Project: Sapien 190
Date: 22.05.2010
Version: 0.14

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	Document merged with AA and PHM stuff. Ch 5 a.o.	PHM
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

Table of Contents

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

[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

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**.

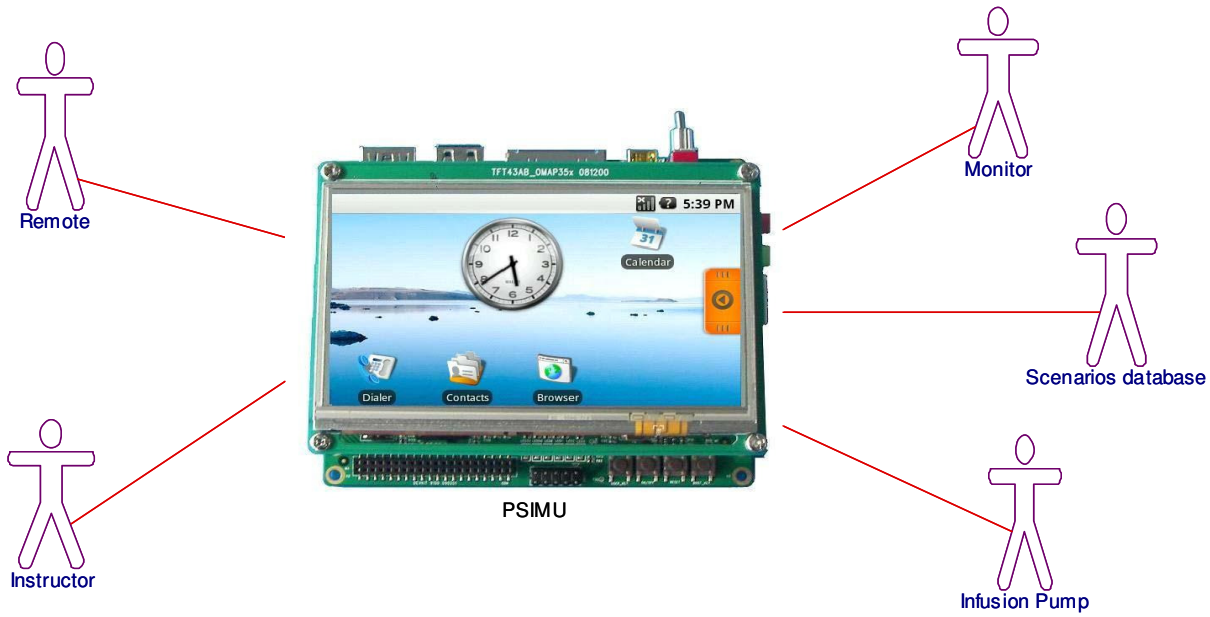
1.5 Document role in an iterative development process

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

WHAT TO WRITE HERE?

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

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 scenarios

Initiate the simulation using a scenario configuration chosen by the instructor.

This use case is not part of the first iteration and will be updated in the next iteration.

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

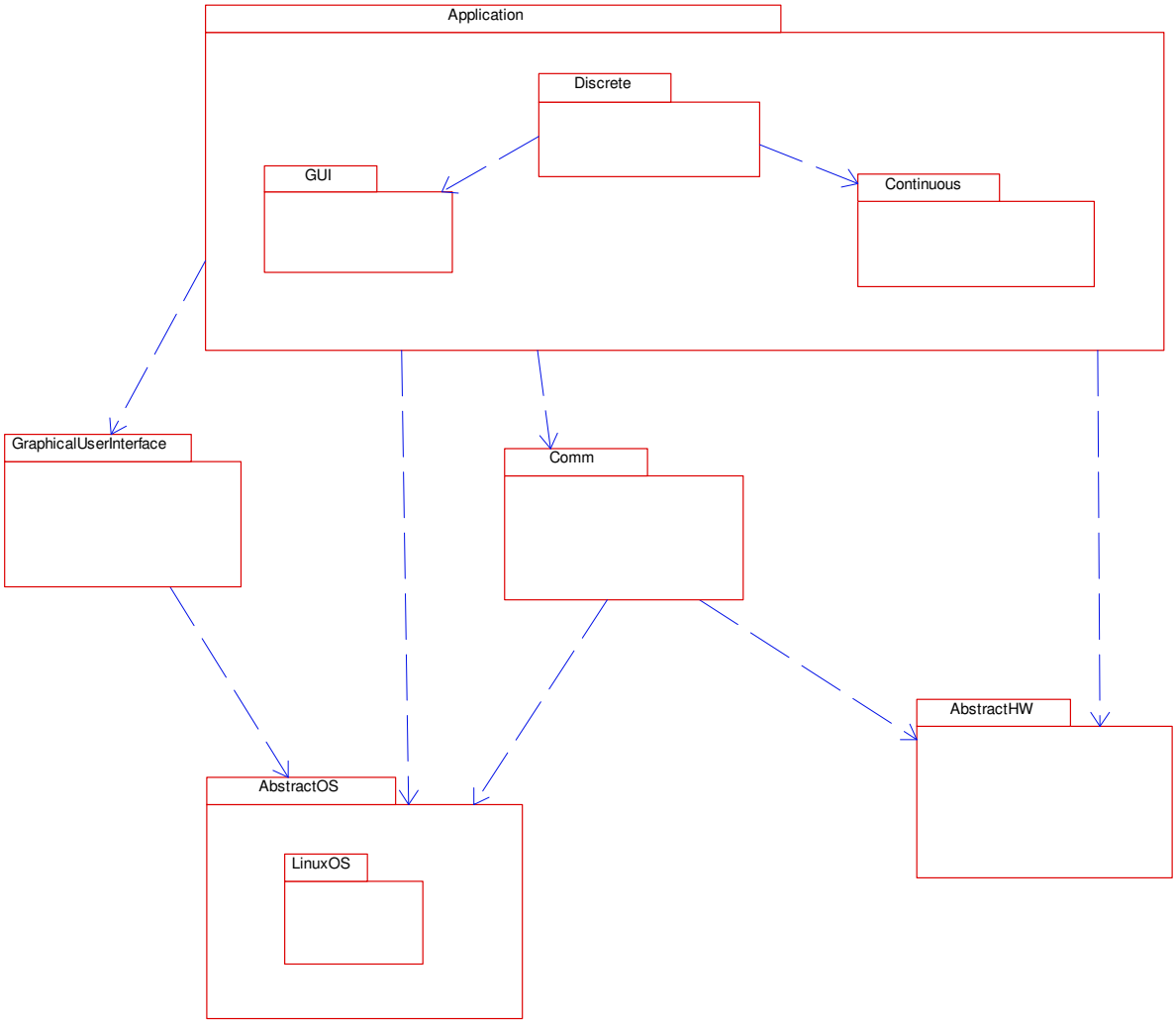


Figure 1 Five layered architecture for logical view

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 again simplifies the process of changing our design or adding new ex. hardware to 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.

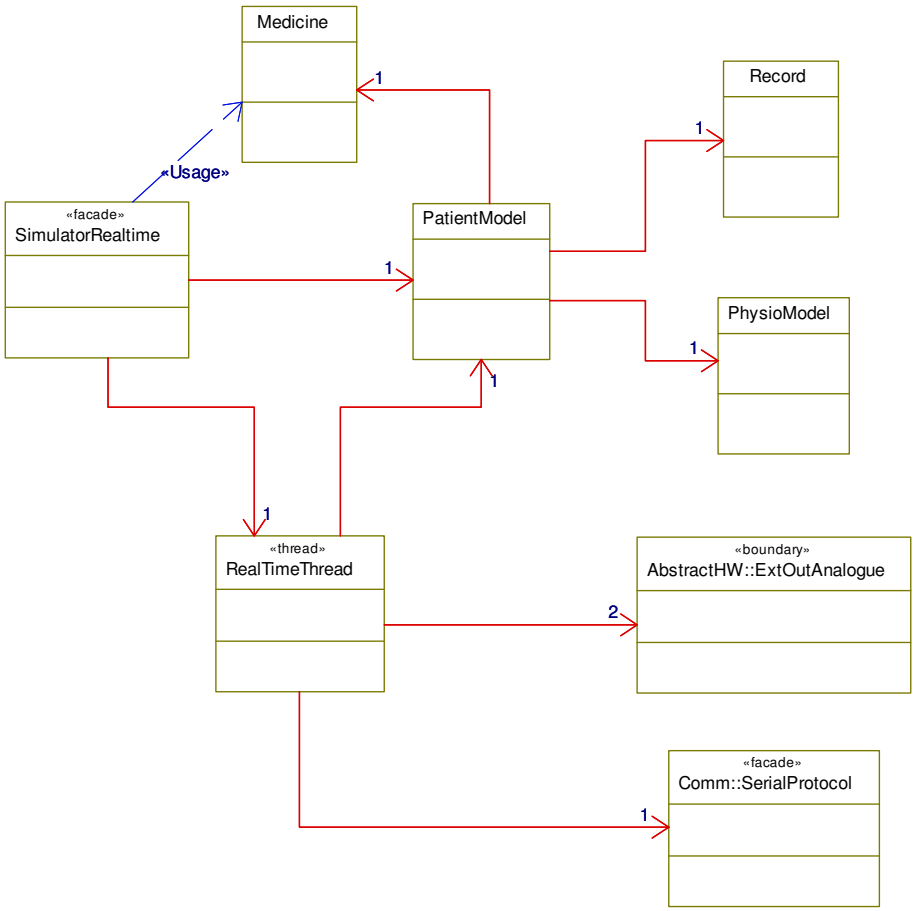


Figure 2 Major Classes in Continuous Package

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.

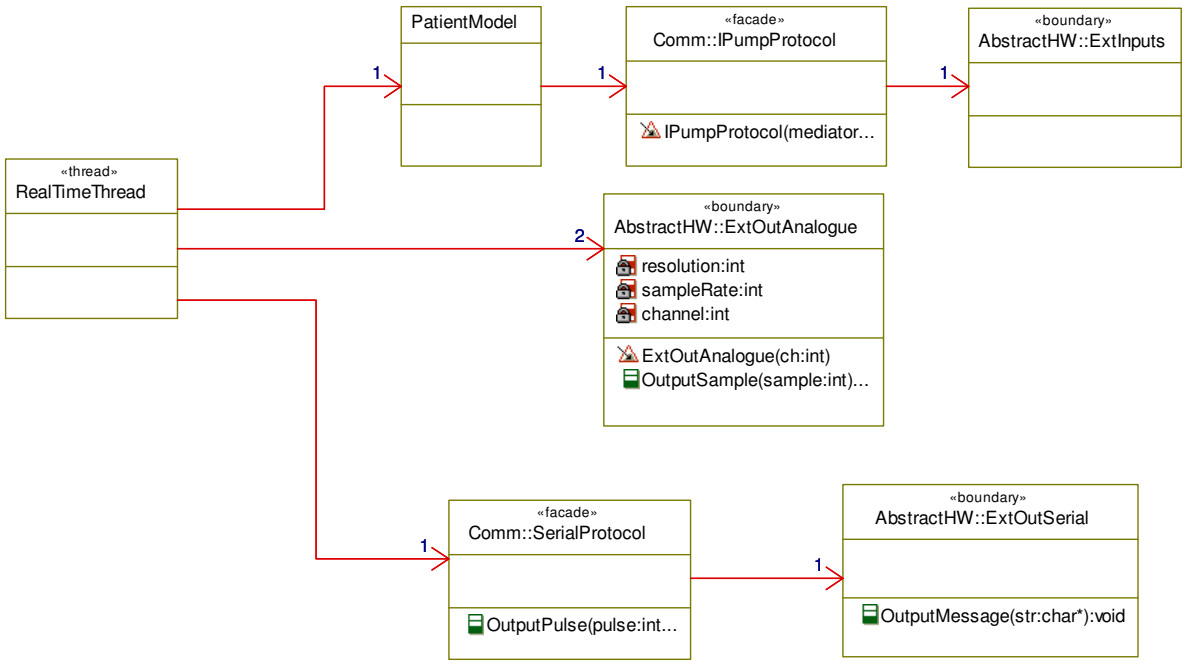


Figure 3 Boundary Classes for RealTimeThread

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* .

