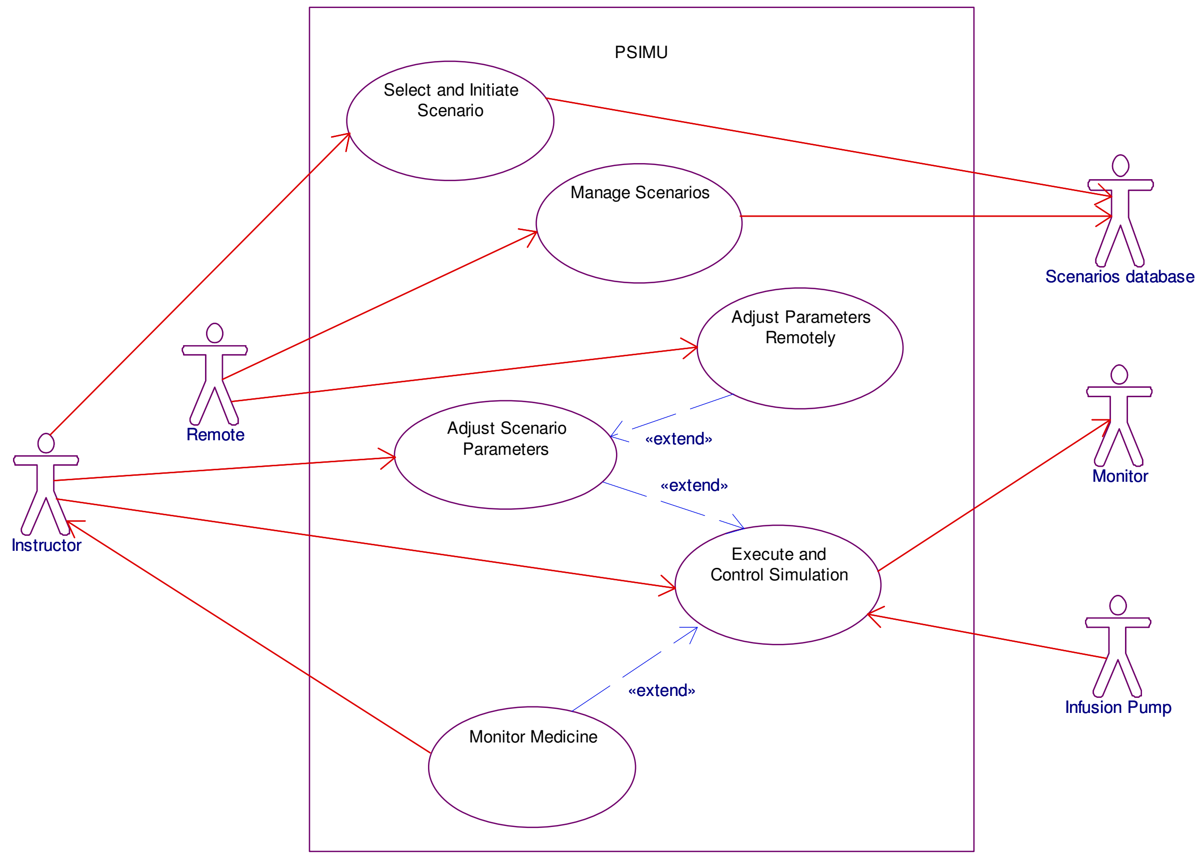

Figure 2 Use case specification and analysis

The next step was to make a domain model for the use case "Execute and Control Simulation" and complete the first iteration in the ROPES spiral micro-cycle. The use case "Execute and Control simulation" is the most complex and significant for the architecture of the real-time part of the patient simulator. This fist official prototype we have delivered to our "Customer" together with a first version of the architectural documentation and a demonstration of the Sapien 190 patient simulator. In the W-Model[9] by Alistair Cockburn he defines external visible deliveries for the external stakeholders of a project being able to follow the progress and give feedback to the project.

First Delivery 11. May 2010

- Updated Requirement Specification
- Draft Product Architecture Document
- Status Report
- Simulator prototype first version (Part of Use Case #1)

Final Delivery 4. June 2010

- Requirement Specification

[9] Slide 43 from Lesson 8 – L3_ROPES_process

- Product Architecture Document
- Project Report
- Simulator prototype (Use Cases parts of #1, #2 and #3)

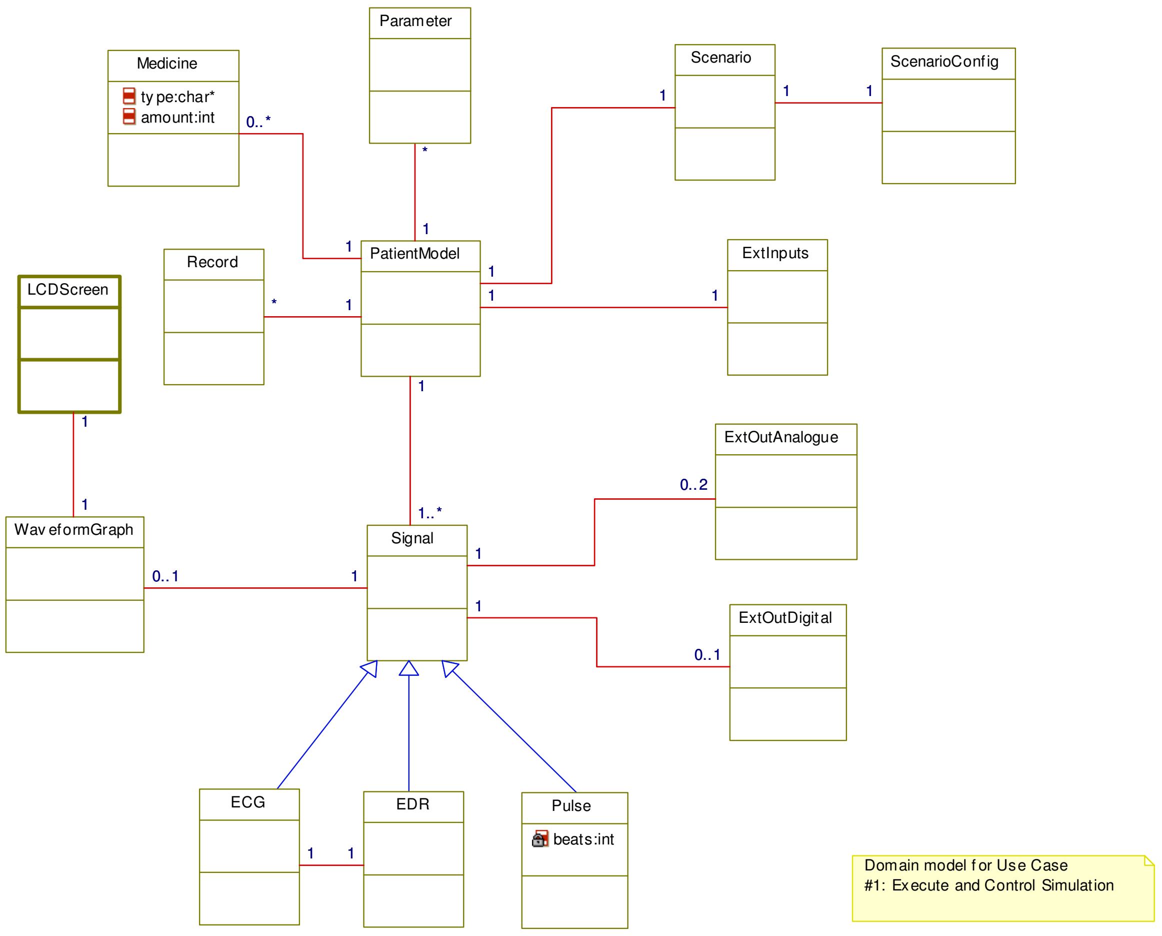

Figure 3 The first domain model for UC#1 Control and Execute Simulation

At the same time using the ROPES micro-cycle spiral to develop the actual product we did make a number of experimental prototypes to reduce the number of unknowns and risk in the project. Some of the questions we had in the beginning were:

- Would it be possible to generate the analogue ECG signal from user space in Linux on the target platform with the sampling rate of 250Hz?
- What would the CPU load be?
- How to plot the waveform ECG signals on the LCD screen using Qt?
- How to cross-compile the patient record WFDB library to the Linux target platform?

The first prototype was to reduce risk by investigate how to use the embedded Linux platform in reading patient record on the target and output the analogue ECG signal. The second prototype was to investigate how to develop with Qt and creating a waveform ECG graph based on the patient record readings.

The following prototypes has been about producing functionality according to the use case specification, where for every iteration parts of a use case or more use cases has been included. These prototypes has been developed by a combination of automatic code-generation from a Rhapsody UML model and source code manual written for the HW and OS abstraction layers. The GUI has been written using Qt from Nokia and integrated with the manual code and code generated from Rhapsody.

ROPES describe the key enabling technologies like visual UML modeling, model execution and mode-code associativity. These key technologies has been possible to realize in this project by use of the IBM Rapsody UML design tool not only for drawing UML diagrams, but also for high level modeling and automatic code generation. This approach has enabled the development process to focus on design instead of implementation. The ROPES key technologies has been a help to make fast iterations between testing new design ideas, by animation of state chats, setting breakpoints in Rhapsody and inspecting variables and states in the model. This higher level modeling approach compared to normally coding, debugging and testing has turned the development process to be more design focused than traditional development. We have used the animated sequence diagrams for test documentation and generating the code directly to Linux and the target platform, which has saved us for time fixing typing errors and trivial debugging and test.

The Rational Unified Process (RUP) has a process workflow that specifies a number of phases: Inception, Elaboration, Construction and finally Transition. In each phase a number of iterations are perform like in ROPES. In the beginning focus is on the process workflow requirements and business modeling. In this project we have primary been working in the elaboration phase working with 2 major iterations of product release. Since focus for this course in some extend has been design patterns, we have use most the time in the elaboration phase with focus on the architectural and mechanistic design of the project.

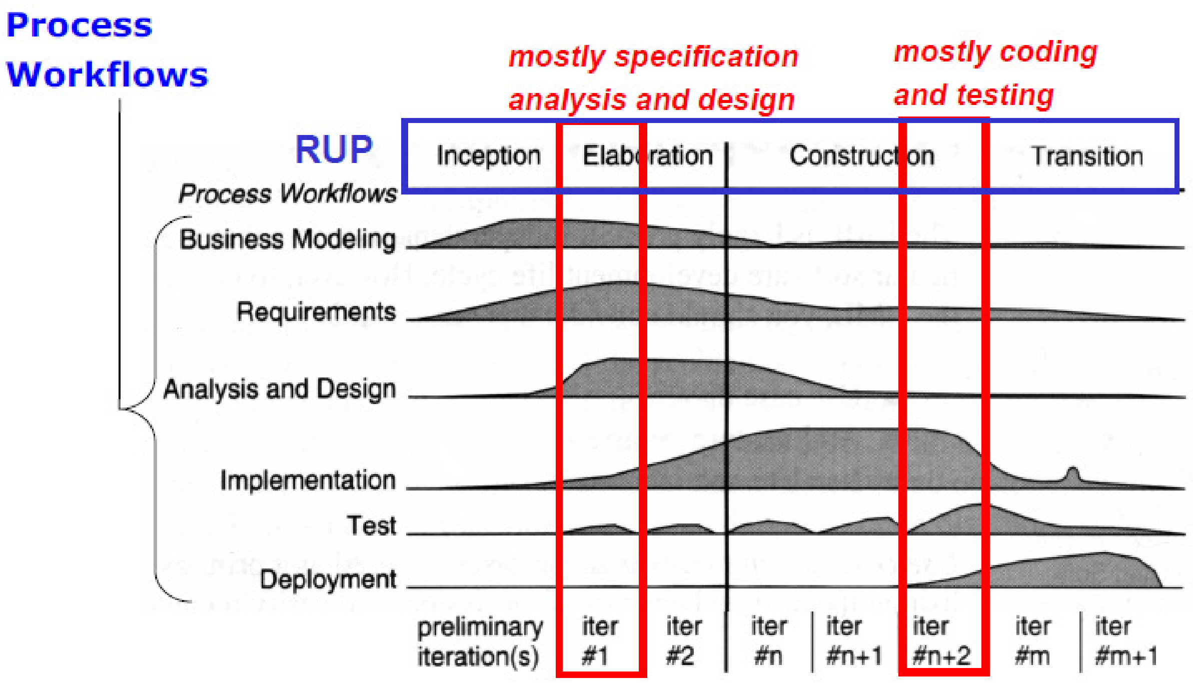

Figure 4 Rational Unified Process workflows

In controlling the progress of the project we have had regularly meetings in which we have been inspired from the way Scrum meetings are organized. In every meeting we have used time on each project member giving a short status of what has been done, issues and problems discovered since last meeting based. The status is based on the planned activities from last meeting. An updated list with notes of meeting status has been updated for every project meetings see appendix A. On every meeting the backlog in Scrum terms has been updated according to our eventually changes priority. We did not define a scrum master or product owner, but on every meeting one of the project members did take the lead on updating the status list and together we did a prioritation of the backlog (Pending activities). Below see example for the contents of meeting status.

Scrum Status 25. May:
____

<Name>:
Done:
- Meeting minutes
- Added text to Chapter 5

Problems:
- What exactly to put into ch 11+12

Next:
- Architecture document chapter 11. and 12.

<u>Next scrum meeting Tuesday at 9:00</u>
- Scrum status
- Status on writing Architecture document
- Status on report writing and assignment of tasks

- Continue to 16:00

Action list (Backlog) to final delivery 04. June:

- Finalize Product Architecture Document
- Finalize Project Report
- Implement ECG to Pulse filter

…..

Notes to be deleted:
Scrum meetings and minutes – Lesson 8 – Scrum short
Schedule – Specification
Part of ROPES – Lesson 8 – L3_ROPES_Process
ROPES Microcycle for UC#1 (3-4 weeks for D1)
Iterations see specification D1 and D2
Prototyping (First prototype with technology clarification)
Model Base Development (Testing design in Rhapsody)
Risk analysis – see minutes notes (NotesMeeting1904_2010.txt)
Fast prototype to reduce risk of new technology
WFDB patient record library on target
HW interfacing on target and load
GUI programming with Qt

3.3. Analysis process and methods (Kim)

In the requirement analysis phase with reference to the ROPES spiral model we have created a number of use cases to define the required functionality of the product. A detailed specification of this work can be found in "Requirement Specifiation for Sapien 190" [5]. The first step has been to define the actor-contex diagram to identify the actors of the patient simulator. Here we have used the specifications of the PSIMU, LMON, IPUMP and interface specification referenced in [5] chapter 1.2

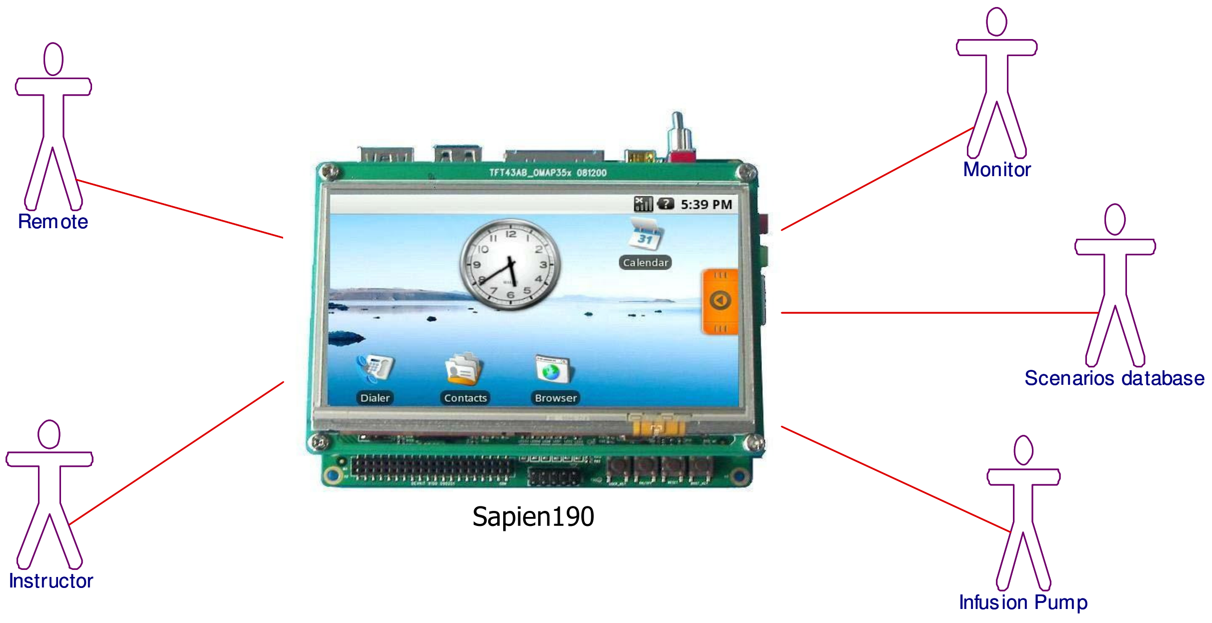

Figure 5 Actor-Context Diagram

The second step in the use case analysis was to identify a number of use cases for each actor as illustrated in use case diagram Figure 2. These use cases is described in details in the identified to be:

1. **Execute and Control Simulation**
2. **Select and Initiate Scenario**
3. **Adjust Scenario Parameters**
4. **Monitor Medicine**
5. **Manage Scenarios**

For each use case specification a main scenario and extension is described like on page 9 in the "Requirement Specification for Sapien 190" [5]. This document contains the textual description of the system context, interface to external actors, non functional requirements like performance, prioritized product qualities and design constrains. Finally the document contains a description of the original planned deliveries of use cases to be contained in the two first deliveries to our "Customer". We have prioritized maintability, correctness and usability as the most important quality factors for the patient simulator. The simulator application is expected to have a longer life time than the hardware and must be able to be maintained for many years in the future. Therefore it is important for the product to focus on these factors in creating the architecture and design.

