System/product architecture document for Sapien 190

Embedded Real-Time Systems (TI-IRTS) Spring 2010

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

Pasien

Document history

Date	Version	Description	Author
24.04.2010	0.00	Initial Version	KBE
10.05.2010	0.01	Updated with Use Case View and initial UML diagrams for first delivery UC#1	KBE
15.05.2010	0.02	Updated diagrams with UC#1 and added IPUMP protocol in communication package	KBE
15.05.2010	0.03	Added command pattern for setting parameters	KBE
16.05.2010	0.04	Started on chapter 13 and 14	KBE
16.05.2010	0.05	Added details for chapter 13 and 14. Still missing error messages.	KBE
16.05.2010	0.06	Details added to chapter 5	PHM
18.05.2010	0.07	UML diagrams updated. Added component diagram. Added mediator pattern for IPUMP protocol. Added UML diagrams for discrete package.	KBE
19.05.2010	0.08	Ch 5.1 updated	PHM
19.05.2010	0.08a	Ch 5.2.2 updated	AA
20.05.2010	0.09	Updated UML diagram for discrete package chapter 5.2.2.	KBE
21.05.2010	0.10	Updated UML discrete 5.2.2 (Qt signals and slots) – Added thread overview diagram.	KBE
22.05.2010	0.11	Updated chapters 5 + 11 + 12 and formatting in general	PHM
22.05.2010	0.12	Updated process view with RMA.	KBE
22.05.2010	0.13	Chapter 5 updated	PHM
22.05.2010	0.14	Added UML diagrams for new chapter 5.3.2 Use case #3 realization for adjust scenario parameter	KBE
22.05.2010	0.15	5.2.5 updated	PHM
23.05.201	0.16	Chapter 6 text added and UML diagrams updated. RMA description made.	KBE
23.05.2010	0.17	Chapter 12 updated	PHM
23.05.2010	0.18	Added sequence diagram for sync of threads in chapter 6.	KBE
23.05.2010	0.19	Updated new use cases #2 + #3. Completed chapter 6 for review.	KBE
25.05.2010	0.20	Updated UML diagrams for deployment	KBE

		and component views. Added more text to chapter 6 and 14.	
27.05.2010	0.21	Updated UML sequence diagrams for chapter 5.3 and added code snippets. Added test results to appendix.	KBE
29.05.2010	0.22	Sections 5.2.2 + 7 + 8 updated	AA
29.05.2010	0.23	Section 5.3.1 finalized	PHM
30.05.2010	0.24	Sections 10 + 11 updated + doc formatting	PHM
31.05.2010	0.25	Figures and text update in ch 10	PHM
31.05.2010	1.00	Initial Release	PHM
1.06.2010	1.01	Added text to 5.3.2 and fixed minor mistakes including version numbers.	KBE
3.06.2010	1.02	Changed color on text box's	KBE

Table of Contents

1. INTRODUCTION..1

    1.1 Purpose and Scope 1
    1.2 References 1
    1.3 Definitions and acronyms 2
    1.4 Document structure and reading guide 2
    1.5 Document role in an iterative development process 2

2. SYSTEM OVERVIEW 3

    2.1 System context 3
    2.2 System introduction 3

3. SYSTEMET INTERFACES...3

    3.1 Interface to human actors..3
    3.2 Interface to external system actors..3
    3.3 Interface to hardware actors 3
        3.3.1 Pulse interface 4
        3.3.2 Infusion pump interface 4
    3.4 Interface to external software actors 4

4. USE CASE VIEW 5

    4.1 Overview of architecture significant Use cases 5
    4.2 Use case #1: Execute and Control Simulation scenarios 5
        4.2.1 Use case goal 5
        4.2.2 Use case scenarios 5
    4.3 Use case # 2: Select and Initiate Scenario..6
        4.3.1 Use case scenarios 6
    4.4 Use Case #3: Adjust Scenario Parameters 7
        4.4.1 Use case goal 7
        4.4.2 Use case scenarios 7

5. LOGICAL VIEW...8

    5.1 Overview 8
    5.2 Architecturally significant design packages..9
        5.2.1 Continuous Package 9
        5.2.2 Discrete Package 17
        5.2.3 Communication Package 21
        5.2.4 AbstractHW Package 22
        5.2.5 Application Helper Classes..23
    5.3 Use case realizations 24
        5.3.1 Use case #1: Execute and Control Simulation scenarios 24
        5.3.2 Use case #3. Adjust Scenario Parameters realization 36

6. PROCESS/TASK VIEW 39

    6.1 Process/task overview 40
        6.1.1 Syncronization between threads 41
    6.2 Process/task implementation..42
    6.3 Process/task communication and synchronization..42
    6.4 Process group 1 - DistributerThread and RealTimeThread 43

Figures

Code Snippets

1. INTRODUCTION

1.1 Purpose and Scope

The Sapien 190 is a patient simulator, simulating human physiological behaviour according to different patient scenarios. Scenarios are written in an open format and can be downloaded to the Sapien 190 that contains patient records from the PhysioBank[1] database.

In conjunction with the Sapien 110 human doll, the Sapien system provides a complete simulated human interface, with EKG, ECG and respiratory measuring spots and medicine injection spots.

1.2 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]   Project specification for PSIMU: Patient Simulator System
         http://kurser.iha.dk/eit/tiirts/Projekter/PSIMU-project.doc
[5]   Project specification for LMON: Local Monitor System
         http://kurser.iha.dk/eit/tiirts/Projekter/LMON-project.doc
[6]   Project specification for IPUMP: Infusion Pump System
         http://kurser.iha.dk/eit/tiirts/Projekter/IPUMP-project.doc
[7]   Project Interface Specification
         http://kurser.iha.dk/eit/tiirts/Projekter/ProjectInterfaces.doc
[8]   Requirement specification for Sapien 190
         http://code.google.com/p/iirtsf10grp5/downloads/detail?name=Sapien190Spec.doc&can=2&q=
[9]   Getting started with Qt
         http://devkit8000.wikispaces.com/Qt+and+Qt+Everywhere
[10]   Specification Abstract, objektorienteret operativsystem-API

---

[1] http://www.physionet.org/physiobank/

1.3 Definitions and acronyms

- **Model** – A model that represents one physical individual. The model may consist of sub-models for different body subsystems. The model uses an algorithm to compute the output signals based on the input signals or patient records.
- **Model Parameters** – Parameters used in the model.
- **Record** – A patient record file taken from the PhysioBank database
- **Scenario** – Model parameters and a collection of records taken from the PhysioBank database.
- **Scenario Configuration** – A set of files that represents model parameters and patient records.
- **Signals** – Signals in form of waveform files or input from external equipment
- **Simulation** – A continuous mode, where the physiological output signals are updated according to the model and the scenario applied to it.
- **ECG** – Electrocardiogram
- **EDR** – ECG–Derived Respiration
- **PDU** – Protocol Data Unit - Information that is delivered as a unit among peer entities of a network
- **D/A Converter** – Digial to Analog Converter
- **OS** – Operating System

1.4 Document structure and reading guide

Chapter 2 and 3 gives an overview of the product and the interfaces to the patient simulator in terms of other devices and user operation.

The document describes the design using the "4+1" view. For each iteration as specified in ROPES [2] we have selected one or more use cases that is used in describing the Use Case, Logical, Process, Deployment and Implementations Views. These views are described in chapters 5 – 8.

In the logical- and process view sections, class names have been written in *italic* and pattern names have been written in **bold**.

1.5 Document role in an iterative development process

This document is updated in using the ROPES development process [2]. The document is updated for every design cycle of the ROPES spiral microcycle. Every design microcycle is using the 4+1 view

2. SYSTEM OVERVIEW

The Sapien 190 is a compact unit with a graphical user interface (GUI) and interfaces to a range of body monitoring and injection equipment. The Sapien 190 can be operated by the instructor and connected to a bedside monitor that used the output signals from the patient simulator. Optional a medicine infusion pump can be connected to the simulator.

2.1 System context

2.2 System introduction

The instructor will be able to control the simulation like forcing a heart attack and monitor the waveform of the signals that is send to the bedside monitor. Two analogue outputs signals can be connected to a bedside monitor to display ECG and EDR signals. It is possible remotely using an Ethernet connection to the patient simulator to update and delete files in the scenarios database on Sapien 190. The optional infusion pump injects medicine into the patient and the flow of medicine can be monitored on the Sapien 190 LCD display.

3. SYSTEMET INTERFACES

3.1 Interface to human actors

Instructor controls the patient simulation by inputs to the LCD touch screen

3.2 Interface to external system actors

Remotely it is possible to update the scenarios database by use of an ftp connection with Ethernet connected to the patient simulator.

3.3 Interface to hardware actors

Analogue outputs signals to the LMON has a resolutions of 12 bits sampled at 250 hz. The analogue output has a voltage range of 0 - 4.096 volts.

3.3.1 Pulse interface

The pulse is send to a connected monitor using an RS232 connection (115200 baud, 8 data bits, no parity, 1 stop bit).

Format of the pulse signal is a maximum 3 bytes ASCII value (beats/minute). It is transmitted from the PSIMU to the LMON with a rate corresponding to the current heart rate or at least every second. The pulse value is terminated with a <CR>.
The format is : <16-bit pulse value > <CR>

Example a terminal can be connected to display the pulse values updated each second:
103<CR>
107<CR>

3.3.2 Infusion pump interface

The infusion pump is connected using a RS232 connection with the protocol described below.

A fixed 64 bytes ASCII PDU is transmitted from IPUMP to PSIMU.

This PDU is transmitted every second when the pump is started.

PDU format:
4 bytes startframe (##?*)
46 bytes medicin name
12 bytes volume infused (since started)
2 bytes CRC checksum

3.4 Interface to external software actors

As defined in the requirement specification [8], it must be feasible to upload and delete scenarios using an FTP protocol.

This will require the installation of an ftp deamon on the DevKit8000.

Scenario, record and annotation files must be stored in the same directory as the application.

4. USE CASE VIEW

In the first iteration the UC #1 has been selected since it provides the main functionality and is the essential use case for the architecture of the patient simulator.

Execute and Control Simulation

When the patient simulation is running it will perform reading of the digitized physiologic signals 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.

4.1 Overview of architecture significant Use cases

The UC #1 has been selected for the first iteration of the ROPES spiral microcycle [2] in making the architectural, mechanistic and detailed design. This use case is significant and provides the central functionality of the patient simulator. 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)
- Generating the analogue output signals (Writing to drivers)
- Display of signal waveform using Qt on target (Working with Qt on target)

4.2 Use case #1: Execute and Control Simulation scenarios

4.2.1 Use case goal

A short description of the goal for the actual Use case (as stated in the requirement specification document)
To simulate signals from the patient based on the selected scenario.
The waveform of signals to monitor and the status of patient are displayed.
The instructor must be able to monitor the simulated patient.

4.2.2 Use case scenarios

Scenario 1 - normal:

1. Opens scenario file
2. Search for record file
3. Opens record file
4. Continues to reads samples from record file and performs:
    a. Use patient model to generate ECG signal
    b. Use patient model and ECG signal to calculate EDR signal
    c. Use patient model and ECG signal to calculate pulse
    d. Update output signals: Pulse, ECG and EDR

Scenario 2 – normal with alternative scenario record:

1. Opens scenario file
2. Search for record file
3. Opens record file
4. Continues to reads samples from record file and performs:
   a. Use patient model to generate ECG signal
   b. Use patient model and ECG signal to calculate EDR signal
   c. Use patient model and ECG signal to calculate pulse
   d. Update output signals: Pulse, ECG and EDR
5. Open alternative record file after specified simulation time and continues at 2.

Scenario 3 – normal with IPUMP:

1. Opens scenario file
2. Search for record file
3. Opens record file
4. Continues to reads samples from record file and performs:
   a. Reads medicine and volume from IPUMP
   b. Use patient model to generate ECG signal
   c. Use patient model and ECG signal to calculate EDR signal
   d. Use patient model and ECG signal to calculate pulse
   e. Update output signals: Pulse, ECG and EDR

Scenario 4 – error opening record file:

1. Opens scenario file
2. Failed to search and open record file
3. Error message on LCD display

4.3 Use case # 2: Select and Initiate Scenario

Initiate the simulation using a scenario configuration chosen by the instructor.
This use case is modified for this version only to start, stop, pause and resume. A fixed
pre-programmed scenario can be started only playing one record file.

4.3.1 Use case scenarios

Scenario 1 – start and stop simulation:

1. Instructor selects to start the simulation
2. The patient simulator will start to perform simulation according to UC #1
3. Instructor selects to stop the simulation
4. The patient simulator will stopped and no signals will be updated

Scenario 2 – pause and resume simulation:
The simulation must be started as specified in the scenario above.

1. Instructor selects to pause simulation
2. The patient simulator will stop updating the output signals and pulse
3. Instructor selects to resume simulation
4. The patient simulator will continue to perform simulation according to UC #1 from where it was paused

4.4 Use Case #3: Adjust Scenario Parameters

4.4.1 Use case goal

Adjust the parameters available in the current model running

4.4.2 Use case scenarios

Scenario 1 – adjusting gain:

5. Instructor adjust the gain parameter
6. The patient simulator is updated with the new gain value
7. The output of the ECG, EDR signal is adjusted according to the new gain setting

Scenario 2 – adjusting rate:

1. Instructor adjust the rate parameter
2. The patient simulator is updated to replay with the new sampling rate
3. The output of the ECG, EDR signal is updated according to the rate independent on the recorded signal rate (Default 250 Hz)

5. LOGICAL VIEW

This section describes the functionality that the system provides to the end user. The system architecture will be described as introduction. Following this, the major design implementations will be described on a package basis.

5.1 Overview
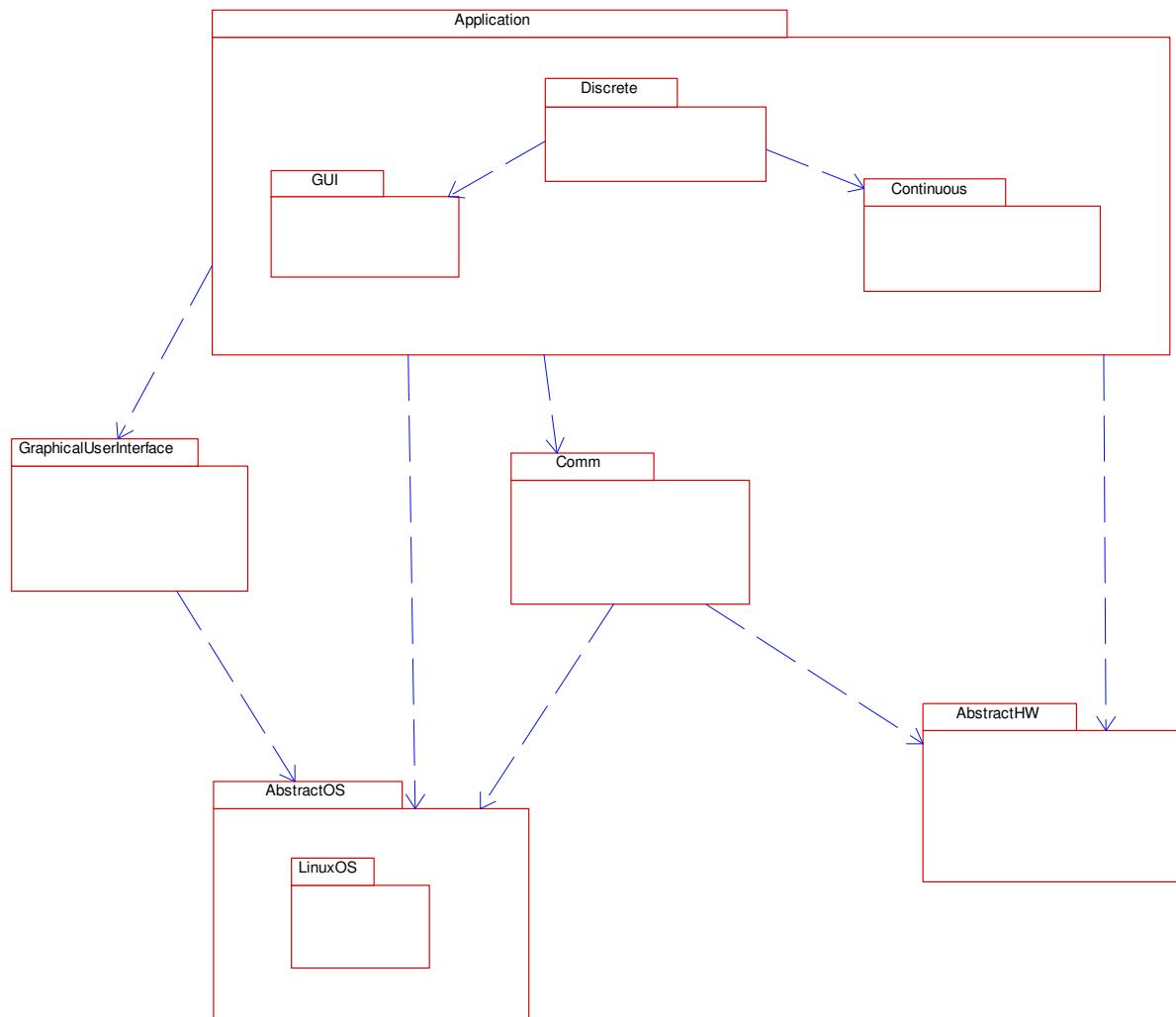
Figure 1 Five layered architecture for logical view

The architecture divides the application into five abstraction layers:
Applications – Business Logic
GUI – Graphical User Interface
Communication – Communication protocols
Abstract OS – OS specific methods
Abstract Hardware – Encapsulated hardware interfaces

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.

5.2 Architecturally significant design packages

The implementation will be explained on a design package basis.

5.2.1 Continuous Package

The continuous package contains all classes to run in the real-time part of the system. This package must respond to events and supply the hardware outputs with real-time data. This is done with stringent requirements to jitter and latency.
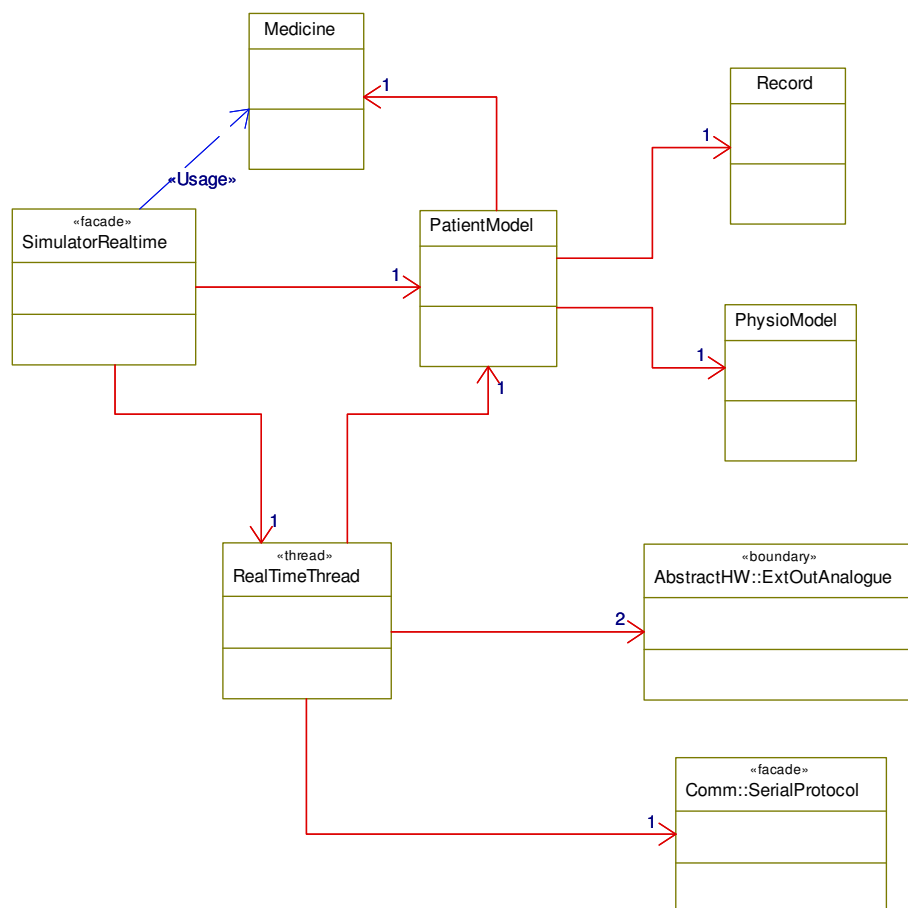
Figure 2 Major Classes in Continuous Package

The *SimulatorRealTime* class provides a façade towards the discrete system. It provides a uniform interface that hides all the underlying logic.
The center of the continuous package is the *PatientModel* class. The patient model emulates a patient, thus it emulates the patients' subsystems, such as heart, lungs etc.
The *PatientModel* uses:

- *Record*, that provides ECG data and signal annotations.
- *PhysioModel*, that provides filters to calculate physiological data based on the record data, medicine and user input (from the discrete system)
- *Medicine*, that provides information about the current medicine injection

The *RealTimeThread* handles the real-time analogue and digital signals. Originally created by the *SimulatorRealTime*, this thread acquires data from the *PatientModel* and outputs its result to the abstracted hardware outputs.
The *ExtOutAnalogue* class wraps the interface to the D/A converter devices and provides

each of the two D/A channels as separate objects to the *RealTimeThread*.
The *SerialProtocol* adds a simple protocol to the raw simulator data provided, before transmitting it using the abstracted serial port hardware (*ExtOutSerial*). Figure 3 explains this in detail. This figure also shows how *Medicine* actually abstracts a physical interface to an injection pump unit.
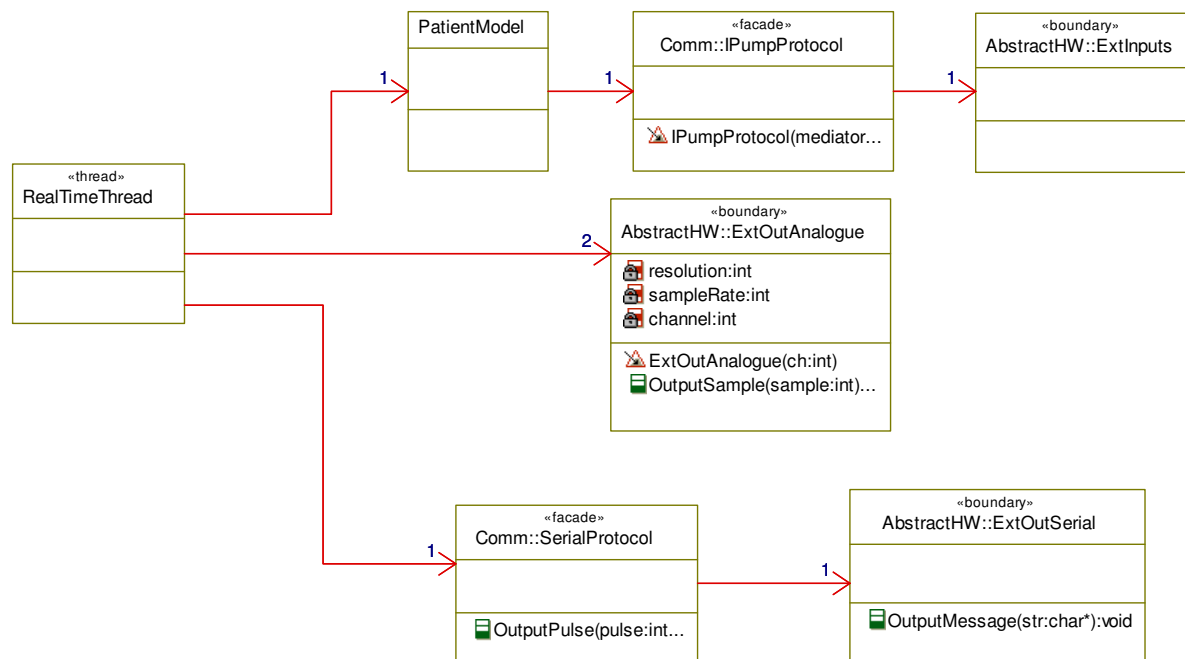

Figure 3 Boundary Classes for RealTimeThread

It has been chosen to implement the collection of record samples in the PatientModel using the **Iterator Pattern** (Figure 4). Compared to the original GOF presentation, *PatientModel* is the Client, *Record* is the Aggregator and *RecordIterator* is the actual iterator. Using this pattern we can let the *PatientModel* iterate through the records without actually knowing anything about them. This makes the system insensitive to changes in the record. The iterator is implemented as an external iterator, as the iteration is controlled from *PatientModel* .
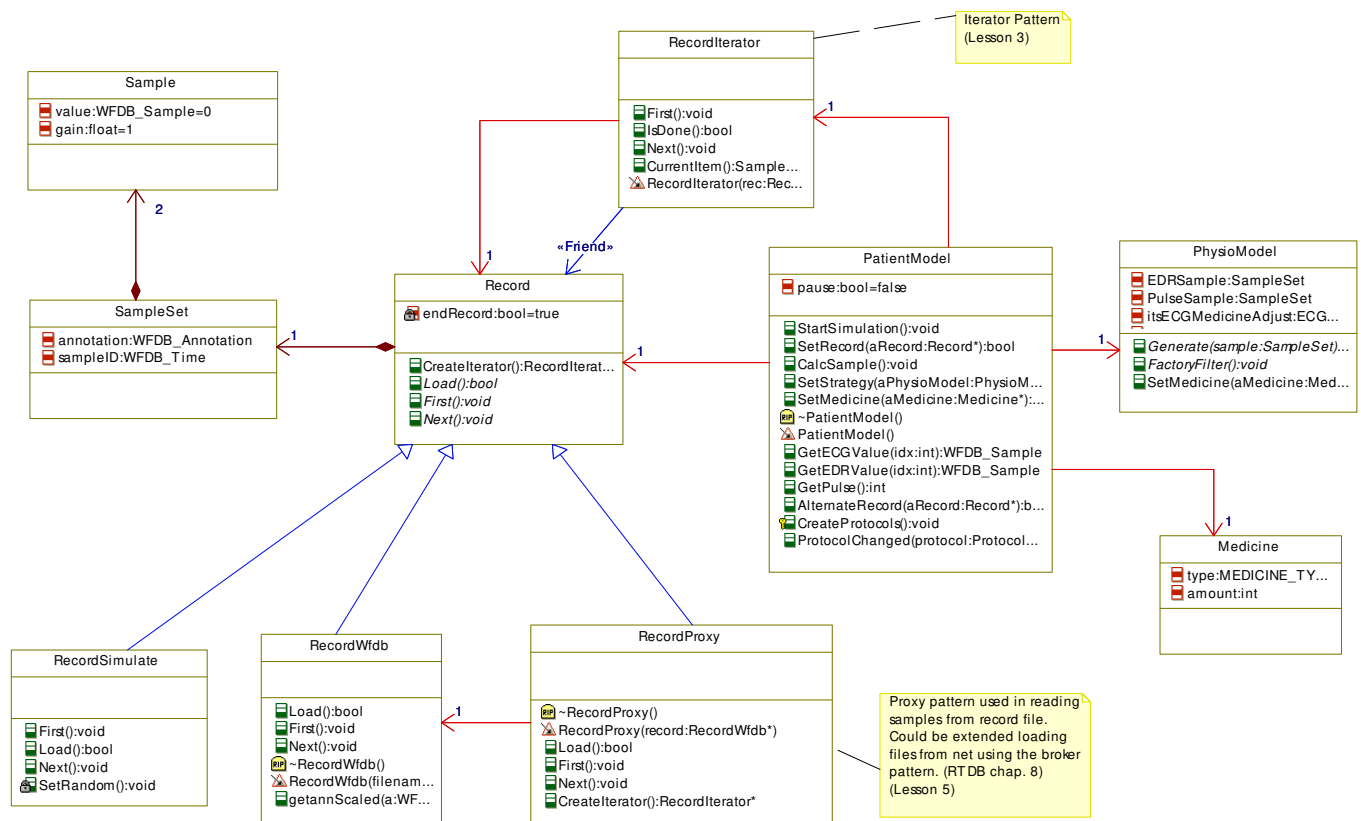
Figure 4 Proxy and Iterator Pattern used to access Records from the PatientModel

The *PatientModel* is implemented similar to the **Mediator pattern**. The *PatientModel* acts as a mediator between the *Record*, the *PhysioModel* and the *Medicine* classes. These classes do not interact directly; rather information is passed among them by means of the mediator class. The mediator promotes loose coupling between its clients and centralizes control of these. The result, again, is a much more rugged design, that is easier to test and maintain. The implementation is not strict to the GOF implementation, where we have the Mediator and Colleague super classes. In a future implementation that includes an infusion pump, it would be natural to let the medicine be a concreteColleauge[2], now being able to notify the Mediator of changes in medicine.