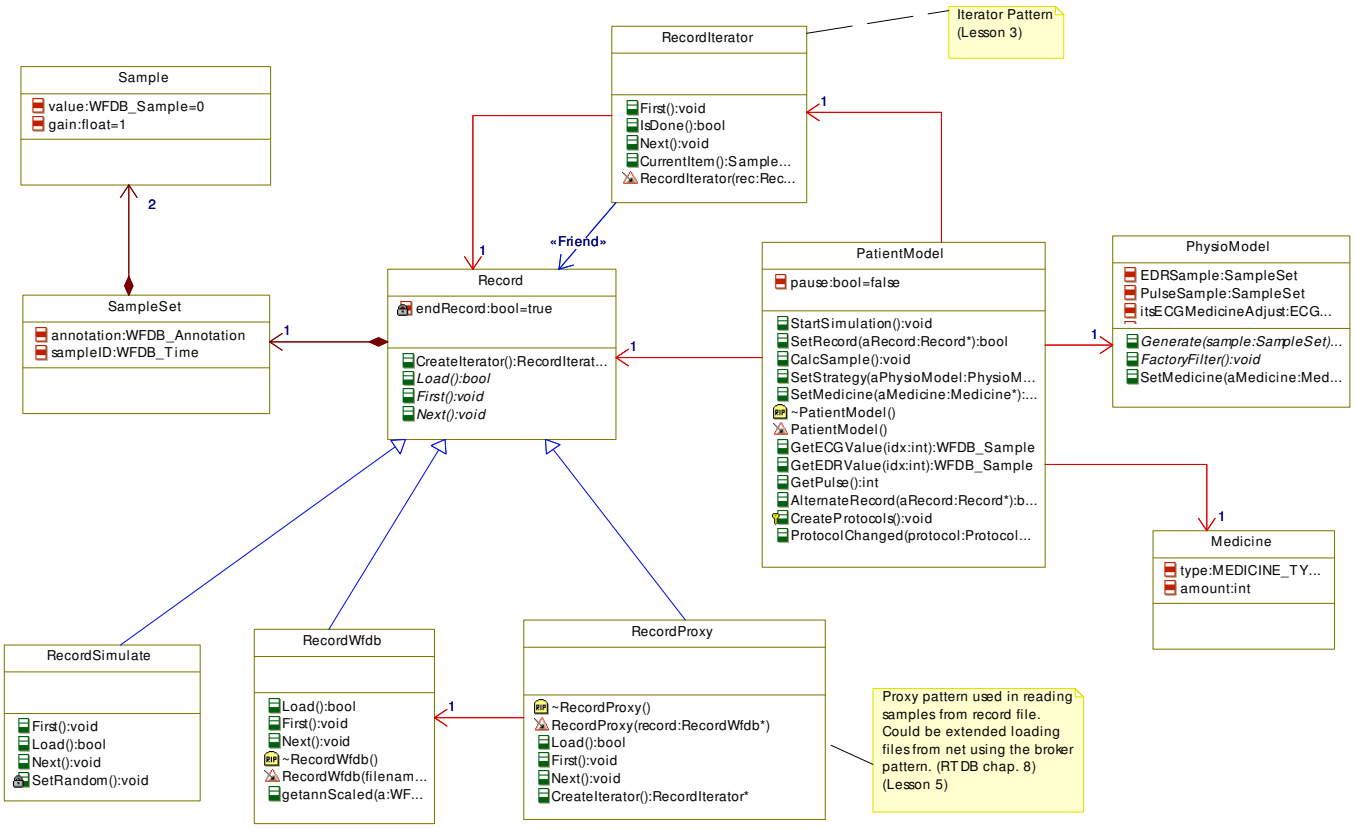


Figure 4 Proxy and Iterator Pattern used to access Records from the PatientModel

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] GOF page xx (Mediator motivation example)

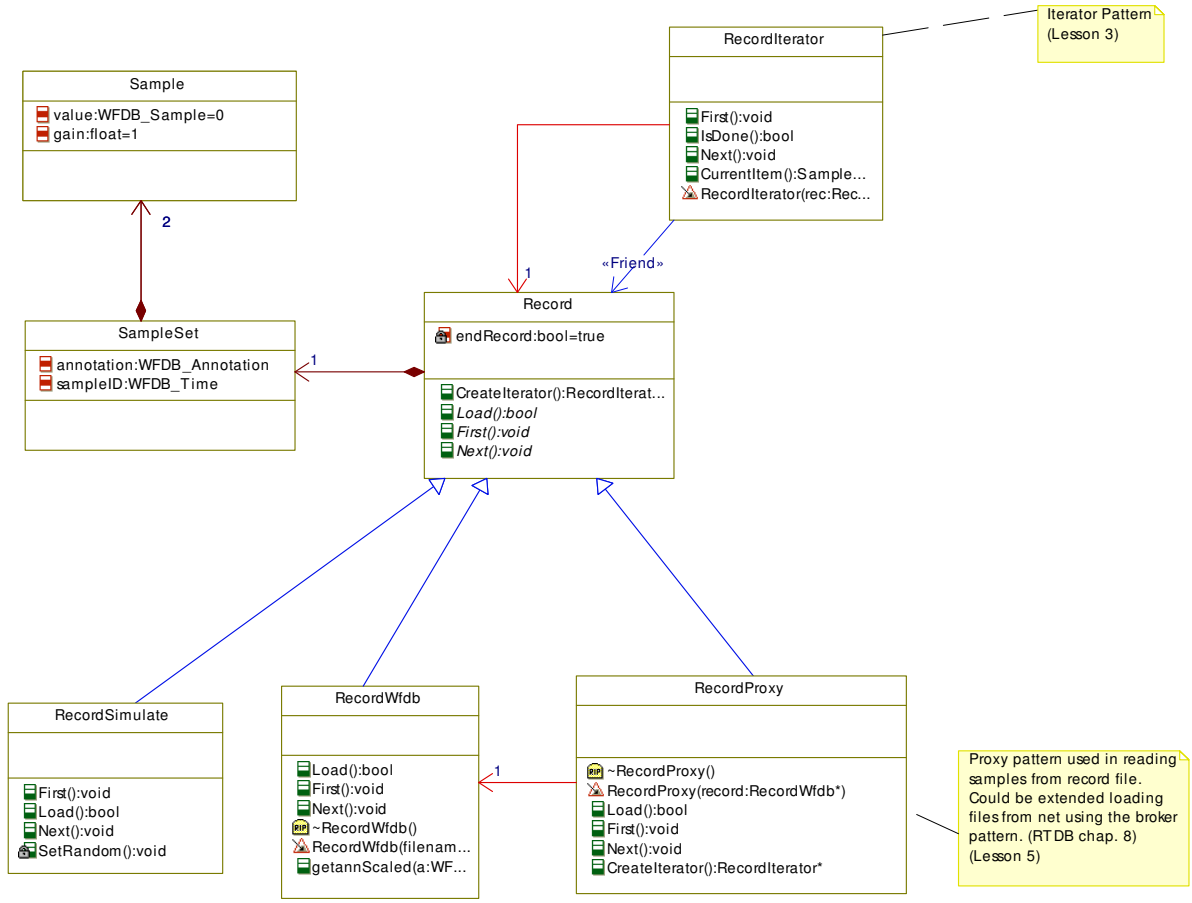


Figure 5 Record and Iterator

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.

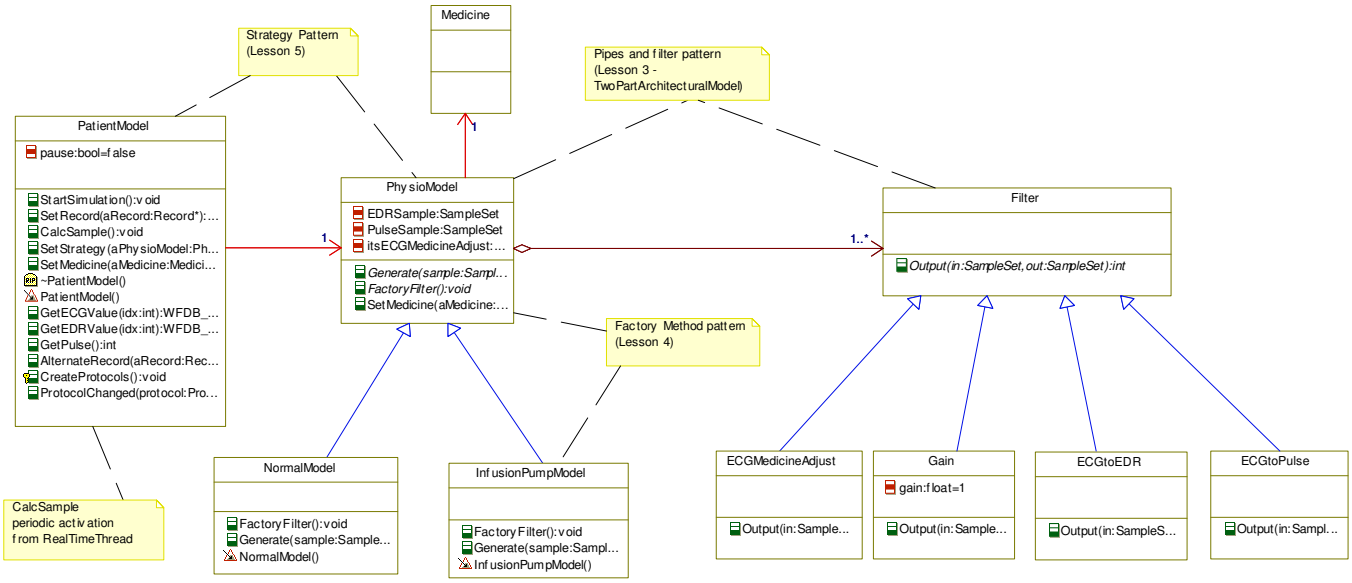


Figure 6 Strategy, Filter and Pipes Pattern used for PhysioModel

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.

.

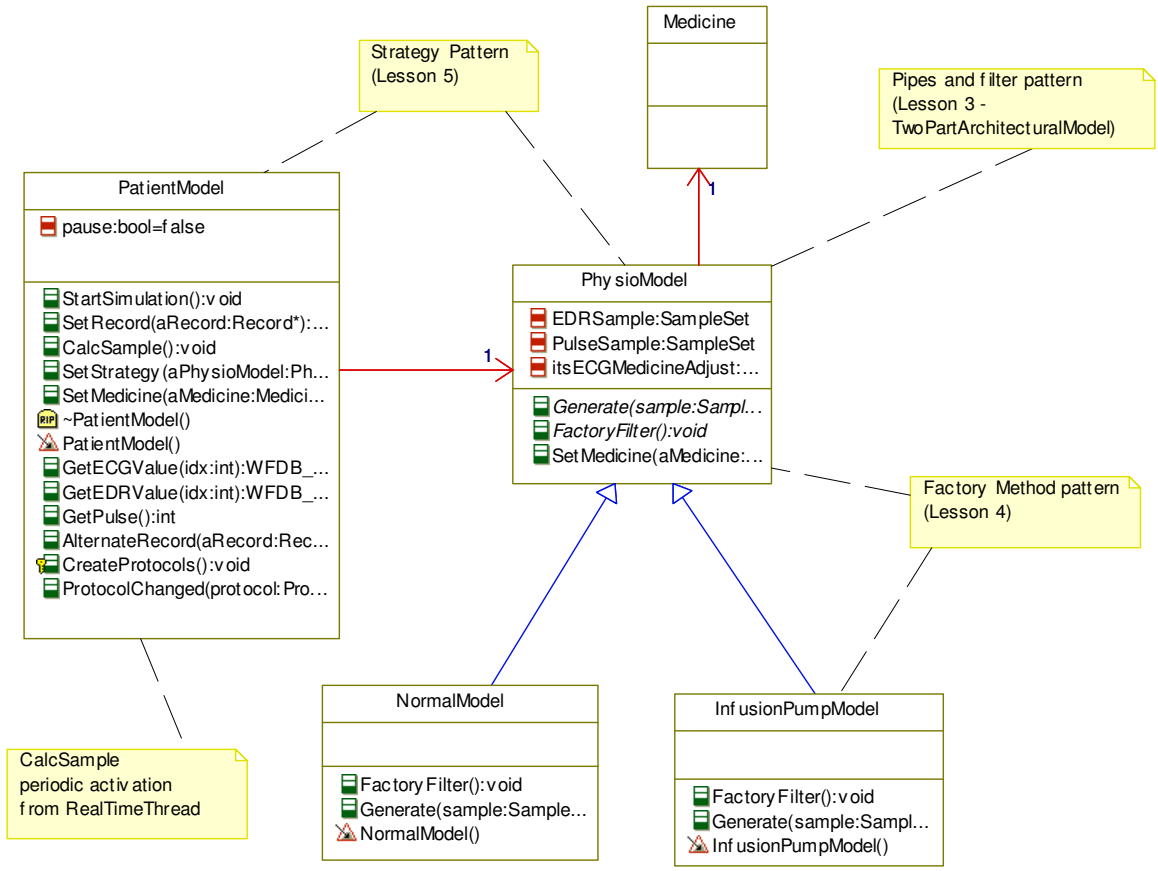


Figure 7 Strategy for PatientModel

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.**

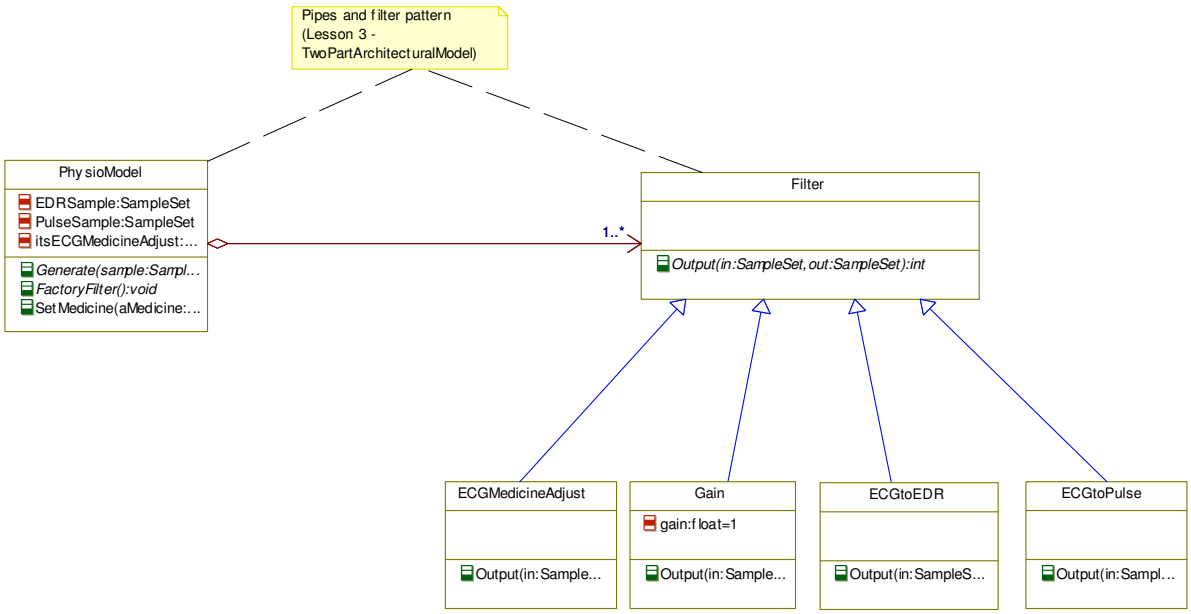


Figure 8 Filter Pattern for PhysioModel

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.