The domain analysis is based on the use case requirement specification where we did start with the use case "Execute and Control Simulation". This approach is defined in the Unified Process[10] (UP). A domain model is created to identify the conceptual domain classes and relations between them to give us a better and deeper understanding of the actual problem. We have chosen the "Execute and Control Simulation" use case to be the first in creating this domain model since it is the most complicated and contains the essential functionality for the patient simulator. The UML domain class diagram is illustrated in Figure 3. To this first domain model we added boundary and one control classes. The purpose of the boundary classes is to separate the model from the external actors and the control class is used to encapsulates the control of the scenario described in the UC#1.

[10] Craig Larman, Applying UML and Patterns, Third Edition chapter 9 Domain Models

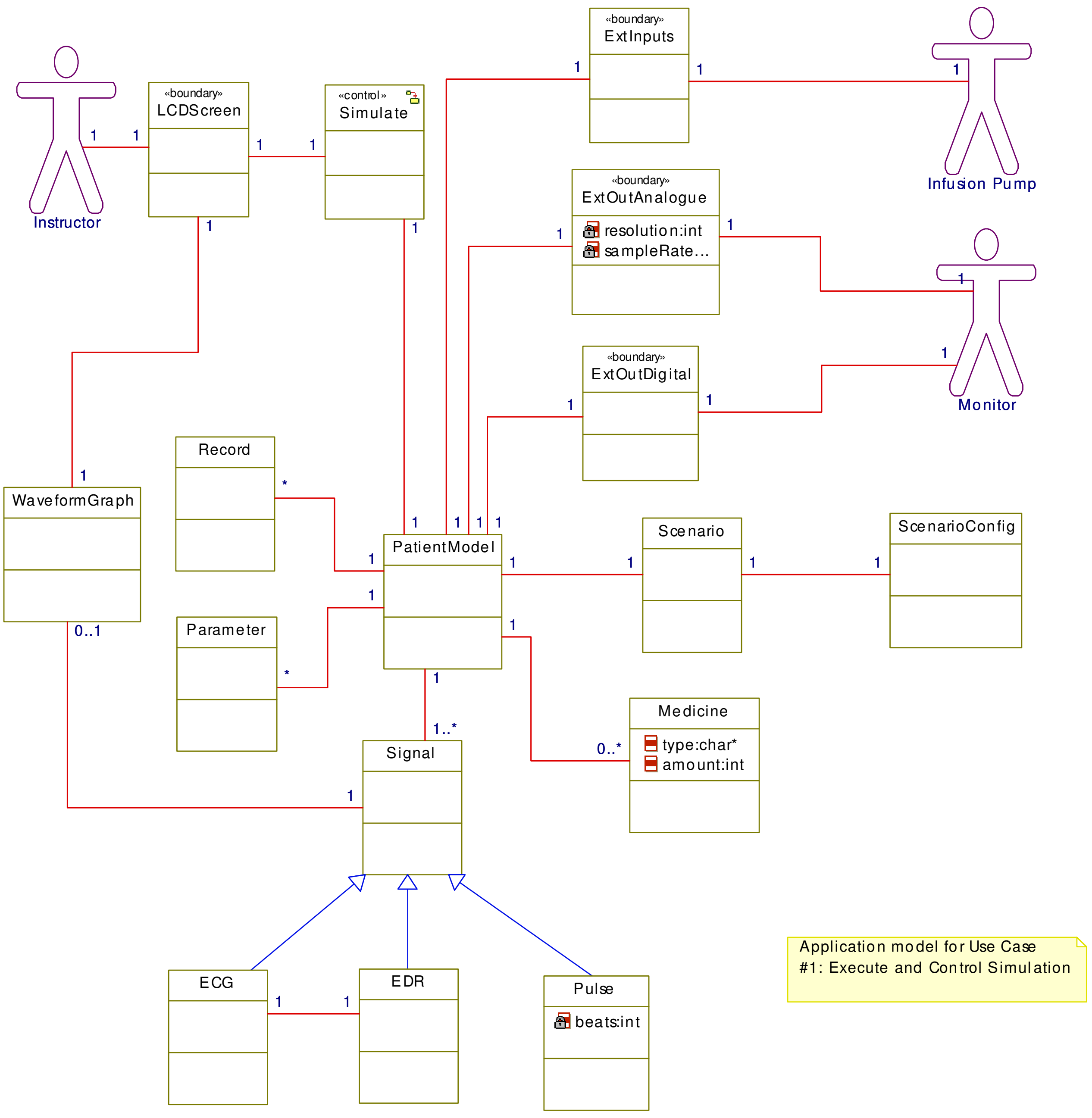

Figure 6 Domain model with boundary and control classes

In the domain model for this first iteration we have identified all the important classes and the relations between them. The domain model is saved in a separate Rhapsody project since the domain model in the following design iterations is changed a number of times. The domain model illustrates how the patient model is the central class for coordinating the patient simulation. When the patient simulation is running it will perform reading of the digitized physiologic signals from a record and send theses "real time" values as analogue signals to local connected bedside monitoring equipments. Up to 2 analogue channels with different signals is possible to be simulated simultaneously. The simulated signals can be ECG or EDR. The pulse will be signaled to the bedside monitoring equipment as a digital signal. The basic idea is that the instructor later should be able to select between different scenarios that describes the configuration of a simulation which include a certain patient model (Normal or with Infusion Pump), patient records from the PhysioBank[11] database and parameter settings for the simulation like gain and rate for replaying the patient record. More details can be found in the requirement specification [5]. In the UML state diagram below we have added a simple functionality for the instructor to start, stop, pause and resume the patient simulation with a fixed scenario configured for the final prototype. That means UC#1, UC#2 and UC#3 is implemented without being able to select a scenario in UC#2.

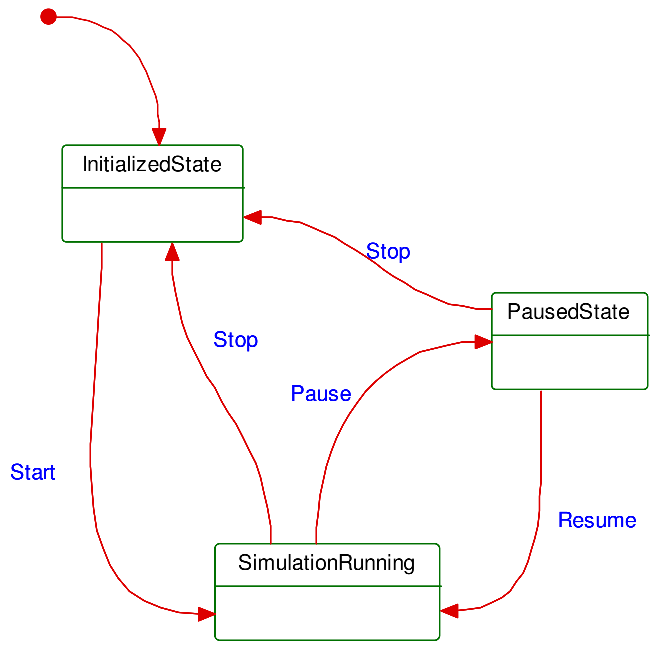

Figure 7 State chart for domain control class simulate

Part of the analysis phase we have identified a number of external and internal generated events that plays an important role in constraining and defining the system behavior. The events listed below are crucial for analysis of the schedulability and deadlines later in the design process (RMA analysis). For this analysis we have defined number of parameter that can be adjusted to find the optimal timing and scheduling being able to generate the ECG, EDR and the pulse signal. At the same time we should be able to processing information from the IPUMP and updating the waveform signal graph on the LCD screen. A frame buffer has been defined for the number of samples to be collected before updating the LCD screen.

[11] http://www.physionet.org/physiobank/physiobank-intro.shtml

Internal and external event list

#	Event Id	System response	Arrival Pattern	Event Source	Response time
1	Sample	Calculate and generate EDR and ECG signals	Frequency of 250 (Fs) max 400	Internal timer	Less than sample periode
2	Pulse	Caculate pulse every 200 (Np) samples	Fs/Np	Internal timer	Less than sample periode
3	PDU	Updates medicine information	Every second (Fi)	IPUMP	Less than ½ second
4	FrameBuffer	Updates signal graph on LCD display	Every 50 (Nf) samples (1/8 of LCD display in pixels)	Internal timer	Less than period updating framebuffer

Parameters identified to be used for RMA analysis in design phase:

-	Time units	Default value	Units
Fs	Sample frequency	250	Hz
Np	Num samples for pulse calculation	200	Number
Fi	PDU frequency	1	Hz
Nf	Frame buffer size	50	Number

Notes to be deleted:
Analysis method (Identification of the essential characteristics of possible correct solution)
Use Case Specification and analysis work
Use case UML domain model analysis
Event analysis

3.4. Design process and methods (Kim)

We have used the 4+1 view designed by Phillipe Kruchten[12] as the process driver for the design work in the project. The Use Case view is the driver for focus on what to design in the other 4 views as show in the figure below. We have started with UC#1 defining a number of scenarios that is used as the foundation for the architectural, mechanistic and detailed design that is described in details in the "System/product architecture document for Sapien 190" reference [4].

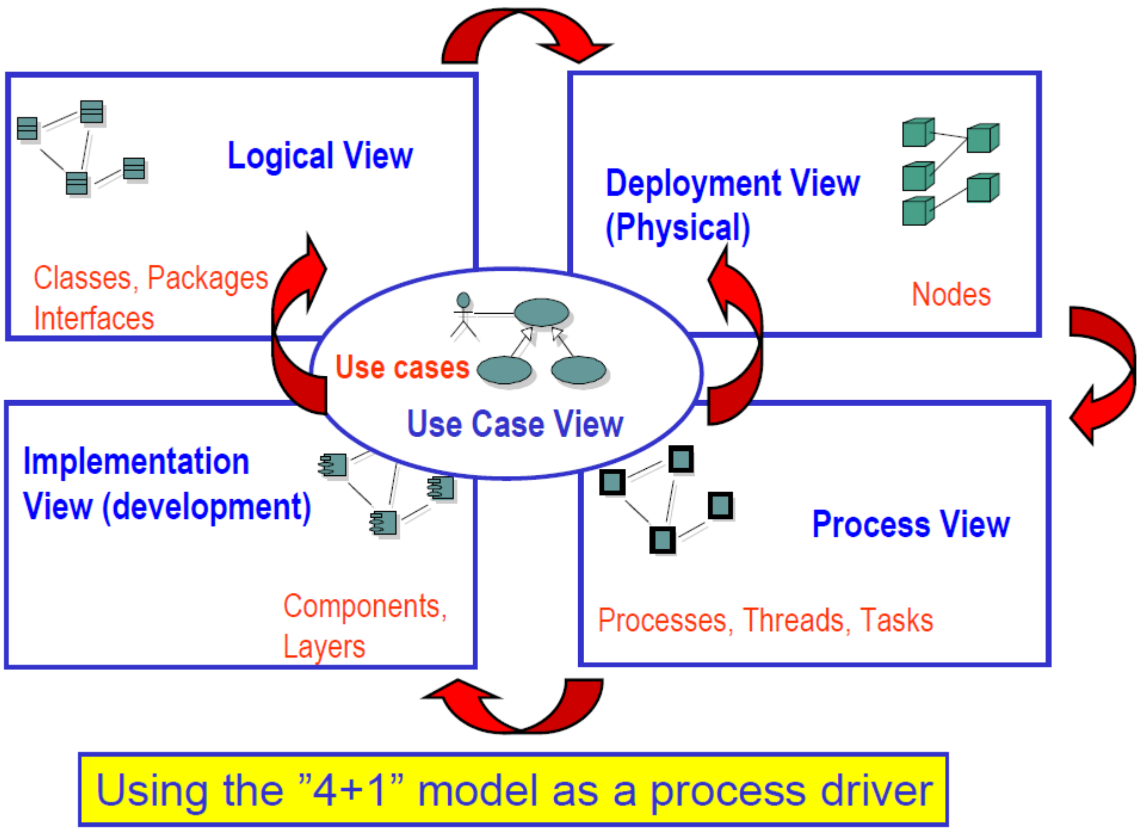

Figure 8[13] "4+1" view model

The UC #1 has been selected for the first iteration of the ROPES spiral microcycle in making the architectural, mechanistic and detailed design. This use case is significant and provides the central functionality of the patient Simulator and contains the real-time constraints. This use case provides the basis functionality that allows the monitor to be connected being able to display the ECG and EDR signals. It also reduces the risk for developing the patient monitor since it covers all the unknown technologies of the product like:

- Reading the patient record files on the target (Linux, WFDB and target)

[12] Kruchten, Phillipe, Architectural Blueprints – The "4+1" View Model of Software Architecture
[13] Slide 4 from Lesson 8, DevelopmentOfRealtimeSystems

- Generating the analogue output signals (Writing to drivers)
- Display of signal waveform using Qt on target (Working with Qt on target

The deployment, logical, process and implementation views are designed according to the steps described in development of real-time embedded systems[14]:

Step 1: Sketch an Actor-Context Diagram

Step 2: Find and document use case #1, #2 and #3 (Chapter 4. and 5.3 Ref. [4])

Step 3: Select a suitable HW architecture (Chapter 7. Ref. [4])

Step 4: Develop a logical model (Chapter 5. Ref. [4])

Step 5: Select a concurrency model (Chapter 6. Ref. [4])

Step 6: Design, Implement, Test and Measure (Chapter 6.5 Rate Monotonic Analysis Ref. [4])

The architectural design is based on the five-layer architecture pattern described in chapter 4.2 reference [2] illustrated in the package diagram shown below.