The *Record* inherits to three children: *RecordSimulate*, *RecordWfdb* and *RecordProxy*. *RecordSimulate* generates random sample values for simulation. *RecordWfdb* acquires samples from the WFDB record specified in the scenario. Finally, *RecordProxy* implements the **Proxy Pattern** to give us a proxy for future interfaces such as access to remote data or access to new data sources. The default implementation of the proxy calls the *RecordWfdb* methods directly

---

[2] [1] page 276
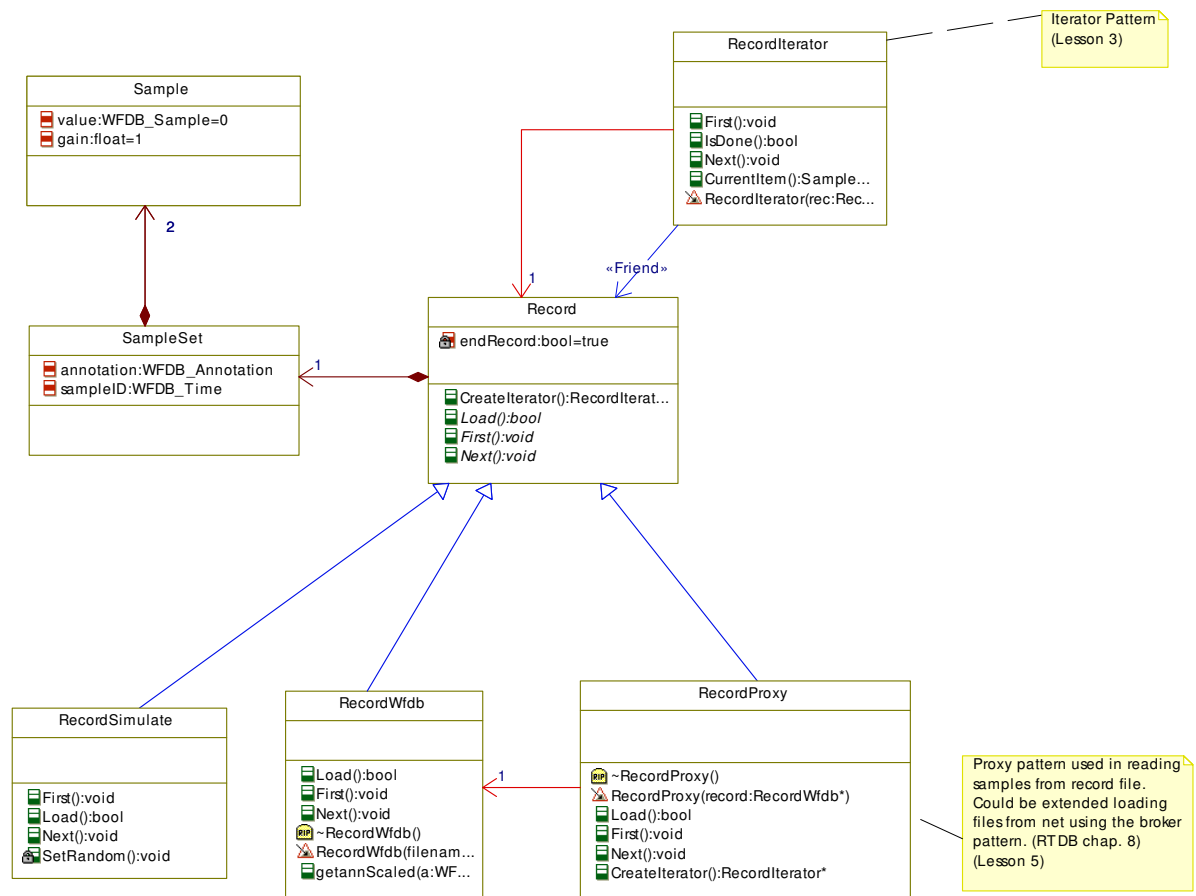
Figure 5 Record and Iterator

Each *Record* object has a sample set. The sample set contains

SampleID – A sequential sample index

Annotation – Annotation text for the current sample index

Value[1..*] – Data sample. Sample time is aligned

Gain[1..*] – Gain of the corresponding sample

One sample is read at a time from *RecordWFDB* or the other classes that inherit from *Record*. This is done in the "next" method, called by the iterator.
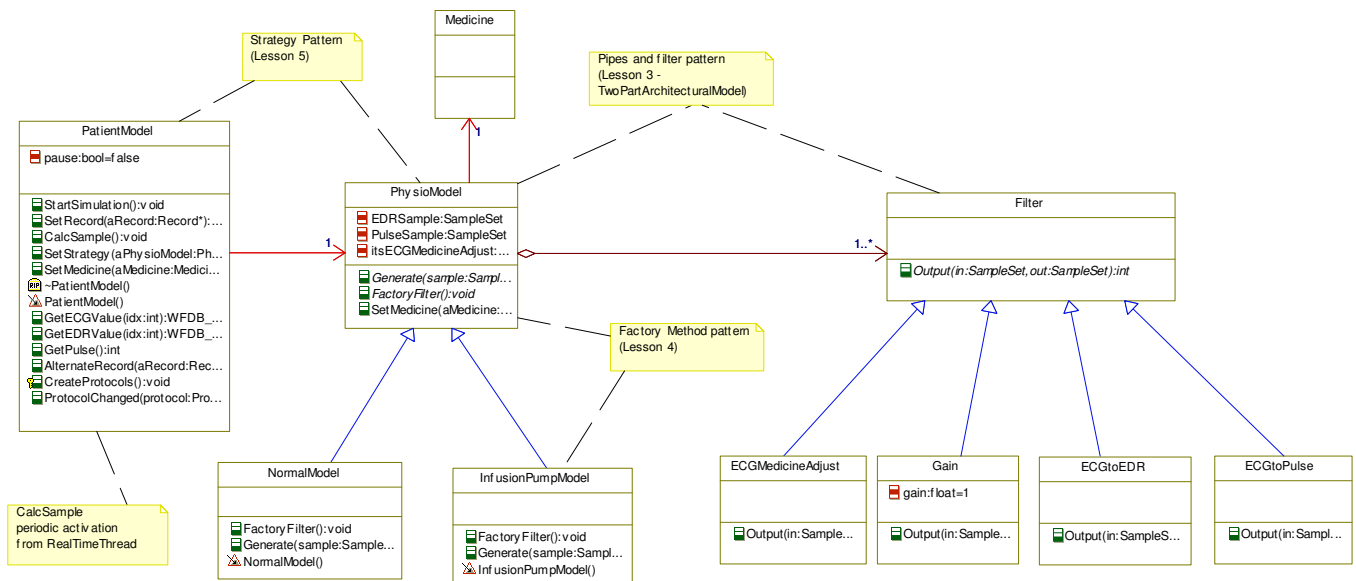
Figure 6 Strategy, Filter and Pipes Pattern used for PhysioModel

The *PhysioModel* is build using a **Strategy Pattern**. The *PatientModel* provides a SetStrategy method to set the *PhysioModel* strategy to either *NormalModel* or *InfusionPumpModel*. This corresponds to the use case scenarios described in 4.2.
.
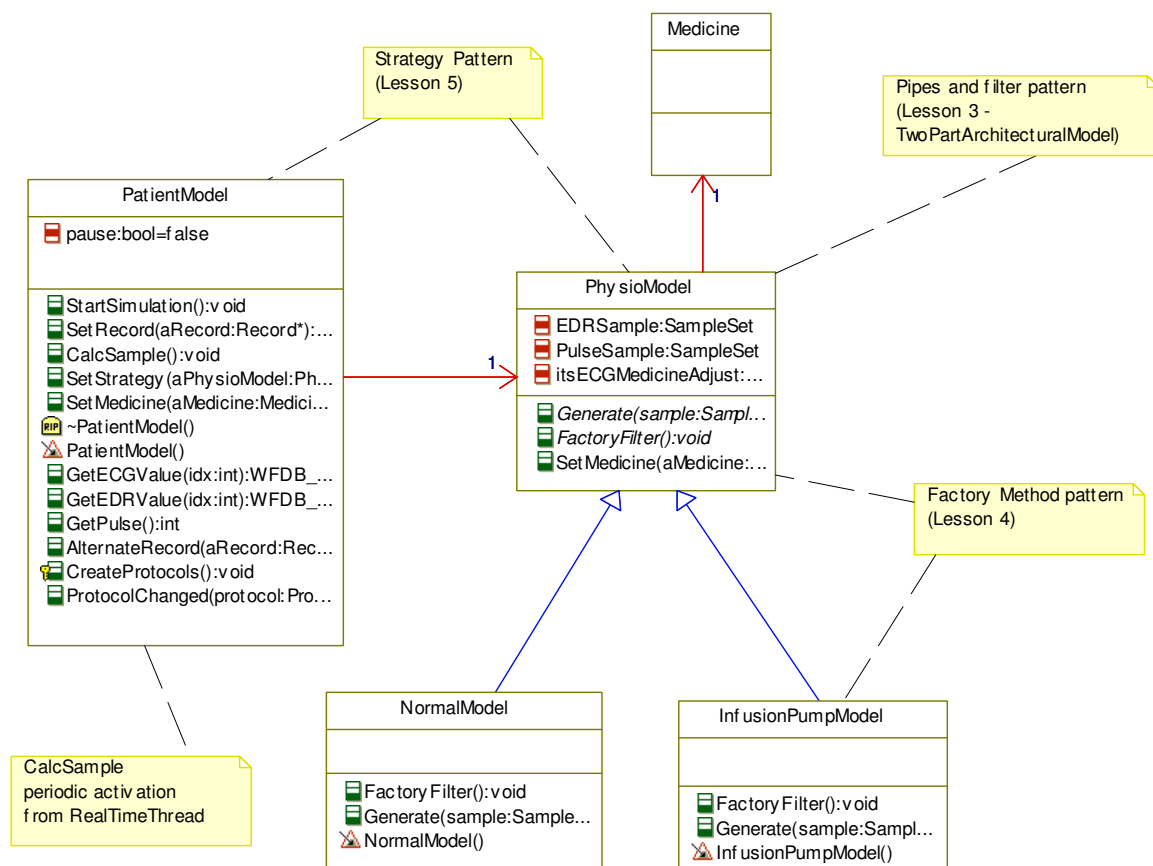
Figure 7 Strategy for PatientModel

The *PhysioModel* uses a collection of filters with a uniform interface, thus making it

possible to connect any two filters in series. This is also known as the **Filters & Pipes Pattern.**
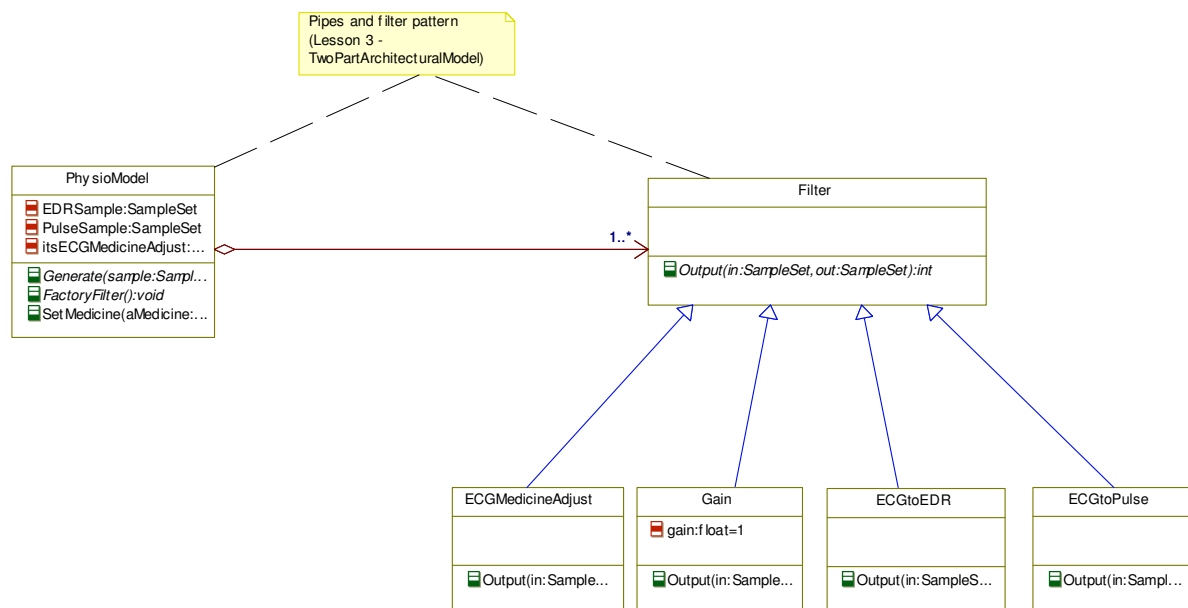
Figure 8 Filter Pattern for PhysioModel

The combination of these two patterns is typical for a two-part architecture. The discrete system[3] chooses a strategy which is then passed from the discrete system to the continuous system, where it is executed.

The classes inherited from *PhysioModel* uses a **Factory Method Pattern** to create filter objects according to the chosen strategy. The "Generate" method implements the actual filter chain.

---

[3] Designing Event-Controlled Continuous Processing Systems, page 8
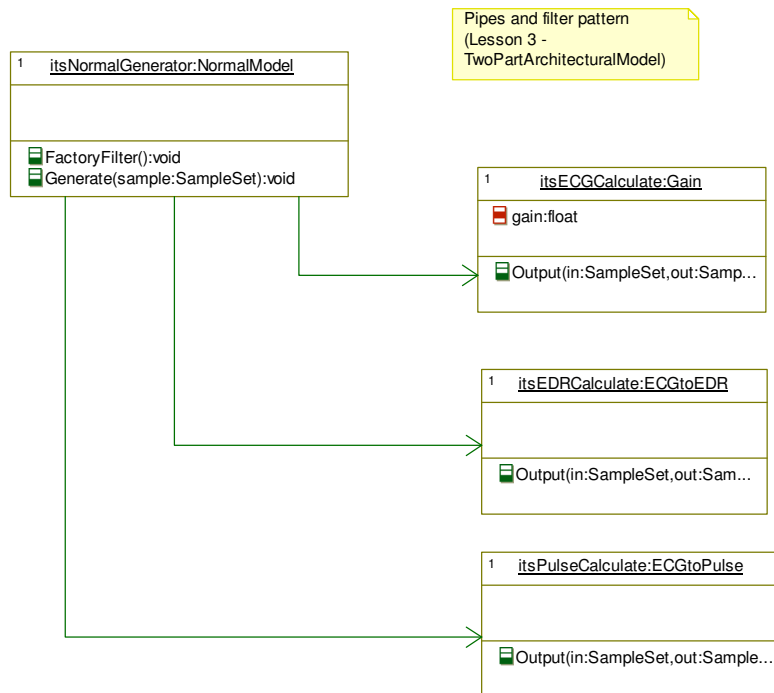
Figure 9 NormalGenerator using filters to generate signals

To provide a uniform interface between the discrete- and the continuous parts of the patient simulator, it has been chosen to wrap the discrete sub system with a **Façade Pattern**. This provides a single class for the real-time part to interface to, thus lowering the coupling between the two sub-systems. As seen in Figure 1, the façade wraps *Record*, *PhysioModel* and their relatives. Another job for the façade is to instantiate all its private classes.
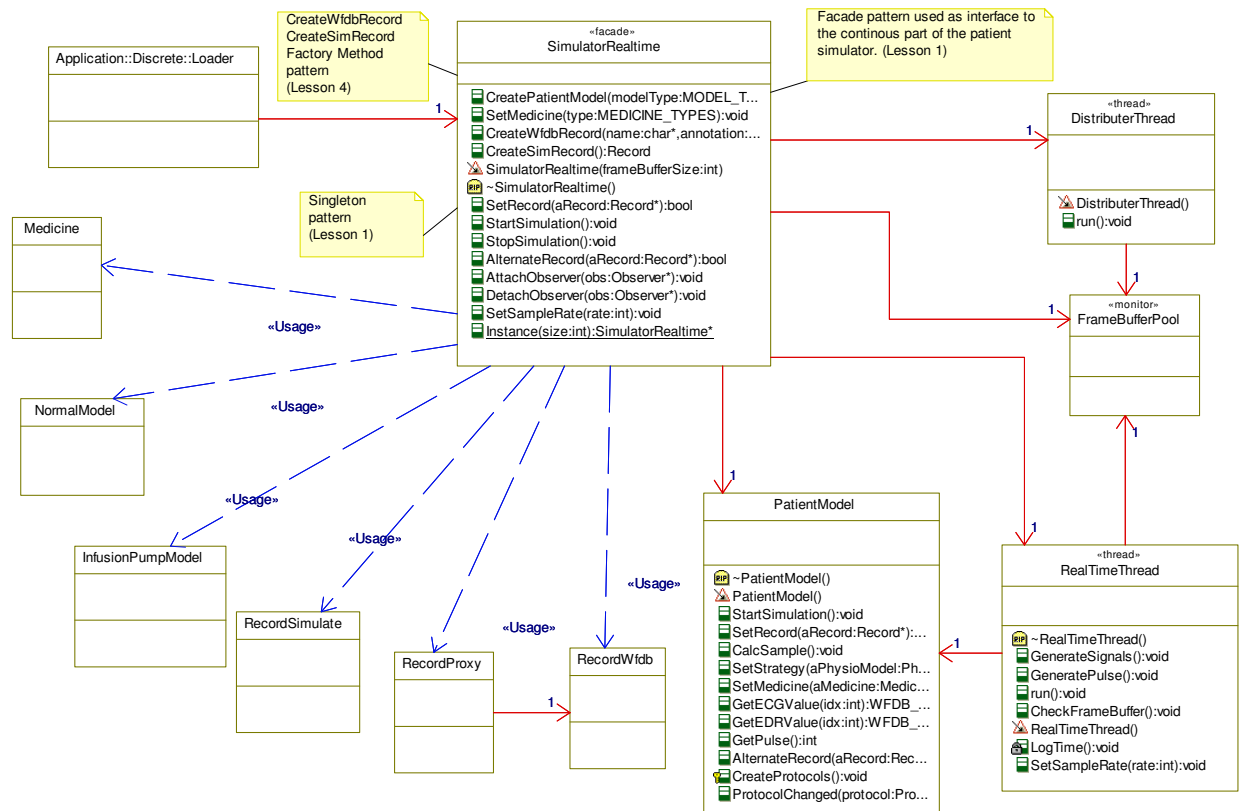
Figure 10 Façade Pattern used for interface to the Continuous Package

5.2.2 Discrete Package
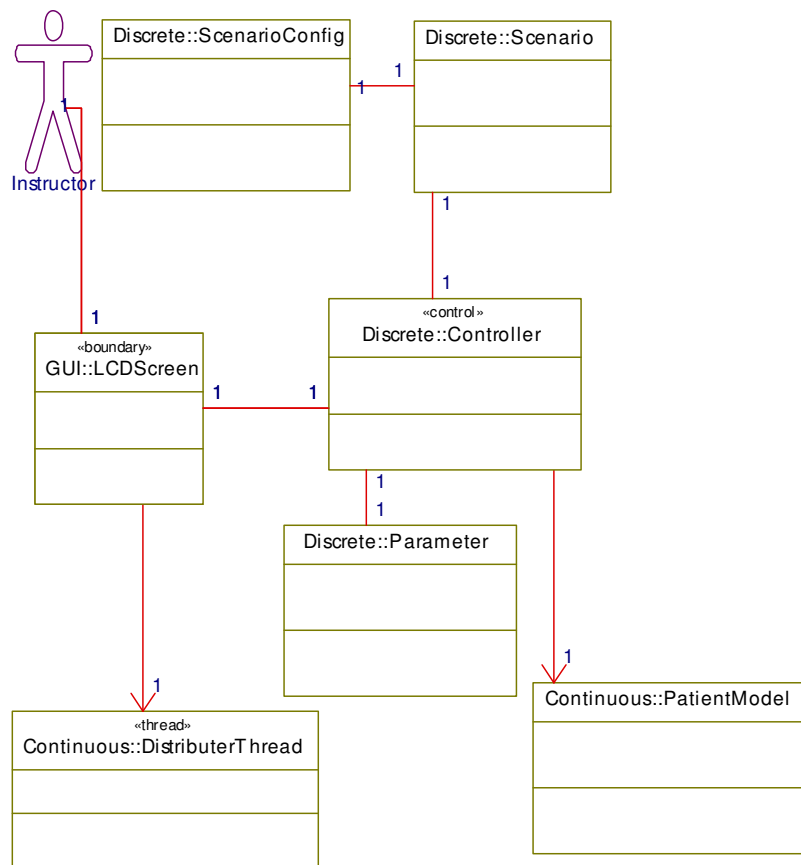
Figure 11 Application model for discrete package

To loosen the coupling between our UI and controller, we have implemented the **Command Pattern**. The command pattern is used to execute requested commands from the UI, on our continuous system via callbacks. We have 2 types of command pattern, one command pattern is used to change parameters (the Parameter class), based on user import, things such as, rate, gain and what record to read. The other one is described later on. See Figure 12.
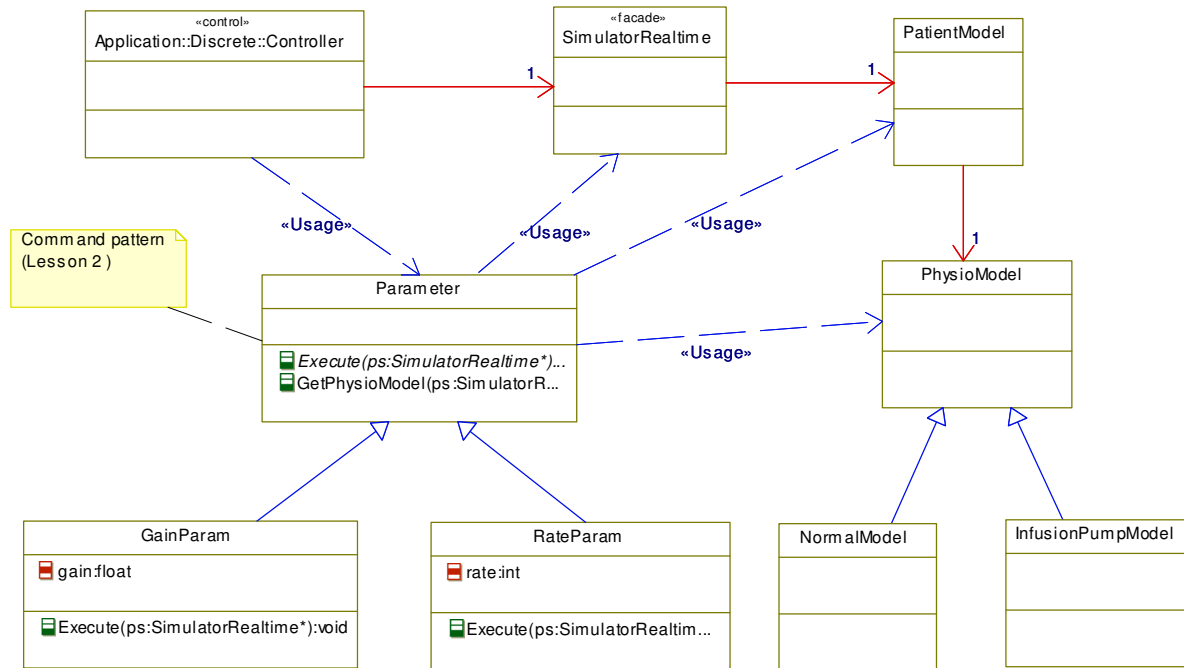
Figure 12 Command pattern used to set parameters in real-time simulator