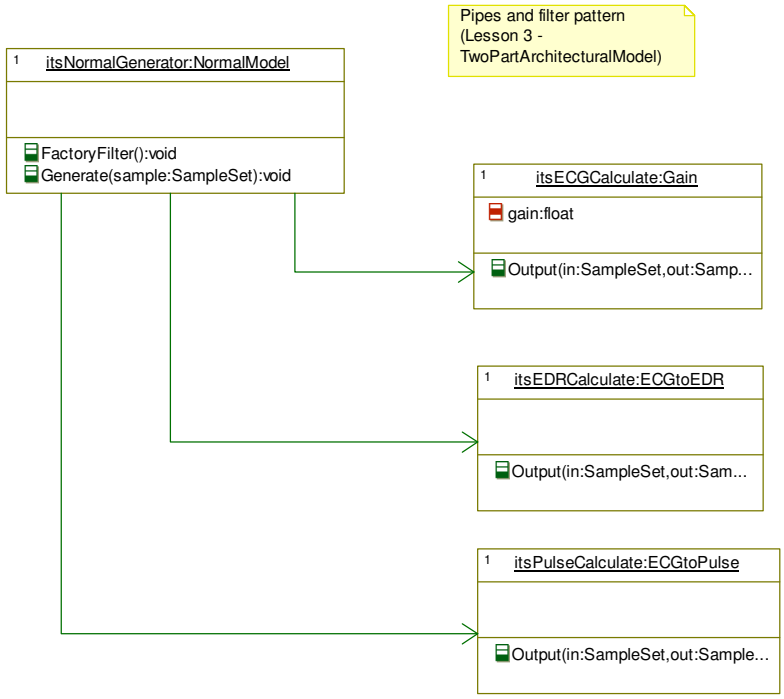


Figure 9 NormalGenerator using filters to generate signals

[3] Designing Event-Controlled Continuous Processing Systems, page 8

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.