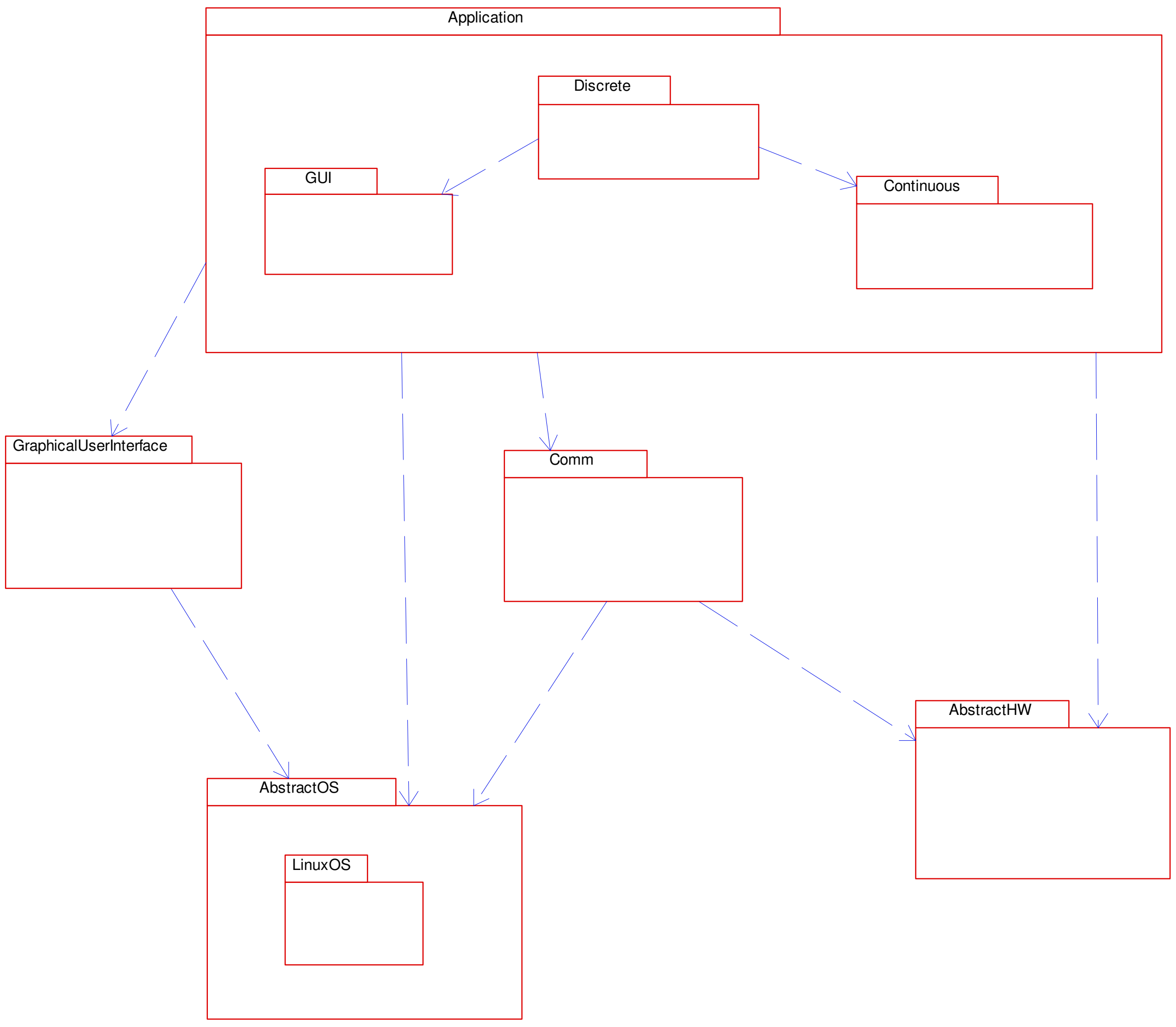

Figure 9 Five layred architecture for logical view

[14] Slides 1 – 17 Lesson 8, DevelopmentOfRealTimeSystem

Each abstraction layer is a logical layer representing a well defined domain. Dividing the system into several layers ensures high cohesion for each domain and low coupling between the domains. This simplifies the process of modifying our design or extending it. The application package uses the two-part[15] architectural model for the discrete and continuous parts of the patient Simulator. These packages contain also the most complex part of our design.
The diagram below shows the essential classes defining the architecture for use case #1. One of the important design considerations is to ensure that the dependency between the layers only is in one direction only. Design patterns have been used between the discrete and continuous package (Observer) and between the continuous and communication packages (Mediator) to ensure this one way dependency. Details about use of design patterns will be described in the following sections.

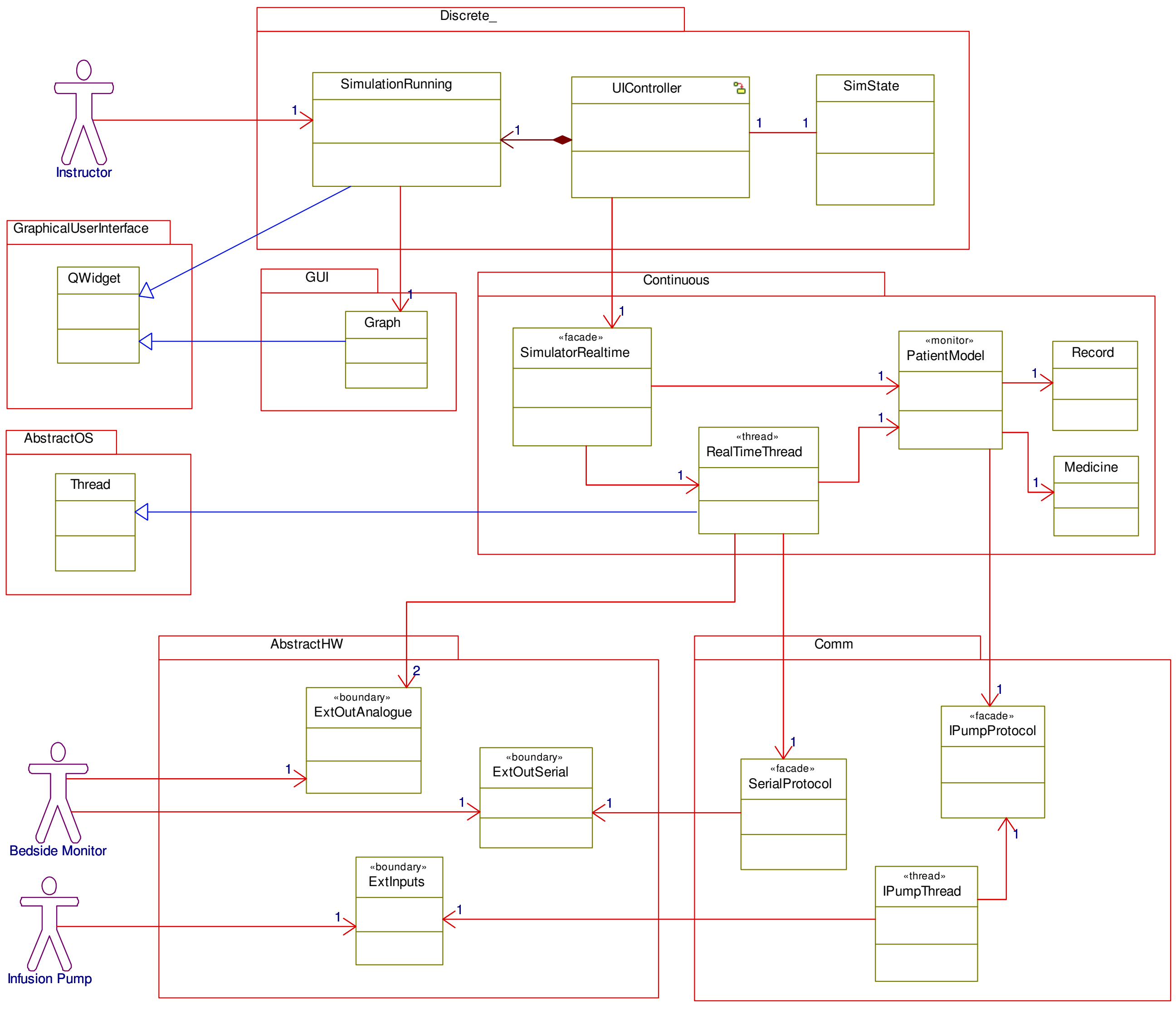

Figure 10 Essential classes in architecture for use case #1 execute and control simulation

[15] Hans Peter Jepsen, Finn Overgaard Hansen, Designing Event-Controlled Continuous Processing Systems

After having defined the overall architecture of the design we have focused on the mechanistic design of the continuos and discrete package by refining the domain model in where we have introduced a number of creational, strutural and behaviroal design patterns from the GoF[16] book.

The list below briefly summarize the patterns that has been used in the design:

Behavioral Patterns (GoF)

- **Observer** (Updating waveform graph) – To notify the UI with a new frame buffer of samples that is updating the waveform graph. By use of this design pattern it would be easy to extend the design with new observers like creation of a FFT view of the ECG signal. This solution separates the continuous package being direct depended on the UI controller in the discrete package.
- **Command** (Setting parameters) – Encapsulates setting parameters in the continuous package by sending a request as an object, thereby letting the UI controller being able to send different type of parameters. It hides the way the parameter is updated in classes that belongs to the continuous package.
- **State** (UI controller in discrete package) – The UI controller depends on its state. The state pattern has been used to handle state changes in combination with the command pattern. Currently this choice is perhaps a bit overkill compared to the simple state machine described in Figure 7, but it will be easy to extend when adding selection of scenarios and new functionality in the future.
- **Command** (In combination with state) – The command pattern has been used in combination with the state pattern to encapsulate generating events to the state machine in the discrete package. This design makes it easy to add new events when the state machine is extended in future software releases.
- **Mediator** (Updating medicine information from IPUMP) – Promotes a loose coupling between the communication package and the continuous package. It provides a way to extend the product by adding different types of devices that can provide inputs to the patient model and thereby manipulating the patient simulation.
- **Strategy** (PhysioModel) – It defines the family of algorithms to compute the patient simulation. Different types of algorithms can easy be added in combination with the filter and pipes pattern. Currently we have designed the NormalModel and InfusionPumpModel more details to be found in chapter 3.4.3.
- **Iterator** (Reading records) – To provide a way to access samples in the records without exposing the underlying representation. The same iterator is used for reading patient records and generating signals for testing.

[16] Gang of four, Erich Gamma and co., Design Patterns, Elements of Reusable OO Software

Structural Patterns (GoF)

- **Proxy** – (Reading records) - To provide a place holder for reading records. The proxy pattern has been selected for future extension in where the proxy pattern could be used in combination with the broker pattern as described in chapter 8 reference [2]. This approach would allow us to access records directly from the PhysioBank database on the internet reference [3].
- **Façade** - (Class in continuous package SimulatorRealtime) – To provide a simple unified interface for the subsystems of the real-time part of the patient simulator. It makes it easier for the UI controller to operate on the complex structure of classes in the continuous package.

Creational Patterns (GoF)

- **Factory method** (Creating records and filters) – The factory method pattern has been used to make it easy to create new patient record objects. Methods in the façade of the SimulatorRealtime class are provided to generate new objects based on the record file name.
- **Singleton** (DAC + Command State pattern) – Ensure that the states only has one instance and is only created ones. This approach saves a lot of new and deletes operation every time a state is changed in the UI controller. The DAC also uses a modified singleton ensuring we only have one instance per DAC channel. The flyweight[17] pattern could as an alternative be used to ensure only one object per channel. Would be a better approach moving to a platform with many DAC channels.

Concurrency Patterns (B.D.)

- **Message Queuing Pattern** (A mailbox of frame buffers) – Used to transfer frame buffers as asynchronous communication messages between the real-time thread and the distributer thread. The frame buffers a used to update the waveform graph by using the observer pattern.
- **Monitor** (Classes PatientModel and FrameBufferPool) – Used to protect shared information between the different threads in the system. The classes PatientModel and FrameBufferPool are designed as monitors since different threads should be synchronized in invocation of methods operating on the same data. This solution separates the communication package being direct depended on the continuous package.

Memory Patterns (B.D.)

- **Pool Allocation Pattern** (Allocation of frame buffers) – Creates a pool of frame buffers that is used by the real-time and distributer threads. Frame buffers are created on startup and available on request by the real-time thread. This approach save time performing new and delete operation and it ensures a more predictable real-time thread.

[17] The Flyweight pattern is a structural pattern in GoF. The pattern is used to share a large number of fine-grained objects efficiently.

Other patterns

- **Filters and pipes** (Filter and calculation of samples) – The filters and pipes pattern is used to compute samples for the different signals (ECG, EDR and Pulse) based on record readings. The structure of the filters and pipes are setup by the strategy (NormalModel and InfusionPumpModel). Different filters is designed for generating the EDR signals, adjusting the gain of the input signal and generating the pulse based on the ECG input signal. The filters and pipes design pattern makes it easy to add more filters or setup of new strategies for computing signals.

In the following chapters we have chosen to give a summary based on the architecture document of some of the essential designs we have implemented. We will only give a summary of the deployment, logical and process view since these views are the most significant for the scope of this project. A fully detailed version of all views is to be found in reference [4]. There will be a discussion on the design considerations and choices we have made. We have also give a more detailed discussion on some of the design patterns described above.

3.4.1. Deployment view

Our target platform is the DevKit8000 that is running Linux. The Sapien 190 application is dependent on Qt libraries and library for the touch screen. The essential hardware components for generating the ECG and EDR signals is the digital to analog converters (DAC) on the add-on board. Linux drivers were already implemented being able to send one sample value at every write to the driver. A better solution would be to write a more complicated DAC driver that was able to send out a buffer of samples based on a sample rate set from the application. This driver could be implemented using the kernel timer API or creating a process in kernel space. This solution would be more accurate than to use a thread running in user space as we have done in this project. Since Linux driver development is not part of this course we have decided not to choose this path in our development strategy. Actual the solution we have made seems to be good enough as long the target platform is not running other heavy CPU consuming applications like Ethernet network traffic.

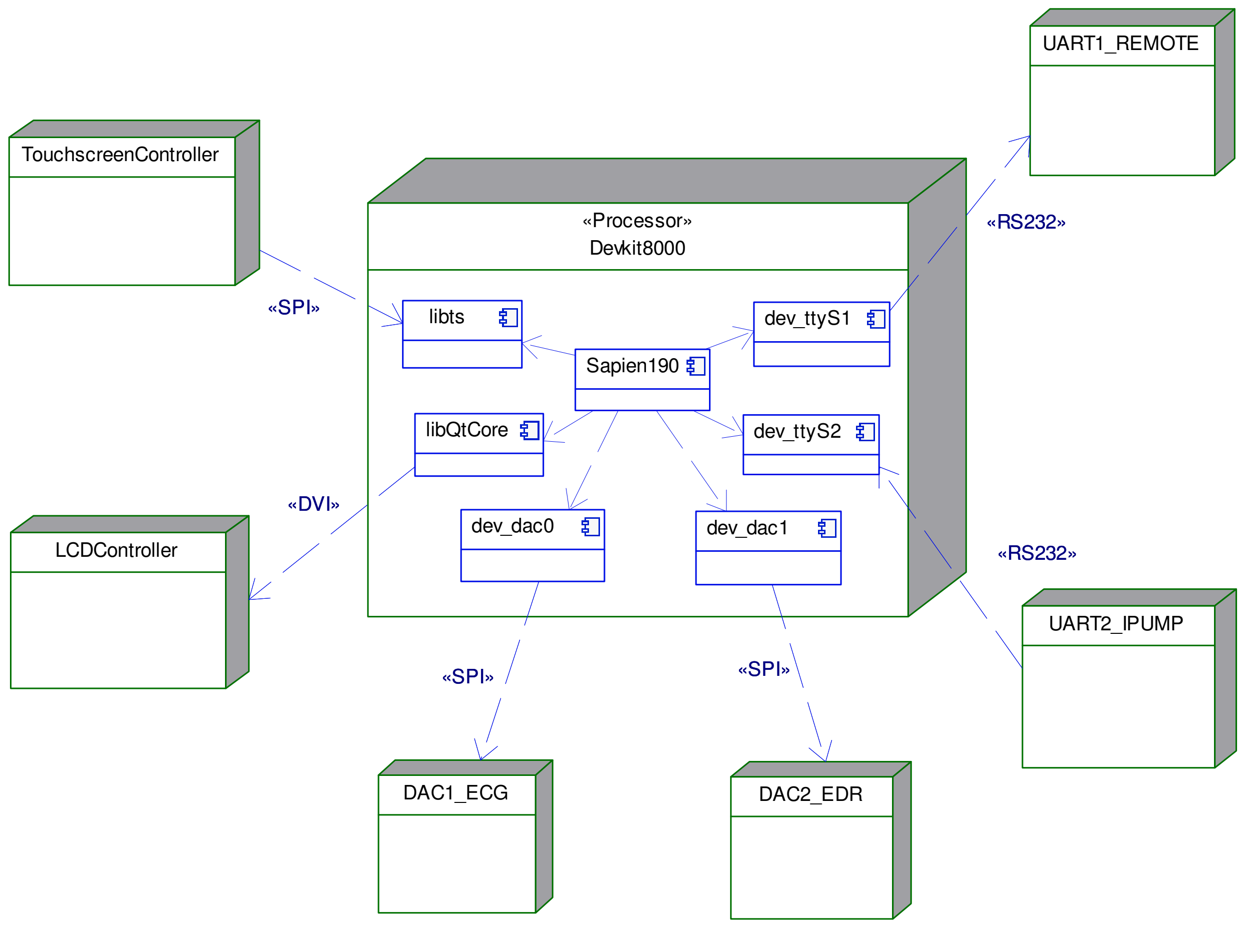

Figure 11 Essential HW nodes used from DevKit8000 used for Sapien190

3.4.2. Logical view - discrete package (Anders)

The figure below illustrates how the command and state pattern is used to implement the state machine for the UI controller in the discrete package. We have modified the state pattern represented with the SimState class and its sub-states. We have implemented a bidirectional association between the context (UI Controller) and state (SimState) to remove passing the reference to the context every time a command is executed and the state is changed. Instead the reference to the context is passed to the state when it is created. The approach has been implemented for the command pattern. Every time a command is created an association to the state is established and thereby saves passing a parameter to the execute method in the command. Since this part of the design is using a number of Qt classes and primitives like slots and signals we have only used Rhapsody as a drawing tool for this part of the design see complete UML class diagram for design of the command and state pattern.