To transfer information to our UI, we have implemented **Observer Pattern**, which with help of the Distributor Thread is used to distribute information in our discrete system.
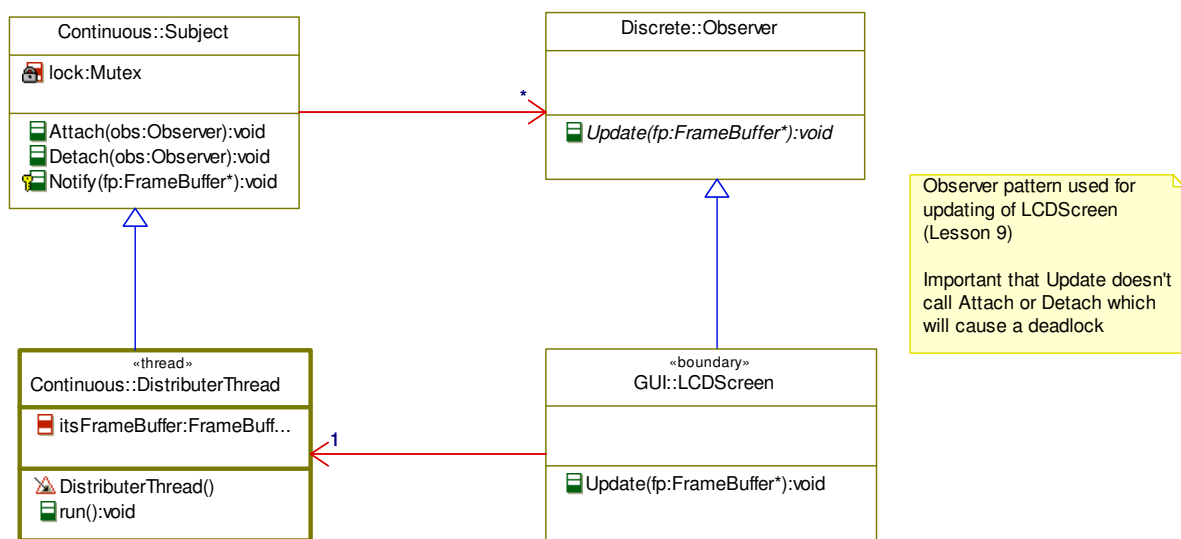
Figure 13 Observer pattern used to notify GUI with new frame buffer
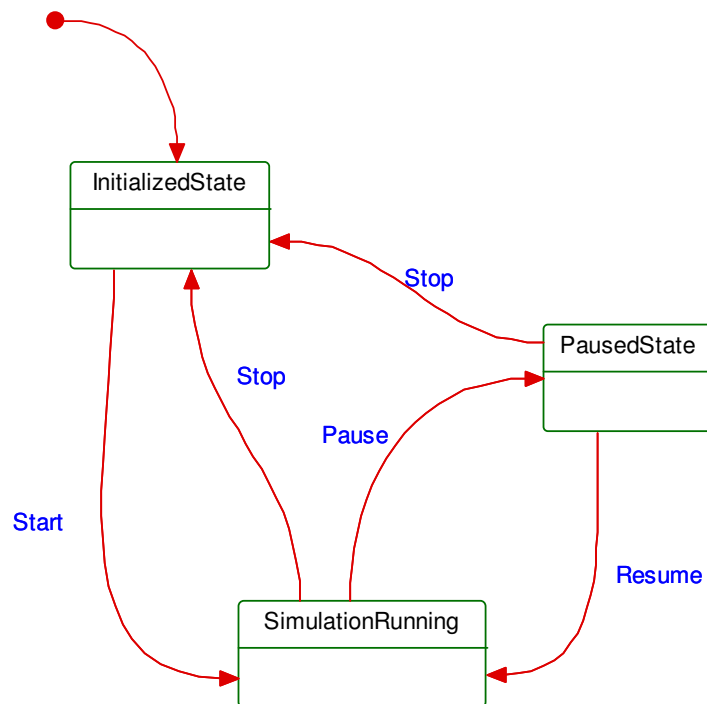
Figure 14 State chart for SapienApplication controller

To control Continuous system, we have implemented a **state pattern** in the presentation layer. This pattern allow us to control the continuous system, by changing state based on user input, and reflect it into our continues system. This is a how we initialize, start, pause, resume and stop our system based on user input.
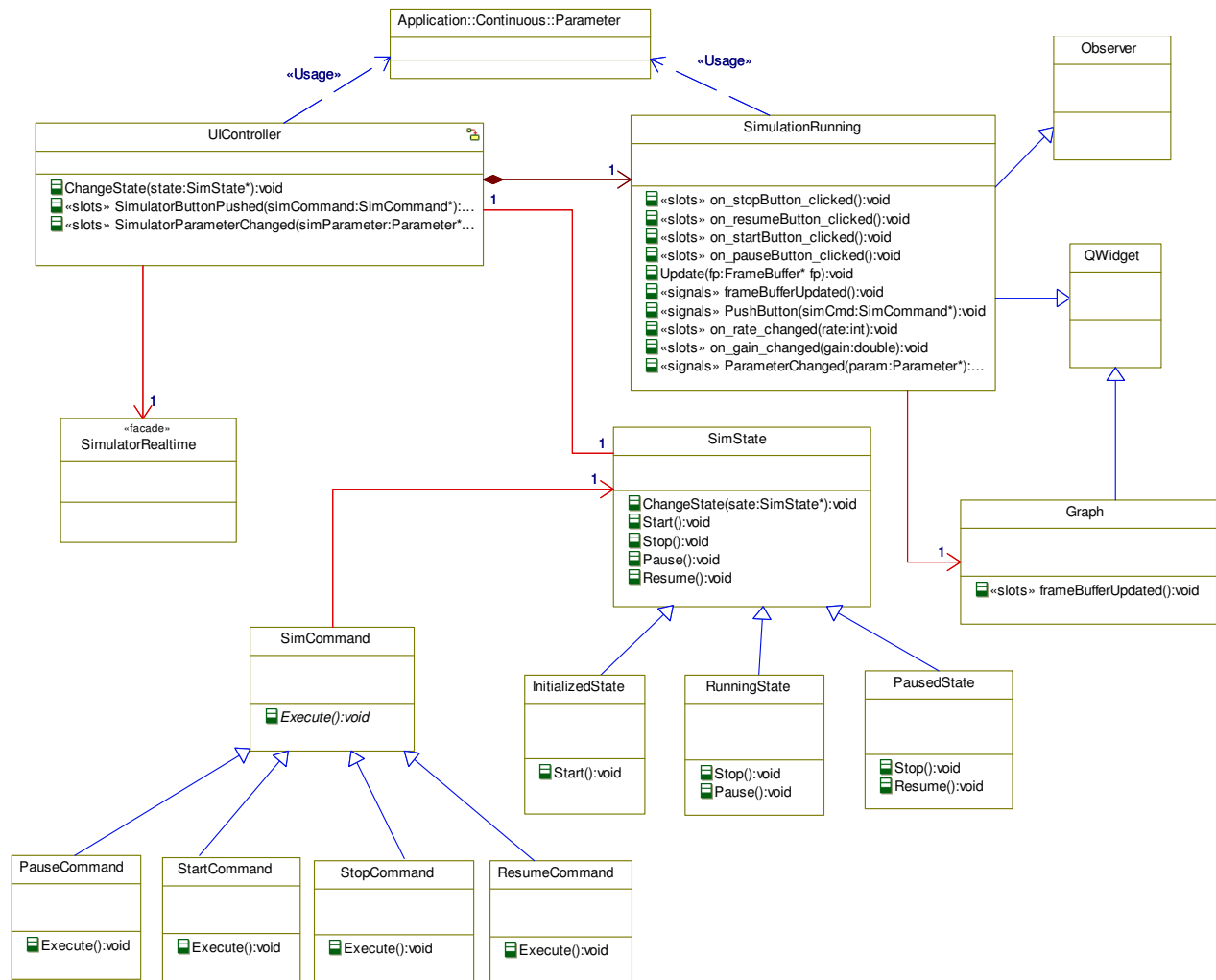
Figure 15 Command, State and Observer pattern used to design SapienApplication (Controller)

As written before we have implemented **Command pattern** to loosen the coupling between our UI and controller. The SimCommand is used to transfer actions from the UI to our state pattern (SimState), controlling state based on user input. The *SimulationRunning* class functions as a subject in our **Observer Pattern**, and transfer information from our continuous system into our UI. The state pattern as mentioned before, control the current state of our continuous system, based on user input from the UI.

5.2.3 Communication Package

The *SerialProtocol* class is provided for creating data packages according to the description in 3.3.1. Integrity check and re-transmission is handled in the abstracted hardware layer, *ExtOutSerial*.
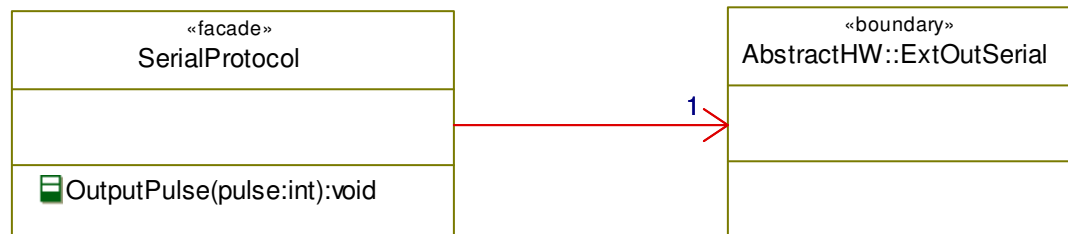
Figure 16 Communication package Serial protocol

The current design of the *PatientModel* uses a Mediator-pattern-like structure. The next version of the Sapien 190 will support interfacing to an IPUMP. Modifying the *PatientModel* design to use the GOF **Mediator Pattern**, will allow the infusion pump to notify the *PatientModel* when new medicine is injected.
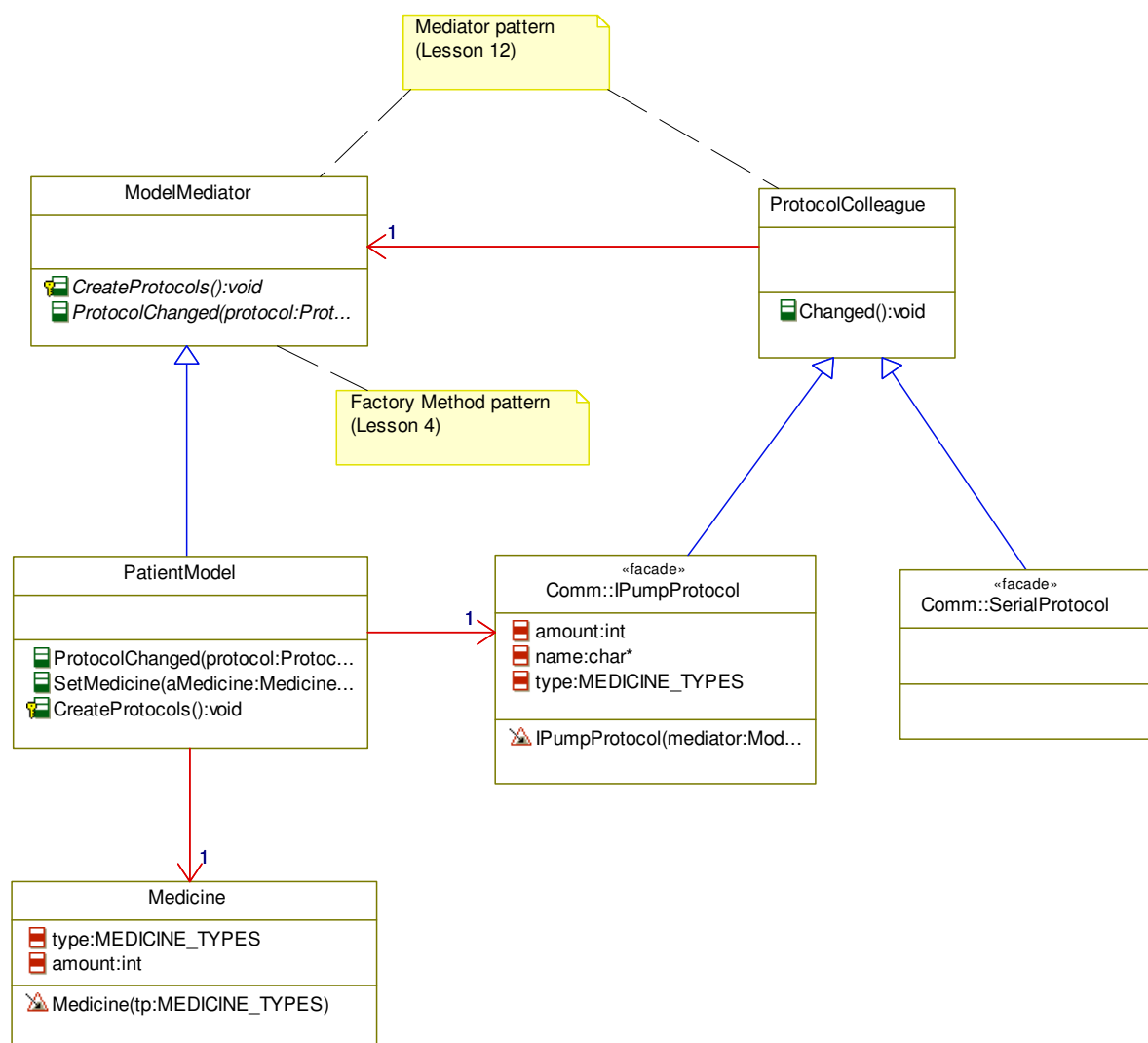
Figure 17 Mediator pattern used to update PatientModel with external input from protocols

The implementation will work like this: When new medicine is injected, that is a data package is received (see 3.3.2), the *IPumpProtocol* will call its inherited Changed() method. The Changed() method will then call *PatientModel*::ProtocolChanged(), to notify the *PatientModel* that something changed in the IPumpProtocol.
The *PatientModel* should then get the new medicine type and amount from the *IPumpProtocol* and create a new corresponding *Medicine* object.
This way, the mediator will decouple the *IPumpProtocol* and the *Medicine* classes and mediate the IPUMP inputs into new *Medicine* objects.
Getter and setter methods are not shown in Figure 17.

5.2.4 AbstractHW Package

The *ExtOutSerial* wraps the serial port interface. The intention is to use a "tty"[4] device. These do not provide means for transmission integrity check and re-transmission. This will have to be implemented in the *ExtOutSerial* class.

The *ExtOutAnalogue* class provides a façade to the abstracted analogue output. Two D/A channels are provided in the hardware. A **Singleton Pattern** has been implemented to limit the number of *Dac* objects to one for each channel. The _instance attribute is an array two elements, one for each D/A channel. Thread protection has been added to ensure that two threads cannot access a *Dac* objects simultaneously.
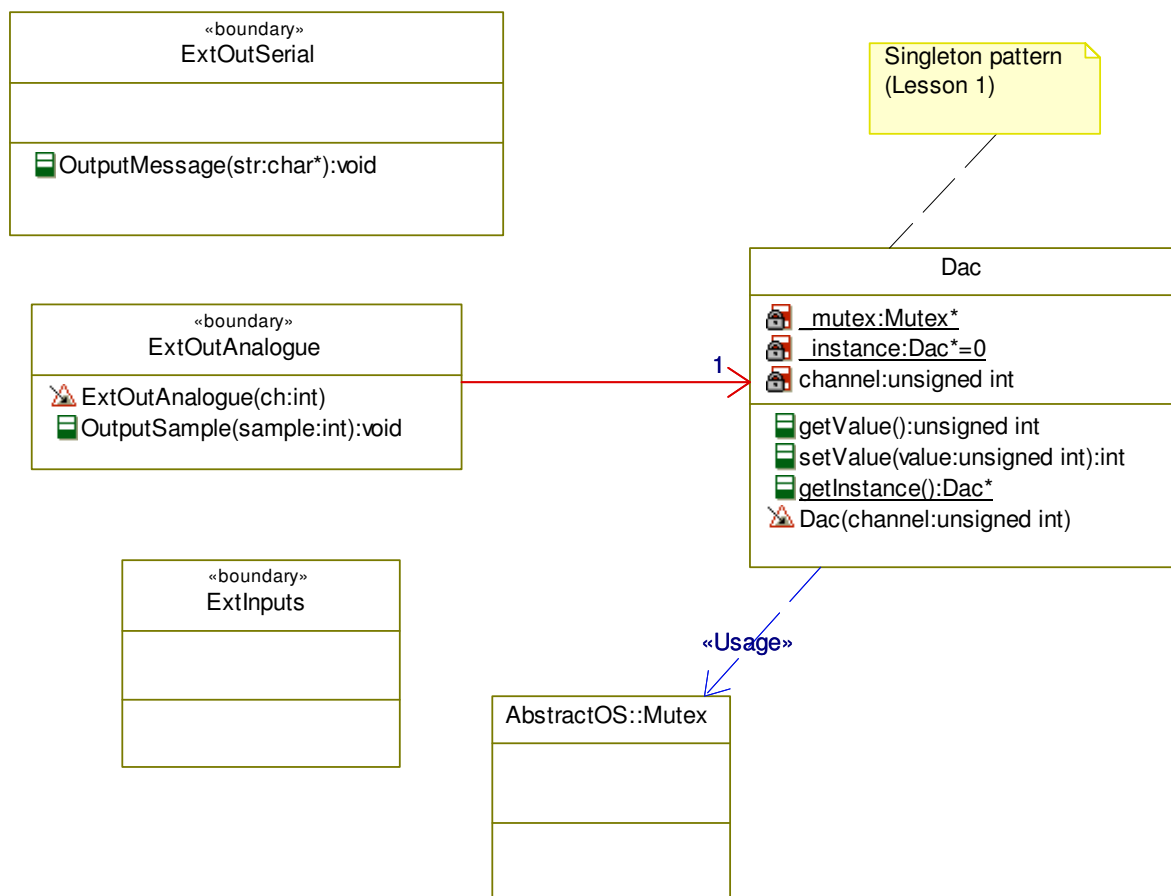
Figure 18 Hardware Abstraction using singletons for Dac abstraction

---

[4] http://www.linuxjournal.com/article/5896

5.2.5 Application Helper Classes

To improve the design a number of helper classes have been created.
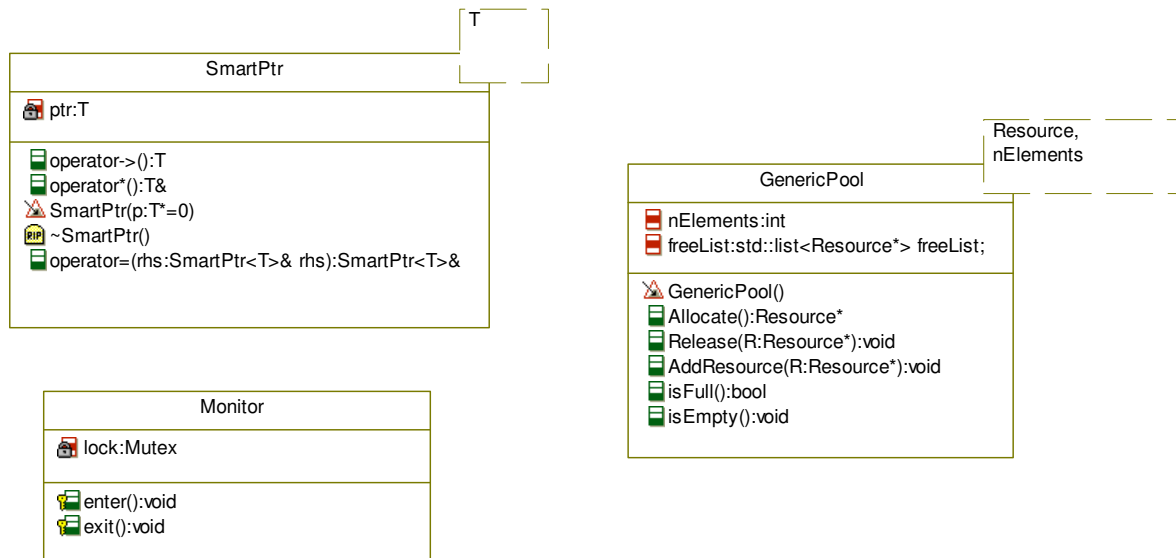
Figure 19 Application Helper classes

The *SmartPtr* class wraps a normal c-pointer. The constructor creates an object and the destructor deletes it, which is the smart thing about this pointer type. The "->" and "*" operators are overloaded, so that the smart pointers can be worked on like ordinary pointers. The *SmartPtr* class uses a **template pattern** that allows it to be used for pointers of different object types. The function of the *SmartPtr* class is very similar to that of the C++ standard library class "auto_ptr"[5].
A *SmartPtr* is used for creation of the *PatientModel* object, when the Test Model button is pressed in the GUI.

The *Monitor* class is used to implement the **Monitor Pattern**. It is used to synchronize access to a shared resource and is basically just a wrapped c-mutex. It is used for the *PatientModel* and the *FrameBufferPool*. The *FrameBufferPool* is accessed by both the Distributer- and RealTime threads. The *Monitor* Class inherits its methods "enter()" and "exit()" to *FrameBufferPool*, who uses them for allocation and release of the pool resource.

The *GenericPool* class implements the memory pattern called **Pool Allocation Pattern**. This pattern allocates a pool of objects at startup. These objects can be requested from the pool and released to it at run-time. The static memory allocation is faster than dynamic allocation at run-time. The *GenericPool* class is used by the *FrameBufferPool* to store frame buffer objects.

---

[5] http://ootips.org/yonat/4dev/smart-pointers.html

5.3 Use case realizations

This section describes the design of the uses cases that have been implemented so far. Refer to the Sapien 190 Requirement Specification [8] for use case details. The description uses the flow of the use case.