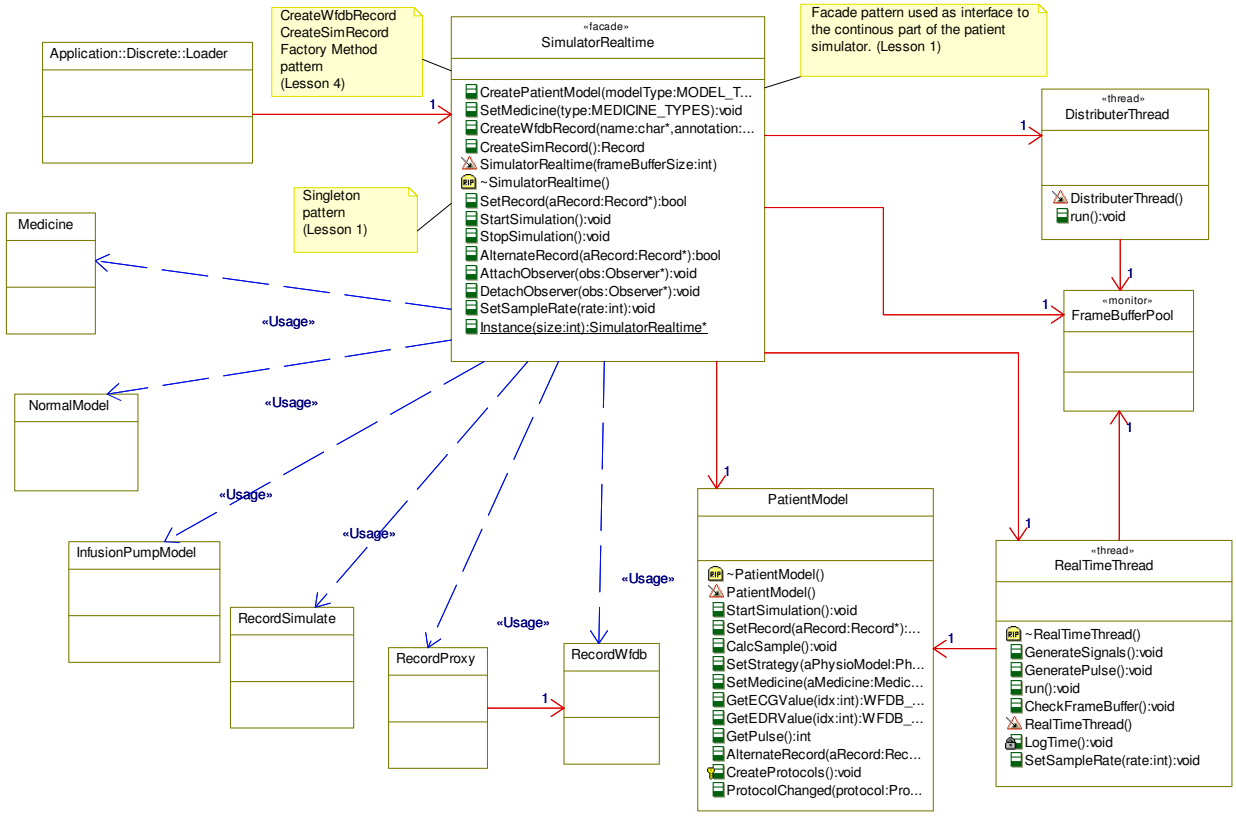


Figure 10 Façade Pattern used for interface to the Continuous Package

5.2.2 Discrete

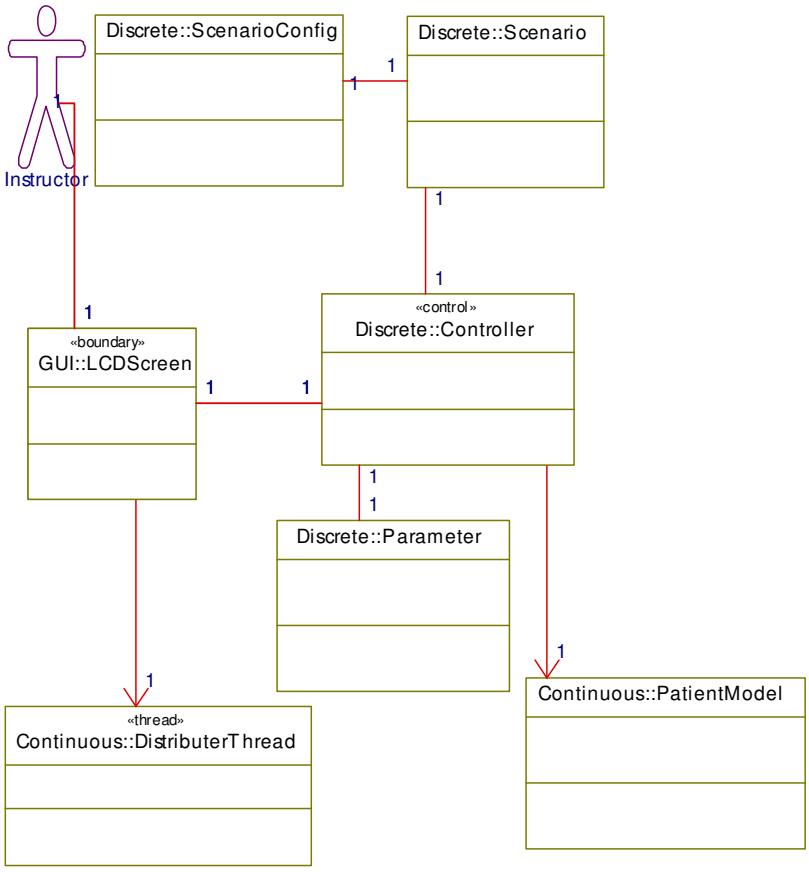


Figure 11 Application model for discrete package

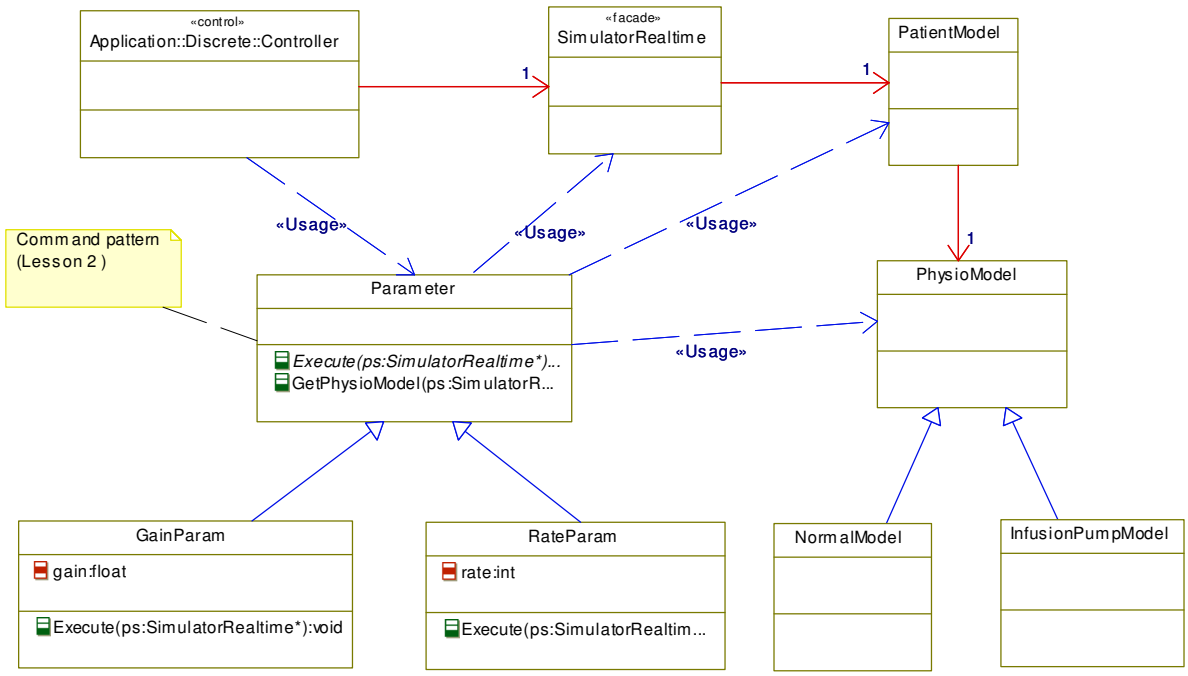


Figure 12 Command pattern used to set parameters in real-time simulator

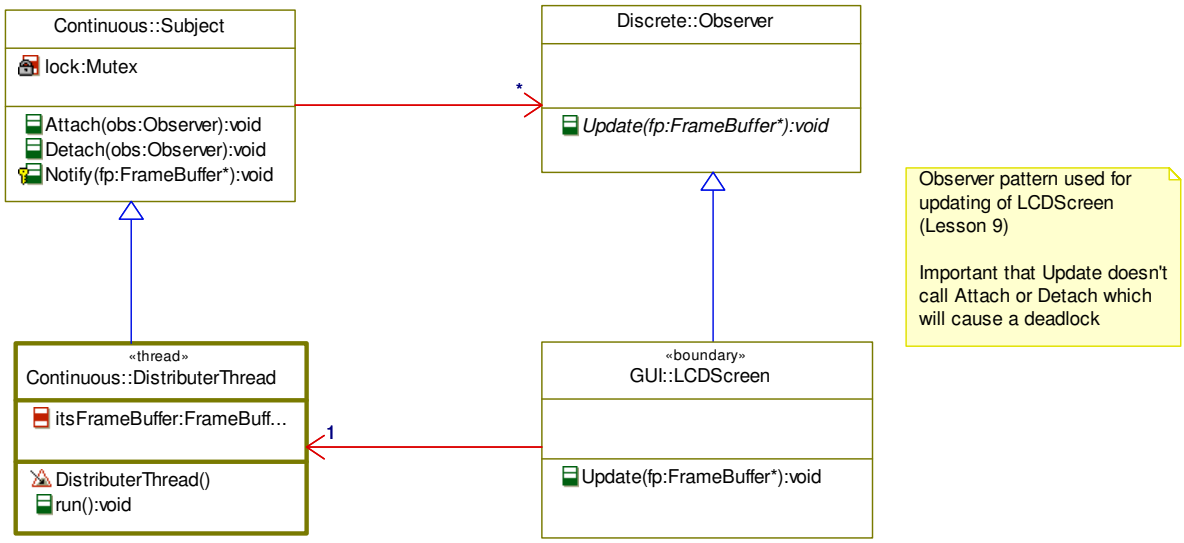


Figure 13 Observer pattern used to notify GUI with new frame buffer

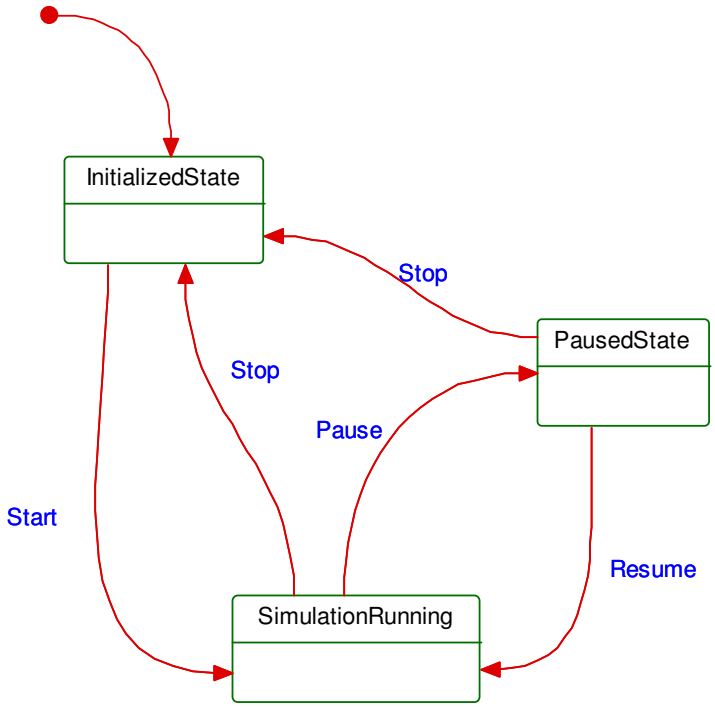


Figure 14 State chart for SapienApplication controller

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.

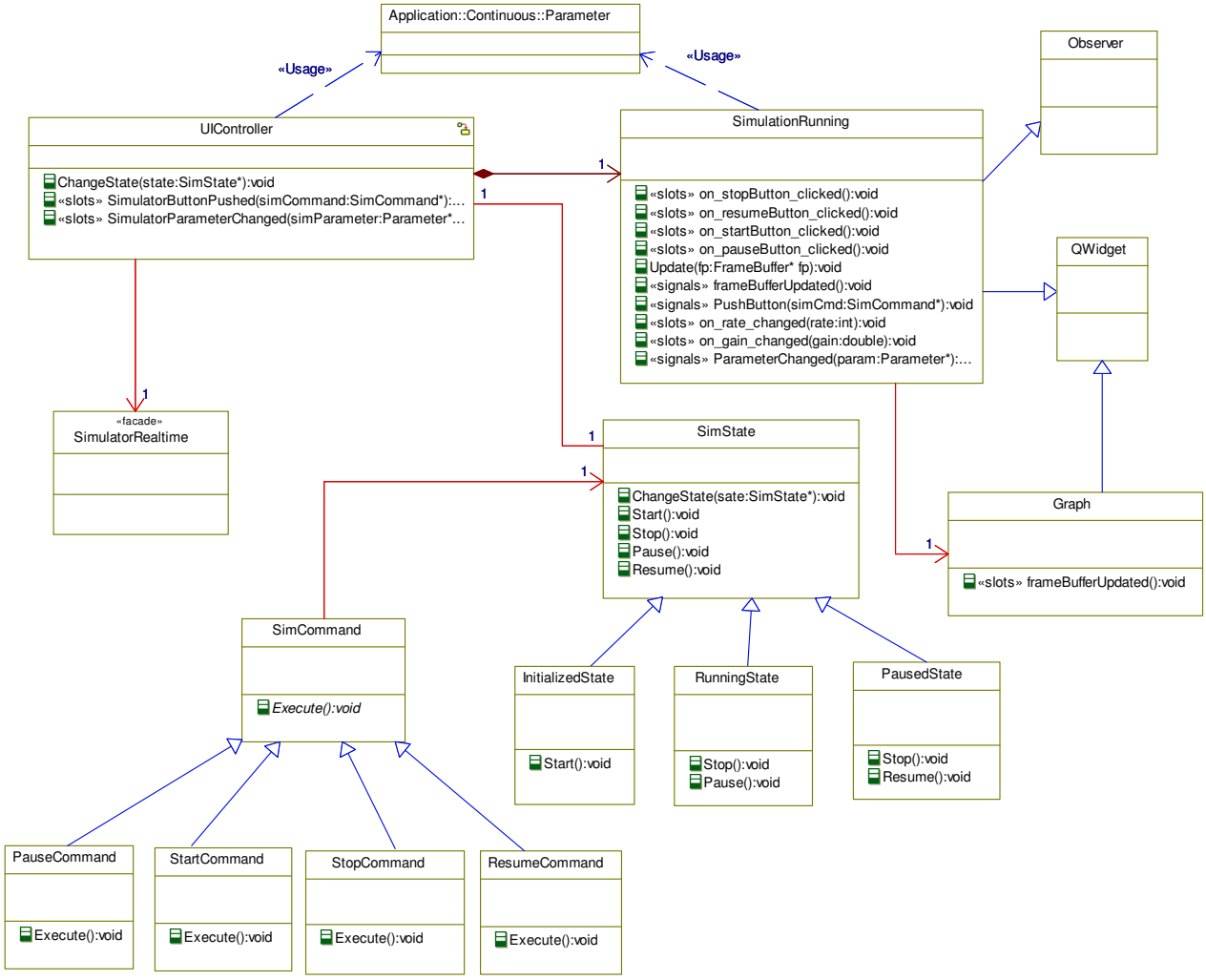


Figure 15 Command, State and Observer pattern used to design SapienApplication (Controller)

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. 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

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*.

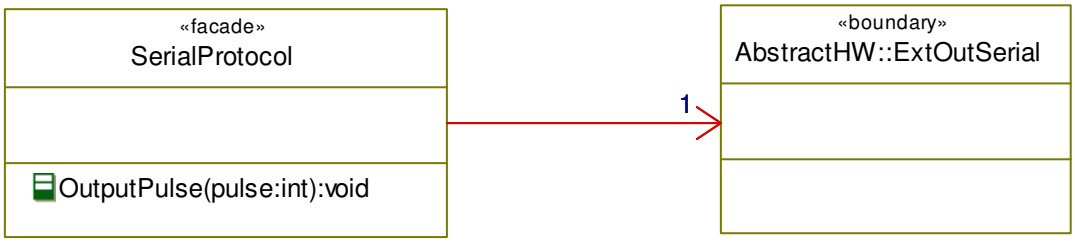


Figure 16 Communication package Serial protocol

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.

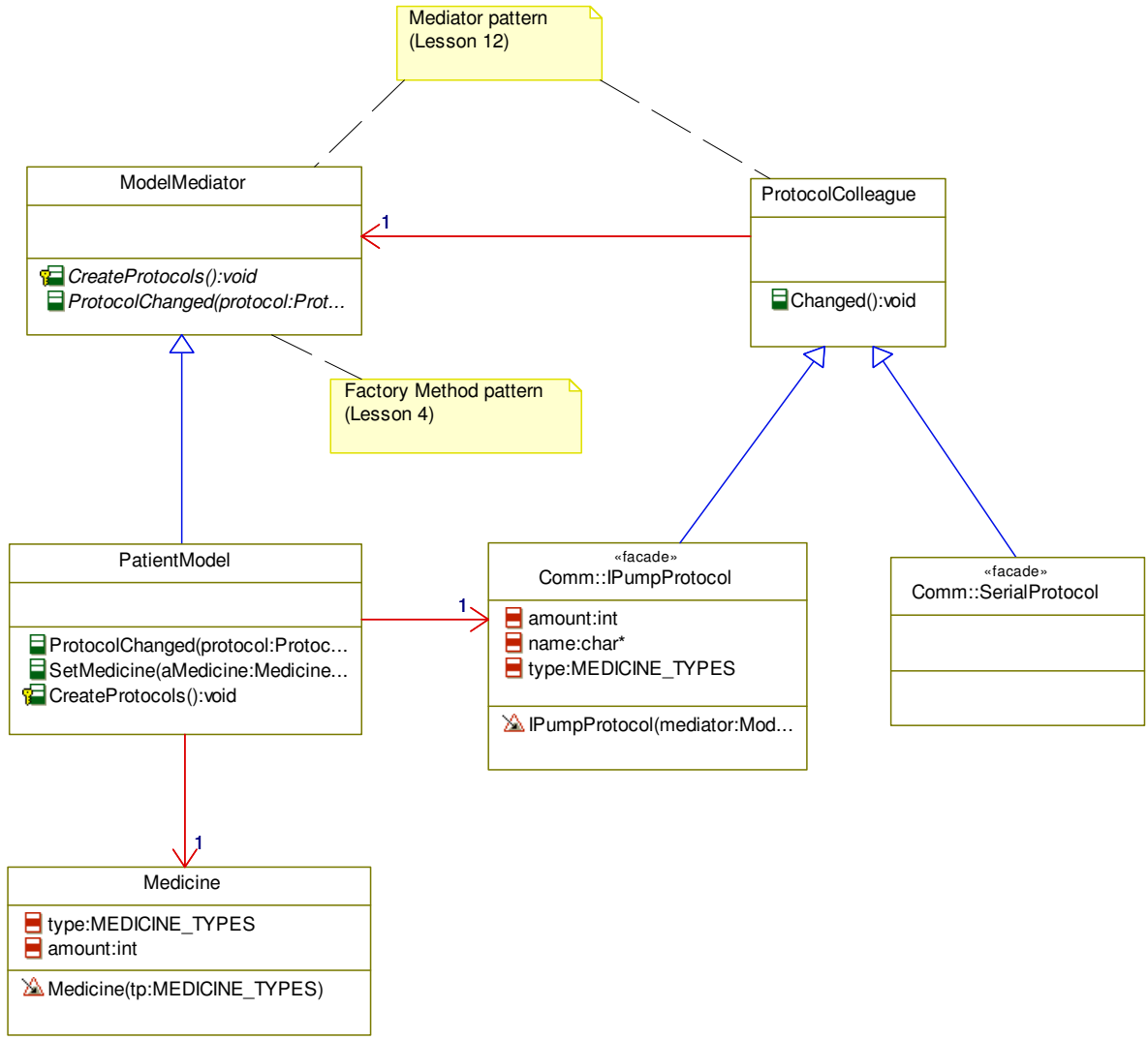


Figure 17 Mediator pattern used to update PatientModel with external input from protocols

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.

5.2.4 AbstractHW

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.