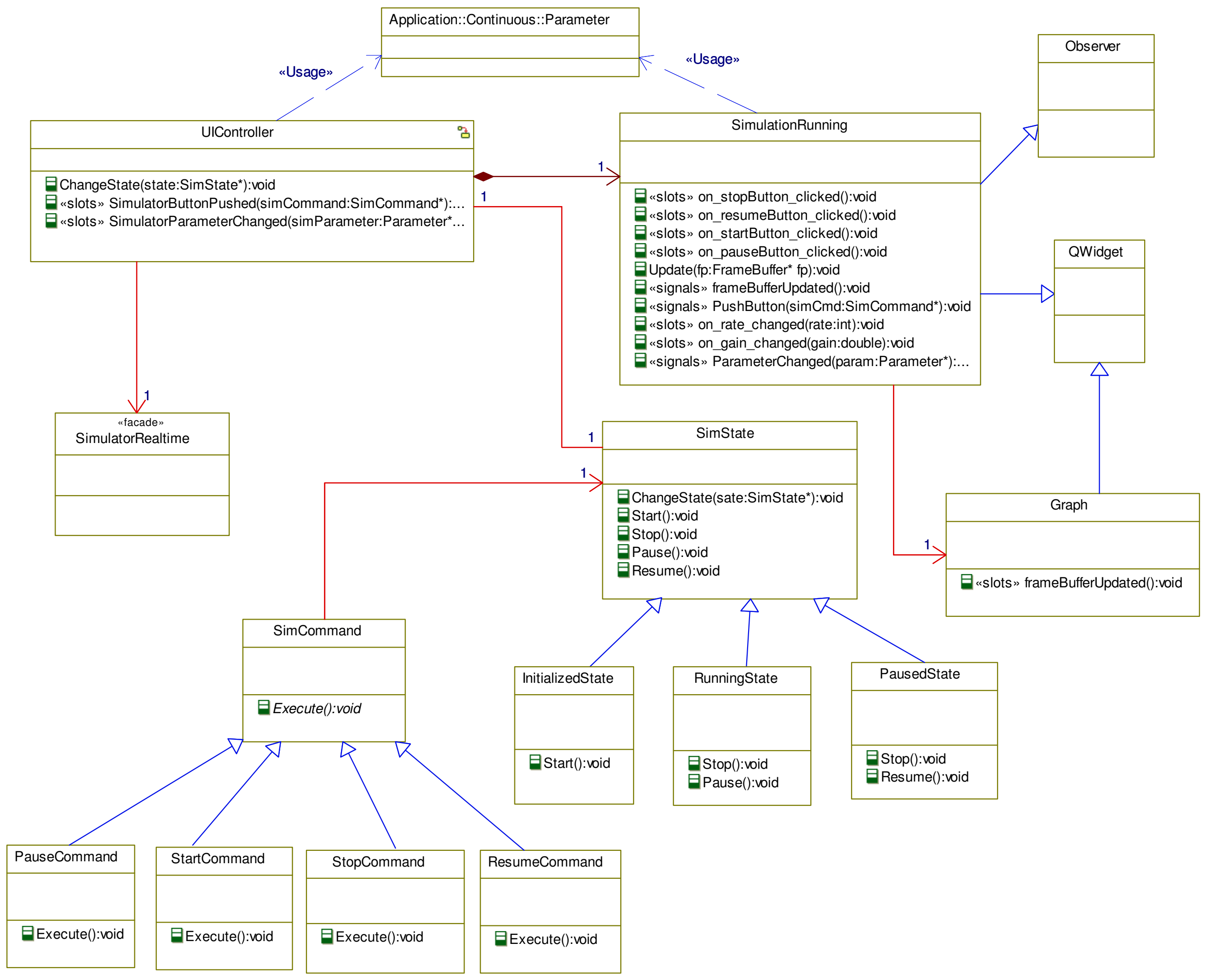
Application::Continuous::Parameter
«Usage»
«Usage»
Observer
UIController
ChangeState(state:SimState*):void
«slots» SimulatorButtonPushed(simCommand:SimCommand*):...
«slots» SimulatorParameterChanged(simParameter:Parameter*...
SimulationRunning
«slots» on_stopButton_clicked():void
«slots» on_resumeButton_clicked():void
«slots» on_startButton_clicked():void
«slots» on_pauseButton_clicked():void
Update(fp:FrameBuffer* fp):void
«signals» frameBufferUpdated():void
«signals» PushButton(simCmd:SimCommand*):void
«slots» on_rate_changed(rate:int):void
«slots» on_gain_changed(gain:double):void
«signals» ParameterChanged(param:Parameter*):...
QWidget
«facade»
SimulatorRealtime
SimState
ChangeState(sate:SimState*):void
Start():void
Stop():void
Pause():void
Resume():void
Graph
«slots» frameBufferUpdated():void
SimCommand
Execute():void
InitializedState
Start():void
RunningState
Stop():void
Pause():void
PausedState
Stop():void
Resume():void
PauseCommand
Execute():void
StartCommand
Execute():void
StopCommand
Execute():void
ResumeCommand
Execute():void
1
1
1
1
1
1

Figure 12 Command, State and Observer pattern used to design SapienApplication (Controller)

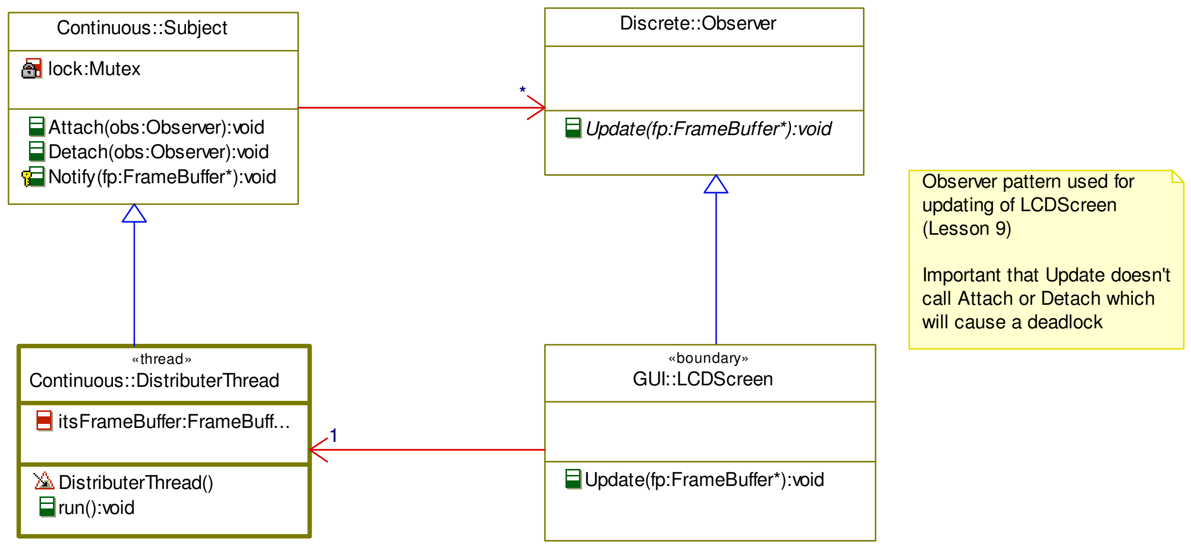
Continuous::Subject
lock:Mutex
Attach(obs:Observer):void
Detach(obs:Observer):void
Notify(fp:FrameBuffer*):void
*
Discrete::Observer
Update(fp:FrameBuffer*):void
Observer pattern used for updating of LCDScreen (Lesson 9)
Important that Update doesn't call Attach or Detach which will cause a deadlock
«thread»
Continuous::DistributerThread
itsFrameBuffer:FrameBuff...
DistributerThread()
run():void
1
«boundary»
GUI::LCDScreen
Update(fp:FrameBuffer*):void

Figure 13 Observer pattern used to notify GUI with new frame buffer

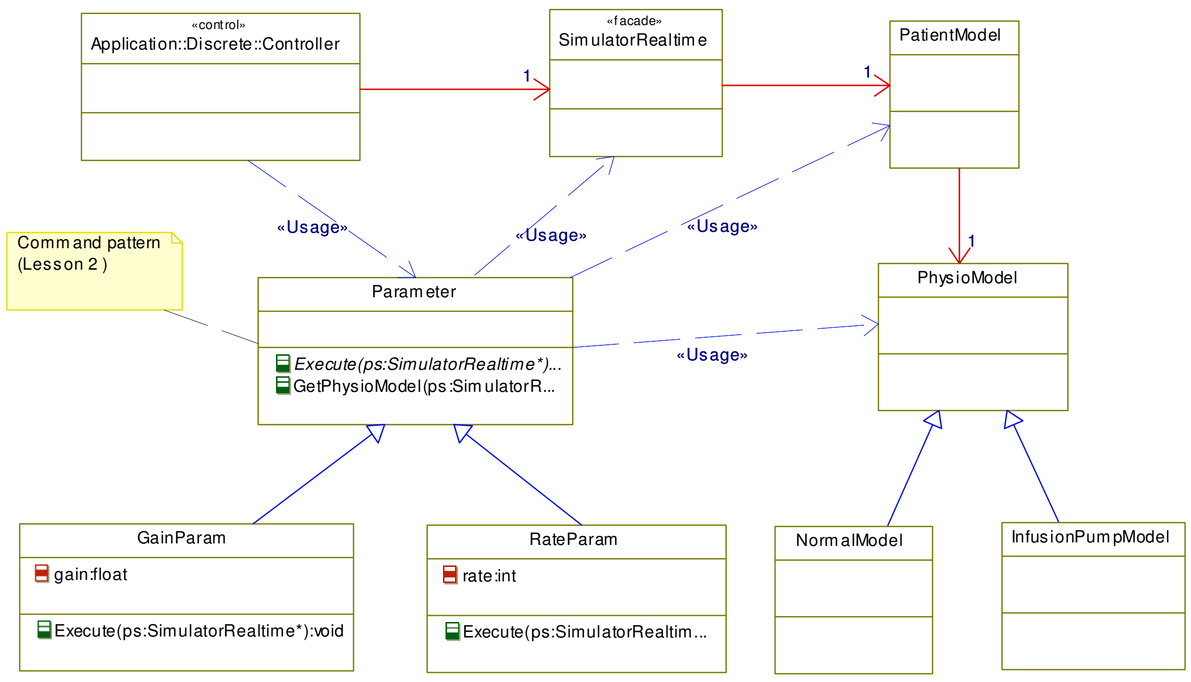
«control»
Application::Discrete::Controller
1
«facade»
SimulatorRealtime
1
PatientModel
«Usage»
«Usage»
«Usage»
Command pattern (Lesson 2)
Parameter
Execute(ps:SimulatorRealtime*)...
GetPhysioModel(ps:SimulatorR...
1
PhysioModel
«Usage»
GainParam
gain:float
Execute(ps:SimulatorRealtime*):void
RateParam
rate:int
Execute(ps:SimulatorRealtim...
NormalModel
InfusionPumpModel

Figure 14 Command pattern used to set parameters in real-time simulator

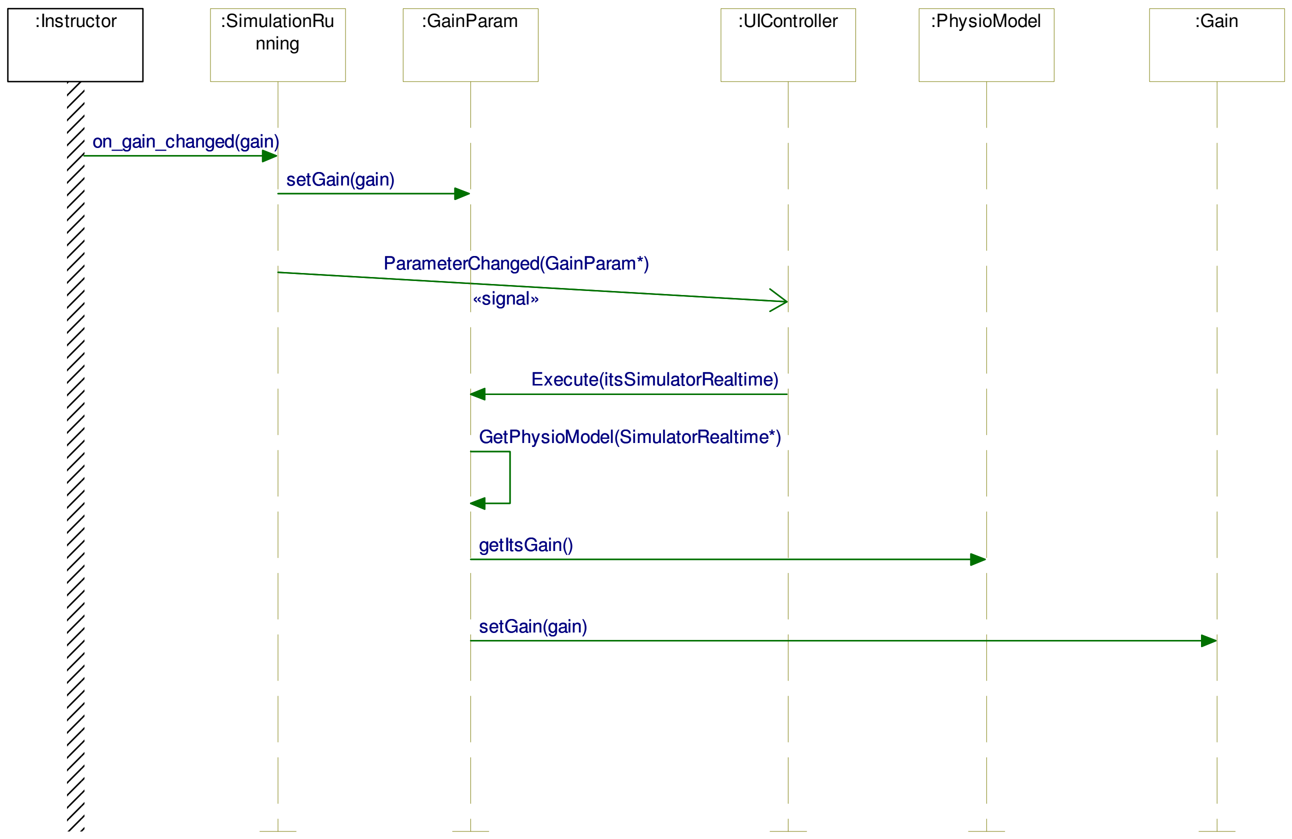

Figure 15 Scenario for instructor adjusting gain parameter

3.4.3. Logical view - continuous package (Peter)

TBD – description of package and design patterns.