5.3.1 Use case #1: Execute and Control Simulation scenarios
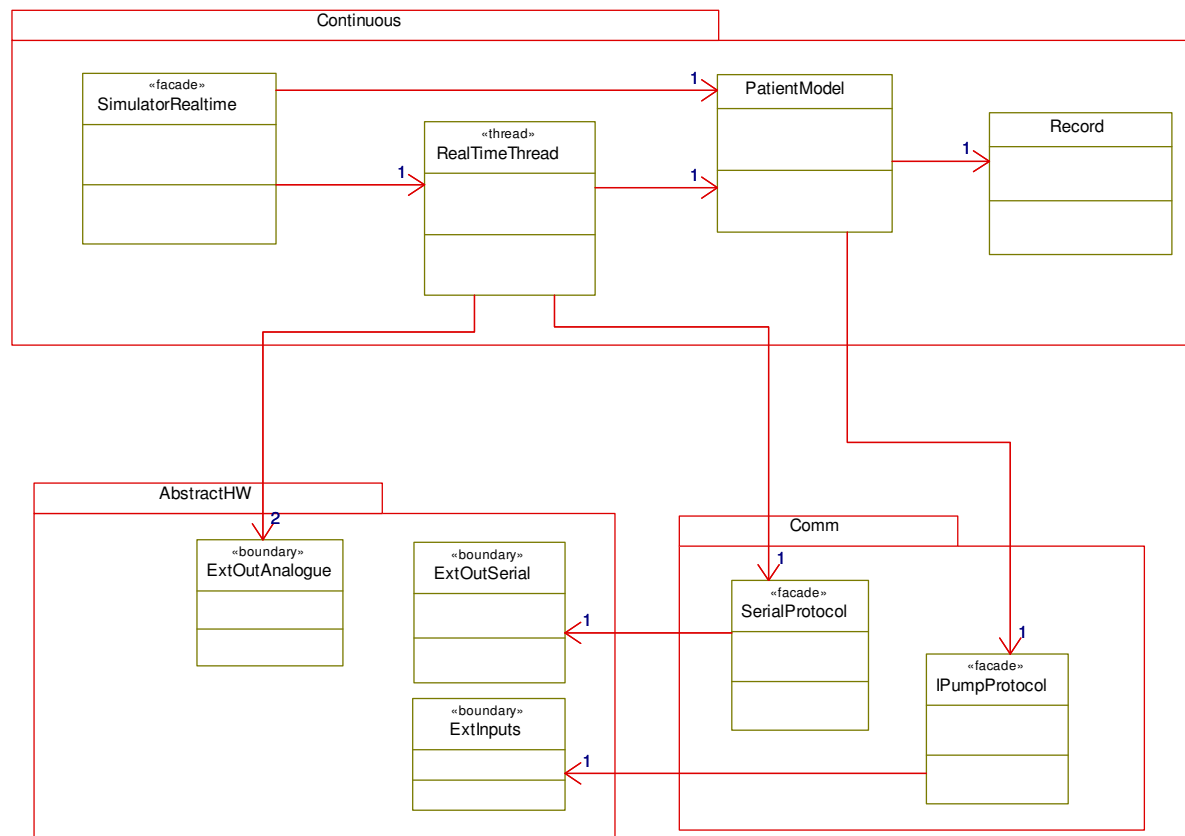
Figure 20 Logical view for use case #1 Execute and Control Simulation

This use case is activated when the simulation is started by the user. This means that to scope of the use case is covered in the continuous part of the design.

The design description following on the next pages is based on a simulation made in the Rhapsody tool.

Test setup in Rhapsody for #UC 1 continuous package SimulatorRealtime
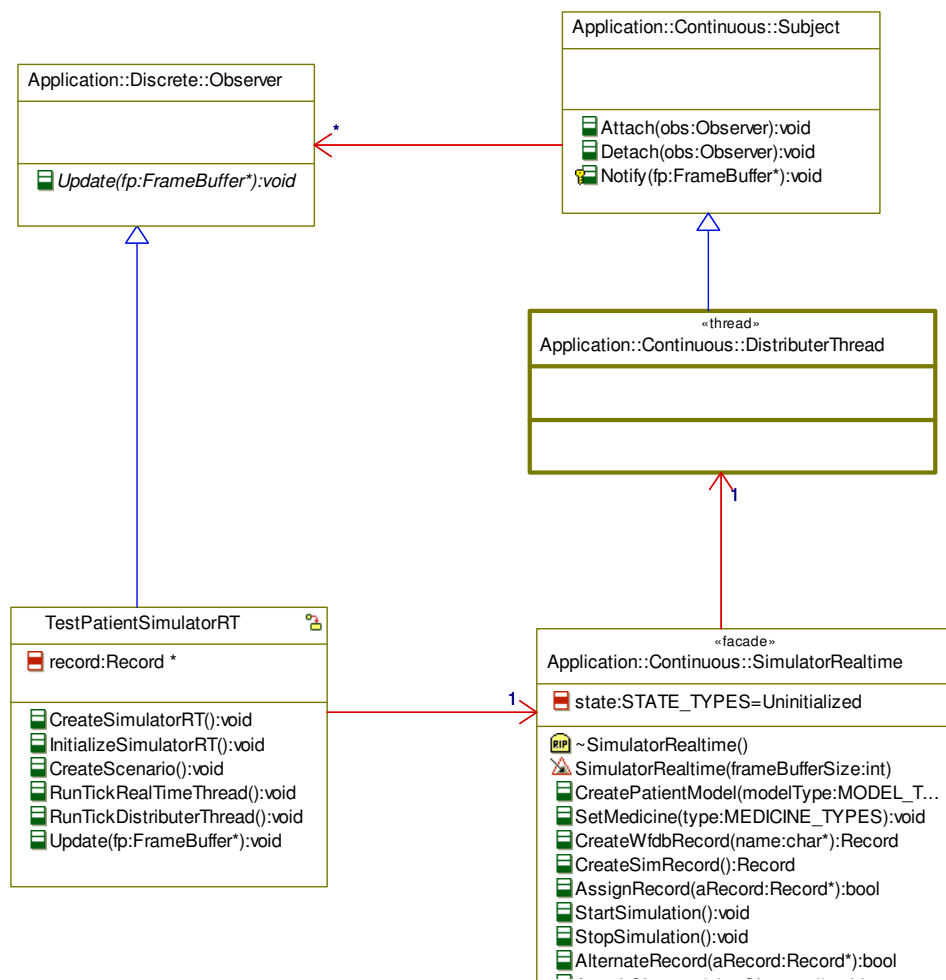
Figure 21 Test setup for UC#1 – test setup in Rhapsody only

The test is created in the *TestPatientSimulatorRT* class. This class creates all objects and implements an observer pattern.
The observer pattern is used to monitor the results of the *SimulatorRealtime*.

The test itself is created as a state machine in the Rhapsody tool (Figure 22).
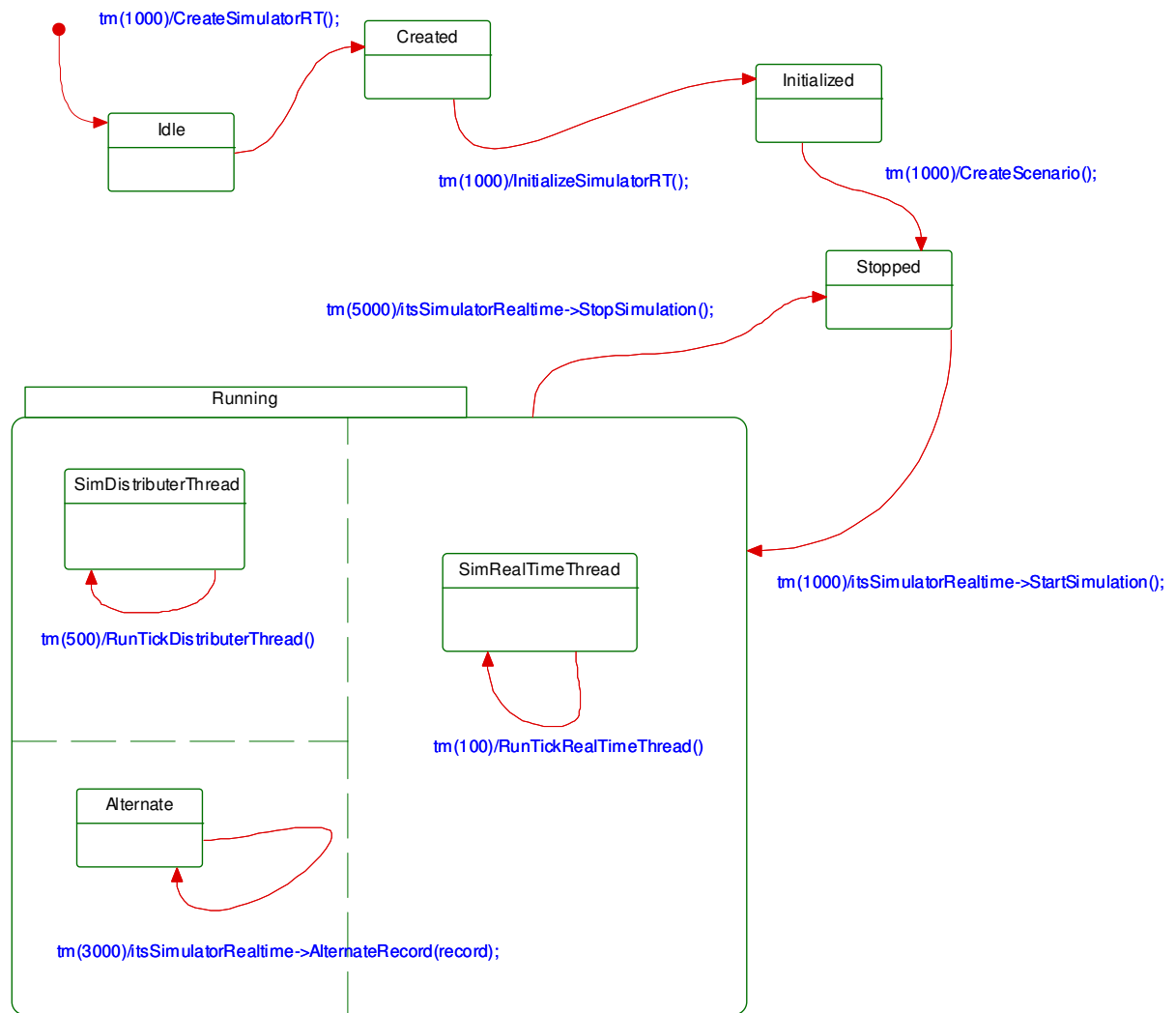
Figure 22 State machine test of UC#1 including simulation of Threads in Rhapsody

As shown in the state chart, the test is run like this:
- After 1000 ms, the SimulatorRT is created.
- After another 1000 ms, the SimulatorRT is initialized.
- After another 1000 ms, a scenario is created.
- After another 1000 ms, the simulation is started and the three "threads" are running.

When the simulator is running, the three threads are scheduled by a timer:
- For every 100 ms the RealTime "thread" is invoked.
- For every 500 ms the Distributer "thread" is invoked
- For every 3000 ms the AlternateRecord "thread" is invoked.

The timer scheduler is implemented to simulate an operating system. I the real application, this will of course be handled by the OS.

The following pages describe the flow of actions performed in the test using sequence charts and code snippets.

The first thing to do, is to create the RealTimeSimulatorRT:
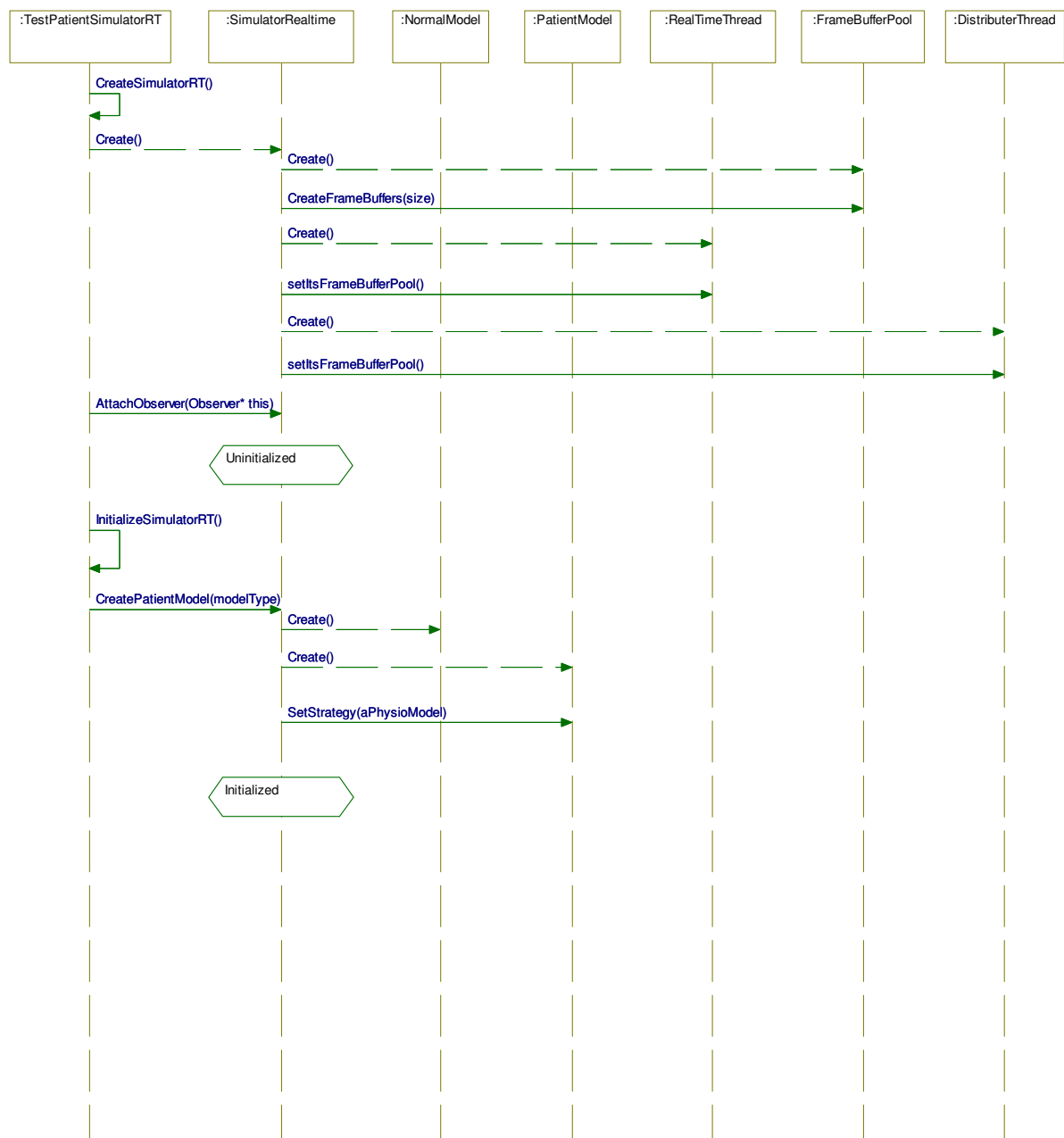
Figure 23 Sequence Diagram: Create Realtime Simulator

The *DistributerThread*, *FrameBufferPool* and *RealTimeThread* objects are created and the simulators observer is attached to the *SimulatorRealTime*.

When the objects have been created, the next thing to do is to initialize the model (See Figure 22). This is done in the SimulatorRealtime, where the *PatientModel* is created and its strategy set.

Code Snippet 1 -Code Snippet 3 shows the creation of objects. Note how the SimulatorRealtime is initialized to a frame buffer size of 5 samples.

```
void TestPatientSimulatorRT::CreateSimulatorRT() {
    itsSimulatorRealtime = new SimulatorRealtime(5);
    itsSimulatorRealtime->AttachObserver(this);
}

SimulatorRealtime::SimulatorRealtime(int frameBufferSize) :
state(Uninitialized)
{
    itsPatientModel = NULL;
    itsFrameBufferPool = new FrameBufferPool();
    itsFrameBufferPool->CreateFrameBuffers(frameBufferSize);
    itsRealTimeThread = new RealTimeThread();
    itsRealTimeThread->setItsFrameBufferPool(itsFrameBufferPool);
    itsDistributerThread = new DistributerThread();
    itsDistributerThread->setItsFrameBufferPool(itsFrameBufferPool);
}
```

Code Snippet 1 TestPatientSimulatorRT::CreateSimulatorRT

```
SimulatorRealtime::SimulatorRealtime(int frameBufferSize) :
state(Uninitialized)
{
    itsPatientModel = NULL;
    itsFrameBufferPool = new FrameBufferPool();
    itsFrameBufferPool->CreateFrameBuffers(frameBufferSize);
    itsRealTimeThread = new RealTimeThread();
    itsRealTimeThread->setItsFrameBufferPool(itsFrameBufferPool);
    itsDistributerThread = new DistributerThread();
    itsDistributerThread->setItsFrameBufferPool(itsFrameBufferPool);
}
```

Code Snippet 2 SimulatorRealtime::SimulatorRealtime

```
void TestPatientSimulatorRT::InitializeSimulatorRT() {
    itsSimulatorRealtime->CreatePatientModel(SimulatorRealtime::Normal);
}
```

Code Snippet 3 TestPatientSimulatorRT::InitializeSimulatorRT

The patient model strategy set to normal, as the infusion pump model is not yet implemented. Code Snippet 4 shows how a switch statement is used to set the strategy.

```
void SimulatorRealtime::CreatePatientModel(const
SimulatorRealtime::MODEL_TYPES& modelType) {
    if (state != Running)
    {
      if (itsPatientModel != NULL) delete itsPatientModel;
      setItsPatientModel(new PatientModel());

      switch (modelType)
      {
       case Normal:
            itsPatientModel->SetStrategy(new NormalModel());
                break;
        case InfusionPump:
            itsPatientModel->SetStrategy(new InfusionPumpModel());
            break;
      };

      state = Initialized;
    }
}
```

Code Snippet 4 SimulatorRealtime::CreatePatientModel

Now that the simulator has been created and is initialized, it is time to create the scenario.
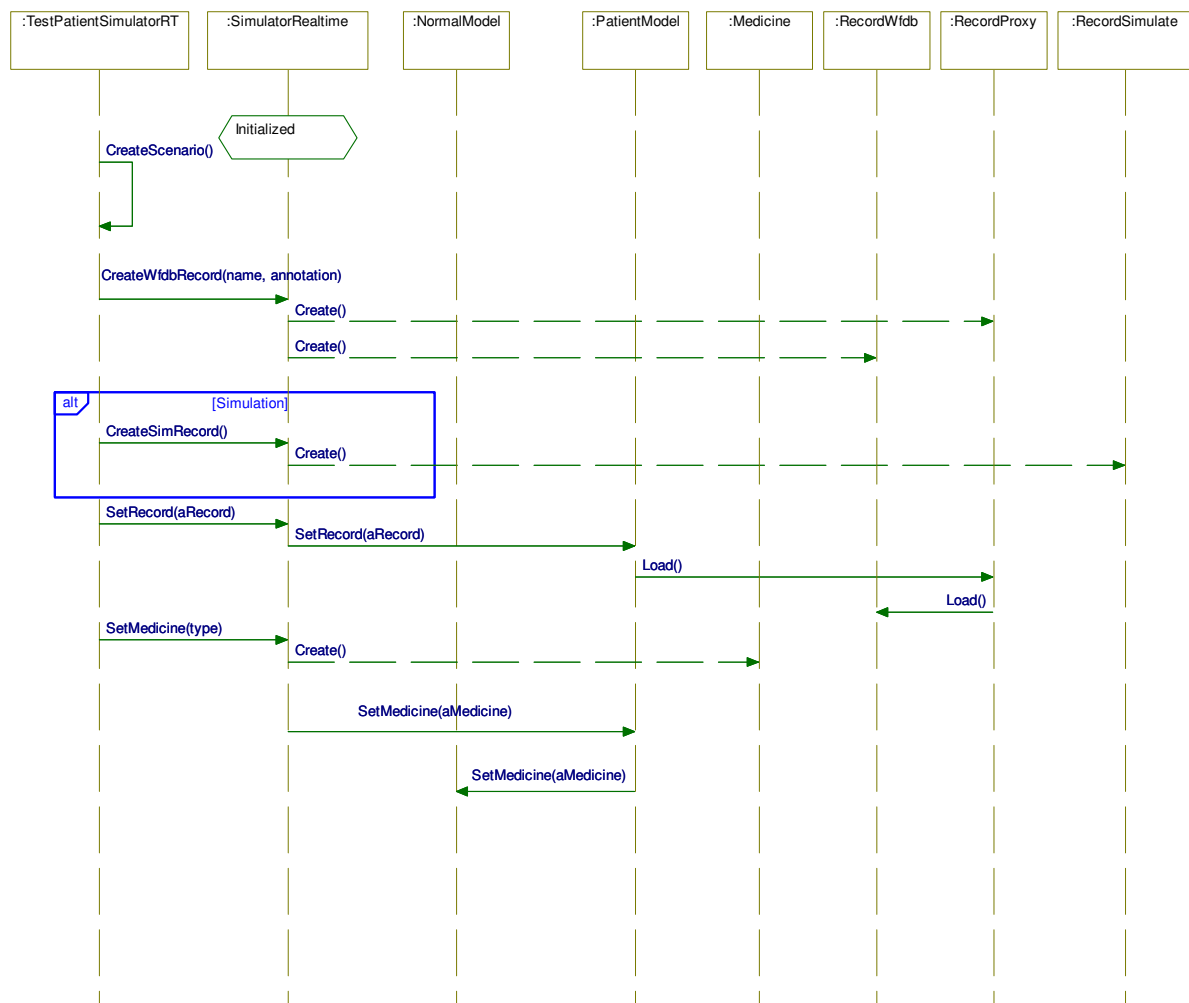
Figure 24 Sequence Diagram: Create WFDB or Simulation Record and Set Medicine

As depicted in Figure 24 and in Code Snippet 5, the *TestPatientSimulatorRT* has a CreateScenario method. This method creates and sets record and medicine for the current scenario. Note that this implementation is static, whereas the final release will create a scenario based on an XML file.

```
void TestPatientSimulatorRT::CreateScenario() {
    GainParam gainParam;
    #ifdef _LINUX
    record = itsSimulatorRealtime->CreateWfdbSimRecord("100s", "atr");
    #else
    record = itsSimulatorRealtime->CreateSimRecord();
    #endif
    itsSimulatorRealtime->SetRecord(record);
    itsSimulatorRealtime->SetMedicine(SimulatorRealtime::Morphine);
    gainParam.setGain(0.50);
    gainParam.Execute(itsSimulatorRealtime);
}
```

Code Snippet 5 TestPatientSimulatorRT:CreateScenario

Note the "#ifdef _LINUX" in the code. When the design compiles in Rhapsody to the windows platform, the record will use simulated data, whereas if the design is compiled to the Linux target platform, it will use an actual WFDB record.

```
Record* SimulatorRealtime::CreateWfdbRecord(char* name, char* annotation) {
    return new RecordProxy(new RecordWfdb(name, annotation));
}


bool SimulatorRealtime::SetRecord(Record* aRecord) {
    if (itsPatientModel != NULL)
    {
       state = Stopped;
       return itsPatientModel->SetRecord(aRecord);
    }
    return false;
}


void SimulatorRealtime::SetMedicine(const
SimulatorRealtime::MEDICINE_TYPES& type) {
    if (itsPatientModel != NULL)
       itsPatientModel->SetMedicine(new Medicine(type));
}
```

Code Snippet 6 SimulatorRealtime::CreateWfdbRecord/SetRecord/SetMedicine

Note how the CreateWfdbRecord method creates a new RecordProxy to *RecordWfdb*.

The simulation objects are now created and initialized. The actual simulation can now take place.
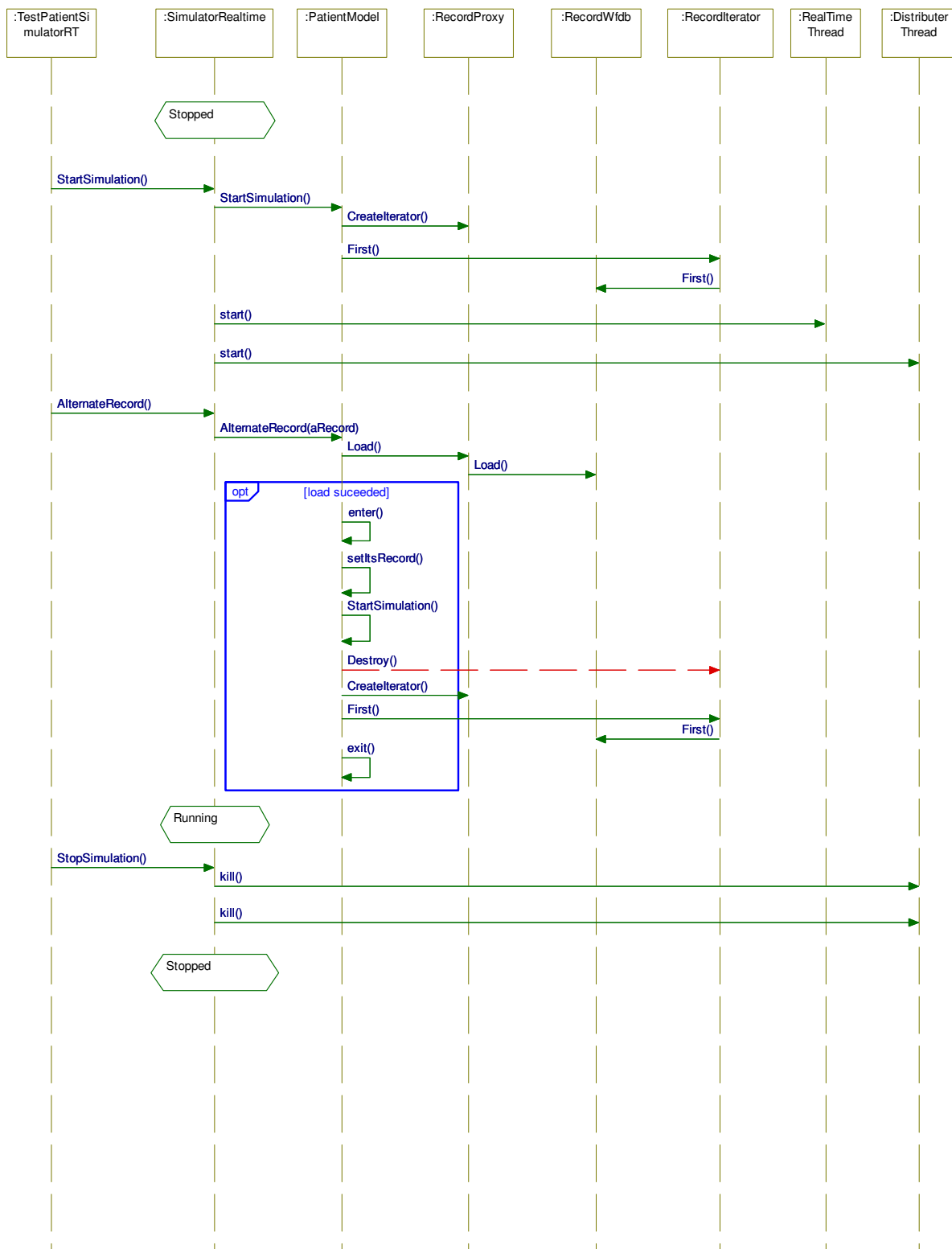
Figure 25 Sequence Diagram: Start Realtime Simulation

The test simulator calls the StartSimulation method in *SimulatorRealtime* (see Code Snippet 7). This method starts the *PatientModel* and runs the two threads Distributer- and

RealTime.

```
void SimulatorRealtime::StartSimulation() {
    if ( ((state == Initialized) || (state == Stopped)) &&
          (itsPatientModel != NULL) )
    {
       itsPatientModel->StartSimulation();
       itsRealTimeThread->setItsPatientModel(itsPatientModel);
       itsDistributerThread->start();
       itsRealTimeThread->start();
       state = Running;
    }
}
```

Code Snippet 7 SimulatorRealtime::StartSimulation

Also note that the state is set to "Running".
PatientModel's StartSimulation method creates the record iterator and calls the First
method to acquire the first sample from Record.

```
void PatientModel::StartSimulation() {
    if (itsRecord != NULL)
    {
       if (itsRecordIterator != NULL) delete itsRecordIterator;
       setItsRecordIterator(itsRecord->CreateIterator());
       itsRecordIterator->First();
    }
}
```

Code Snippet 8 PatientModel::StartSimulation

When the AlternateRecord is invoked by the timer/"scheduler", it tries to switch from the
current *Record* to the new *Record* given by the parameter.

```
bool PatientModel::AlternateRecord(Record* aRecord) {
    if (aRecord->Load()) {
       enter();
       setItsRecord(aRecord);
       StartSimulation();
       exit();
       return true;
    }
    return false;
}
```

Code Snippet 9 PatientModel::AlternateRecord

If the Record loads succesfully, this new record is set as the working Record and
simulation is re-started with StartSimulation.
What is shown in Figure 25 but not in Code Snippet 9 is that the old Record object is
deleted.