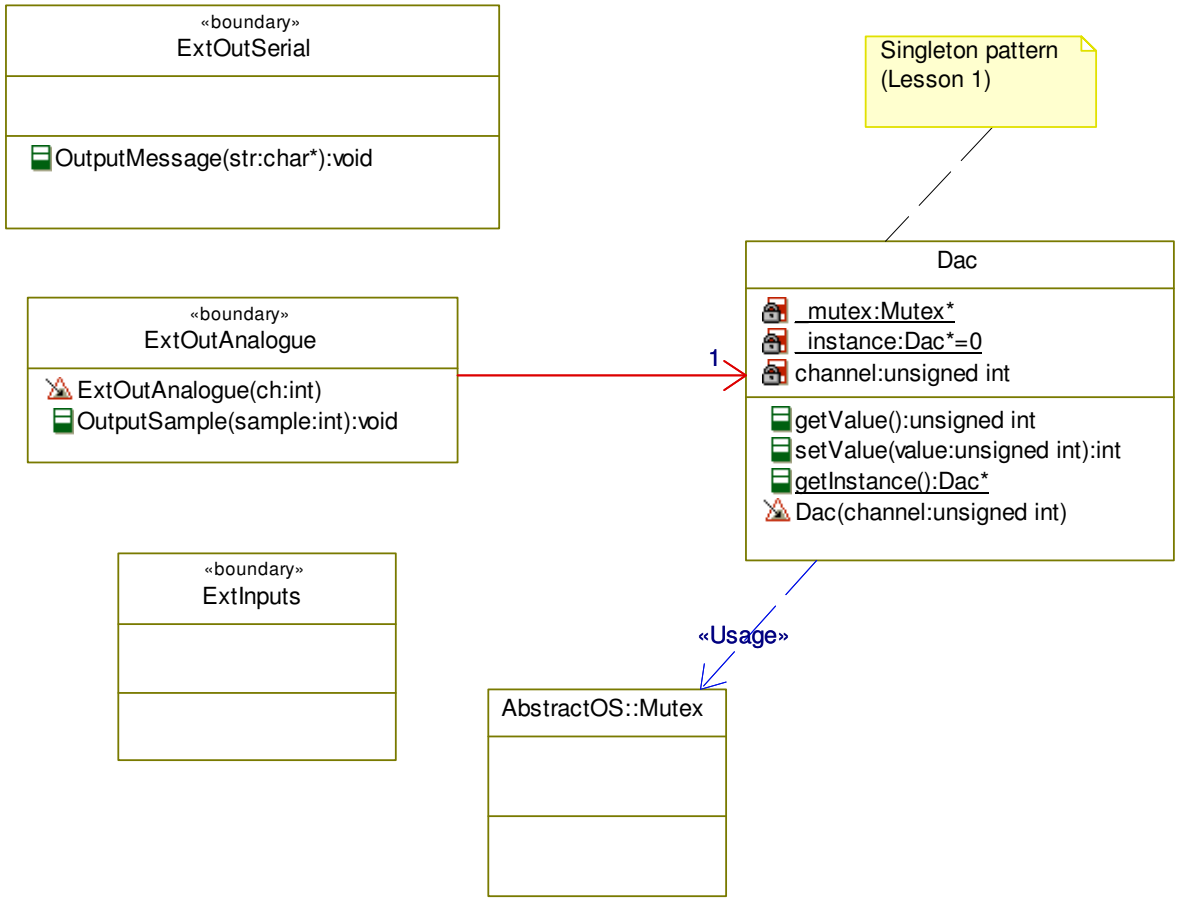


Figure 18 Hardware Abstraction using singletons for Dac abstraction

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

5.2.5 Application Helper Classes

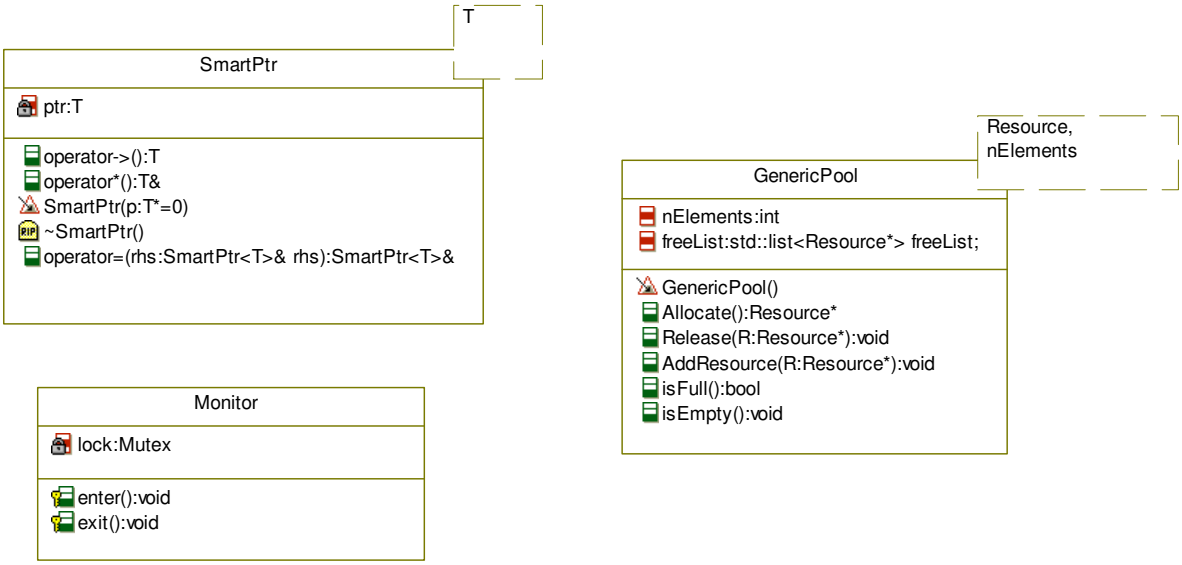


Figure 19 Application Helper classes

5.3 Use case realizations

5.3.1 Use case #1: Execute and Control Simulation scenarios

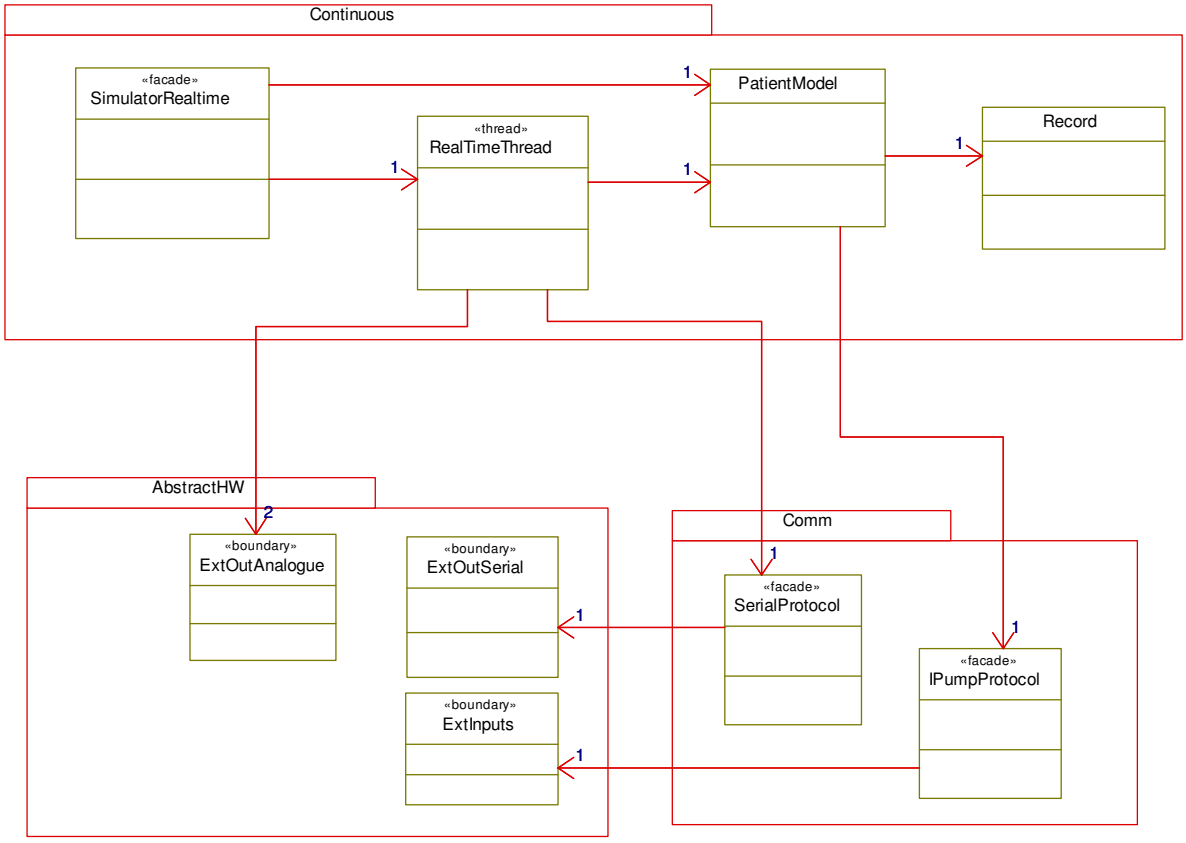


Figure 20 Logical view for use case #1 Execute and Control Simulation

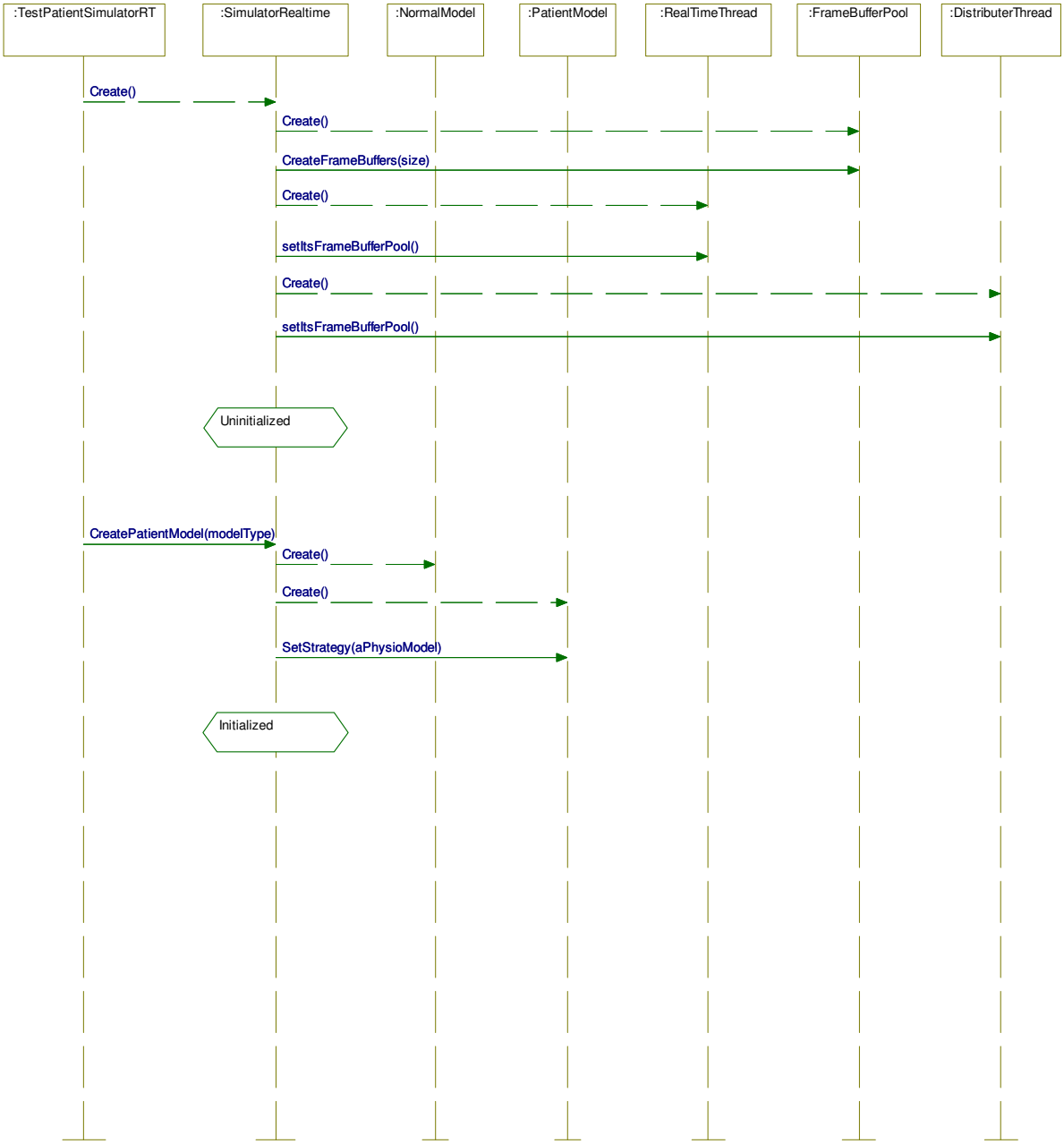