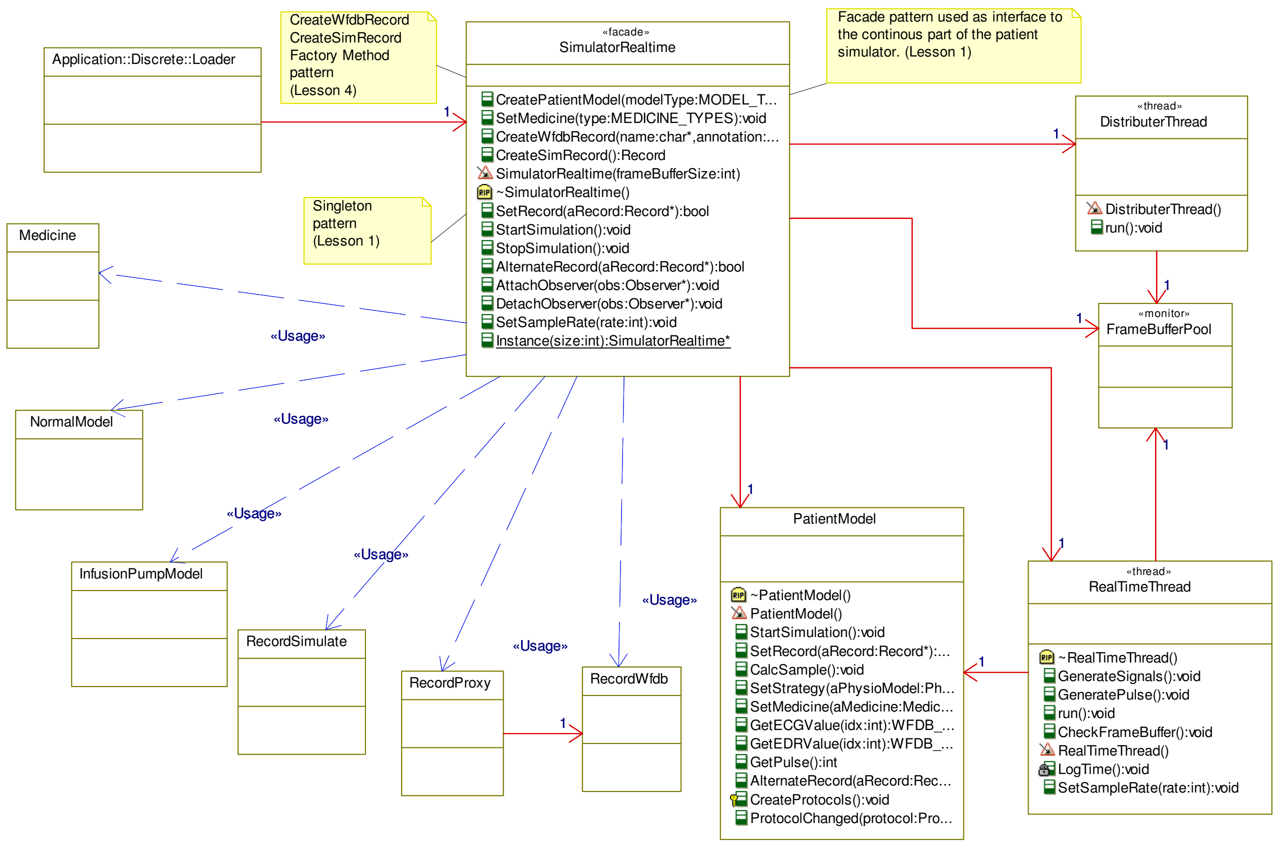
CreateWfdbRecord
CreateSimRecord
Factory Method
pattern
(Lesson 4)
Facade pattern used as interface to the continous part of the patient simulator. (Lesson 1)
Singleton pattern (Lesson 1)
Application::Discrete::Loader
«facade» SimulatorRealtime
CreatePatientModel(modelType:MODEL_T...
SetMedicine(type:MEDICINE_TYPES):void
CreateWfdbRecord(name:char*,annotation:...
CreateSimRecord():Record
SimulatorRealtime(frameBufferSize:int)
~SimulatorRealtime()
SetRecord(aRecord:Record*):bool
StartSimulation():void
StopSimulation():void
AlternateRecord(aRecord:Record*):bool
AttachObserver(obs:Observer*):void
DetachObserver(obs:Observer*):void
SetSampleRate(rate:int):void
Instance(size:int):SimulatorRealtime*
«thread» DistributerThread
DistributerThread()
run():void
«monitor» FrameBufferPool
Medicine
NormalModel
InfusionPumpModel
RecordSimulate
RecordProxy
RecordWfdb
«Usage»
PatientModel
~PatientModel()
PatientModel()
StartSimulation():void
SetRecord(aRecord:Record*):...
CalcSample():void
SetStrategy(aPhysioModel:Ph...
SetMedicine(aMedicine:Medic...
GetECGValue(idx:int):WFDB_...
GetEDRValue(idx:int):WFDB_...
GetPulse():int
AlternateRecord(aRecord:Rec...
CreateProtocols():void
ProtocolChanged(protocol:Pro...
«thread» RealTimeThread
~RealTimeThread()
GenerateSignals():void
GeneratePulse():void
run():void
CheckFrameBuffer():void
RealTimeThread()
LogTime():void
SetSampleRate(rate:int):void
1

Figure 16 Façade Pattern used for interface to the Continuous Package

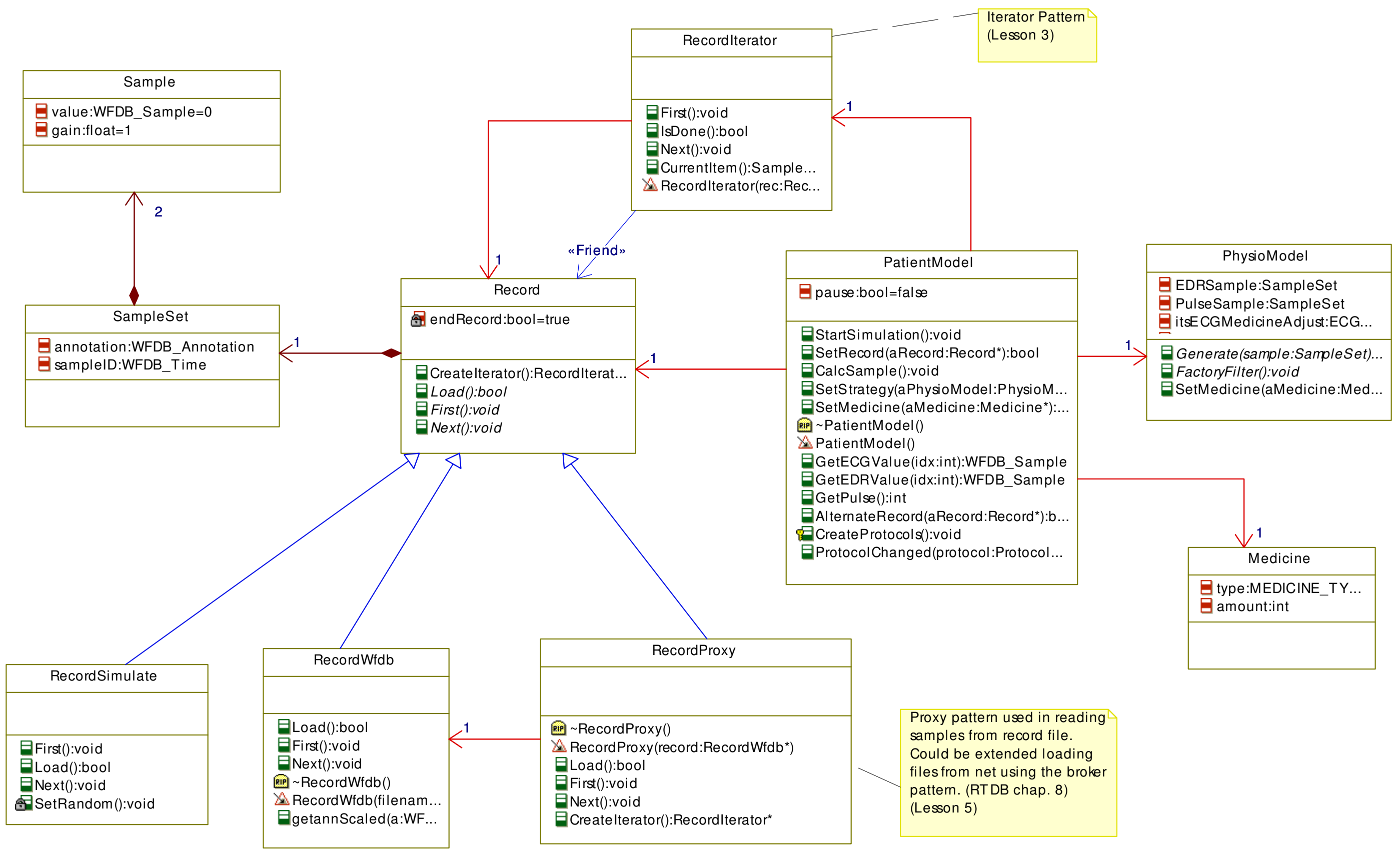

Figure 17 Proxy and Iterator Pattern used to access Records from the PatientModel

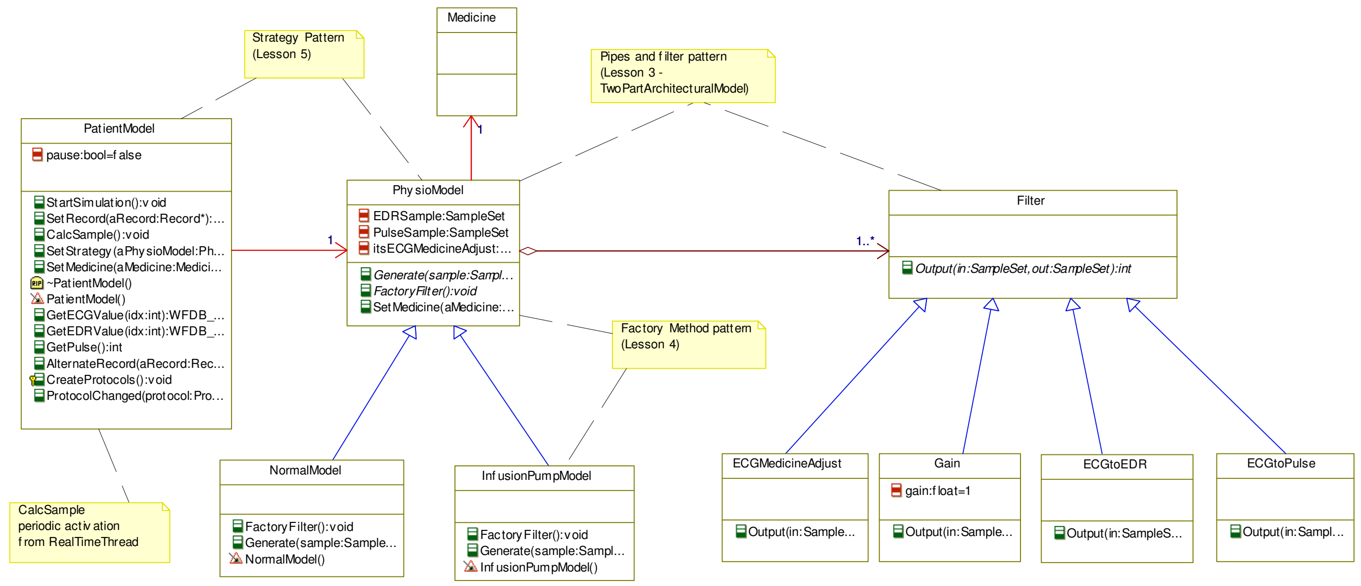

Figure 18 Strategy, Filter and Pipes Pattern used for PhysioModel

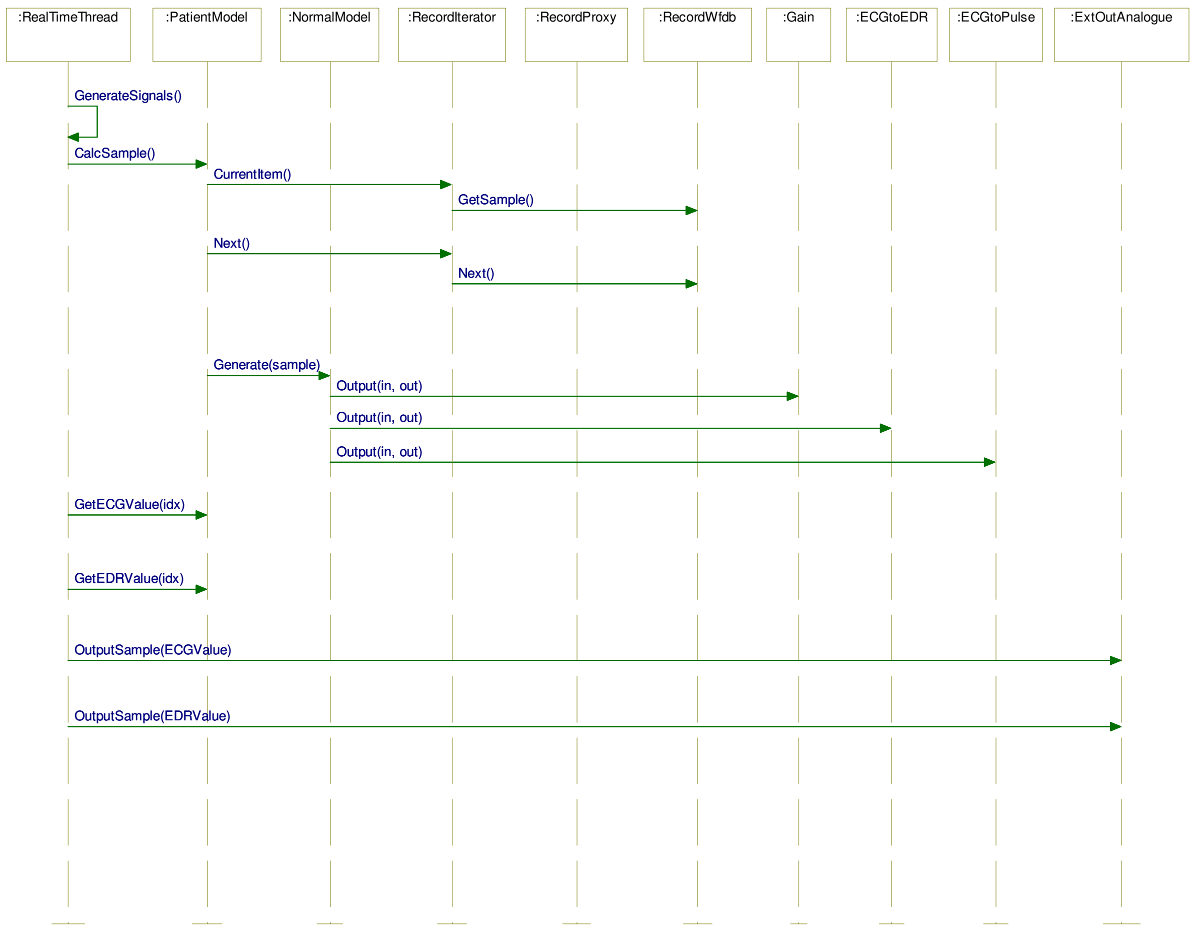

Figure 19 Sequence Diagram: GenerateSignals using filter and pipes

3.4.4. Logical view - communication package (Anders)

TBD – description of package and design patterns.

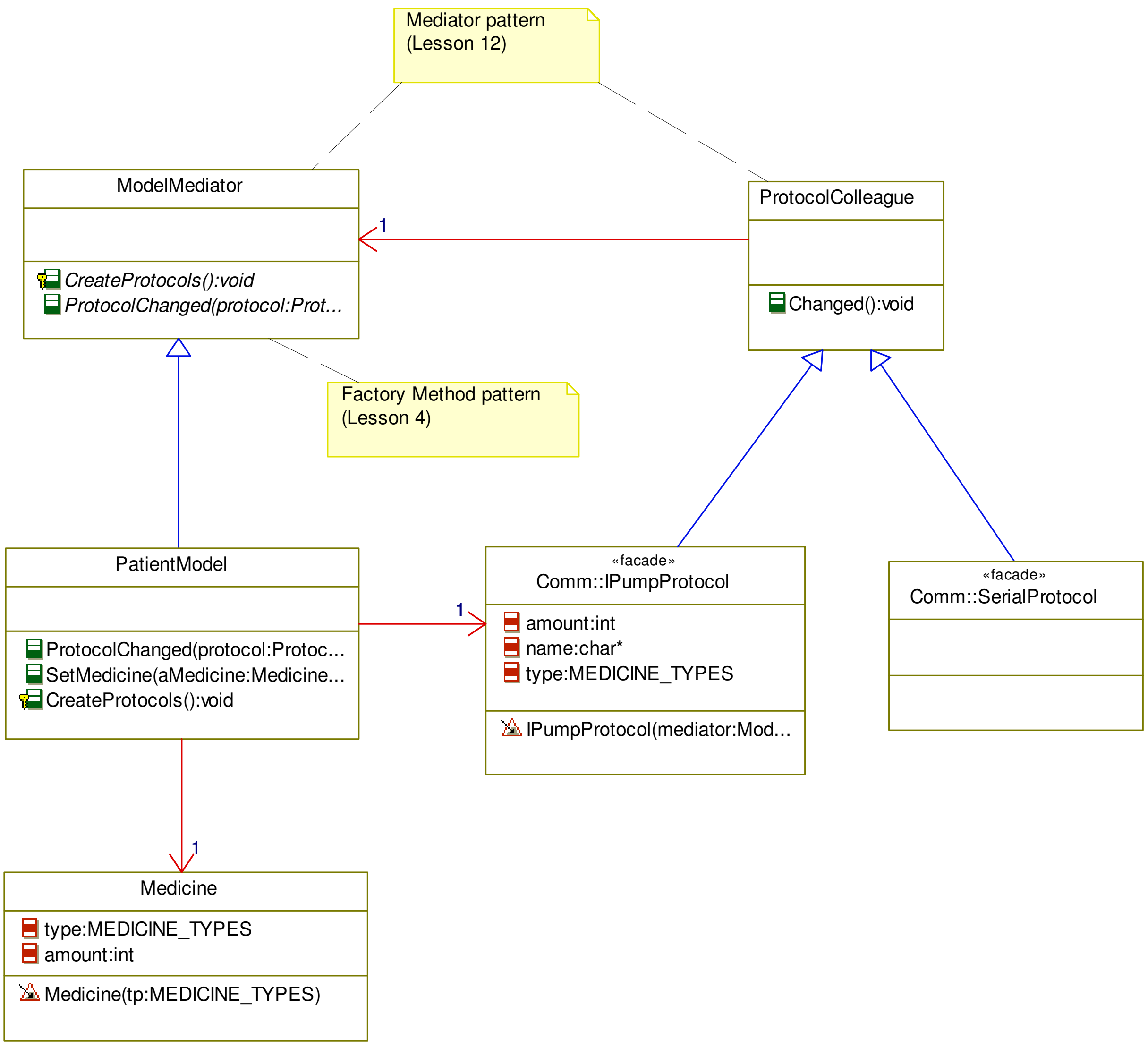

Figure 20 Mediator pattern used to update PatientModel with external input from protocols

3.4.5. Process view (Kim)

In this chapter we will briefly describe the process view of the Sapien 190 simulator. The following threads are running in patient simulator:

Controller:
The controller will be part of the discrete package that contains the state machine to control the continuous part of the two layered application model. Synchronization is needed between the controller and the continuous part of the system.