When the simulation is stopped, the two threads are killed.

```
void SimulatorRealtime::StopSimulation() {
    if (itsPatientModel != NULL)
    {
        if (state == Running)
        {
            itsDistributerThread->kill();
            itsRealTimeThread->kill();
        }
        state = Stopped;
    }
}
```

Code Snippet 10 SimulatorRealtime::StopSimulation

Figure 25 shows how samples are acquired from the WFDB record, processed and output to the D/A converter.
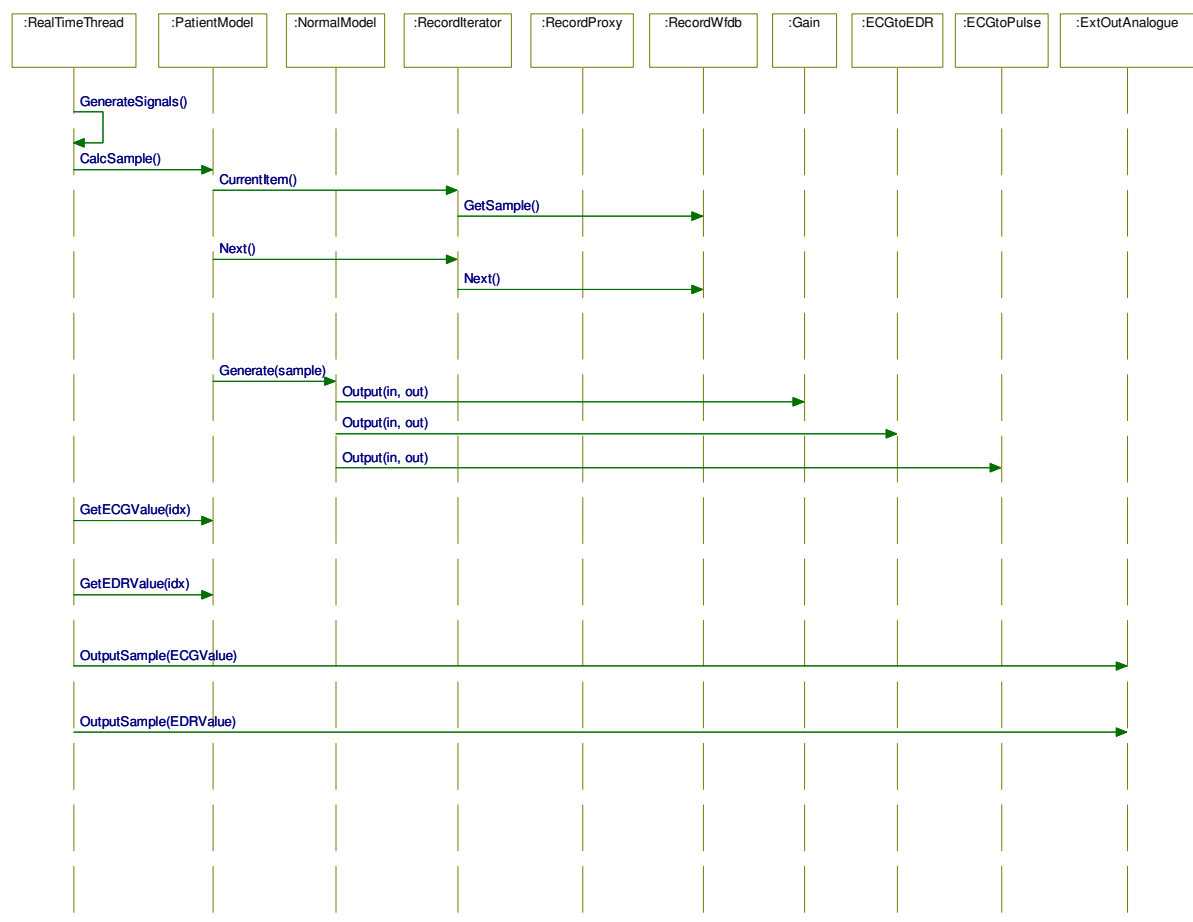
Figure 26 Sequence Diagram: RealTime thread GenerateSignals

As mentioned previously in this chapter the real time thread is called by the scheduler at a fixed interval in the simulation. In the real application, we would also like this to happen. This is not possible, as the current OS in not pre-emptive, that is timing cannot be guaranteed. I has been attempted to make the real-time thread run at a fixed time interval. The approximation is done by measuring the execution time of the real-time

thread, and letting it sleep for the remain time until the next time interval.

Tsleep = Tinterval – Texecution

The implementation can be viewed in Code Snippet 18.

A sample is acquired from the WFDB file and processed in the *PatientModel*.
The results returned are output to the D/A converter.

```cpp
void RealTimeThread::GenerateSignals() {
    itsPatientModel->CalcSample();
    WFDB_Sample e0 = itsPatientModel->GetECGValue(0);
    WFDB_Sample r0 = itsPatientModel->GetEDRValue(0);
    itsExtOutAnalogue[0]->OutputSample(e0);
    itsExtOutAnalogue[1]->OutputSample(r0);
    if (itsFrameBuffer != NULL)
      itsFrameBuffer->Insert(e0);
}
```

Code Snippet 11 RealTimeThread::GenerateSignals

```cpp
void PatientModel::CalcSample() {
    if (itsRecordIterator != NULL)
    {
      enter(); // Enter Monitor (Lock resource)
      itsSampleSet = itsRecordIterator->CurrentItem();
      if (itsRecordIterator->IsDone())
            itsRecordIterator->First();
      else
            itsRecordIterator->Next();

      if (itsPhysioModel != NULL)
                itsPhysioModel->Generate(*itsSampleSet);
      exit(); // Exit Monitor (Free resource)
    }
}
```

Code Snippet 12 PatientModel::CalcSample

The model calls the filters sequentially as described for the filters and pipes design.

```cpp
void NormalModel::Generate(SampleSet& sample) {
    itsGain.Output(sample, ECGSample);
    itsECGtoEDR.Output(ECGSample, EDRSample);
    itsECGtoPulse.Output(ECGSample, PulseSample);
}
```

Code Snippet 13 NormalModel::Generate

5.3.2 Use case #3. Adjust Scenario Parameters realization
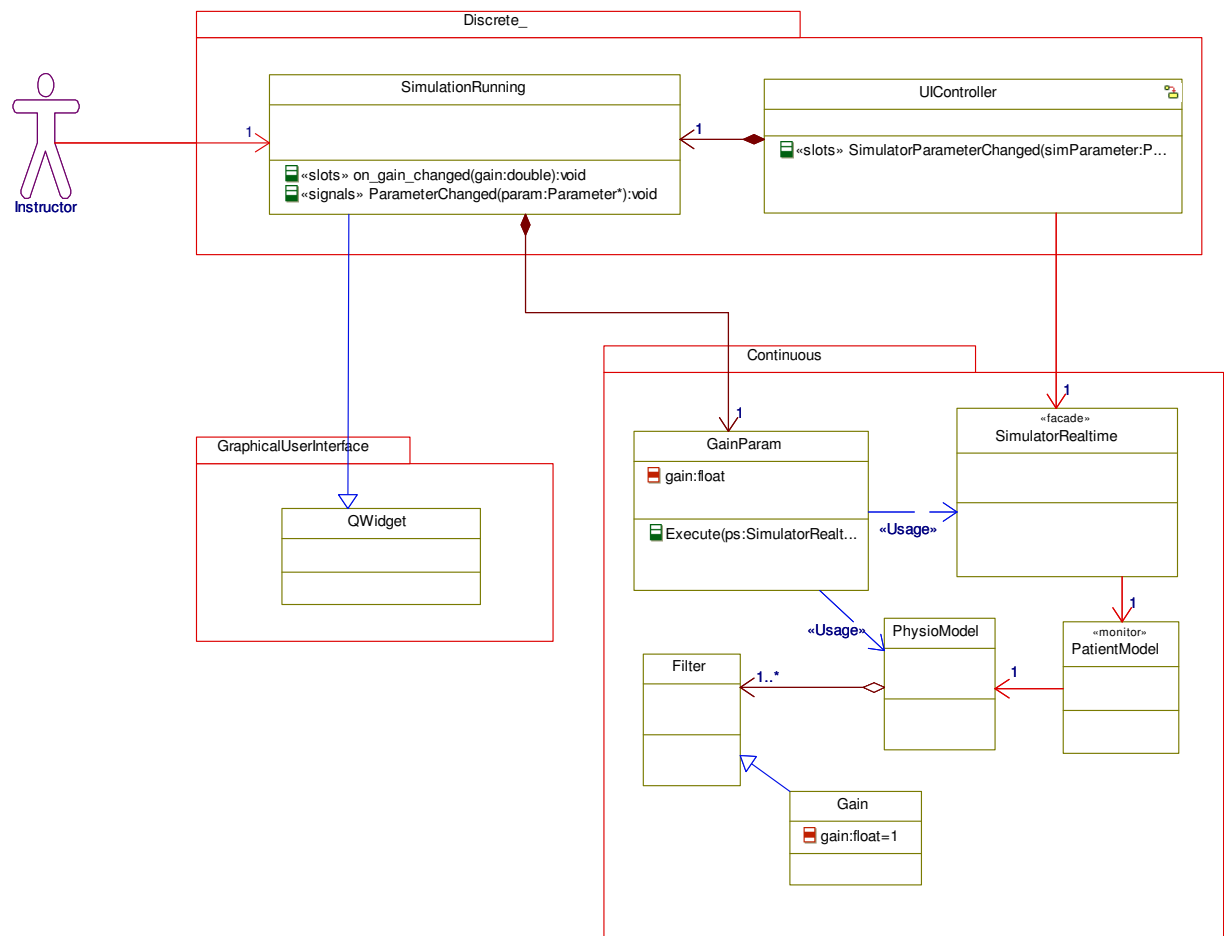
Figure 27 Actor, Classes and packages involved in use case

Change in the UI is returned to the *UIController* via events or callbacks that are used in Qt. In Qt callbacks are introduced with help of two types of functions.

- Signals
- Slots

Signals make the class function like a subjects in Observer Pattern, where the signal functions are a function which can be fired. The subscriber of a subject needs to have slot function for incoming callback. Whenever the *UIcontroller* receive a parameter change, it receives it as an object, using the command pattern, and execute the functionality on the simulation object.
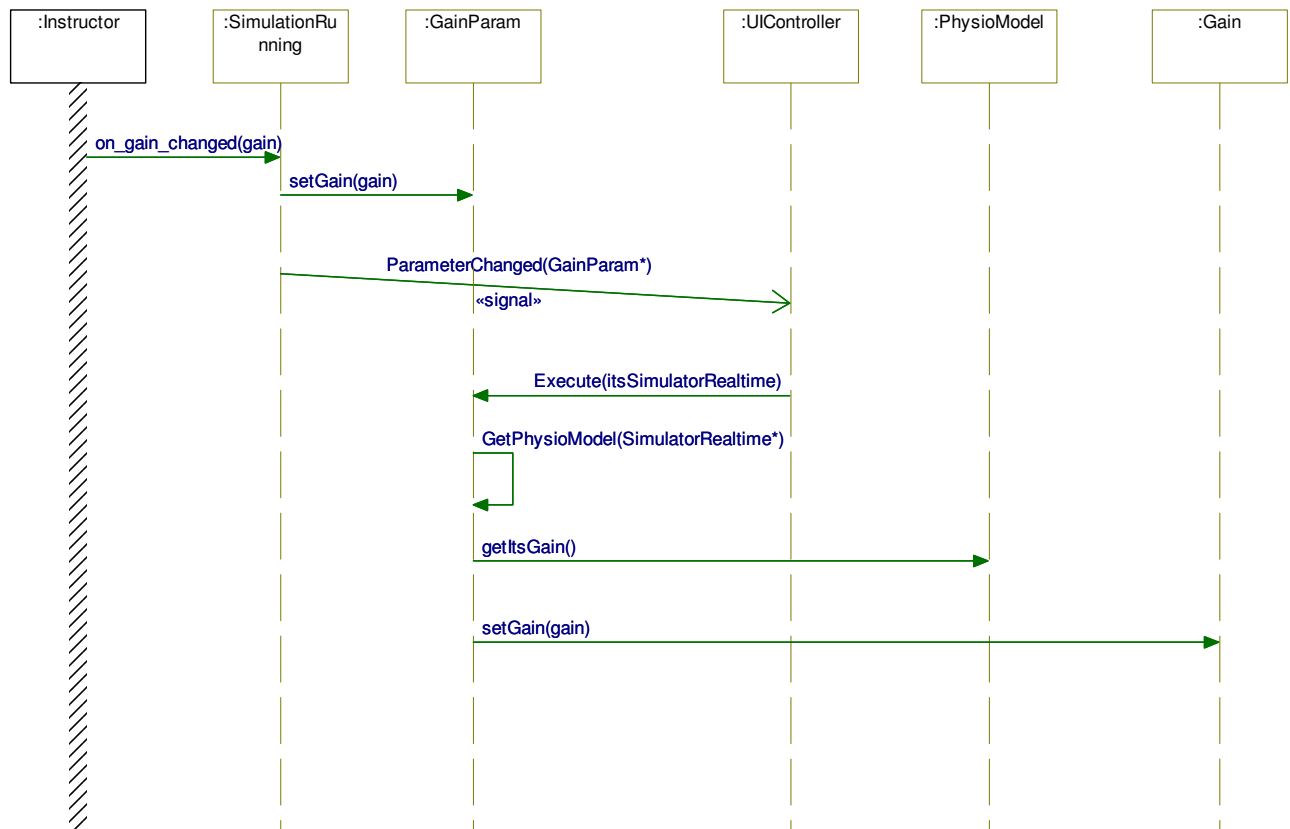
Figure 28 Scenario for instructor adjusting gain parameter

The above scenario illustrates what happens when the instructor adjusts the gain parameter. When the gain spin slider is change on the GUI the on_gain_changed method is invoked setting the gain "command" parameter. The gain is set and a callback is emitted asynchronously to the UIController by use of the signals and slots provided by Qt. Actually Qt are in this situation using a combination of an observer and message queue pattern.

```cpp
class SimulationRunning : public QWidget, public Observer {
    Q_OBJECT

private:
    Ui::SimulationRunning *ui;
    GainParam gainParam;

private slots:
    void on_gain_changed(double);

signals:
    void ParameterChanged(Parameter*);
};

void SimulationRunning::on_gain_changed(double gain)
{
    gainParam.setGain((float)gain);
    emit ParameterChanged(&gainParam);
}
```

Code Snippet 14 SimulatorRunning::on_gain_changed

The connection between emit of the event ParameterChanged and receiving the event in the UIController is done in the UIController constructor class by creating a connection between the signal ParameterChanged and slot SimulatorParameterChanged. The connection is established as a queued connection. This is the same principle as in the Attach method of the observer pattern. Here we have actually combined the command with the observer and message queue pattern.

```cpp
class UIController : public QObject
{
Q_OBJECT
public:
    explicit UIController(QObject *parent = 0);

    SimulatorRealtime* GetSimulatorRealtime();
    void ChangeState(SimState*);
    void start();

private:
    SimulationRunning sim;
    SimState *simState;
    SimulatorRealtime *itsSimulatorRealtime;

public slots:
    void SimulatorButtonPushed(SimCommand*);
    void SimulatorParameterChanged(Parameter*);

signals:

};

UIController::UIController(QObject *parent) :
    QObject(parent)
{
    connect(&sim,SIGNAL(ParameterChanged(Parameter*)),this,
            SLOT(SimulatorParameterChanged(Parameter*)),
            Qt::QueuedConnection);

    // Create the SimulatorRelatim system and configurate
    itsSimulatorRealtime = SimulatorRealtime::Instance(50);
    itsSimulatorRealtime->AttachObserver(&sim);

    this->simState = SimState::GetInstance();
    this->simState->SetSapienApplication(this);
}

void UIController::SimulatorParameterChanged(Parameter * simParameter)
{
    simParameter->Execute(itsSimulatorRealtime);
}
```

Code Snippet 15 UIController::UIController

6. PROCESS/TASK VIEW

This chapter describes the view of threads for the real-time part of Sapien 190. It contains an overview of the different threads in the architecture and synchronization used between them. In this chapter task and threads will have the same meaning and threads will be used in the text and UML diagrams.

There is included a Rate Monotonic Analysis (RMA) for the utilization of threads based on the calculation of utilization bounds including analysis of blocking threads.

Finally an analysis is included for the utilization of updating the ECG graph on the LCD display.

6.1 Process/task overview

In the below class diagram is illustrated threads in the Sapien 190 architecture.
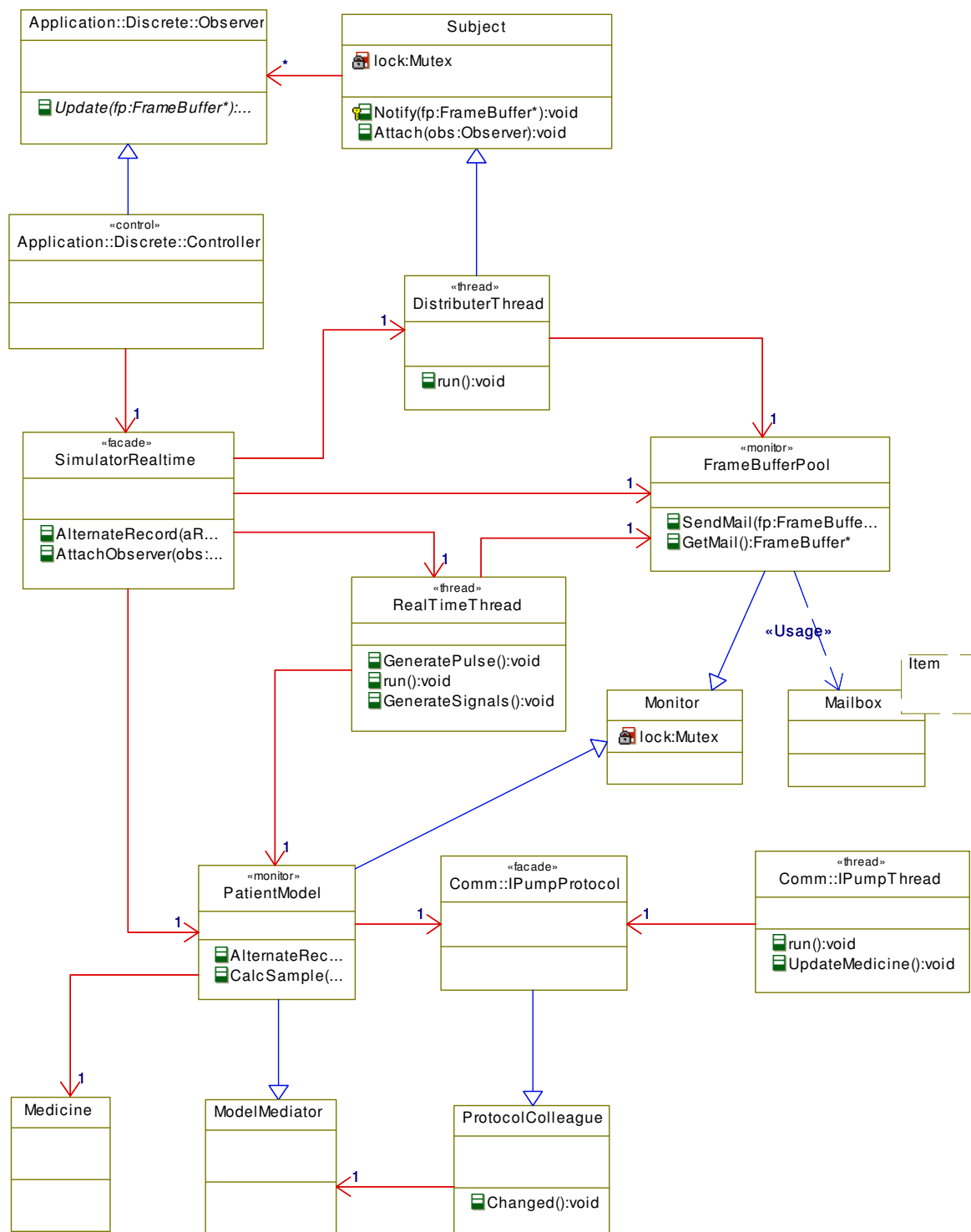
Figure 29 Overview of all threads for the Sapien 190 design

The following threads will be running in Sapien 190 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 model. States will be controlled from the main GUI thread (Qt) of the Sapien 190 and therefore synchronization is needed between controller and other continuous pats 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.

DistributerThread:

This thread is used to collect a buffer of samples that is updated to the observers that in this case will be the controller part of the GUI thread. This thread 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.

6.1.1 Syncronization between threads

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 Distri*b*uterThreads when they request the pool of free frame buffers. This part will be explained in details in chapter 6.4.

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.

6.2 Process/task implementation
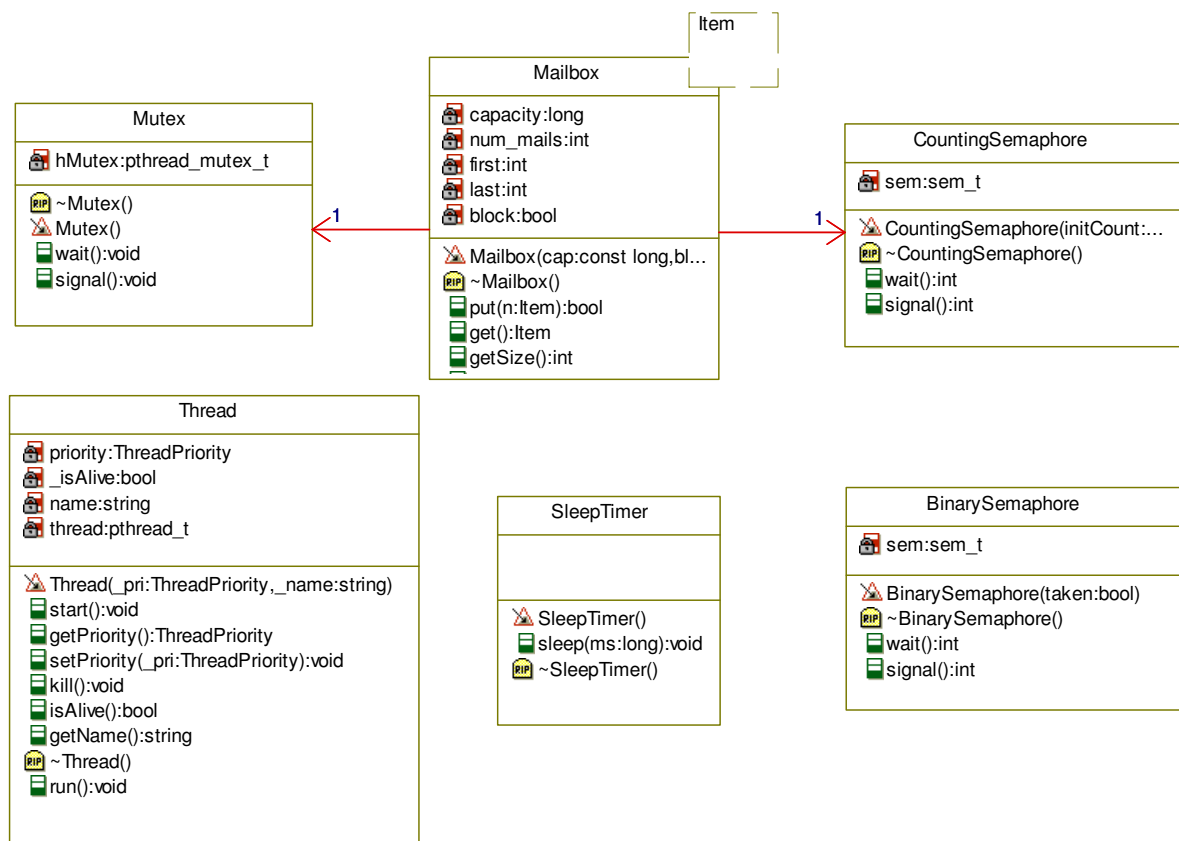
Figure 30 Abstract OS (Linux)

The operating system is encapsulated in the classes displayed above. The specification of this operating system is to be found in [10]. 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.