Figure 21 Sequence Diagram: Create Realtime Simulator

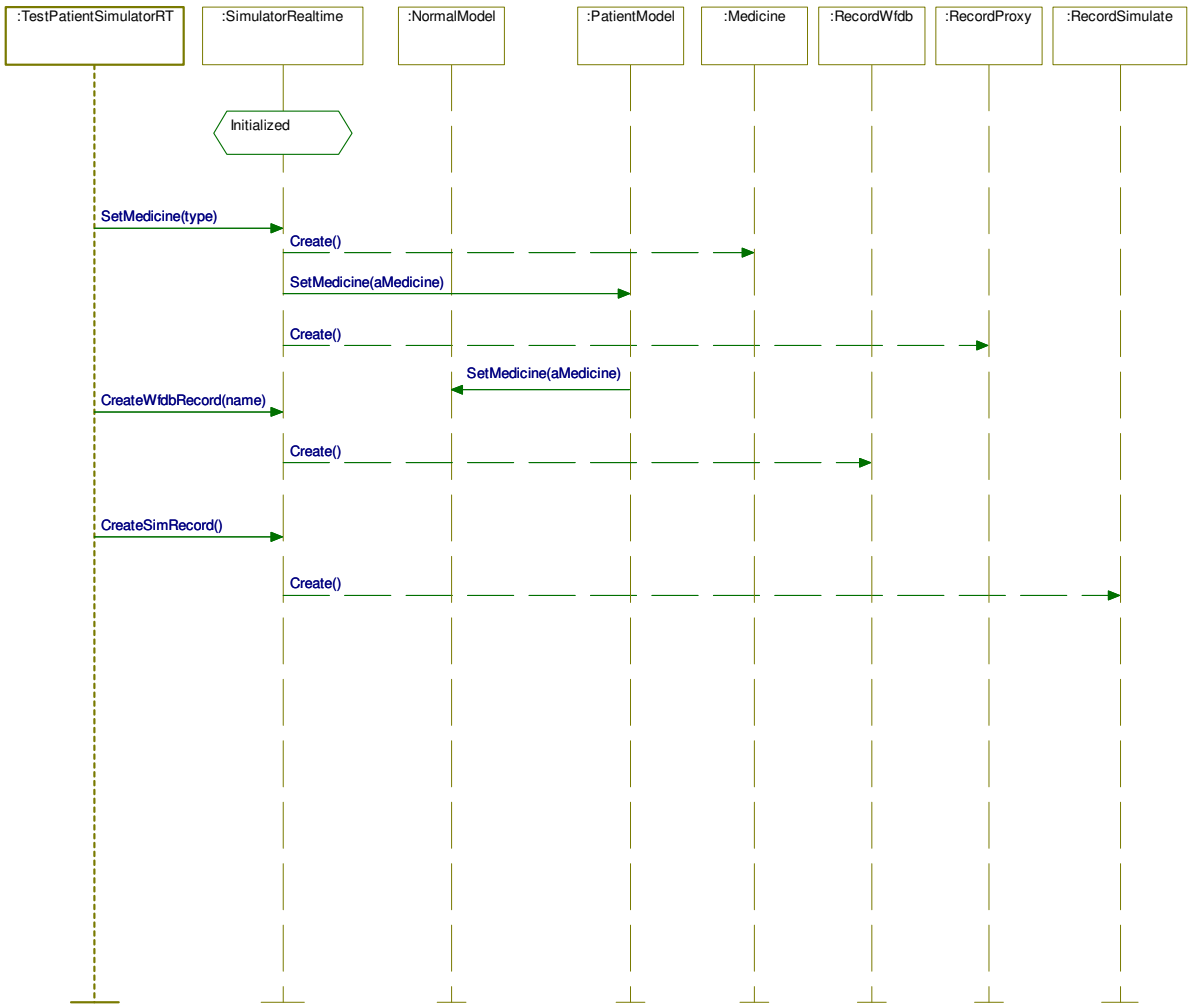


Figure 22 Sequence Diagram: Create Simulation Record and Set Medicine

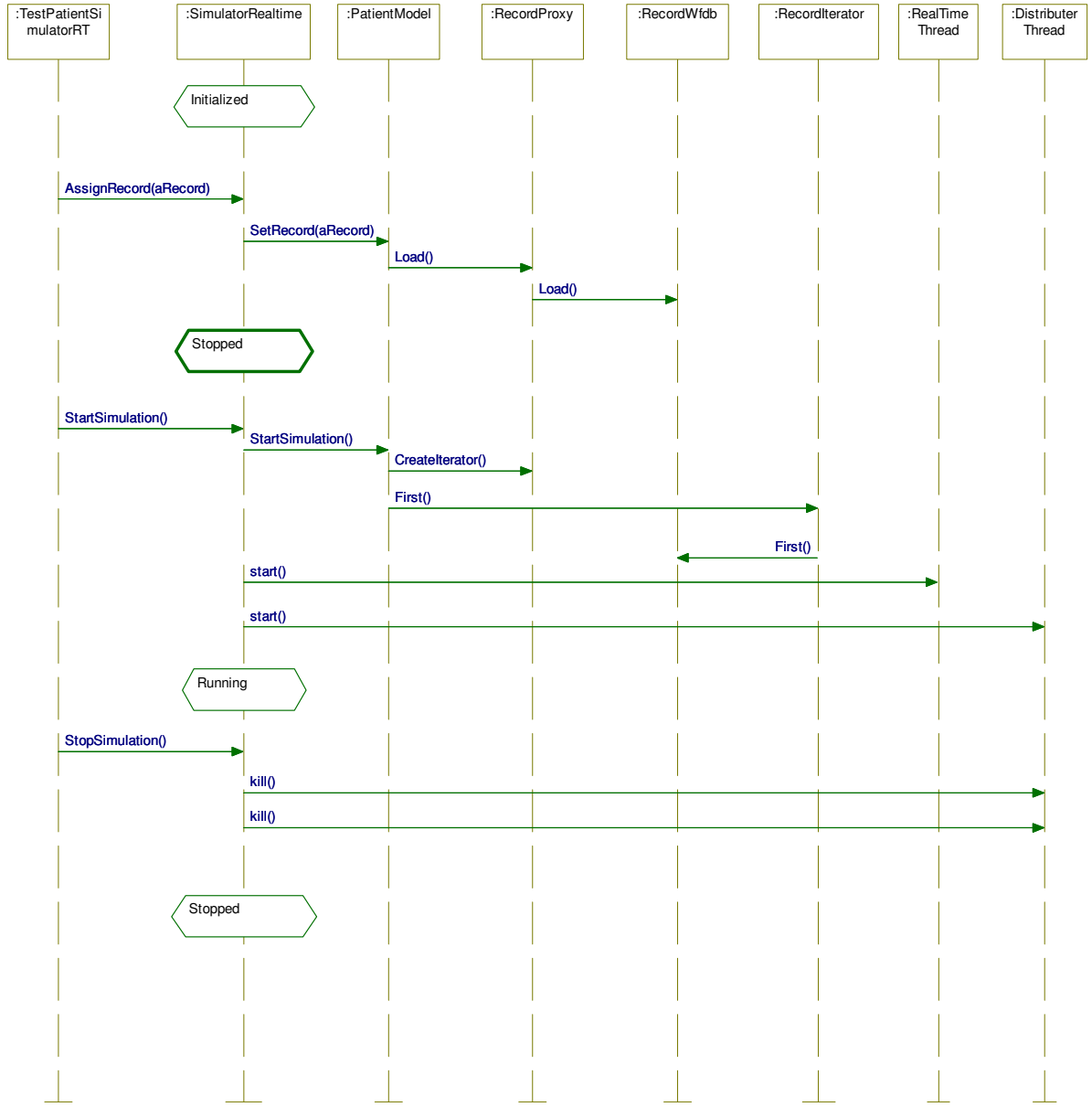


Figure 23 Sequence Diagram: Start Realtime Simulation

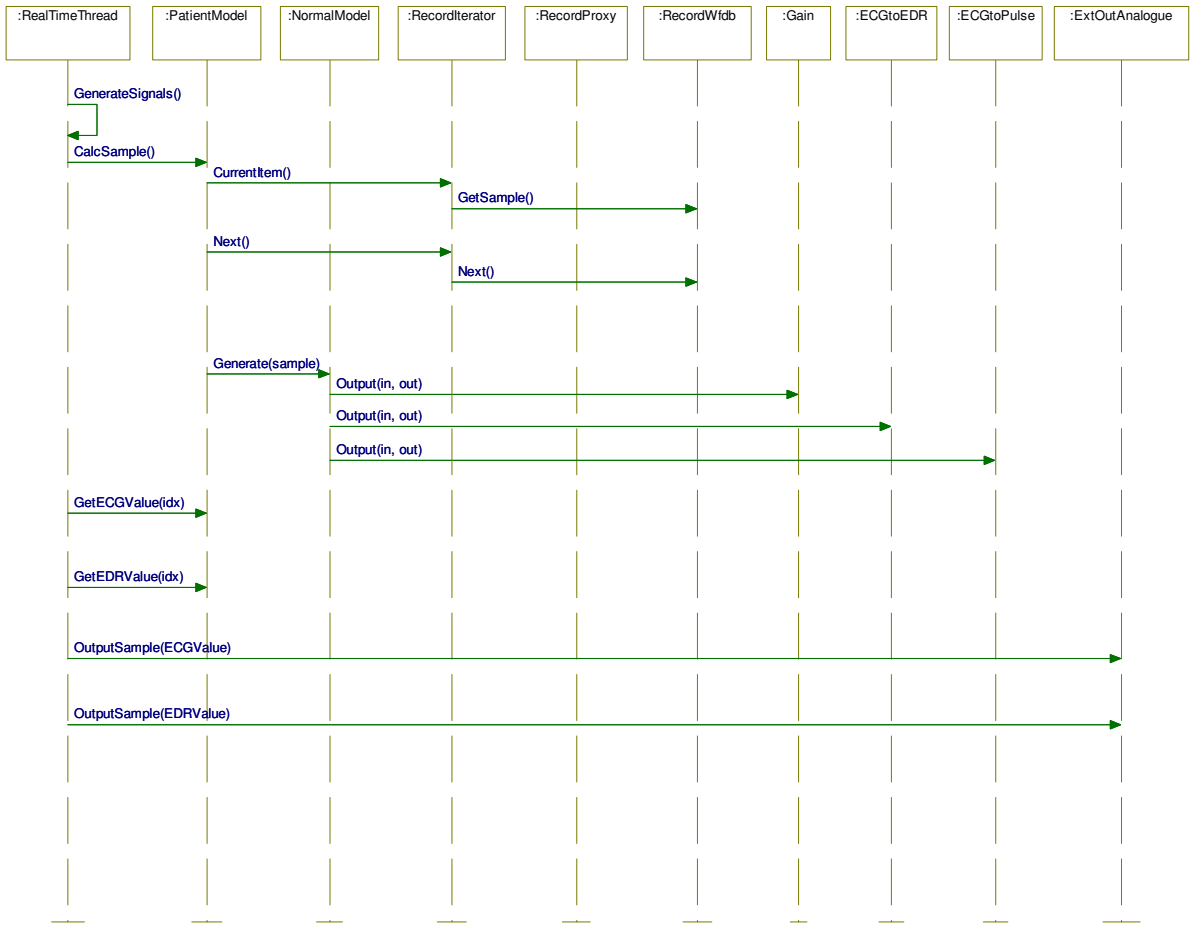


Figure 24 Sequence Diagram: RealTime thread GenerateSignals

```
// RealTimeThread.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);
}
```

```
// PatientModel.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)
    }
}
```