RealTimeThread:
This thread is the essential thread of the systems that is periodic with the sample rate and is responsible for generating signals and outputs them to the environment (Push). Alternatively this thread could have been implemented creating a buffer of samples and sending it to the DAC driver. The DAC driver should then be responsible for sending the buffer at a given sample rate (Pull). This approach would have been more efficient and accurate, since timing in kernel space of Linux would be more predictable in calculation of WCE. It also introduces less overhead in copy of data from user to kernel space.

DistributerThread:
This thread is used to collect a frame buffer of samples that is updated to the observers which in this case will be the controller class part of the "Qt-GUI thread". The DistributerThread is periodic with the sample rate times the number of samples in the frame buffer.

IpumpThread:
This thread is used to handle PDU messages received from the IPUMP and updating the medicine volume. PDU messages are received with a rate of 1 sec.

The controller class from the discrete part of the system will interact with the DistributerThread where the observer pattern has been used to update the GUI with frames of samples. A mutex has been added to this observer pattern to ensure synchronization between the controller thread and the DistributerThread. This synchronization will ensure that Attach is not called the same time as Notify, meaning that new observes are not allowed to be added or deleted at the same time we are updating the observes. The RealTimeThread generates samples and collects a number of samples before a new frame buffer are sent to the DistributerThread by using a mailbox (SendMail and GetMail). The FrameBufferPool is implemented as monitor to ensure synchronization between the RealTimeThread and DistributerThreads when they request the pool of free frame buffers.

Synchronization between the IpumpThread and RealTimeThread is not yet finalized since this part is still in the design phase. I could be done by adding a monitor or mutex to the mediator pattern to ensure that the medicine class is updated and access exclusively between the two threads.

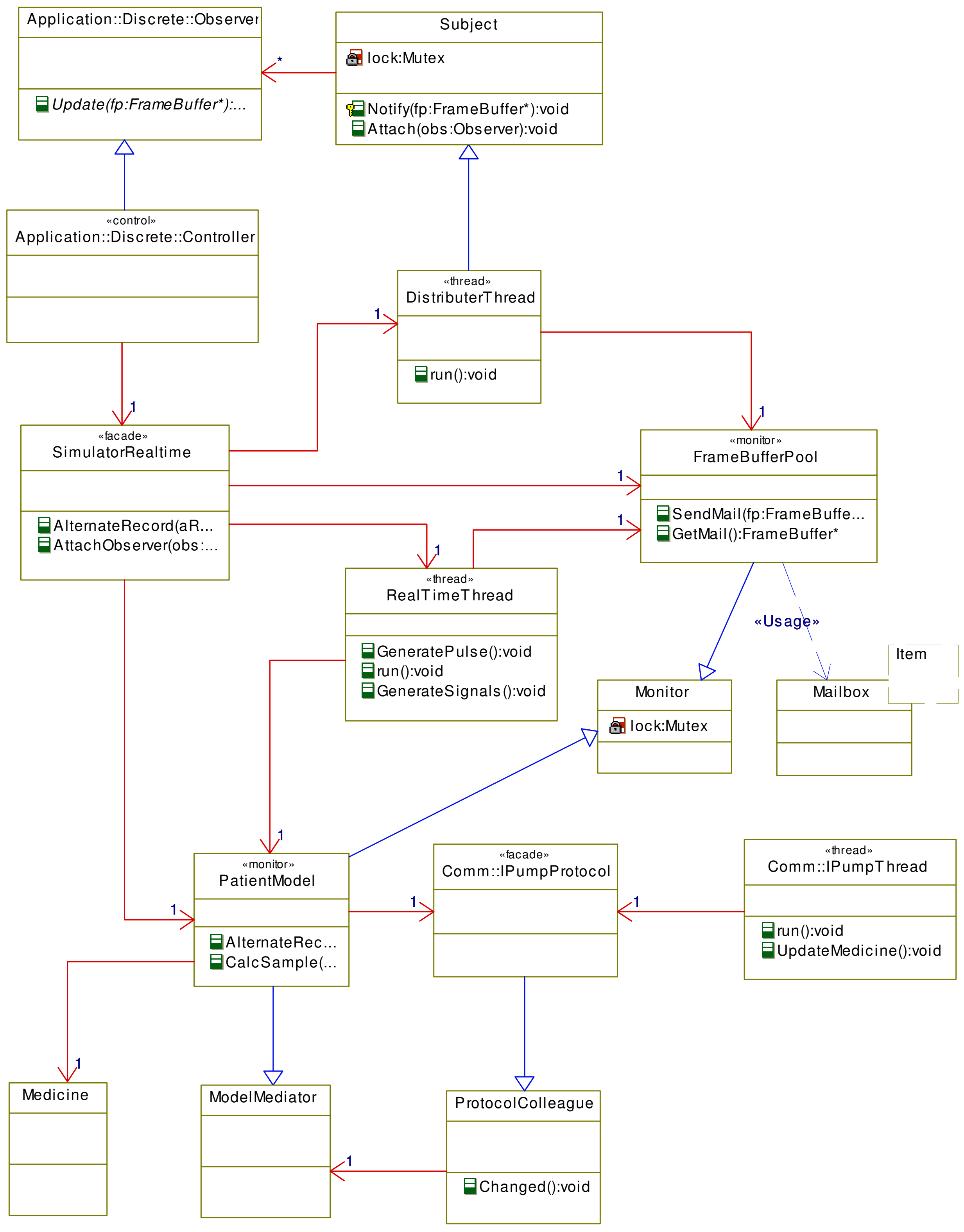

Figure 21 Overview of threads for the Sapien 190 design

The essential threads for Sapien 190 are the DistributerThread and RealTimeThread. These threads exchanges frame buffers with samples by use of a Mailbox implemented according to the Message Queuing Pattern. This pattern uses asynchronous communication between the two threads. The FrameBufferPool is implemented according to the Pool Allocation Pattern. This approach saves allocation of a new frame buffer from the heap every time the RealTimeThread will be filling the next buffer with samples. There is only allocated 2 frame buffers in the pool since the DistributerThread needs to distribute the newest samples to the waveform graph. In case it cannot follow the speed of updating the graph if the sampling rate is too high samples will automatically be skipped. We could also have used the Static[18] Allocation Pattern to achieve the same effect, but the GenericPool class is a more generic solution that can be used in other parts of the design for future extensions. The monitor is used by the FrameBufferPool to synchronize allocation and release of FrameBuffer's to the GenericPool.

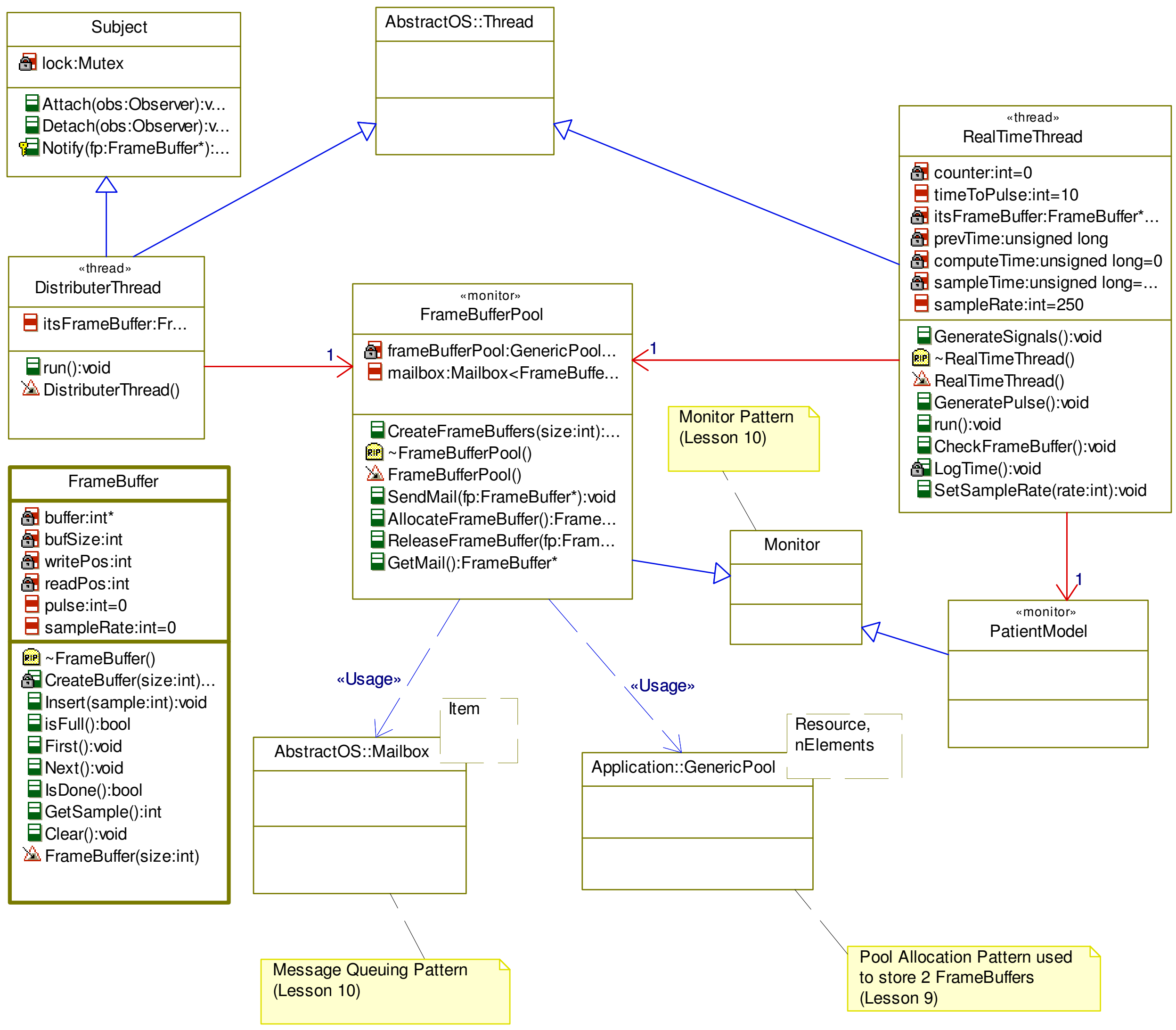

Figure 22 Process view for Distributer and RealTime threads and mechanism for synchronization

[18] Slide 3 Lesson 6 – MemoryPatters.pdf

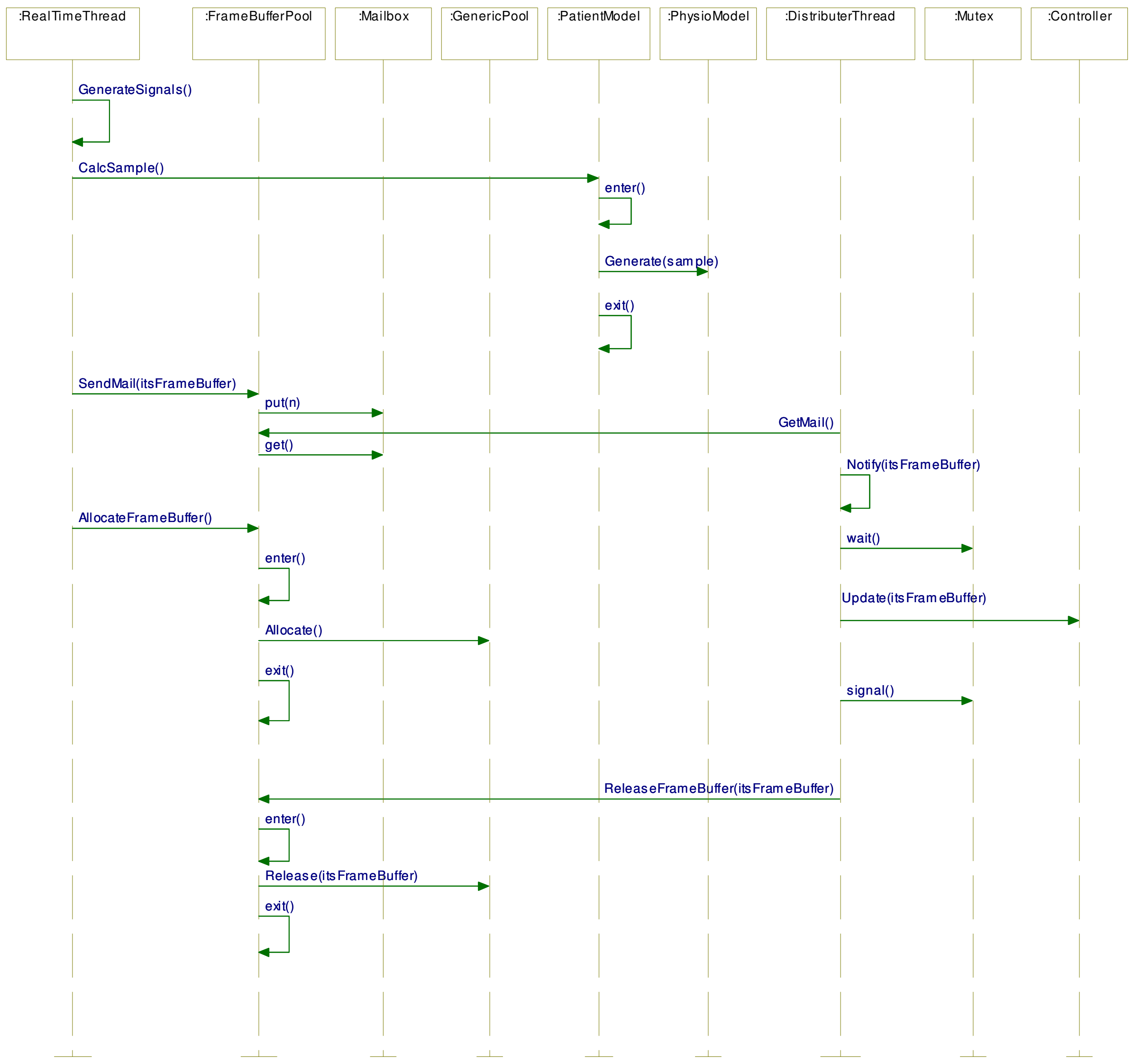

Figure 23 Synchronization between threads

The operating system is encapsulated in the classes displayed below. These classes are implemented in two versions. The first is to be used for simulation and test in Rhapsody. The second is implemented as an abstraction of the POSIX thread API used on Linux.

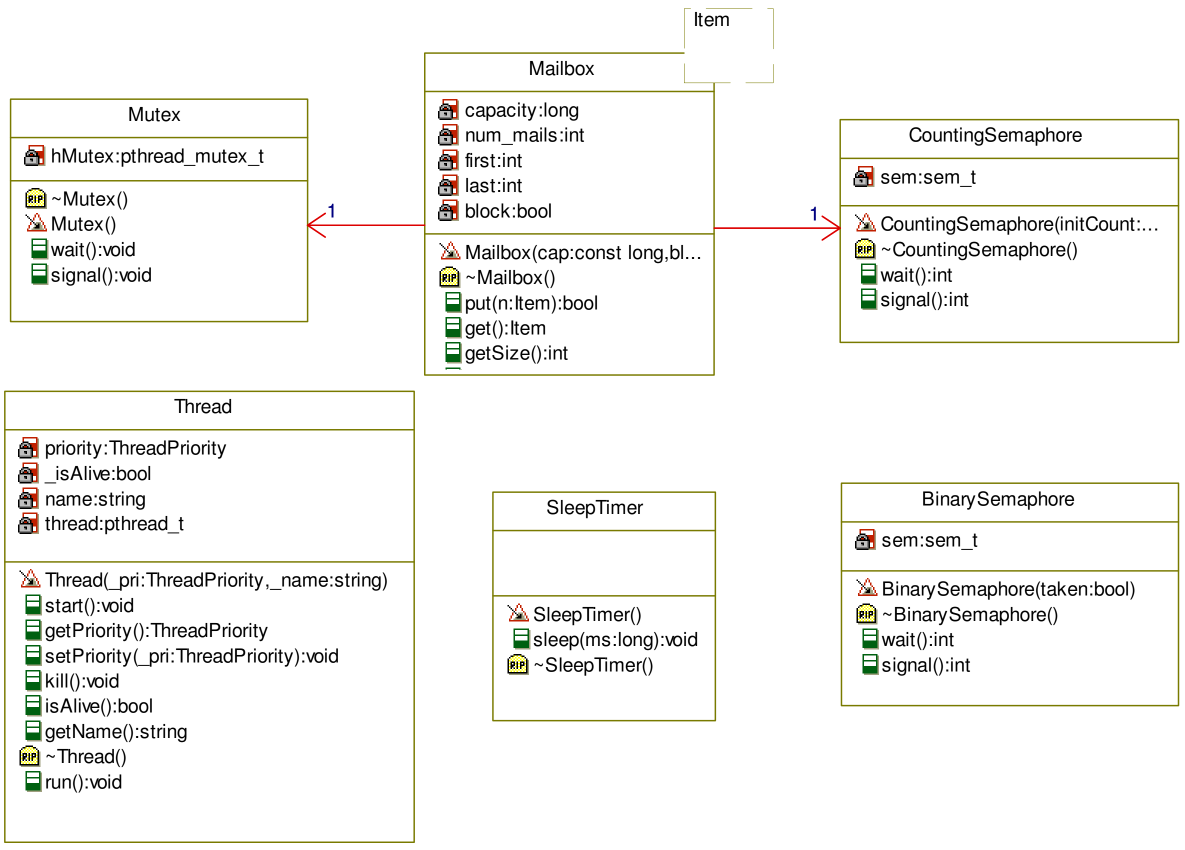

Figure 24 Abstract OS (Linux)

Based on the event analysis made in chapter 3.3 we have included a Rate Monotonic Analysis of the scheduling of threads. The WCE times and blocking delays is based on measurements and estimates. We have measured the time it takes to generate new ECG and EDR samples WCE (250) and the time it takes to update the waveform graph to only ~4000 us instead of the 100 ms (Bi for DistributerThread) used in the RMA below. All times below is in us.