The mailbox is implemented by use of a mutex to ensure synchronization between put and get and a semaphore where the caller will be waiting if the mailbox is empty. Thread, Mutex and Semaphore is implemented by using the POSIX pthread, mutex and semaphore API for Linux.

6.3 Process/task communication and synchronization

A monitor is implemented using a mutex to implement the enter and exit functions of the monitor. The monitor is used by the *PatientModel* and *FrameBufferPool* as illustrated in class diagram Figure 29.
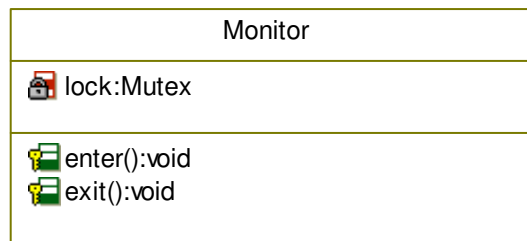
Figure 31 Monitor implemented using mutex
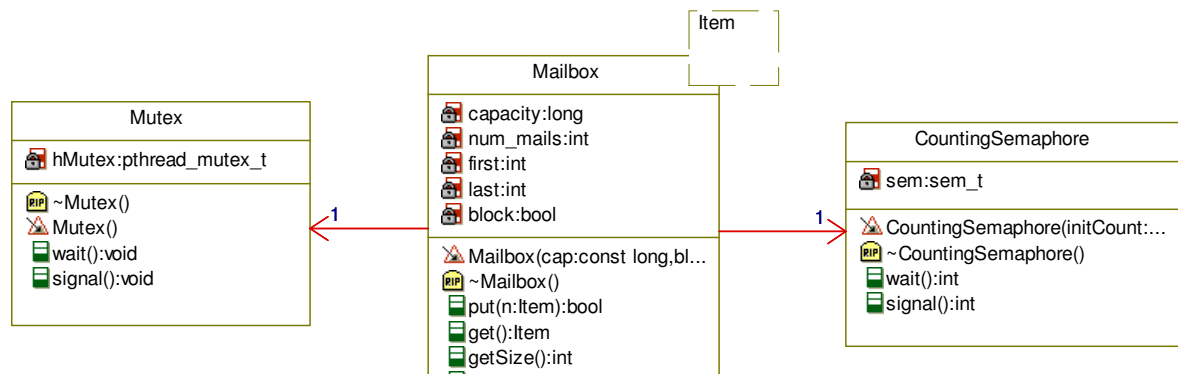
Figure 32 Mailbox implementation for Linux

The mailbox is implemented for the linux version using a buffer, mutex and counting semaphore. The mailbox can be used as a blocking or non blocking mailbox.

6.4 Process group 1 - DistributerThread and RealTimeThread

In this chapter we will describe in details how the interaction between *DistributerThread* and the *RealTimeThread* is performed.

6.4.1 Process communication in group 1
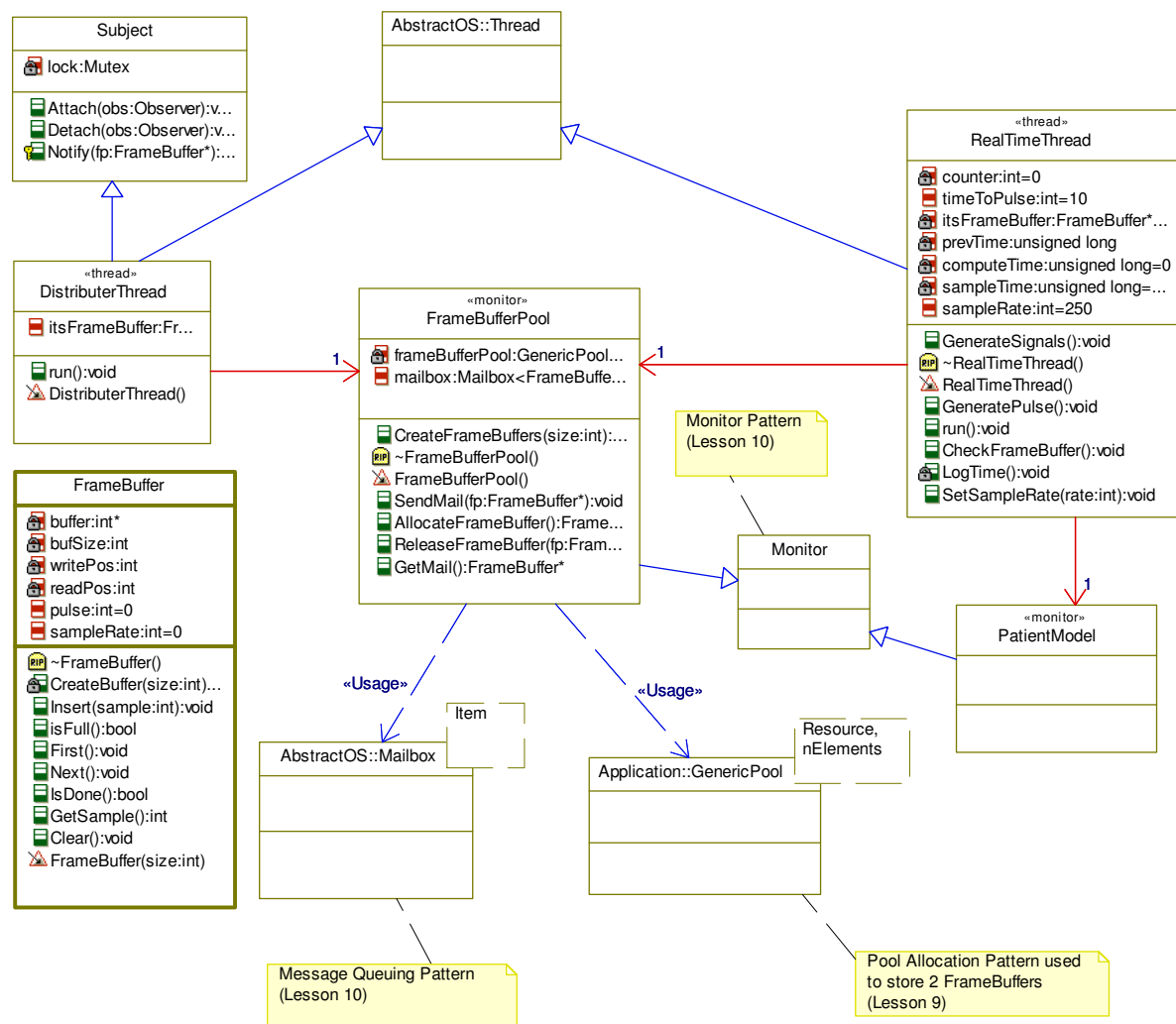
Figure 33 Process view for Distributer and RealTime threads and mechanism for synchronization

The essential process groups 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. A pointer to the frame buffer is passed through the mailbox by use of the methods SendMail and GetMail. GetMail will be blocking if there is no new frame buffers in the mailbox letting the *DistributerThread* being blocked. The pointer reference to the frame buffer is exchanged between the two threads to minimize copy of data.

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.

The monitor is used by the *FrameBufferPool* to synchronize allocation and release of *FrameBuffer's* to the GenericPool. The *FrameBuffer* contains a number of samples, pulse and sample rate all information used to be updated on the GUI. The *FrameBuffer* contains information that is send to the GUI controller using the observer pattern.
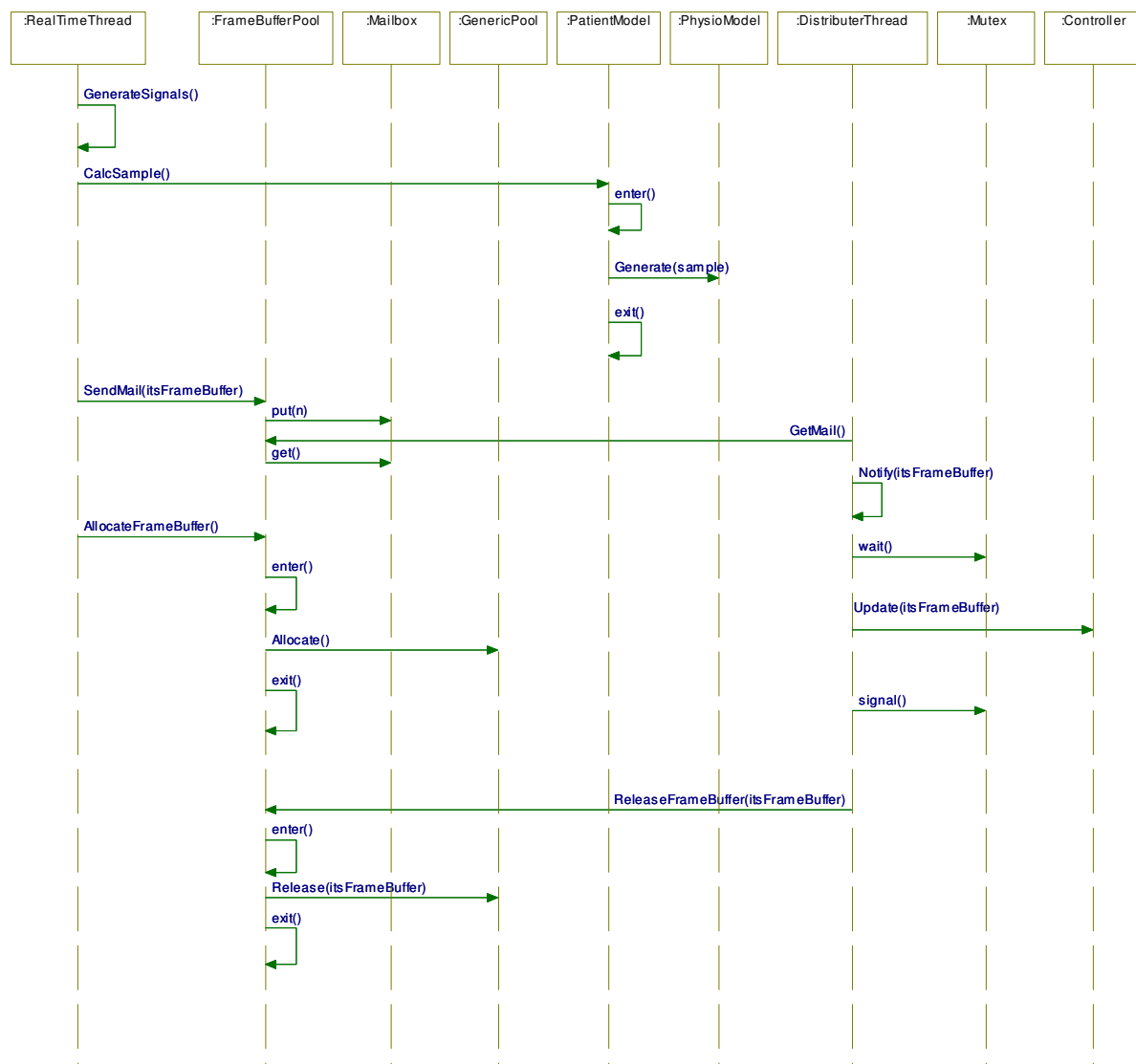
Figure 34 Synchronization between threads

The sequence diagram above illustrates part of a scenario where the *RealTimeThread* is calculating a new sample. When the Generate method is called in the *PatientModel* the enter and exit monitor functions are called to ensure that the GUI is not at the same time altering the record currently used to generate samples of the ECG, EDR and pulse signals.

When a frame buffer is filled with samples the *RealTimeThread* sends a filled FrameBuffer to the *DistributerThread* by calling the SendMail method in the *FrameBufferPool* that puts the pointer of the *FrameBuffer* in the Mailbox. The filled *FrameBuffer* is collected by the *DistributerThread* by calling the GetMail method that will be blocking in the get call to the Mailbox if it is empty.

A mutex internal in the *Distributer* thread is used to synchronize updating the UI controller class in case a call to Attach or Detach in the observer class is invoked at the same time. A monitor is also used by the *FrameBufferPool* to synchronize calling the methods Allocate and Release in the *GenericPool*.

6.4.2 Distributor thread

The distributor thread separates updating the GUI with information of the continuous part of the system and is active as long the discrete system is in the running state. It is activated for every update of the frame buffer that contains a configured number of samples. Every time the *DistributorThread* notifies the observers a lock (Mutex) is used to ensure mutual access to the list of observers.

```
void DistributerThread::run() {
    while(isAlive()){
      itsFrameBuffer = itsFrameBufferPool->GetMail();
      if (itsFrameBuffer != NULL) Notify(itsFrameBuffer);
      itsFrameBufferPool->ReleaseFrameBuffer(itsFrameBuffer);
    }
}
```

Code Snippet 16 DistributerThread::Run

```
void Subject::Notify(FrameBuffer* fp) {
    lock.wait();
    std::list<Observer*>::const_iterator iter;
    iter = itsObserver.begin();
    while (iter != itsObserver.end()){
        Observer *pObserver = *iter;
        pObserver->Update(fp);
        iter++;
    }
    lock.signal();
}
```

Code Snippet 17 Subject::Notify

6.4.3 Real-time thread

The real-time thread is a periodic thread that is executed every sample period running with the default sample rate of 250 Hz. This thread reads samples from the patient record and generates the ECG, EDR and pulse samples that are send to the analogue and serial interfaces. It collects a number of ECG samples that is transmitted to the distributer thread to separate updating the waveform graph from the real-time part of the system. To compensate for the computing time of generating the signals we have added the method LogTime that measures the time it takes to execute the run loop of the *RealTimeThread*. This time is subtracted from the sampleTime and will be limited to this time. Meaning if the sampling rate is increased too high or the compute time takes a very long time this thread will be running all time.

```
void RealTimeThread::run() {
    while(isAlive()){
      usleep(sampleTime - computeTime);
      LogTime();
      GenerateSignals();
      counter++;
      if (counter > timeToPulse)
      {
            GeneratePulse();
            counter = 0;
      }
        CheckFrameBuffer();
        LogTime();
    }
    // Ensure that Distributer thread will die
    if ((itsFrameBuffer != NULL) && (itsFrameBufferPool != NULL))
            itsFrameBufferPool->SendMail(itsFrameBuffer);
}
```

Code Snippet 18 RealTImeThread:::run

6.5 Rate Monotonic Analysis (RMA) with Task Blocking

In this chapter an analysis of the scheduling of all threads are calculated. An excel sheet is attached with the project that can be used to calculate the RMA based on a number of parameters and measured Worst Case Execution (WCE) times.

Two questions are important for this analysis.

Will we be able to calculate and generate the EDR and ECG signals with the given sampling frequency (250 – 400 Hz)?

What is the best configuration of parameters (frame buffer size) for the RMA analysis being able to update the LCD display with the ECG or EDR signal waveforms?

Below is listed the internal and external events that is important for this analysis. Here we have the sample event that will be generated by the sampling frequency based on an internal timer. The pulse and frame buffer are updated every Np or Nf samples given a periodicity that is in sync with the sample rate. The only external event is the PDU frame that is received from the IPUMP every second and will not be in sync with the sample rate.

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 50 (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

A number of parameter has been identified to be important for the scheduling and calculation of utilization bounds of the RMA analysis. These parameters can be used to adjust the Sapien 190 thread scheduling in finding an optimized configuration being able to run the system without any missed deadlines.

RMA analysis with Task Blocking (Fs = 250 Hz)

Parameters identified from internal and external event list:

_	Time units	1000000	us per second	
Fs	Sample frequency	250	Hz	
Np	Num samples for pulse calculation	10	Number	**Nf > Np**
Fi	PDU frequency	1	Hz	**Ti(3) > Ti(4)**
Nf	Frame buffer size	50	Number	

The Rate Monotonic Scheduling (RMS) is used to schedule the threads as described in the previous chapter. There are assigned fixed priorities to all threads, they are all periodic tasks and the highest priority has been assigned to the thread with the shortest period.

The table below contains the fundament for calculating the RMA. The WCE times is only estimates based on a measurement made in the *RealTimeThread* run part.

#	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

6.5.1 Calculation of Utilization Bound (250 Hz)

The RMA calculation is done using the steps and formulas presented in slides ThreadsAndSchedulability from Lesson 6.

S1. Utotal – calculate the total utilization for all of the events
S2. Btotal – calculate the blocking term
S3. Ubound – calculate the utilization bound
S4. UBtotal – compare total utilization (Utotal + Btotal) to the utilization bound (Ubound)

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*

6.5.2 Utilization Bound for the FrameBuffer event sequence (250 Hz)

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. The *FrameBuffer* event is schedulable if the utialization bound calculated in step 3 is bigger than total effective utilization for the *FrameBuffer* event calculated in step 2.

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**

6.5.3 RMA calculation for 370 Hz

In the next calculation we can see that with a sample rate of 370 Hz we will not be able to schedule the threads using a frame buffer of 50 samples. All other parameters are the same as for 250 Hz.

Rate Monotonic Analysis with Task Blocking (370 Hz)

(U = Utilization, UB = Utilization and Blocking, B = Blocking)

Utotal	sum (Ci/Ti)	0.10	
Ubound	n(2^1/n - 1)	**0.76**	
Btotal	max(Bti)	0.74	
UBtotal	Utotal + Btotal	**0.84**	*Ubound > Ubtotal*

Utilization bound for FrameBuffer (e4) valid as long Nf > Np (370 Hz)

Step 2: Calculate f
f4 = sum(Cj/Tj)|Hn + 1/Ti(4) (Ci(4) + Bi(4) + sum(Ck)|H1   *0.84*

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)	**FALSE**

7. DEPLOYMENT VIEW

7.1 System configurations overview
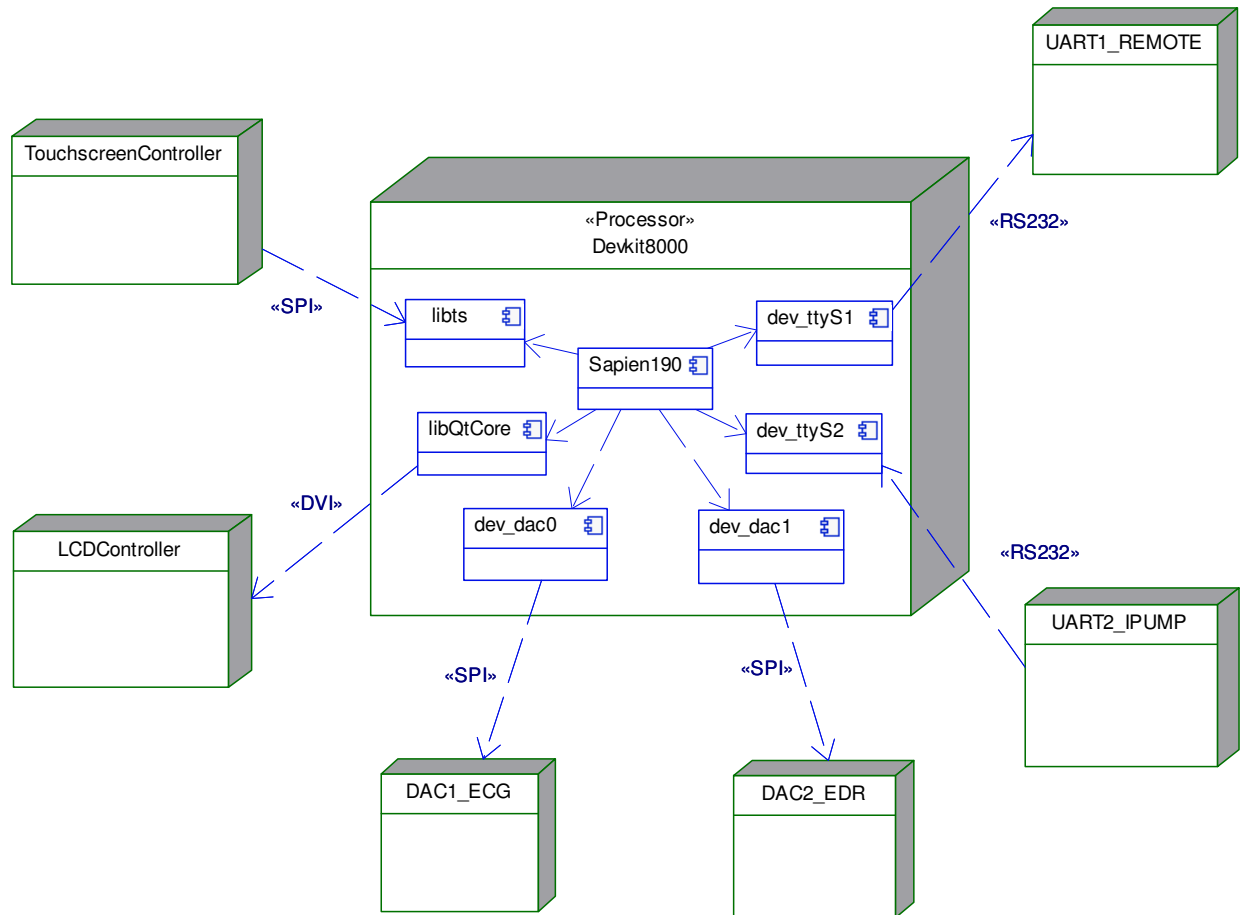
Figure 35 Essential HW nodes used from DevKit8000 used for Sapien190

On Figure 35 the deployment of our system is shown. The libts components contain driver information, which enable the Sapien190 to use the TouchScreenController. LibQtCore is used to communicate with our LCD controller, via DVI.

For converting our values into analog signal the system are using component dev_dac to communicate with our digital / analog converter.

8. IMPLEMENTATION VIEW

8.1 Overview

8.2 Component descriptions
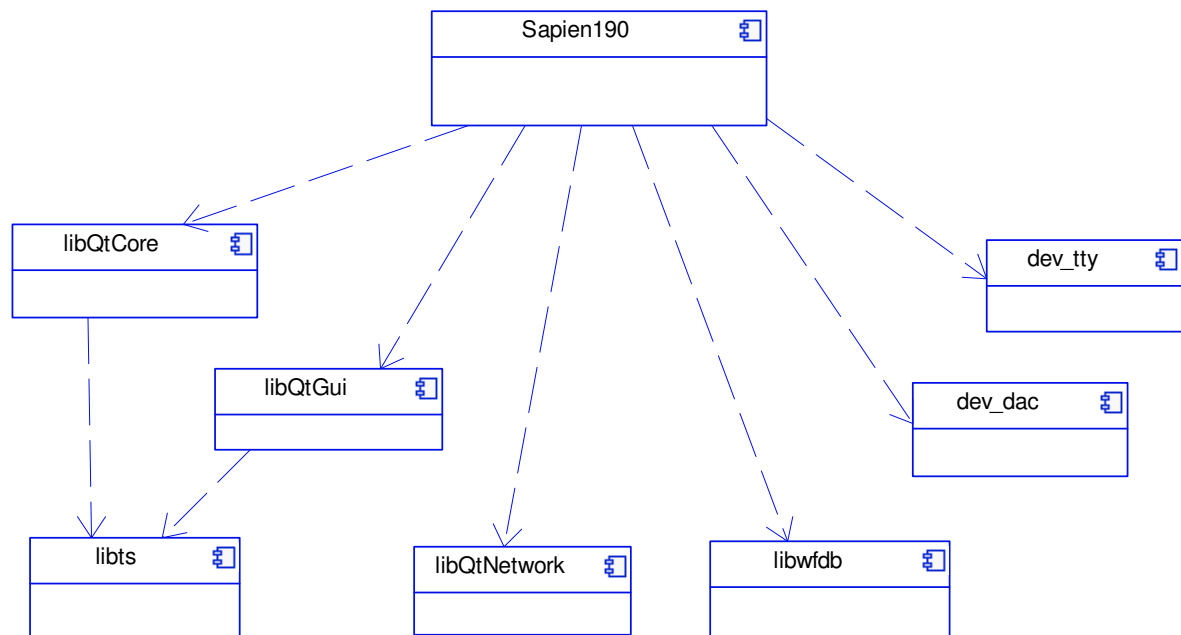
Figure 36 Component diagram for Sapien190 and libraries used from Qt and WFDB

8.2.1 Sapien190

Containing main program for Sapien190

8.2.2 libQtCore

LibQtCore is a component containing standard classes used when programming with the QT framework

8.2.3 libwfdb

Libwfdb provide drivers for reading all our patient records.