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

5.3.1.1 Test setup for #UC 1 continuous package SimulatorRealtime

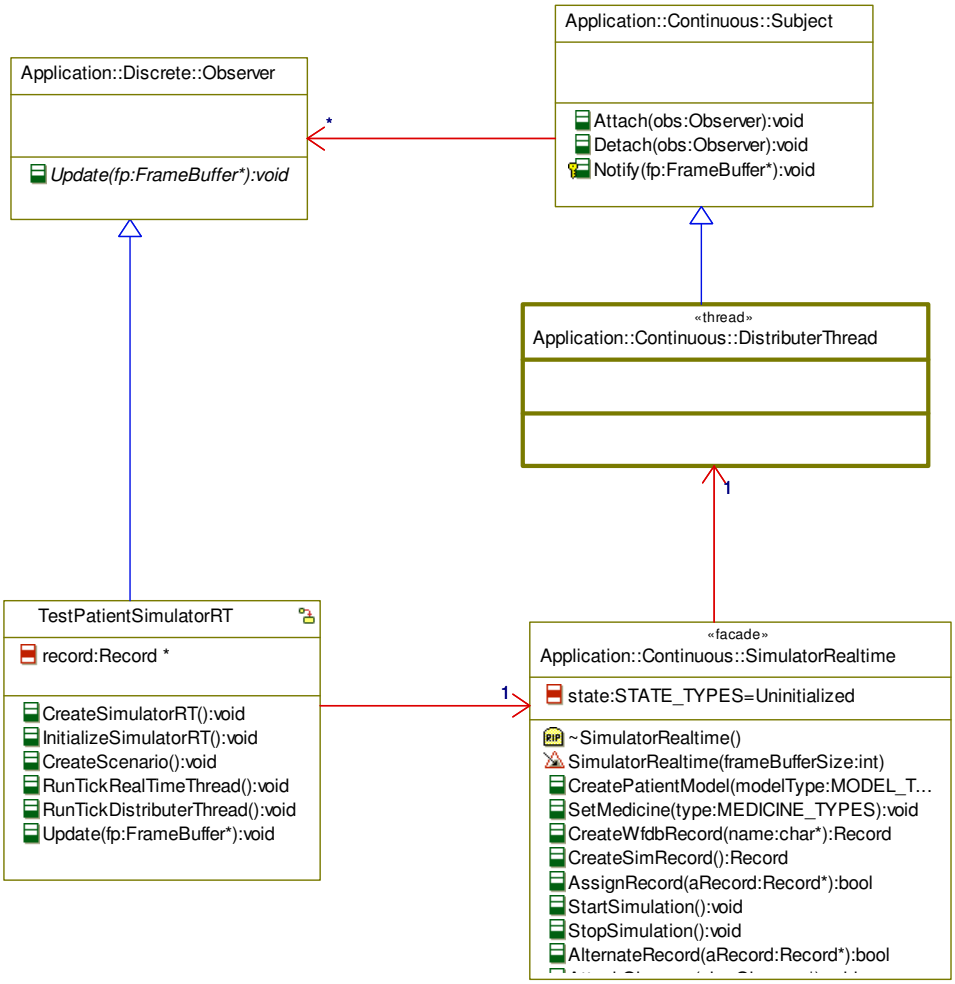


Figure 25 Test setup for UC#1 – test setup in Rhapsody only

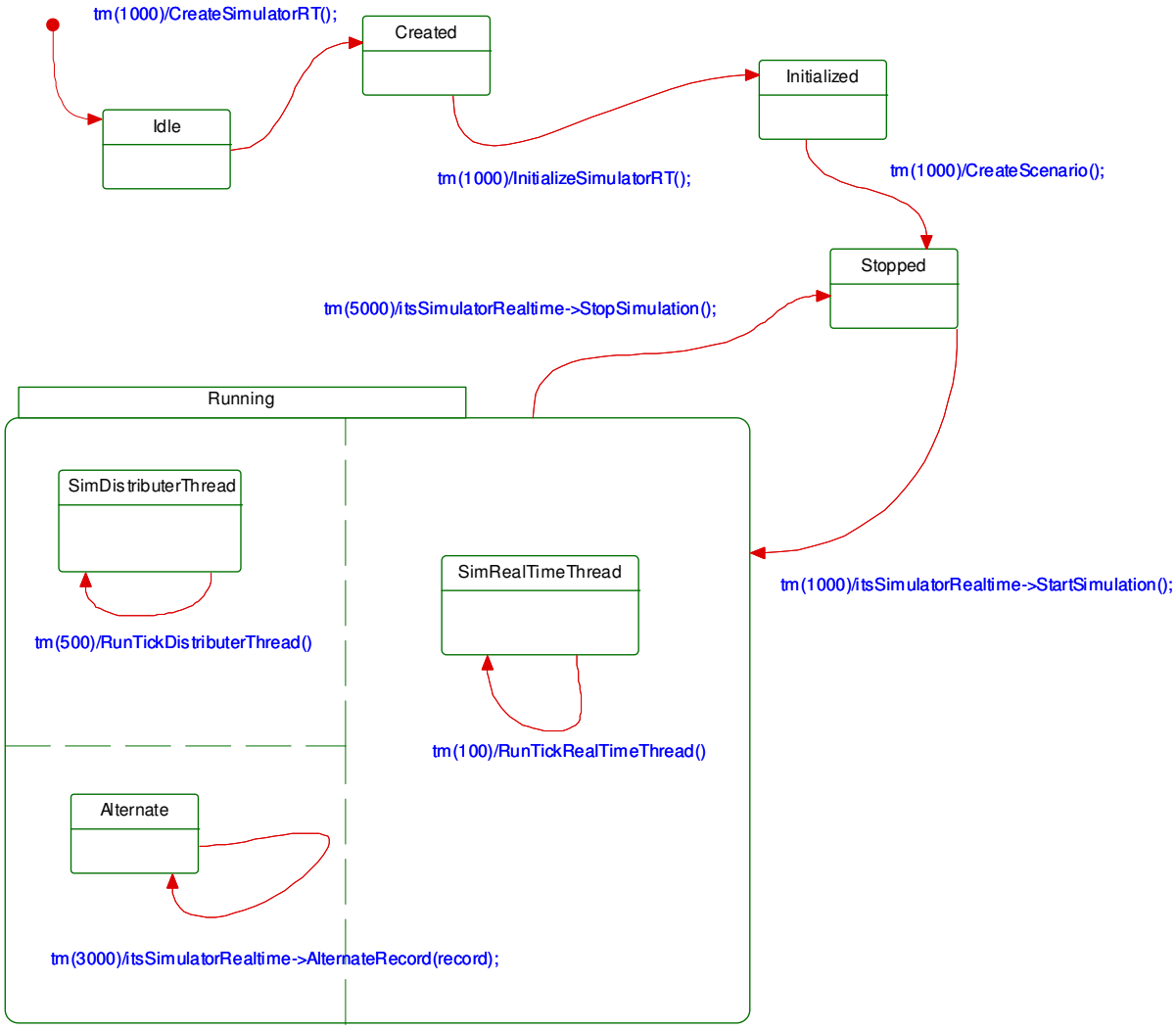


Figure 26 State machine test of UC#1 including simulation of Threads in Rhapsody

5.3.2 Use case #3. Adjust Scenario Parameters realization

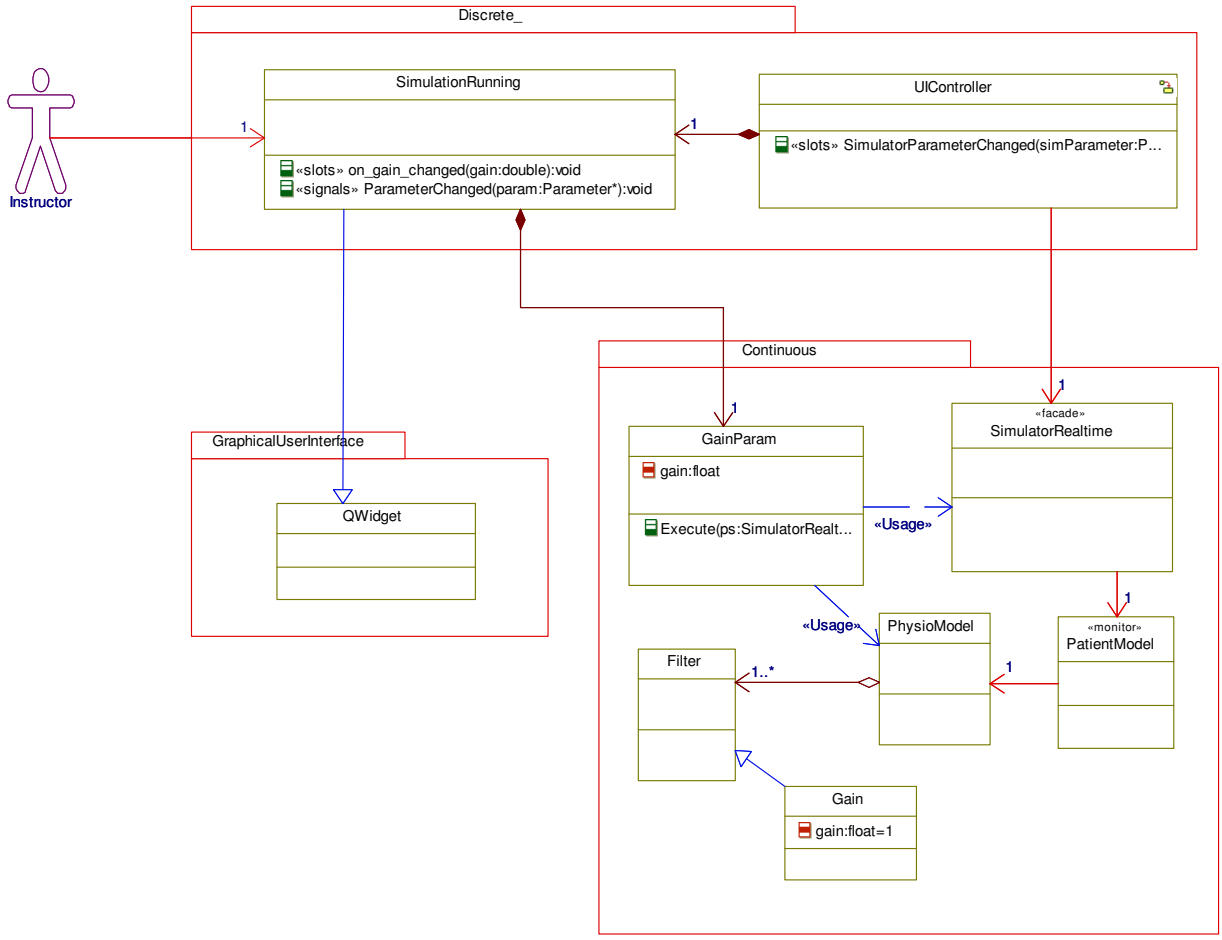


Figure 27 Actor, Classes and packages involved in use case

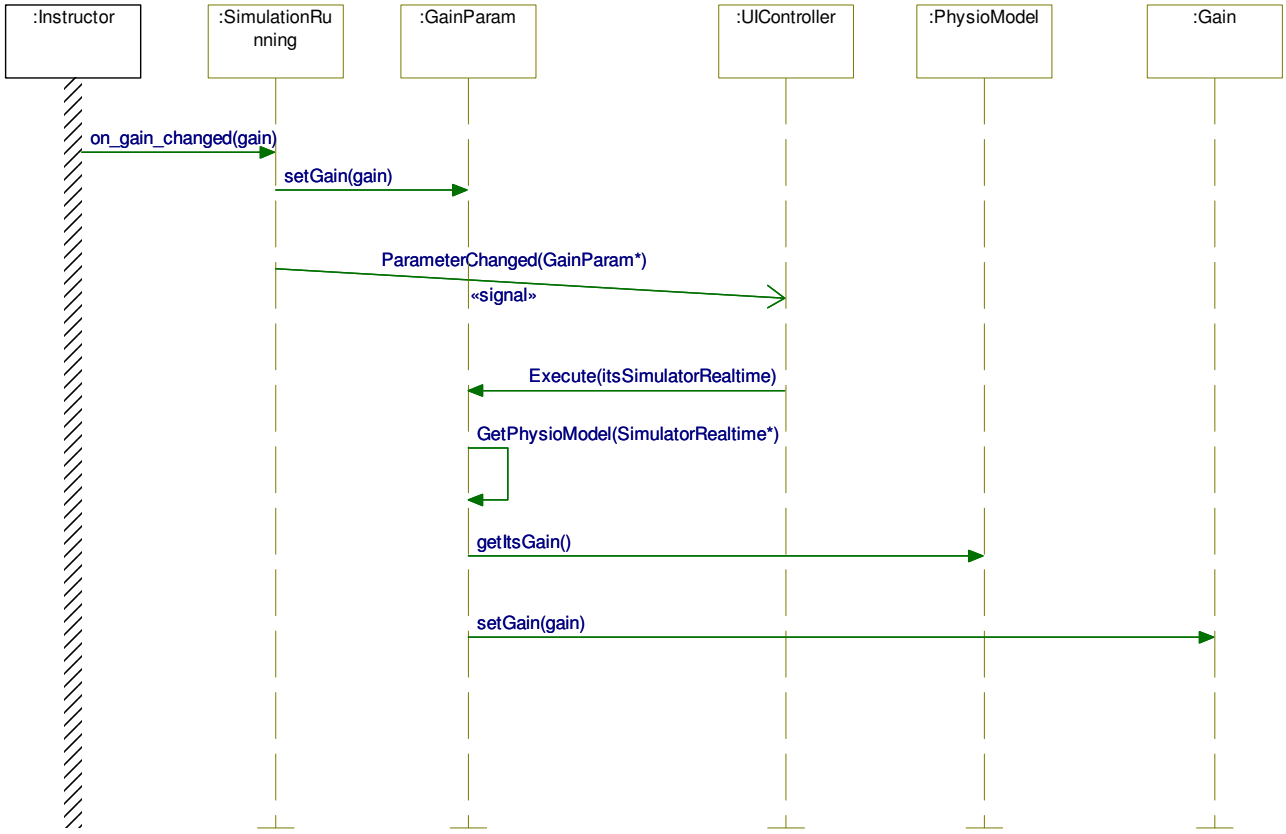


Figure 28 Scenario for instructor adjusting gain parameter

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

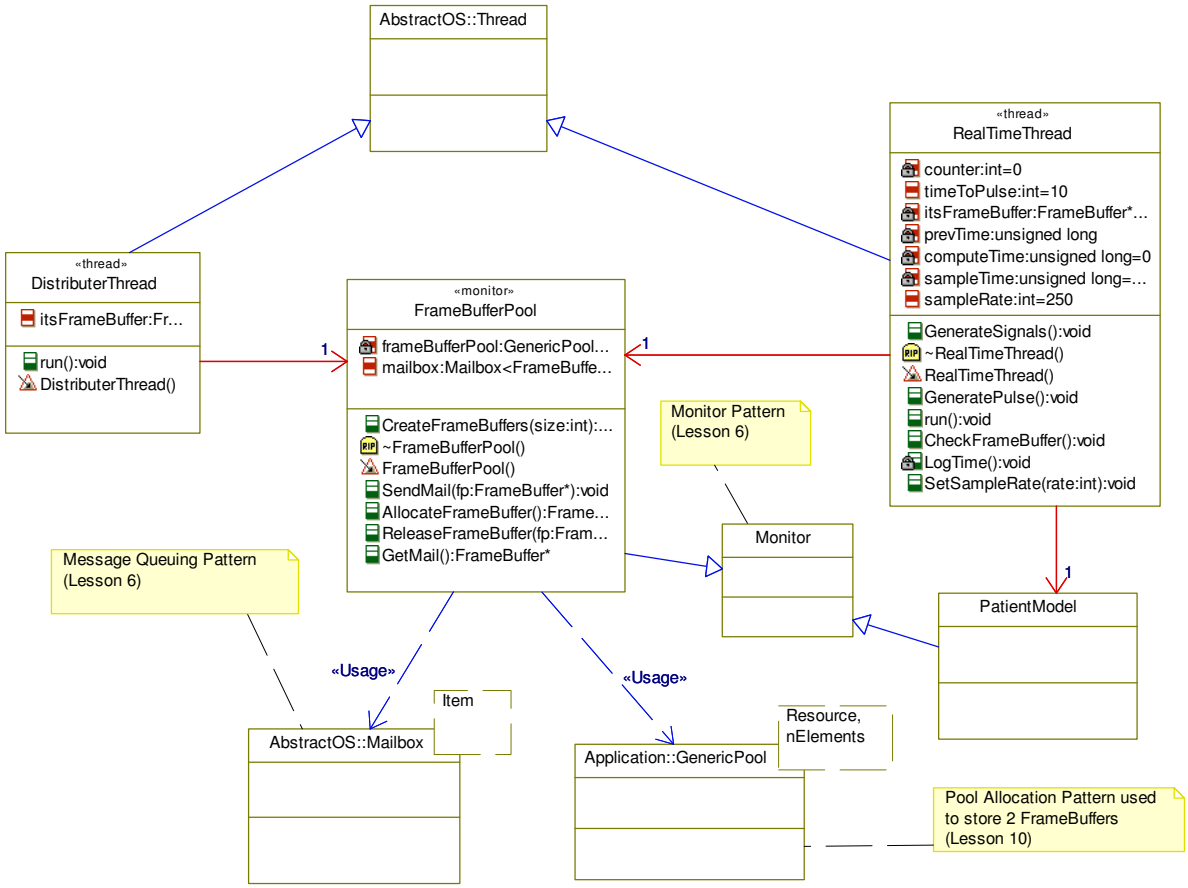


Figure 29 Process view for Distributer and RealTime threads and mechanism for synchronization