#	Event Id	Arrival Period (Ti)	Action	Thread
1	Sample	4000	Run (GererateSignals)	RealTimeThread
2	Pulse	40000	Run (GeneratePulse)	RealTimeThread
3	PDU	1000000	Run (UpdateMedicine)	IPumpThread
4	FrameBuffer	200000	Run (Notify)	DistributerThread

WCE Time (Ci)	Priority	Blocking Delays (Bi)	Blocking term (Bti = Bi/Ti)	Deadline (Di)
250	Very High	40	0.01	4000
50	Very High	0	0	4000
200	High	40	0.00004	10000

200	Medium	100000	0.5	200000

In the below calculation we can see that with a sample rate of 250 Hz we will be able to schedule the threads using a frame buffer of 50 samples. We can see that the calculated total utilization (UBtotal) is less than the utilization bound (Ubound).

Rate Monotonic Analysis with Task Blocking
(U = Utilization, UB = Utilization and Blocking, B = Blocking)

Utotal	sum (Ci/Ti)	0.06	
Ubound	n(2^1/n - 1)	**0.76**	
Btotal	max(Bti)	0.50	
UBtotal	Utotal + Btotal	**0.56**	***Ubound > UBtotal***

In the next part we have added calculation of the utilization bound for the FrameBuffer event, since this is the part that is most complicated and critical for the scheduling updating the waveform graph. More details can be found in chapter 6.5 of the architecture document reference [4].

Utilization bound for FrameBuffer (e4) valid as long Nf > Np

Step 1: Identify H		**Step 2: Calculate f**
- Higher priority events - e1, e2, e3		
H1 = e3	Ti >= Di(4)	f4 = sum(Cj/Tj)|Hn + 1/Ti(4) (Ci(4) + Bi(4) + sum(Ck)|H1
Hn = e1, e2	Ti < Di(4)	***0.57***

Step 3: Utialization bound		
u(n,di) = n((2*di)^1/n -1) + 1 - di	***0.83***	(0.5 < di <= 1)
di = Di(4) / Ti(4)	**1.00**	(di < 0.5)

Step 4: Compare effictive calculated utilization with bound	
Caluclate f < Utializtion bound (f4 < di or u(n,di)	***TRUE***

Notes to be deleted:
Design process for development of real-time systems – Lesson 8 - DevelopmentOfRealtimeSystem.pdf
Design method (Adds elements to the analysis the defines one particular solution)
Ropes design activities – Lesson 5 – ArchitectureAndUML.pdf – slide 20
Architectural
Mechanistic
Detailed
Architecture – Five layered and two layered patterns
UC# 1 most important for the architecture discrete and continuous
Two layered and five layered based on UC#1
Alternative solutions …..

Mechanistic design using patterns from course
Detailed design by adding methods and using sequence diagrams for UC

Description of the design process and an evaluation of this.
Why we havn't used ports – complicated to implement (Lesson 5 – DesigningWithPortsInUML2.pdf)
Open-Close Principle where in design patterns it is used (Lesson 5 – GeneralDesignPrinciples-2.pdf)
Liskov Substituion Principle (LSP) (Lesson 5) –Strategy pattern is a good example PhysioModel
Arguments for the design choises:
Mediator for PatientModel and Medicine – to be extended
RMA even analysis on the screen update and sample calculation (RealTimeThread and DistributerThread and GUI) – see Lesson 6 – ThreadsAndSchedulability
Scheduling policy – see Lesson 10 – Critcal Section Pattern used.
Sample based vs. block based processing in driver layer – push and pull

3.5. Translation and testing (Kim)

Rhapsody (version 7.5) has been used to create an UML model for the Sapien 190 patient simulator that is used for test of the simulator model before automatically generated C++ source code to be compiled on Linux and target. The Model Driven Architecture (MDA[19]) approach has been used where we have created a platform independent model (PIM – on Windows) that later is translated to the platform specific model (PSM – on Linux) target the Linux platform by using the code generation from Rhapsody. This approach has fully been used for the continuous package of our design.

We have created test scenarios by use of Rhapsody state diagrams. In Rhapsody it is possible to set breakpoints in the animated state diagrams and perform an inspection of the state variables of the model (Instances of classes and attributes) by using the high level visual abstraction in Rhapsody. During execution of the model animated sequence diagrams has been created to verify the design model see example in Figure 26. This approach has reduced the development time in removing a lot of the manual C++ coding and debugging of the model. The translation from UML to C++ code is fully automated. The contents of methods in Rhapsody still need to be coded manual in the UML model.

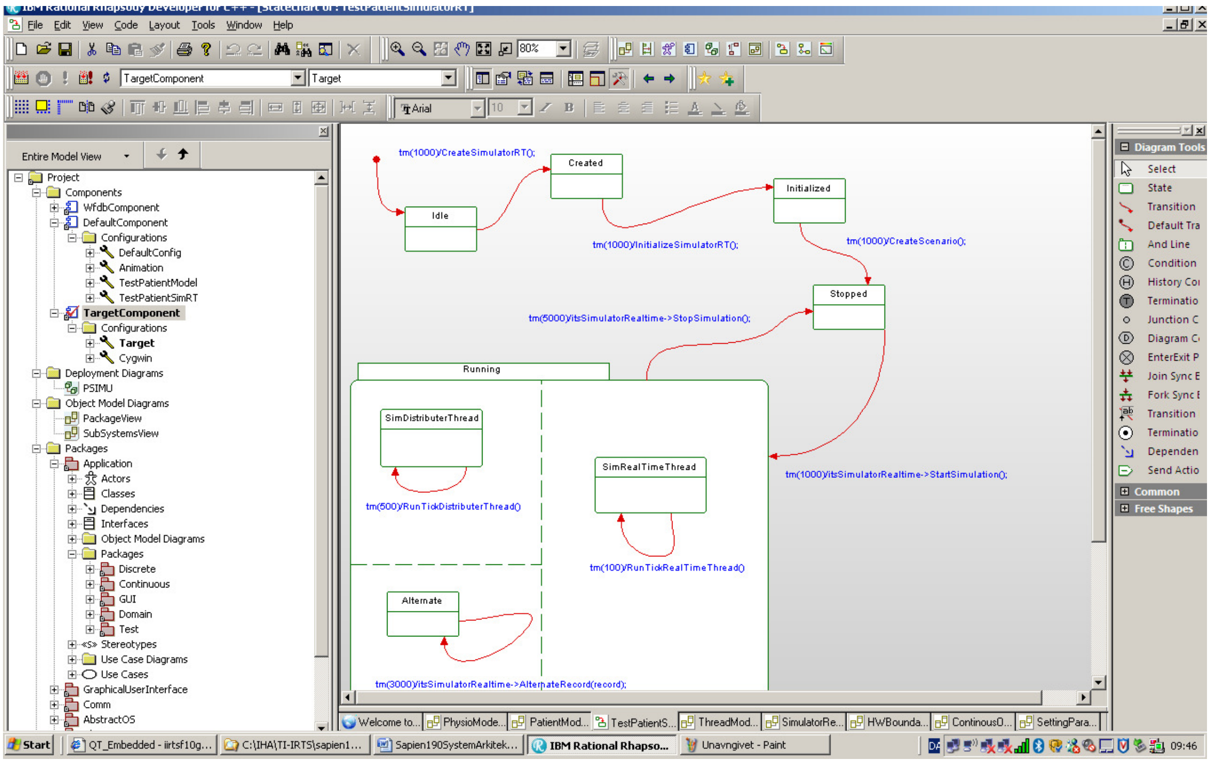
tm(1000)/CreateSimulatorRT();
Created
Idle
Initialized
tm(1000)/InitializeSimulatorRT();
tm(1000)/CreateScenario();
Stopped
tm(5000)/itsSimulatorRealtime->StopSimulation();
Running
SimDistributerThread
SimRealTimeThread
tm(1000)/itsSimulatorRealtime->StartSimulation();
tm(500)/RunTickDistributerThread()
tm(100)/RunTickRealTimeThread()
Alternate
tm(3000)/itsSimulatorRealtime->AlternateRecord(record);

Figure 25 Rhapsody UML model used for simulation and testing

[19] Slides 31 – 34 from Lesson 8 – L3_ROPES_Process.pdf

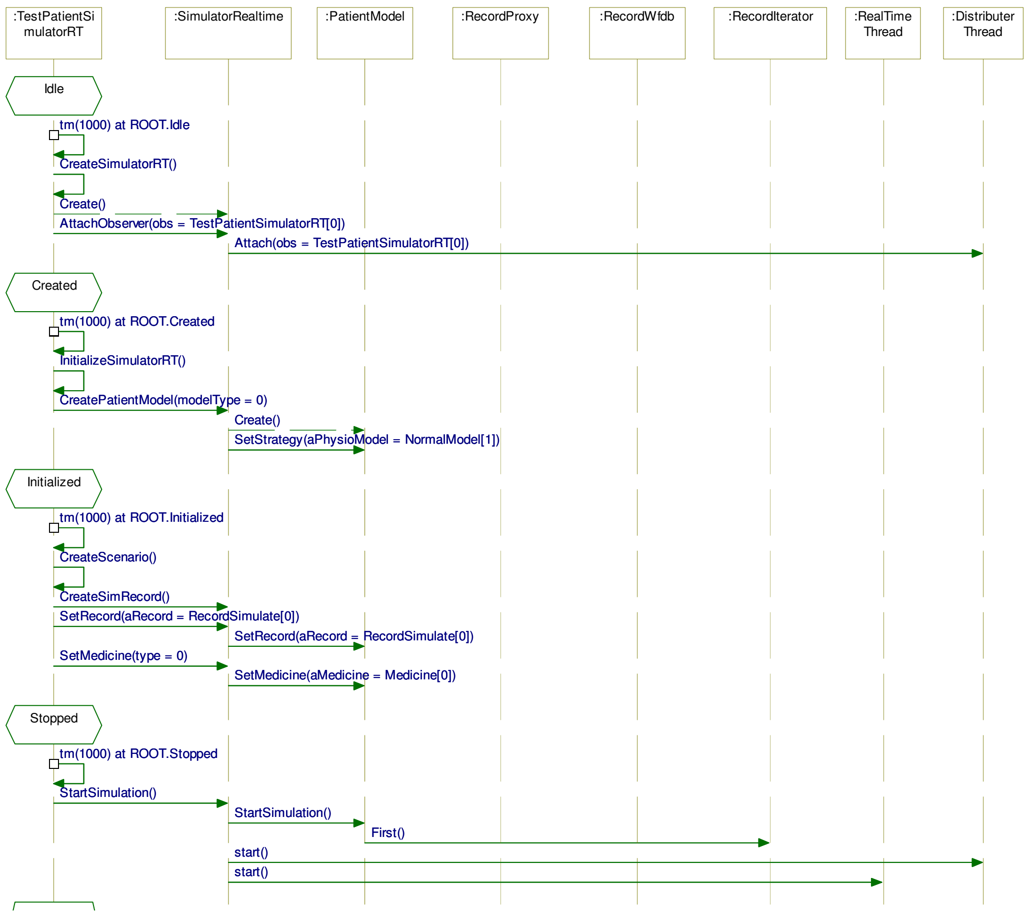

Figure 26 Animated sequence diagram for testing in Rhapsody

The operating system is encapsulated in abstract OS classes which are written by hand and tested in a separate Linux Eclipse project. Interface to the hardware drivers as DAC is manual written in C++ and tested in another separate Eclipse project with setup for cross-compilation to the target.

The Qt framework provides an abstraction for the whole GUI. It is a cross-platform framework that allows the GUI design to be ported to several operating systems such as Linux and Windows Embedded. By using Qt we have made it easier to port the design to other operating systems. This part has been manual implemented and tested on a Linux host computer with cross-compilation to target.

We have configurated Rhapsody to generate the C++ source code directly to a sub directory of the Qt project that is compiled on the Linux host. We have shared a directory between Windows and the Linux host platform in where the final debugging and integration of all parts of the software are compiled and tested before the final cross-compilation to the target platform DevKit8000. Different makefiles are automatic generated using the qmake function in Qt that creates a makefile for Linux host or target. More details on compilation and installation are to be found in chapter 12 and 13 of the architecture document see reference [4].

Test design in high level modeling with Rhapsody
Test design integrated with Qt on Linux
Translation
Creates executable deployable realization of the design
Automatic code generation from Rhapsody
GUI programming in Qt
HW and OS Abstraction manual programming

3.6. Development tools (Anders)

Qt, Eclipse, Linux, WFDB, Rhapsody, Crosscompiler, DevKit8000, Cygwin

See ArchiteturalDocument chapter 13 + 14

File structure and how tools have been integrated.

3.7. Results (Kim)

The final prototype of the product is running on target and contains the basic and essential functionality for the real-time part of the product covering functionality based on the use cases #1, #2 and #3.

The prototype contains a graphical user interface in where the simulator can be started and stopped. When the simulation is running it is possible to alternating playing different records. Parameters can be adjusted during the simulation by setting the gain to zero simulating a heart stop. The rate of samples send out to the DAC can be adjusted to measure performance limits. The signal waveform graph of replaying the ECG records will be simultaneously updated on the LCD screen at the rate of each 50 samples. The pulse is computed and output is displayed on the serial console.

The design has been tested in Rhapsody before testing on the Linux host platform and final testing on target. Details about the size, performance and quality measurements are to be found in chapters 10 and 11 of the architecture document [4].

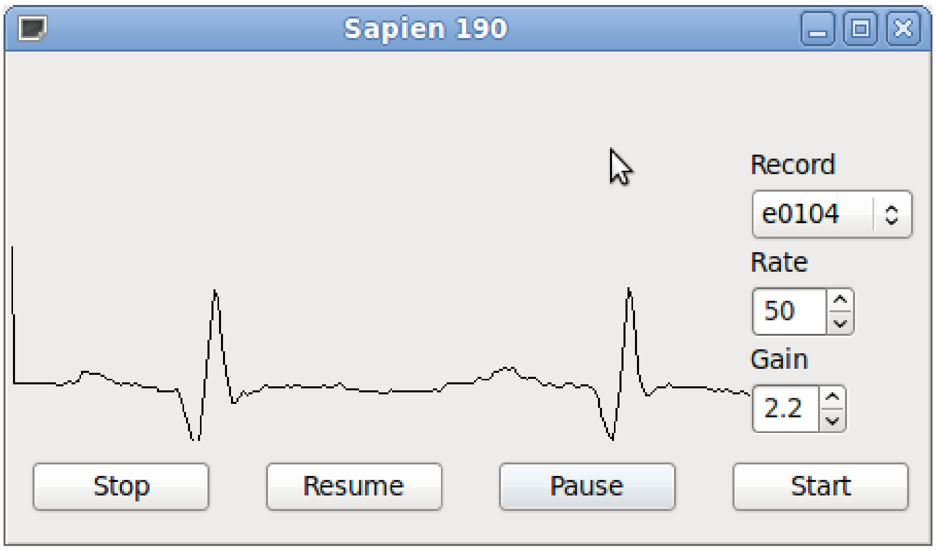

Figure 27 Sapien 190 Final prototype

The figure above shows the GUI of the patient Simulator updating the ECG waveform graph. Below is listed the serial console output that generates the calculated pulse.

```
Pulse 52
Pulse 55
Pulse 60
Pulse 58
Pulse 62
Pulse 62
Pulse 61
```

The figure below shows the ECG signal form the analouge output from the patient Simulator, though it is rather noisy. The noise issues are electrical related and not within the scope of this project.