8.2.4 libts

Providing drivers to be able to display our user interface on the Devkit8000

8.2.5 dev_dac

Providing drivers to be able to write to the digital to analog converter

8.2.6 dev_tty

Drivers for writing to the serial port.

8.2.7 libQtGui

Providing a framework to draw and controler the userinface of Sapien190

9. GENEREL DESIGN DECISIONS

9.1 Architectural goals and constraints

The architectural goals are to make the system modifiable and provide high performance. Since the system is a real time system, there are constraints on how modular system can be, since modular and performance work in the opposite direction. It should be easy to extend the system new types of medicine, different kind of inputs and output. The system should be encapsulated so it's easy to port the system to different platform.

9.2 Architectural patterns

Observer Pattern:

    Observer pattern helps out with mass distribution of information in a system. If a number of processes are interested in the same information, observer pattern is the pattern to choose. In Sapien the observer pattern is used to distribute information generated by our simulation to the user interface. This a very common way to use observer pattern since it also notify the user interface when new data has arrived so it can be refreshed.

Five Layered Architecture:

    Layered pattern organizes domains into a hierarchical organisation based their level of abstraction. The advance of layered architectural is that it make it easier to find relevant code, and make it easier to replace whole layers, for instance if you want to port the system to another device. There used open layered architectural so it's allowed to call more than one layer down.

9.3 General user interface design rules

9.4 Exception and error handling

There is no exception handling implemented, if something is going wrong, the functionality is wrapped, so the systems does not crash.

9.5 Implementation languages and tools

This section lists the chosen implementation language and tools with version numbers. Implementation languages:

- C++
- Tools
- Ecliplse
- TrollTech QT Creator

9.6 Implementation libraries

- Qt-Everywhere-4.6.1
- WFDB

10. SIZE AND PERFORMANCE

10.1 Sample Computation Time

The most important quality measure of the Sapien 190, is the output stability. The requirements to the output signal are:
- Output sample rate: 1-500 Hz
- Output sample jitter: < 1 ms

This measure has been tested by measuring the sample computation time. As described in 6.4.3 the computation time must be less than the output sample time.

Figure 37 illustrates the results of a test where the DAC outputs are updated at 20Hz.
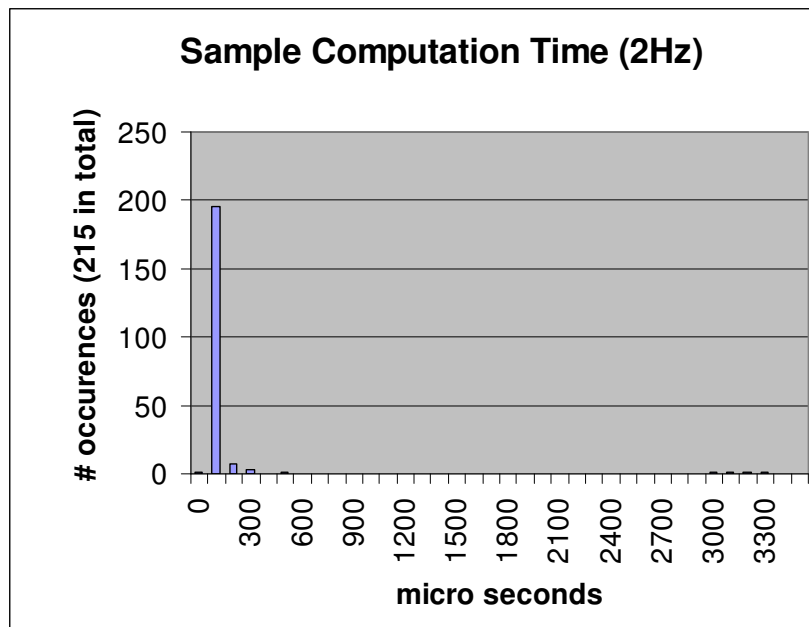
Figure 37 Sample Computation Time (20Hz)
Source: /Test Results/SapienComputeTime2HzTest2.txt

The result is that the average computation time is approximately 130 micro seconds. Note however the very small bars at the right.
When the LCD display is updated, then the computation time runs up in excess of 3300 us. It takes approximately 3200 us to update the display, so it seems that the real time thread is sometimes scheduled during a display update.

A max computation time of 3200 limits the output frequency to:

Fmax = 1/3300 us = 303 Hz

Increasing the DAC output frequency, we see that the effect becomes more pronounced.
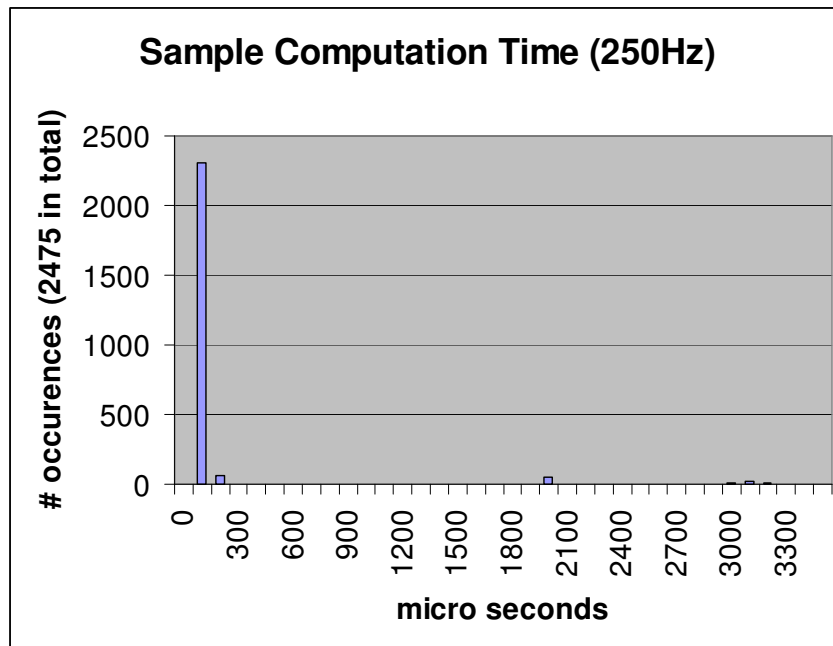
Figure 38 Sample Computation Time (250Hz)
Source: /Test Results/SapienComputeTime250HzTest2.txt

A solution to the problem could be to differentiate the priority of the distributor- (LCD update) and Realtime threads.

10.2 Graph Update Time

It is important that refreshing the display does not take too long time. If the update rate becomes lower than 30Hz, the display will appear to be flickering and will become uncomfortable to look at.
The update time has been measured using a timer.
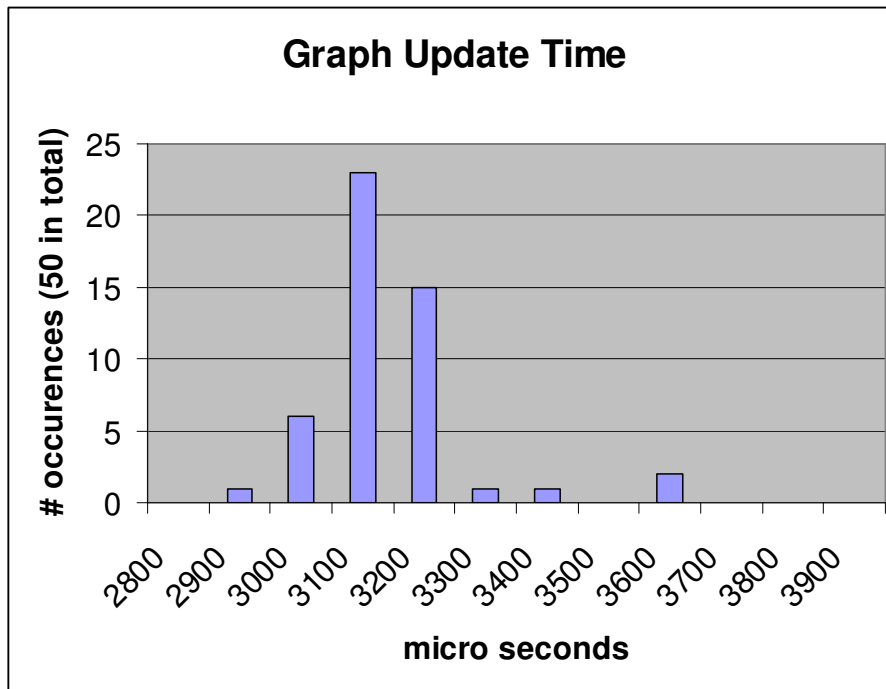
Figure 39 Graph Update Time
Source: /Test Results/SapienGraphUpdate.txt

It takes on average 3150 micro seconds to update the display. At 30Hz refresh rate, this means a CPU utilization of:
Ncpu = (30Hz * 3150 us)% = 9 %

10.3 DAC Response Time

The actual DAC output has been measured with an oscilloscope. To check the output stability, a well defined output has been output continuously. The output simply toggles for each sample with a 250Hz output sample rate. The result can be viewed in Figure 40.
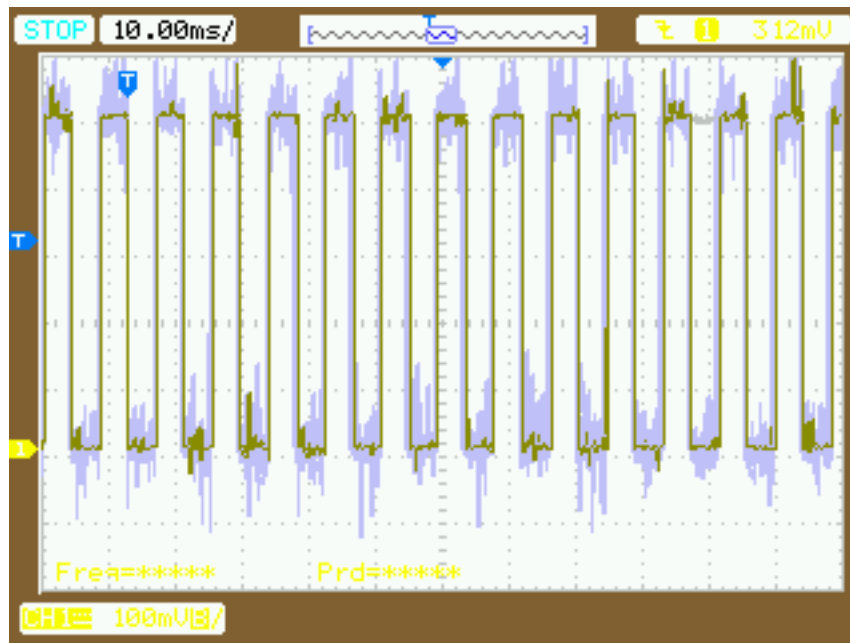
Figure 40 DAC Output toggeling

The output toggles every 4 ms as expected. At the right it can however be seen that a bit is missed once in a while. This occurs quite seldom as seen in Figure 38. It is caused by a prolonged graph update.

Looking at the ECG output shown in Figure 41, we can see that the signal is clearly visible, though it is rather noisy. The noise issues are electrical and not within the scope of this project.

Figure 41 DAC Output ECG

10.4 Code Size

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
```

Application:
```
Sapien190                 0.15 MB
```

Total:
```
Lib+Application          16.50 MB
```

The total code size may be reduced by linking the Qt libraries statically to the application.

10.5 Data Size

The WFDB record files vary in size but have a typical duration of 30 minutes. A 30-minute file has a size of approximately 5 MB at 250 Hz sample rate.

A scenario may contain several record files. Depending on usage, at least 10MB of data area should be available.

10.6 Memory Usage and Processor Load

The resource usage during application execution has been reviewed.

```
stud@ubuntu: ~

File  Edit  View  Terminal  Help
Mem: 41560K used, 84412K free, 0K shrd, 2740K buff, 29976K cached
CPU:   1% usr   0% sys   0% nice  97% idle   0% io   0% irq   0% softirq
Load average: 0.00 0.01 0.00
  PID  PPID USER     STAT   VSZ %MEM %CPU COMMAND
 1351  1334 root      S    36180  29%   2% ./sapien190_tgd6 -qws
 1344  1342 root      R     2852   2%   0% top
 1333  1312 root      S     2220   2%   0% /usr/sbin/dropbear -r /etc/dropbear/dr
  139     2 root     SW<       0   0%   0% [omap2_mcspi]
  434     2 root     SW<       0   0%   0% [mmcqd]
 1334  1333 root      S     2852   2%   0% -sh
 1342  1341 root      S     2852   2%   0% -sh
 1320     1 root      S     2740   2%   0% /sbin/syslogd -n -C64 -m 20
 1322     1 root      S     2676   2%   0% /sbin/klogd -n
 1341  1312 root      S     2220   2%   0% /usr/sbin/dropbear -r /etc/dropbear/dr
 1312     1 root      S     2048   2%   0% /usr/sbin/dropbear -r /etc/dropbear/dr
  476     1 root      S <   1816   1%   0% /sbin/udevd -d
 1329     1 root      S     1712   1%   0% /sbin/getty 115200 ttyS2
 1330     1 root      S     1712   1%   0% /sbin/getty 38400 tty1
    1     0 root      S     1472   1%   0% init [5]
  338     2 root     SW<       0   0%   0% [mtdblockd]
  439     2 root     SW<       0   0%   0% [kjournald]
    6     2 root     SW<       0   0%   0% [khelper]
  159     2 root     SW<       0   0%   0% [kmmcd]
    2     0 root     SW<       0   0%   0% [kthreadd]
```
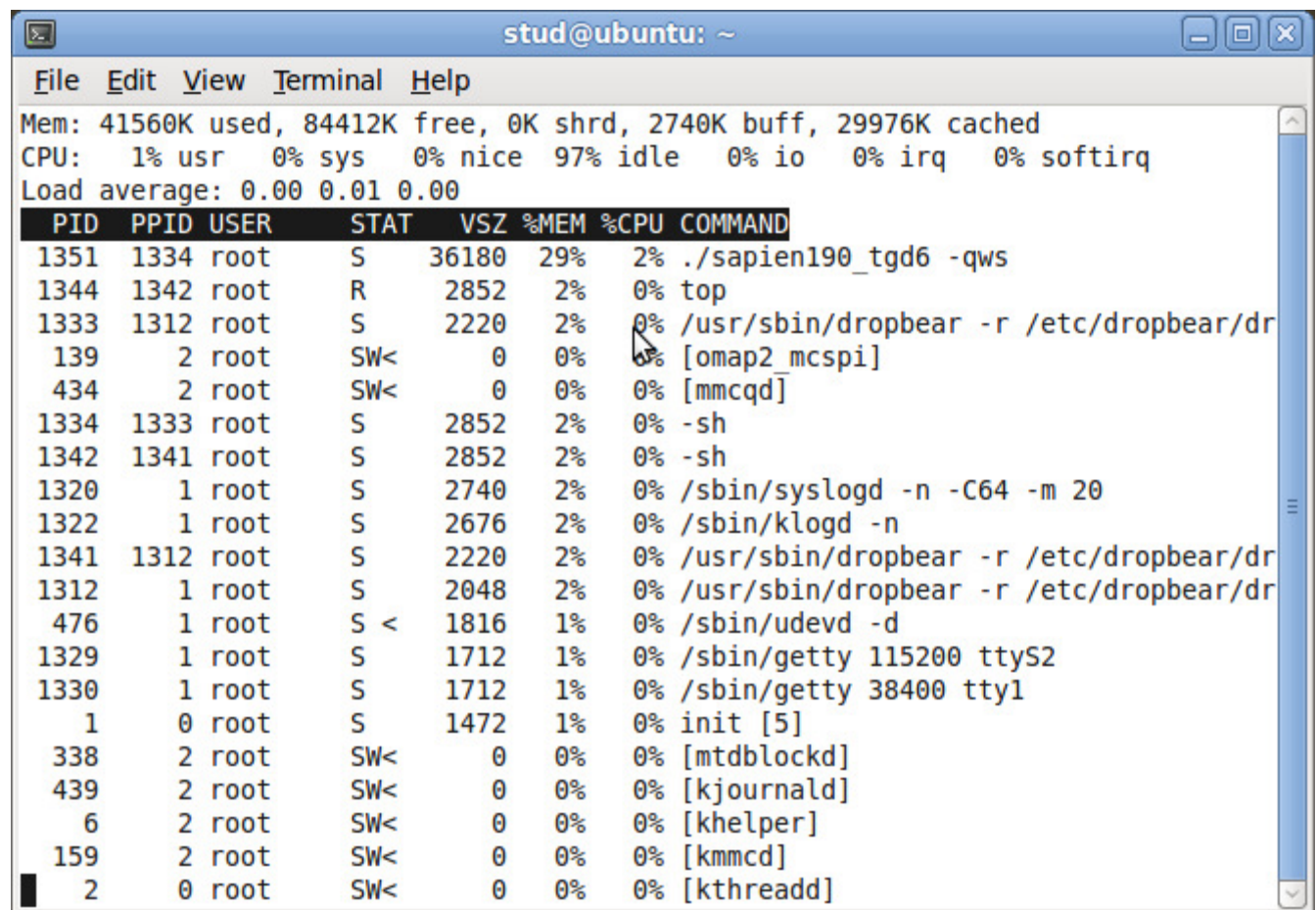
Figure 42 Memory and CPU usages on target

From the plot it can be derived that the application uses 29% of the boards 128 MB RAM memory. This equals 36MB.

Running the application, but without pressing the start button, the WFDB records are not loaded. This resulted in memory consumption of 20MB.

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 remaining memory usage may come from heap usage and Qt related activities.

The memory usage should be investigated further using a memory profiling tool. This job should have high priority for the next iteration.

The CPU load is light. The 2% result is lower than estimated in 10.2. 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 %. The current design seems to be efficient enough for the current hardware platform.

11. QUALITY

This section describes the non-functional qualities of the system.

11.1 Operating Performance

The required MTBF is not specified directly in the specification [8]. However the system has been designed to minimize bugs by applying GRASP[6] principles such as "Low Coupling", "High Cohesion" and by using a layered architecture.

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 should replace the existing pointers used in factories and similar places, especially where objects are created and deleted often.

Another performance measure is the jitter of the D/A output. The design uses the D/A output a sink, that is sourced from the input records. That is, the output pulls data from the input. The input itself does not take initiative to source data to the output.
This simple source-sink design protects against data contention and provides the lowest possible jitter as it is driven by the D/A output.

Jitter performance could be improved by providing fifo buffers on the D/A output data, and letting the device driver control the output timing.

11.2 Quality Targets

The main targets specified ([8] section 5) are:
- Maintainability
- Correctness
- Usability

How the design seeks to fulfil these goals will be described next.

11.2.1 Maintainability

From a pure software design perspective, the following sections on Extensibility and Portability explain how it has been implemented.

From an architecture point of view, the 5-layered architecture allows to modify one layer without affecting the other layers, thus making the design easier to maintain.

From a programming tool perspective, the top-down design approach using the Rhapsody tool has been very effective. Changing the design in the future, can be done "by the click on a mouse button". Rhapsody generates most of the code, and modified code is placed in a separate folder, making it easy to identify which files need to be updated.
Source Code File structure:

```
--- Sapien              - Main file
    `--- rpy            - Folder with original Rhapsody created files
```

---

[6] Craig Larman, Applying UML and Patterns

```
`--- rpyLocal        - Locally modified version of selected Rhapsody gen files
`--- presentation    - Qt files and user interface (GUI and Discrete packages)
`--- hw              - Hardware specific files (DAC)
`--- oxf             - Definitions files (Rhapsody)
`--- os              - OS specific files (AbstractOS package)
   `--- posix        - Posix specific files
```

11.2.2 Correctness

As described in 11.1, the system has been designed to provide minimum jitter on the DAC output with the current non-pre-emptive setup.

Quantization errors in the filters have not been considered, but should be subject to further investigation.

11.2.3 Usability

This first iteration does not provide much usability to the end-user. However a great framework has been created for adding a lot of functionality in an easy way. This again has something to do with the extensibility and portability described in the following sections.

11.3 Extensibility

The software design allows the software to be extended with minimum effort and to keep domain specific code together. This satisfies the Open-Closed Principle[7]

11.3.1 Patient Model

The design is extensible to new data record sources. The *Record* class can use the *RecordProxy* to point a new source (Figure 5).

The *SampleSet* is not limited to use two samples per set as it is used at the moment.

Using the *RecordIterator* to acquire samples makes the *PatientModel* insensitive to changes in the *SampleSet* (Figure 4).

11.3.2 PhysioModel

The *PhysioModel* is build using a Strategy Pattern and supports extension of models by adding new strategies (Figure 7).

Filters can be added or reused in the *PhysioModel* as the filters and pipes pattern is used (Figure 8). New filters inherit from the *Filter* class and are instantiated in the filter factory in the appropriate model (Figure 6).

11.3.3 DistributerThread

The *DistributerThread* uses an observer pattern to notify the *LCDScreen* of frame buffer updates.
The system can be extended with data processing- or other algorithms that subscribe to

---

[7] http://www.objectmentor.com/resources/articles/ocp.pdf

the frame buffer data, by implementing the "Update" method (Figure 13).

11.4 Portability
Care has been taken to create a design that wraps OS- and hardware specific code. The layered architecture encapsulates hardware/OS specific to increase portability.

11.4.1 Hardware
The hardware boundaries have been encapsulated in the AbstractHW package and contain (Figure 3):
- ExtInputs – Input from infusion pump
- ExtOutAnalogue – D/A outputs
- ExtOutSerial – Serial digital interface for transmitting pulse values

The abstraction provides a well-defined boundary to the hardware interfaces. Changing the hardware will only require changes in the *AbstractHW* implementation.

The LCD display and touch panel is not interfaced to directly, rather through the QT framework. The QT framework can be seen as an abstraction of the whole GUI.

The hardware abstraction is as such not an abstraction to the raw hardware, this is the job of a device driver. The abstraction is actually an interface to the resources provided by the operating system.

11.4.2 OS
Code specific to the operating system is placed in the AbstractOS package (and in the AbstractHW package, see 11.4.1).

The AbstractOS package currently holds an abstraction of the Posix library "pthread". Porting the design to a different operating system will require an update of this class. OS specific classes are placed under Sapien/os/ in the project file structure.

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, but its performance in example a non-preemptive Linux version must be investigated.

11.4.3 Spoken Language
Currently, no measures have been taken to support multiple languages. This could however be implemented with an abstract factory pattern. The factory should create objects that use the proper text source (xml file or similar).

12. COMPILATION AND LINKING
This section describes the process of compiling and linking a program.
This project has been developed using a number of different tools.

IBM Rhapsody has been used for UML modelling, simulation and testing. It has been

used for automatic source code generation and compilation on windows using Cygwin and generating C++ source code classes for Linux.

Eclipse has been used on Ubutu Linux in WMware for implementation and testing of the WFDB interface and hardware interface to the DAC and serial ports.

Qt has been used on Ubutu Linux for the final application by integrating and compilation of the graphical user interface, UML model and interface implementation made in eclipse.

The ARM cross compiler has been used to compile eclipse and Qt projects for the target.

In the following chapters the above tools usages for compilation the software for the Sapien patient simulator to target (Devkit8000) is described.

12.1 Rhapsody modeling and testing

Rhapsody (version 7.5) has been used to create an UML model for the Sapien patient simulator that is used for test of the simulator model including generated C++ source code before compilation on Linux and Target. We have created test scenarios by use of the Rhapsody state diagrams and used them for testing the model by setting breakpoints in the states and made inspection of the state variables of the model by using the high level abstractions of that is possible with Rhapsody. This approach has reduced the development time in removing a lot of the manual C++ debugging and coding of the model. In this chapter we will describe how Rhasody has been configured for testing and code generation for Linux and Target.
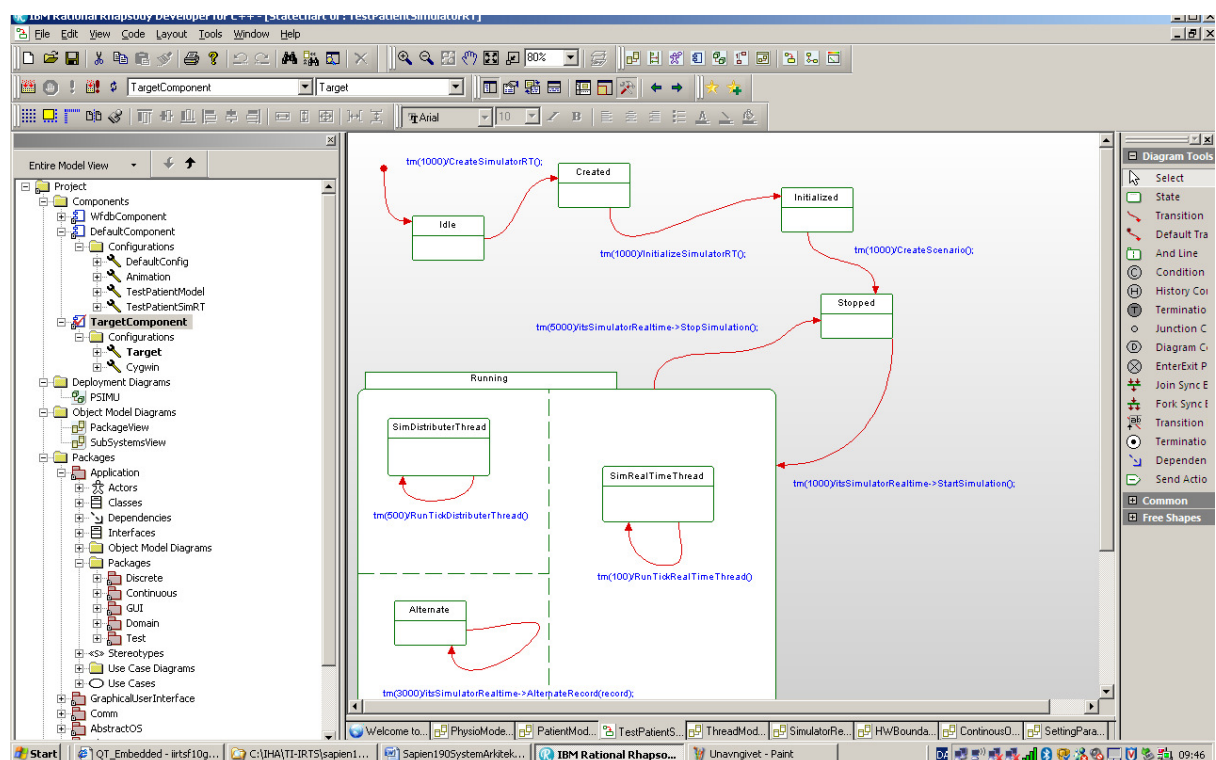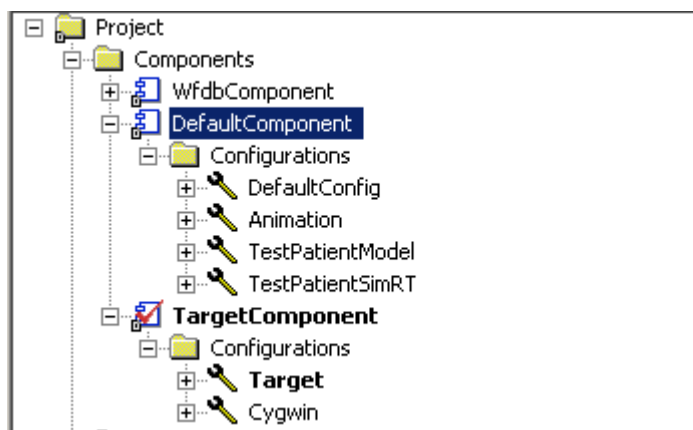

Figure 43 Rhapsody UML model for Sapien 190 used for simulation and test

Figure 44 Rhapsody components for testing and source code generation

The figure above illustrates the components we have created for the Rhapsody project. The DefaultComponent and TargetComponent specifies the **scope** for classes for where source code is generated. Classes used for test of the model are not in scope of the TargetComponent since this component contains classes that are part of the final product. Simulation and test on windows is done by using Cygwin this is done by selecting the DefaultConfig settings to Cygwin and specifying the wfdb headerfiles that we have used for the project:

DefaultConfig settings for windows simulation of DefaultComponent:

Standard Headers: wfdb/wfdb.h,math.h,wfdb/ecgcodes.h
Instumentation Mode: None or Annimation
Settings -> Enviroment: Cygwin

Rhapsody automatically generates the Makefile and compiles the generated source files in this configuration by using cygwin. We have made a configuration for each test scenario we have done in Rhapsody. (TestPatientModel, TestPatientSimRT)

TargetComponent is settings for generating source code for Linux and target:

Directory: C:\Ubuntu_share\sapien190\source\Sandbox\sapine_v1\rpy
Settings -> Enviroment: Enviroment

Here we have configured Rhasody to generate the C++ source code directly to a sub directory of the Qt project that is compiled on Linux. We have shared a directory between the Windows and Linux platforms. The generated source classes is included in the Qt make project by using qmake.

12.2 Linux host Compilation-software

On the host Linux computer Qt and Eclipse must be installed. Qt has been used for the final product and Eclipse to test and develop parts of the Sapien software like hardware access and reading of WFDB records.

We have used the open source Qt Creator version 1.3.1 and the Qt SDK 2010.02 to be downloaded and installed from:

http://qt.nokia.com/downloads

We have used the open source Eclipse Galileo release to be downloaded and installed from:

http://www.eclipse.org/cdt/downloads.php

12.3 Linux Cross Compilation and linking process

The guide "Getting started with Qt" [9] describes how to install Qt on the Linux host which must be performed prior to the below steps. The following chapters describe how to cross compile the needed libraries for Qt and the WFDB library for the target Devkit8000 platform. For touch screen support see "Getting started with Qt".

12.3.1 Qt Cross Compilation with qt-everywhere

This chapter describes how to download the embedded version of Qt that does not use the Linux X11 graphic library. It is therefore a suitable compact cross platform framework for the simulator platform where Devkit8000 is used for the first version of the product.

The following process describes how to configure Qt everywhere for compiling and linking a given program. Include a specification of special compiler and link switches.

This section is about how to modify qt-everywhere for cross compilation to target.

```
#
# qmake configuration for building for ARMv7 devices with arm-none-linux-gnueabi-g++
#

include(../../common/g++.conf)
include(../../common/linux.conf)
include(../../common/qws.conf)

# modifications to g++.conf
QMAKE_CC  = /opt/CodeSourcery/Sourcery_G++_Lite_2007q3/bin/arm-none-linux-gnueabi-gcc
QMAKE_CXX  = /opt/CodeSourcery/Sourcery_G++_Lite_2007q3/bin/arm-none-linux-gnueabi-g++
QMAKE_LINK  = /opt/CodeSourcery/Sourcery_G++_Lite_2007q3/bin/arm-none-linux-gnueabi-g++
QMAKE_LINK_SHLIB  = /opt/CodeSourcery/Sourcery_G++_Lite_2007q3/bin/arm-none-linux-gnueabi-g++
QMAKE_CFLAGS += -O3 -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp
QMAKE_CXXFLAGS+= -O3 -march=armv7-a -mtune=cortex-a8 -mfpu=neon -mfloat-abi=softfp
#-mfpu=vfp


# modifications to linux.conf
QMAKE_AR  = /opt/CodeSourcery/Sourcery_G++_Lite_2007q3/bin/arm-none-linux-gnueabi-ar cqs
QMAKE_OBJCOPY = /opt/CodeSourcery/Sourcery_G++_Lite_2007q3/bin/arm-none-linux-gnueabi-objcopy
QMAKE_STRIP = /opt/CodeSourcery/Sourcery_G++_Lite_2007q3/bin/arm-none-linux-gnueabi-strip
```

QMAKE_INCDIR += /home/stud/tslib_arm/include
QMAKE_LIBDIR += /home/stud/tslib_arm/lib

load(qt_config)

Configure the libraries to use the linux-armv7-g++ configuration:

$ cd /home/stud/qt-everywhere-opensource-src-4.6.2
$ ./configure -embedded arm -xplatform qws/linux-armv7-g++ -qt-kbd-linuxinput -qt-mouse- tslib -opensource -verbose -R /home/stud/tslib_arm/lib/

Make the libraries (Grab ...something, this takes several hours):
$ make

and install them:
$ make install

WFDB Installation and Cross Compilation

This chapter describes how to install the WFDB library from physionet.org and how to modify it for cross compilation to the ARM target running Linux. The WFDB library is an open source project used to read and manipulate patient records from the PhysioBank database. The host Linux platform must have installed the "`arm-none-linux-gnueabi-gcc`" prior to this installation.

First download the WFDB library from here:

http://www.physionet.org/physiotools/wfdb.shtml#downloading

Steps to install WFDB on the Linux host:

1. Install XView for Linux
$ sudo apt-get install xviewg xviewg-dev
2. Unpack the downloaded wfdb.tar.gz
/home/stud$ tar –xvf wfdb.tar.gz
3. Run configure in the wfdb directory
/home/stud/wfdb-10.5.1$ ./configure
4. Build and make the wfdb library
/home/stud/wfdb-10.5.1$ make
5. Install the wfdb libray on Linux host
/home/stud/wfdb-10.5.1$ sudo make install

Steps for cross compile of the WFDB library to target:

This part describes how create the libwfdb.so.10.5 library that must be copied to the target platform. Prior to this step the WFDB library must be installed on the Linux host see description above.

1. The first step is to copy the source library files to a new directory. Copy the lib directory from the wfdb-10.5.1 installation to a new directory named lib-arm.

```
/home/stud/wfdb-10.5.1/lib
   Copy to
/home/stud/wfdb-10.5.1/lib-arm
```

2. Edit the Makefile manual in the new lib-arm directory, change the following lines to:

```
SRCDIR = "/home/stud/wfdb-10.5.1"
WFDBROOT = /home/stud/wfdb-arm
CC = arm-none-linux-gnueabi-gcc
BUILDLIB = arm-none-linux-gnueabi-gcc $(MFLAGS) -shared -Wl,-
soname,$(WFDBLIB_SONAME) $(LL) \
-o $(WFDBLIB)
```

These modification will now used the arm cross compiler (arm-none-linux-gnueabi-gcc) and install the WFDB header and library files in /home/stud/wfdb-arm

3. Make, Compile, link and install the wfdb library for arm target
```
/home/stud/wfdb-10.5.1/lib-arm $ make
/home/stud/wfdb-10.5.1/lib-arm $ make install
```

Ignore errors when "make install" is performed this is due to some files it tries to install on the Linux host that is not needed on target.

Steps for using the WFDB library in eclipse:

1. Change the eclipse application project to include wfdb:
/home/stud/wfdb-arm/include

2. Change the eclipse project to include the wfdb library and path:
-l wfdb
/home/stud/wfdb-arm/lib

Steps for using the WFDB library in Qt:

This step describes how to use the qt-everywhwere version of qmake to generate the makefile for cross compilation to target. The following parameters to qmake must be added as described below.

These parameters adds information about the paths for find the wfdb header files and wfdb library. The below line also specifies to include the touch screen library (-lts) for the Devkit8000 platform.

/home/stud/qt-everywhere-opensource-src-4.6.1/bin/qmake LIBS+="-L/home/stud/wfdb-arm/lib -lts -lwfdb" DEFINES+=_LINUX DEFINES+=_USE_HW_DAC INCPATH+=/home/stud/wfdb-arm/include

If the _USE_HW_DAC is not specified a version will be compiled that just prints out record samples on the standard output. (Terminal output)

13. INSTALLATION AND EXECUTING

This section describes how to install the sapien190 software on the DevKit8000 target running Linux. We will describe how to install the needed libraries and the application program on target.

13.1 Installation

The sapien190 executable can be copied to the target by copy of the cross compiled binary executable program to the SD card or using Ethernet over an USB connection to the target.

Ethernet over USB to target

Configure Ethernet over USB after the USB wire is connected between host and target powered on with the target IP address of 10.9.8.2:

First configure the IP address of the Linux host:

$ sudo ifconfig usb0 10.9.8.1/24 up

Use secure copy of binary cross compiled application to target in the default directory /home/root:

$ scp sapien190 root@10.9.8.2:

Copy of files using SD card:

Copy the sapien190 binary file to the SD card an insert it on the target.
The SD card will automatic be mounted on the target at:

/media/mmcblk0p1/

13.2 Executing-hardware

To run sapien190 on the host development platform Qt and the WFDB library must be installed to start sapien190 enter the following command in Linux:

$./sapien190

To run Sapien190 on the target after installed:

$./sapien190 –qws

The driver for the Add-on board must be installed according to the description found here: http://devkit8000addon.wikispaces.com/

The Add-on board contains the following features:

- 8-Channel 12-bit A/D Converter with build-in programmable PGA

- 2-channel 12-bit D/A Converter

- 16 GPIO ports via the I2C Interface

- 8 GPIO ports directly mapped to the OMAP processor

- CPLD for future VHDL programs

- Flexible clock generator for the CPLD

- Additional LEDs for debugging

- 3 RS-232 Interfaces including

13.3 Executing-software

This chapter describes the need libraries to be installed on target to run the Sapien190 patient simulator.

Installing Qt Everywhere on target by moving the cross compiled libraries and some fonts to the DevKit8000 target.

Fonts: At least one font must be present on target to show text. Fonts can be found in `$HOME/qteverywhere-opensource-src-4.6.1/lib/fonts` on the host and must be copied to `/usr/local/Trolltech/QtEmbedded-4.6.1-arm/lib/fonts/` on target (create the directories as necessary).

(At least) the below libraries must be copied from `$HOME/qt-everywhere-opensourcesrc-4.6.1/lib/` on host to `/usr/lib` on target:
- **libQtCore.so.4**
- **libQtGui.so.4**
- **libQtNetwork.so.4**

The wfdb library must be copied from host `/home/stud/wfdb-arm/lib/` on host to `/usr/lib` on target:
- **libwfdb.so.10**

The standard C++ library must be copied from host `/opt/CodeSourcery/Sourcery_G++_Lite_2007q3/arm-none-linux-gnueabi/lib/` to `/usr/lib` on target:
- **libstdc++.so.6**

Install and enable the touch screen on Devkit8000 by following the steps in "Getting started with Qt" [9] and install the touch screen library as described below:

Copy the touch screen libraries from `/home/stud/tslib_arm/lib/` to `/usr/lib` on target and enable the touch on the DevKit8000:
- **libts-0.0.so.0**
- **libts.so**

Enable touch screen on DevKit8000:

```
$ chmod a+rw /dev/input/event2
$ export QWS_MOUSE_PROTO=Tslib:/dev/input/touchscreen0
```

Add the environment variable QWS_MOUSE_PROTO in the startup profile listed below.

```
/etc/profile
```
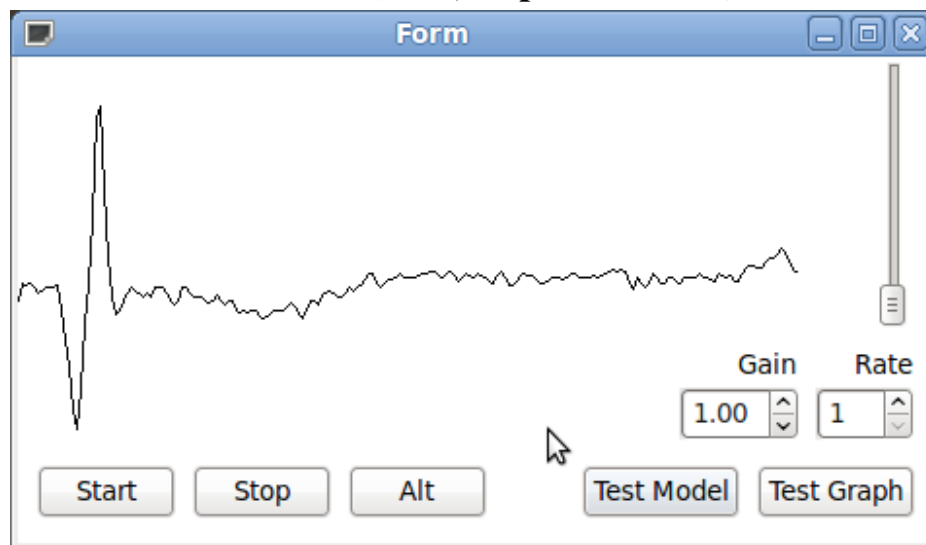
13.4 Execution-control (start, stop and restart)

Figure 45 Sapien190 test version

The first version of the Sapien190 patient monitor is only able to start and stop playing a specific patient ECG record e0104 alternating with patient ECG record e0103. When test model or test graph is selected the contents of patient record files is printed on the display independent of the DAC signal outputs. The final prototype of the Sapien190 patient monitor is shown below. In this version the graph is updated simultaneously with output of the ECG signal on the analogue output. It is possible to alternate between patient records e0103 and e0104, when the simulation is running and setting gain and rate of replaying the patient record. Pause and resume buttons will temporary stop the simulation and continue from the point it was stopped.
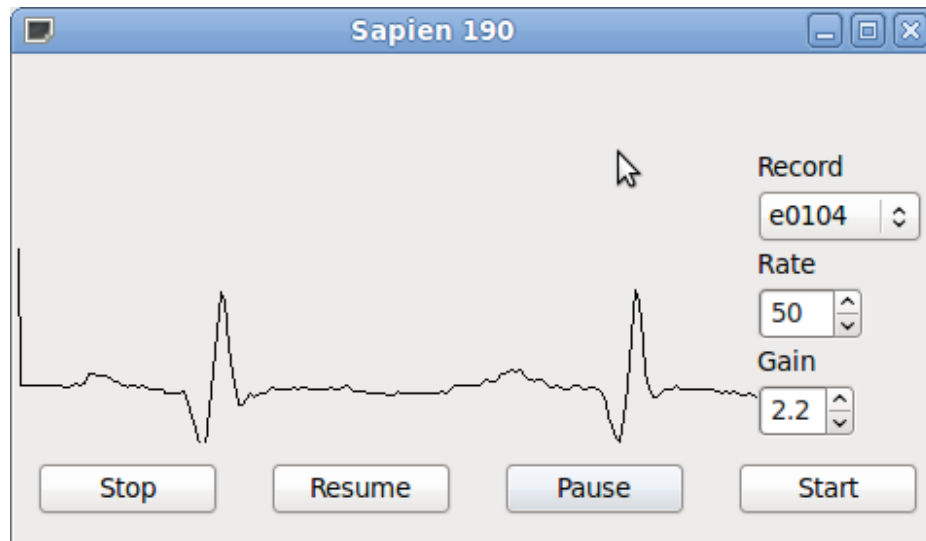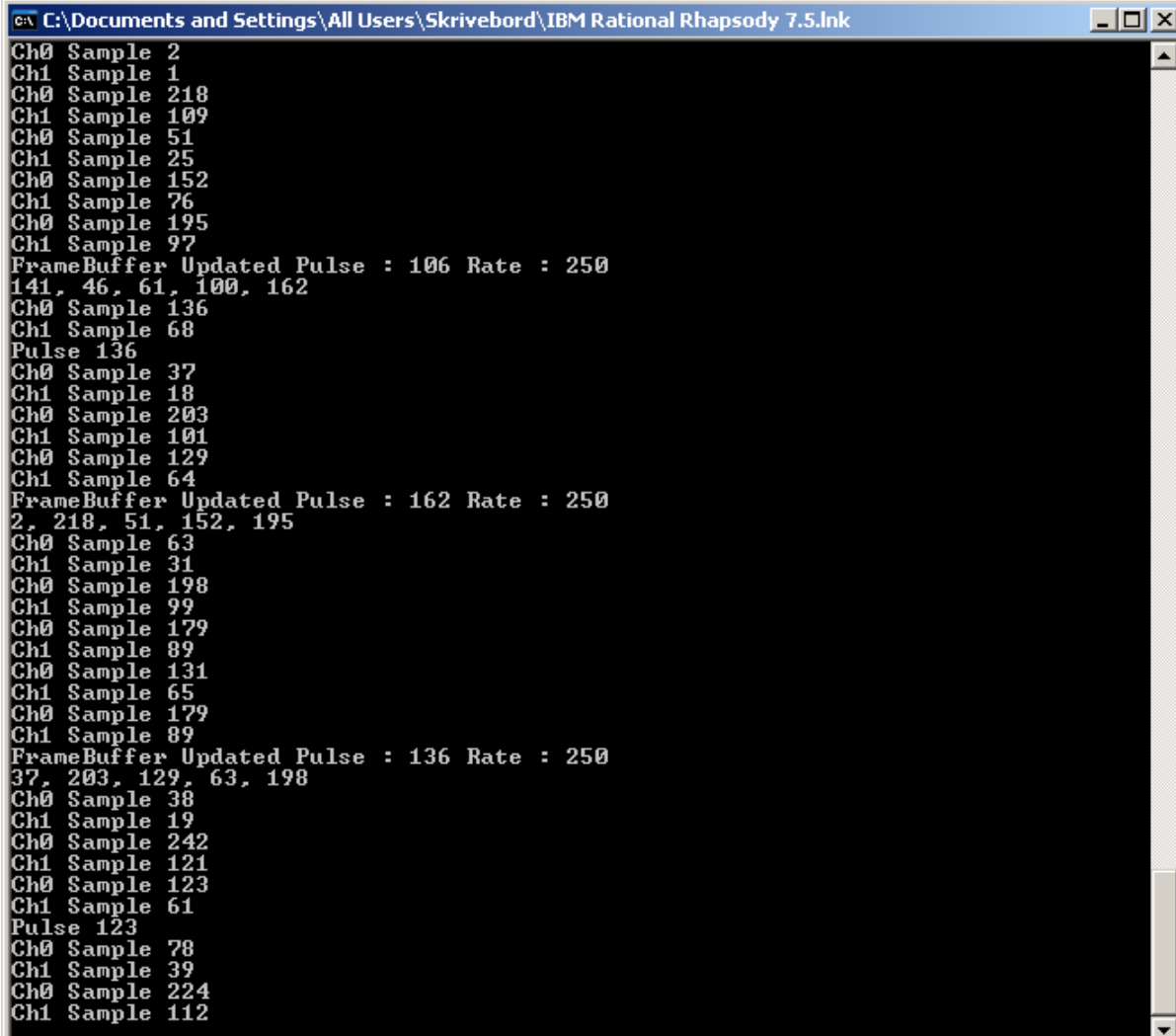
Figure 46 Sapien 190 Final prototype

13.5 Error messages

The current version will be displaying no error messages if as an example the patient record could not be found. The simulator will in this case just output zero voltages output for ECG and EDR signals.
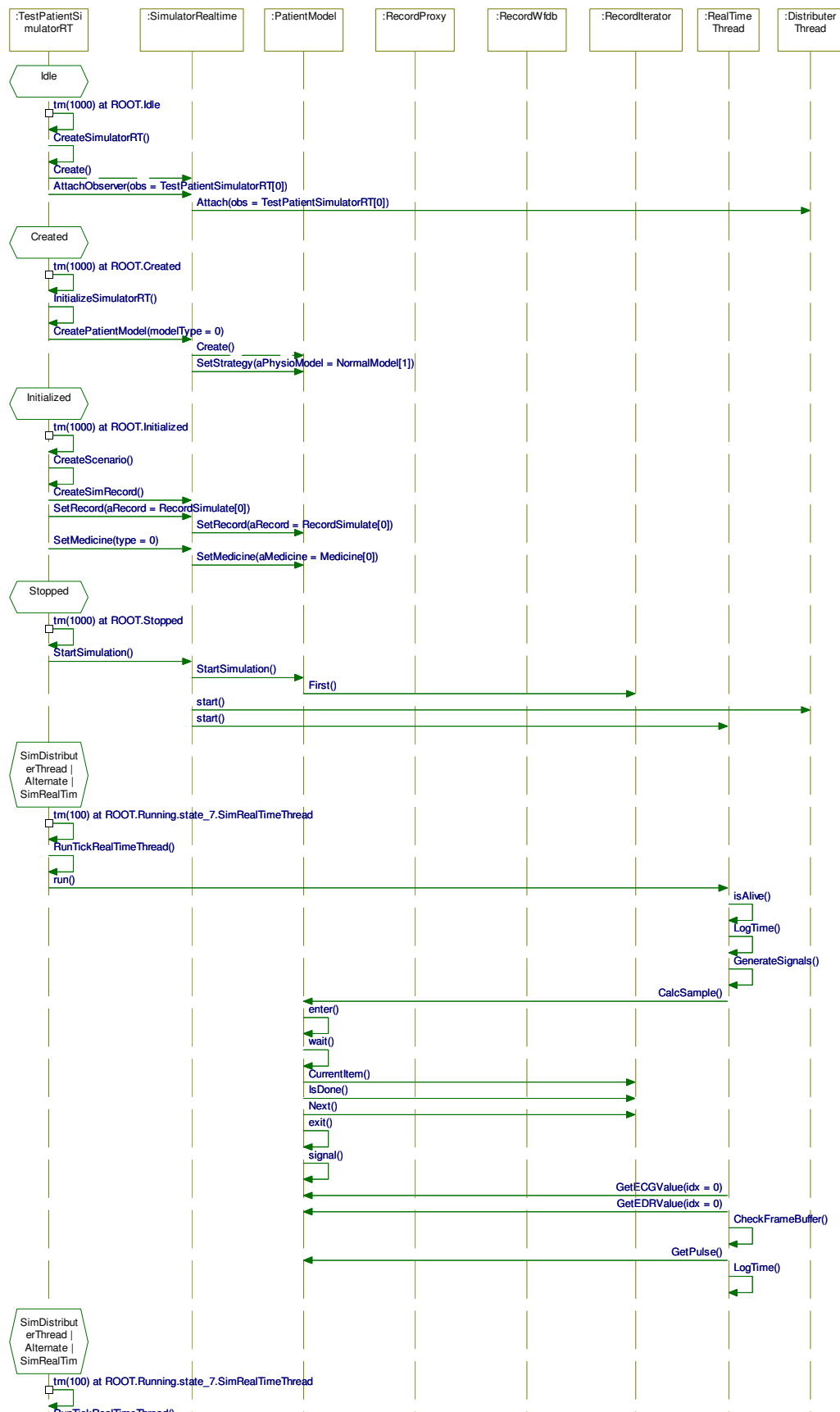
14. APPENDICES

Test: Console output from test in Rhapsody

Figure 47 Animated Rhapsody of testing real-time simulator