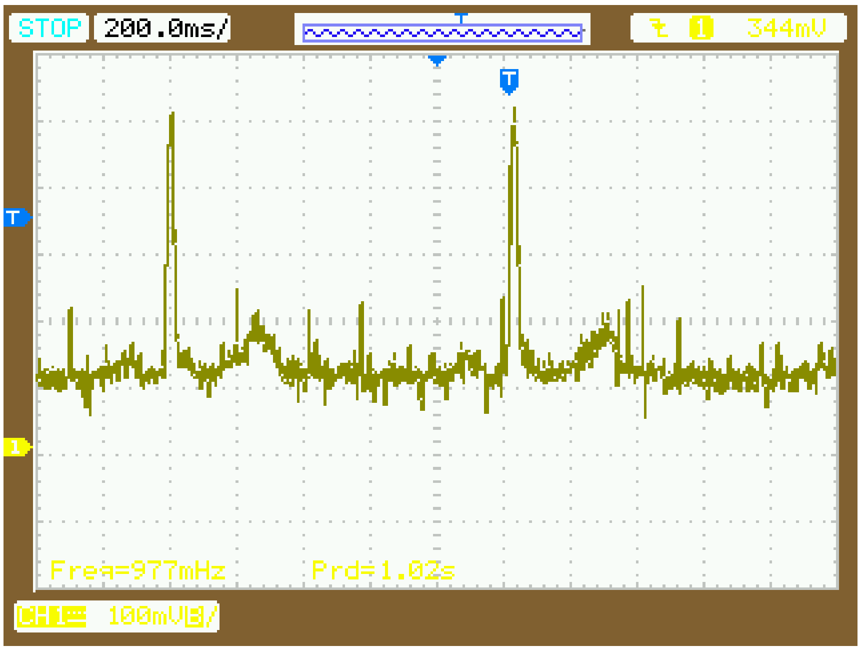

Figure 28 DAC Output ECG

It is important that refreshing the display does not take too long time. If the update rate becomes too slow the display will appear to be flickering and will become uncomfortable to look at. These measurements will also be used in the RMA analyses to calculate if thread scheduling would be possible. The update and compute times are measured by adding code to log time (us) in Linux. Measurements are calculated for updating the waveform graph and the time the RealTimeThread takes to compute and generate output samples (ECG, EDR and pulse).

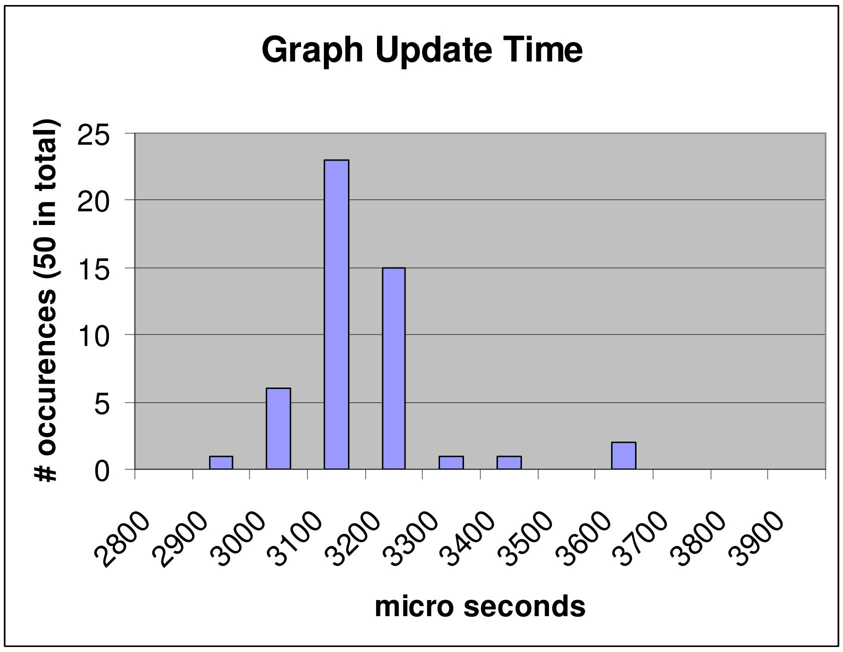

Figure 29 Graph Update Time

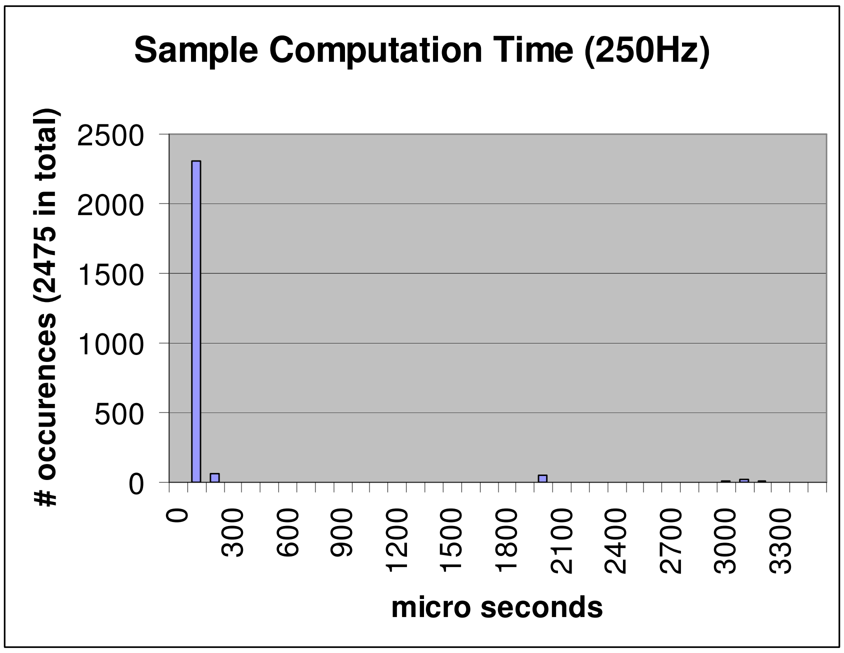

Figure 30 Sample Computation Time (250Hz)

The required flash disc size by the application and libraries is indicated below:

Application:	
Sapien190	**0.15 MB**
Libraries:	
libQtCore.so.4	**3.10 MB**
libQtGui.so.4	**11.00 MB**
libQtNetwork.so.4	**0.89 MB**
libwfdb.so.10	**0.16 MB**
libstdc++.so.6	**1.18 MB**
Libts-0.0.so.0.1.1	**0.01 MB**
Total:	
Lib+Application	***16.50 MB***

Additional the WFDB record files should be counted. A 30-minute file has a size of approximately 5 MB at 250 Hz sample rate.

We have also meassured the memory usage to 36MB equal to 29% of the memory available on the target platform. This rather massive memory usage is caused by the following:

Libraries:	**16.45 MB**	
Application	**0.15 MB**	
WFDB record file	**16.00 MB**	**(36-20MB)**
Video Frame Buffer	**0.16 MB**	**(480 x 320 x 8-bit)**
Total	***32.76 MB***	

The CPU load is light only 2% of the CPU. The reason is that the test application only updates the display at a 5Hz rate. This gives a workload of = (5Hz * 3150 us)% = 1.5 %. Currently it is not the compute and generate of new samples that is critical. The current design seems to be efficient enough for the current hardware platform.

3.8. Discussion on achieved results (Kim)

In the process of making the architecture and design for the patient simulator, we have been inspired by the ROPES and the SCRUM process which has been the fundament for an efficient and structured way of achieving the results.

We have made a prototype of the patient simulator that has incorporated many design patterns that is part of the course TIIRTS where we have covered both traditional GoF patterns combined with real-time patterns to handle concurrency and memory handling. In some areas the product is perhaps overdesigned like use of the proxy pattern reading records. This design choice is not "the agile way" of working since we are designing for functionality reading records remotely from the Internet not part of the specification. We have achieved a product that contains the solid fundament for the design and architecture of the patient simulator. Many real-time aspects have been covered in the project like design of the continuous packaged including a rate monotonic analysis of the thread scheduling.

The design is optimized for maintability, correctness and usability as stated in the requirement specification document. The maintability is realized use of design patterns like it would be easy to add new patient models by use of the strategy and filter and pipes patterns. It would be easy to add new medicine and parameters realized using the command pattern and adding additional views on the signals by use of the observer pattern. The whole design is carefully documented in the product architecture document [4], making it easy to continue working on the product. The software can easily be moved to another platform due to the design abstractions and use of Qt. (HW, OS and GUI abstractions)

We have verified the correctness of the patient simulator prototype by measurement of the signal output from the DAC and the jitter of the signal. The design uses the DAC output as a sink supplied by samples of the source from the input records. This simple source-sink design protects against data contention and provides the lowest possible jitter as it is driven by the DAC output. Jitter performance could be improved by letting the device driver control the output timing.

The performance and resource requirement of the target platform for the final prototype is quiet good only 2% of the CPU performance is used. It turns out to be the memory consumption that is the most critical for the current prototype using 29% of the memory available. Since we could not figure out how to changing the priority of threads running on Linux we have discovered that our real-time thread can be skipping its deadline. Since all threads are running with the same priority updating the waveform graph causes that real-time thread to be skipping sample periods very rarely.

The design has not been tested heavily and may have memory leaks or other bugs that have not been identified yet. The use of smart pointers is an effective way of minimizing memory leaks and stray pointers. Smart pointers have not been widely used, but could replace some of the existing pointers like when creating new records. Smart pointers are especially good where objects are created and often deleted.

We did expect in the start of the project being able to fully implement all the use cases we original did specify, but we prioritized to reduce the functionality and use time on getting the use cases implemented we did design in the second iteration. We did also focus on completing a good documentation for the architecture and design. We have tried to make the design with as few limitations as possible and being able to extend and port the design to other platforms. The current hardware platform is in some aspects limited only to handle 2 analogue output ports, but our design could easy be extended to generate and handle many simultaneously generated signals or even more patient simulators running on the same platform.

The choice of using Rhapsody not only for documentation but also for code generation requires education and experience with the modelling tool, only one in the project did have the need qualifications. Later in the project we decided not to use Rhapsody for code generation of the discrete package but only for documentation. Using Rhapsody in some aspects saves development time (Reduces coding errors and speeds up debugging and testing) in other aspects it increases the development time in mastering the tool being able to do what you want.

3.9. Experience obtained (Kim)

During this project we have improved our skills in developing of and embedded real-time system based on the OOA/D methodology incorporating design patterns and handling of concurrency. We have managed to use and transform the theory from the TIIRTS course into a first prototype of a product that is running on an embedded Linux target with acceptable performance. We have learned how to write product documentation based on a use case requirement specification and a product architecture document based on the "4+1 view". We have gained experience with part of the ROPES methodology and Scrum process in the way we have structured the project and organized our work and meetings. Part of the ROPES methodology concerns about analysis and design in where we have gained a better practical experience. Especially in the mechanistic phase of the spiral model in where design patterns are applied.

We have gained experience in designing a concurrent system by combining GoF patterns and concurrency patterns. The continuous part of the design is using a number of concurrent patterns like monitor and the message queue pattern to synchronize threads exchanging information in combination with the observer pattern. We have gained insight in how to make event analysis and rate monotonic analysis (RMA) of the concurrent design. We have found that it actually can be used in an iterative development process where WCE measurements on the prototype can be used to verify how close we are to the limits for scheduling of threads in our product. More details can be found in chapter 6 of the architecture document [4].

Development of embedded systems requires that you have a good understanding on the underlying hardware, the development tools and target environment. In this project we have used a large amount of time trying to sort-out cross compiling in Qt and Eclipse and incorporating libraries for compilation to both Linux development host and target. We have learn it is important to spend time setting up the environment tool properly from the start instead of trying to make repetitive tinkering.

We have chosen to use Qt as the "Creator" development tool to support developing of graphical embedded applications. It has been a learning experience to see how the Qt framework is using patterns in the way it is extending the C++ classes with slots and signals. Actually the Qt framework is using a combination of an observer and message queue pattern we have learn about in the course. More details on this topic can be found in chapter 5.3.2 of the architecture document [4].

We have had problems when we checked in our project in Subversion and out on another development host platform. Second, debugging is very limited and slow there is very scarce information available during debug, which sometimes made it unnecessarily cumbersome to find bugs. We have in the meantime found that it is possible to get a Qt plug-in for Eclipse which would have been of interest to use instead. Rhapsody does also exist as an Eclipse plug-in. To create an Eclipse based development tool chain incorporating Qt and Rhapsody and thereby combining MDD and GUI for development of embedded real-time systems in one tool could be an interesting project.

For implementation of the architecture itself we have used Rhapsody. Rhapsody is a very cumbersome program and requires a great insight into how the program works before it really gives a good output for code generation. In this project we have used a large amount of time to get this insight in using Rhapsody. The magnitude of this project could perhaps have been reduced just using Rhapsody for documentation and writing the code manually. The fortune is however when a system is first created in Rhapsody, it is very easy to maintain adding new functionality or changing and testing the design. Rhapsody produces some code overhead that virtually all code generation systems do.

3.10. Excellence of the project (Kim)

The perhaps most important part of the design for this project is the layered architecture in where we have divided the design into different packages. Especially the separation of the application into the discrete and continuous packages where the dependency is minimized by use of the façade, observer and command pattern in interchanging information between these two layers.

The application is made very easy to move to another platform by the HW and OS abstraction layers and use of Qt. We have already proved this by the OS abstraction implement for Rhapsody modeling and testing. The design is also very flexible since it would be easy to add a new physiological patient model by use of the strategy in combination with the filter and pipes patterns.

The design is very generic the way we have designed handling the protocol for the infusion pump. I would be easy to add another external device using the mediator pattern to interact with the patient model. This new device just has to implement the interface specified by the class ProtocolColleague see Figure 20.

The user interface is design using Qt and the command pattern combined with the state pattern. We have used the architectural pattern Model-View-Controller [6] that is in family with the observer pattern in combination with the way Qt handles the GUI design. It would be possible to add a new view to present a FFT plot of the ECG signal without changing the real-time model. New events can easy be added for the state machine of the GUI controller and new parameters could be defined setting values in the patient simulator. This is possible due to use of the command pattern.

The tool chain setup we have made enables the developer to continue with an iterative design methodology based on Model Driven Development in Rhapsody. New design patterns can be added to the existing model and verified before generating code to Linux host platform. The GUI can continue to be developed manually in Qt for the discrete package of the application integrating the whole project debugging and testing on the Linux host and cross-compiling to target.

The excel[20] file we have created that contains an RMA scheduling analysis can be used for future extensions to calculate if the scheduling of threads will be possible. This could be the case if the product is extended with new computation of signals then the measurements of WCE must be updated or if configuration parameters for the product are changed. (Sample rate, graph updating rate)

[20] See CD-rom directory documents and Sapien190_RMA.xls

3.11. Suggestion to improvements (Anders)

- Project – we should have started with design from ex 1-5
- Memory consumption on target – fix of memory leaks
- Implementation of use case select scenario and UC#4 + UC#5
- Product – to be completed – design patterns that could improve and been implemented
- Use of patterns not used…
- **Composite, Template Method, Abstract Factory, Decorator, Chain of Responsibility, Prototype, Visitor, Bridge, Adapter, Memento, Builder, Interpreter, Flyweight**
- Handling of scenarios – composite pattern – scenario and sub-scenarios
- Adapter to encapsulate the C implementation generating the EDR signals
- Template Method to load different types of scenarios
- Memento to implement an undo functionality for the GUI in combination with state machine
- Abstract factory to handle instantiation of different OS abstractions
- Flyweight to be used to handle many DAC channels instead of singleton

4. Conclusion

Learning outcomes and competences:

The participants must at the end of the course be able to:
– ***analyze and describe*** the requirement for an
embedded real-time system
– ***design and constructs*** an architecture for an
embedded real-time system
– ***evaluate and apply*** design patterns in development
of an embedded real-time system
– ***prepare*** a product documentation for an embedded
real-time system using UML

See guide

Eg: "We have realized in this project that modelling is a powerful tool in helping us to **evaluate different ideas** applied to the problem being studied. While analyzing the situation we came up with several strategies that have been modelled using UML and VDM++. It was possible to determine whether the solutions to the problem were a correct approach or not and the advantages and disadvantages they presented”

5. References

[1] Erich Gamma et al., Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley (GoF)

[2] Bruce Powell Douglass, Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems

[3] PhysioNet and PhysioBank the research resource for complex physiologic signals. http://www.physionet.org/

[4] System/Product architecture document for Sapien 190

[5] Requirement specification for Sapien 190
http://code.google.com/p/iirtsf10grp5/downloads/detail?name=Sapien190Spec.doc&can=2&q=

[6] Buschmann, Meunier, Rohnert, Sommerlad, Stal, Pattern-Oeriented Software Architectures