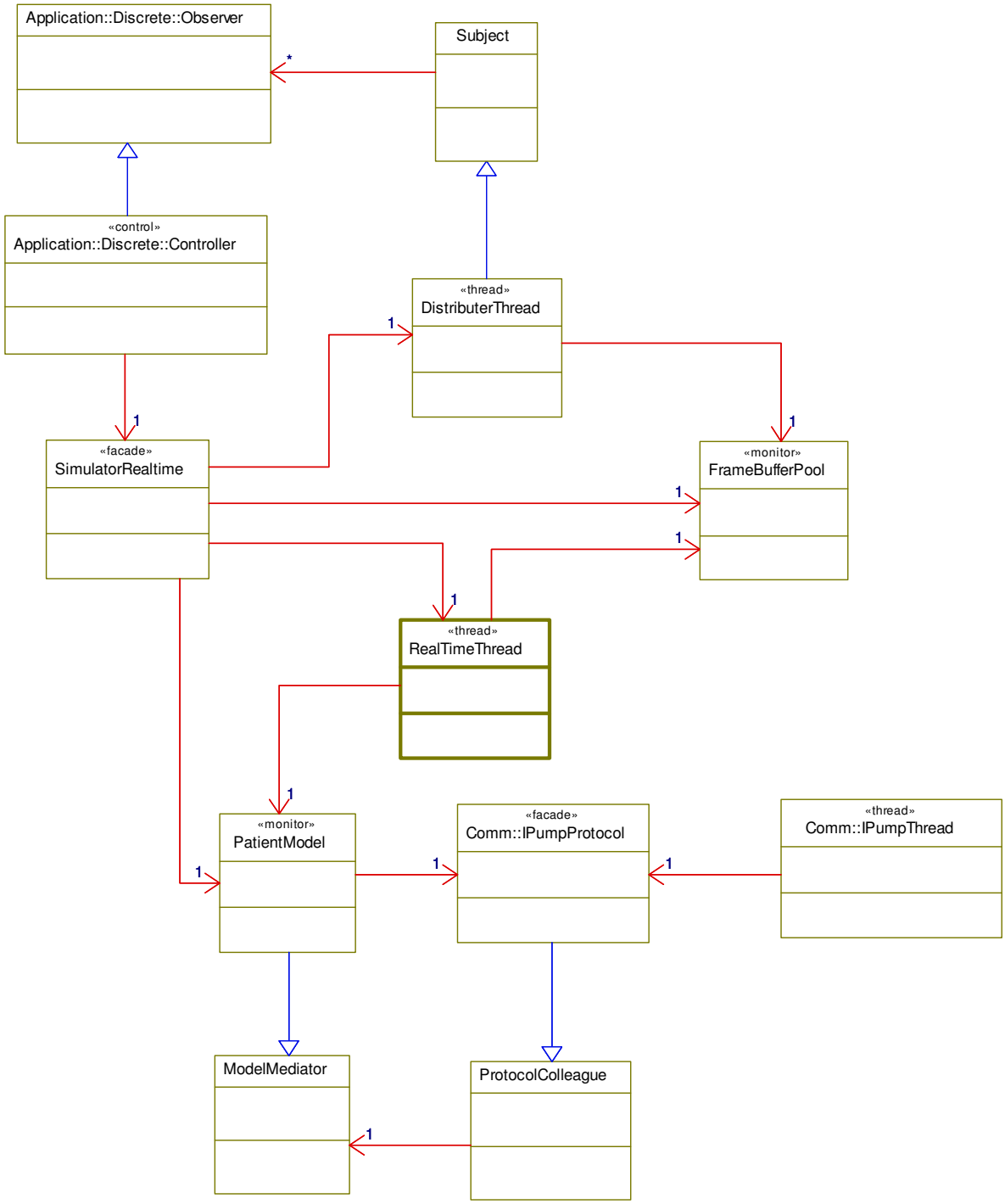


Figure 30 Overview of all threads for RMA analyses

Internal and external event list

#	Event Id	System response	Arrival Pattern	Event Source	Response time
1	Sample	Caculate and gerenrate EDR and ECG signals	Frequence 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 periode updating framebuffer

RMA analysis with Task Blocking (Fs = 250 Hz)

Parameters identified from internal and external event list:

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

#	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

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***

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)	***SAND***

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) ***FALSK***

6.2 Process/task implementation

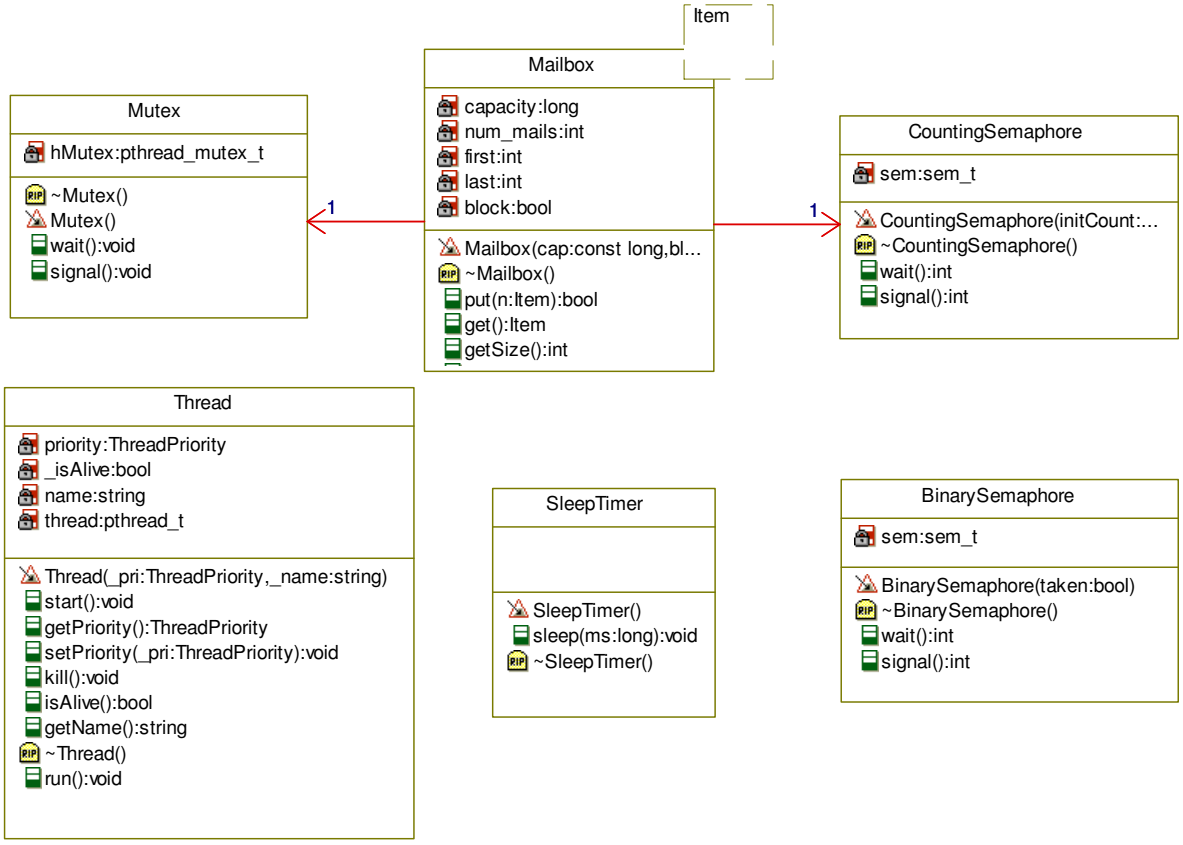


Figure 31 Abstract OS (Linux)

6.3 Process/task communication and synchronization

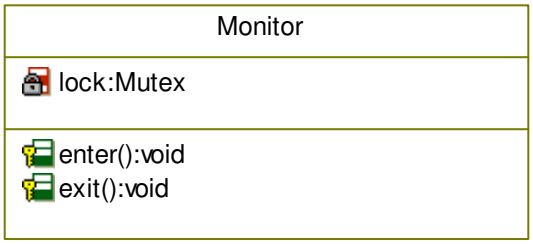


Figure 32 Monitor implemented using mutex

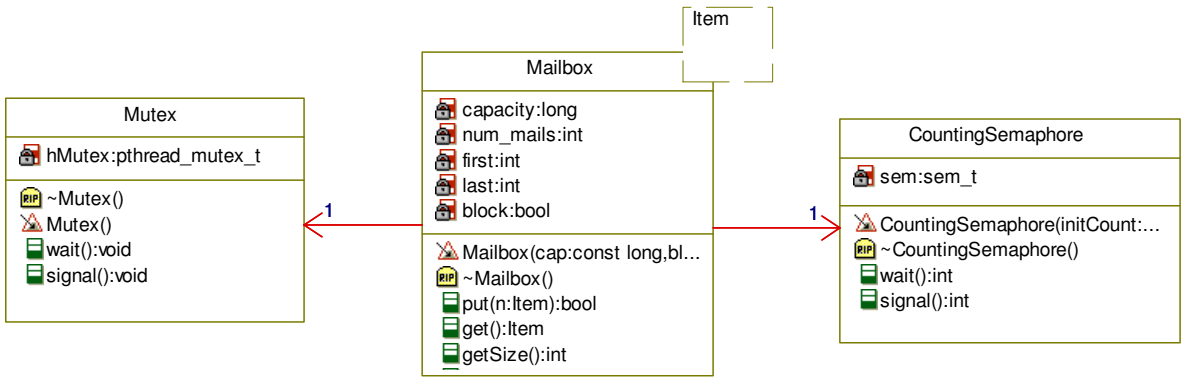


Figure 33 Mailbox implementation for Linux

6.4 Process group 1.

6.4.1 Process communication in group 1

6.4.2 Process 1. description

6.4.3 Process 2. description

6.5 Process group 2.

7. DEPLOYMENT VIEW

7.1 System configurations overview

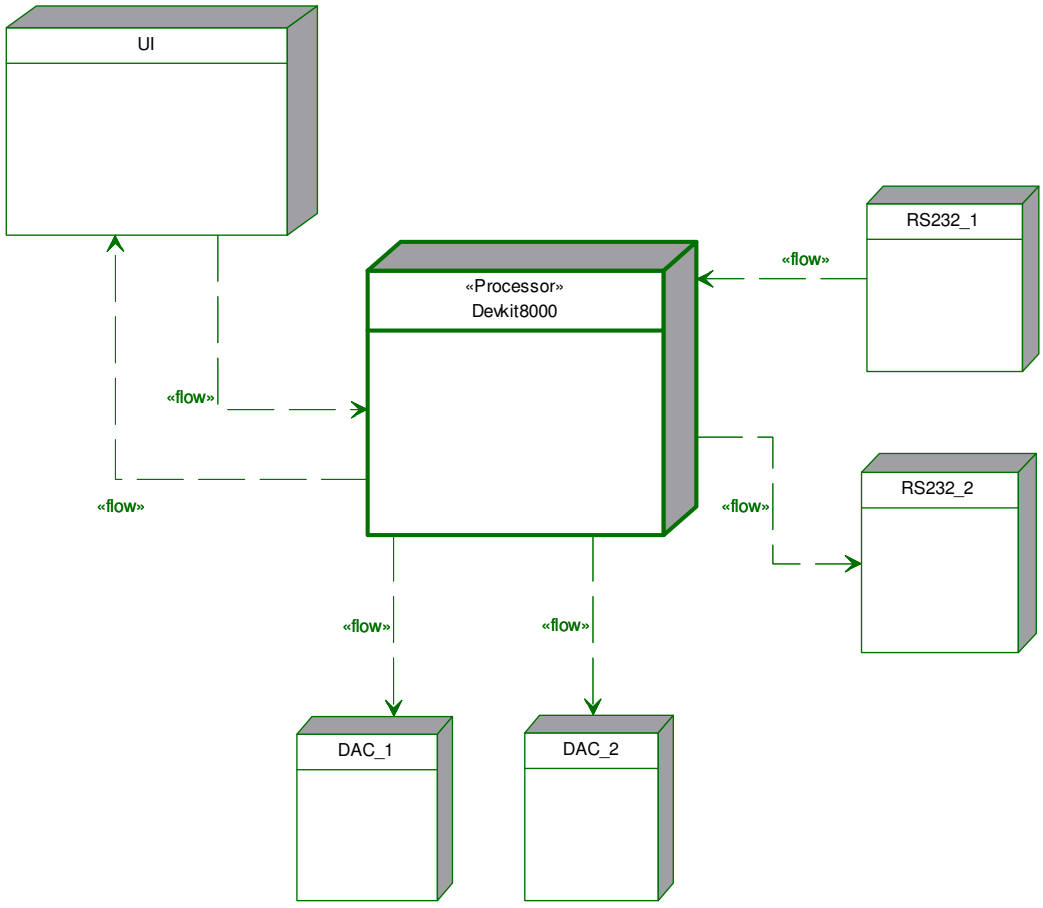


Figure 34 Essential HW nodes used from DevKit8000 used for Sapien190

7.2 System configurations

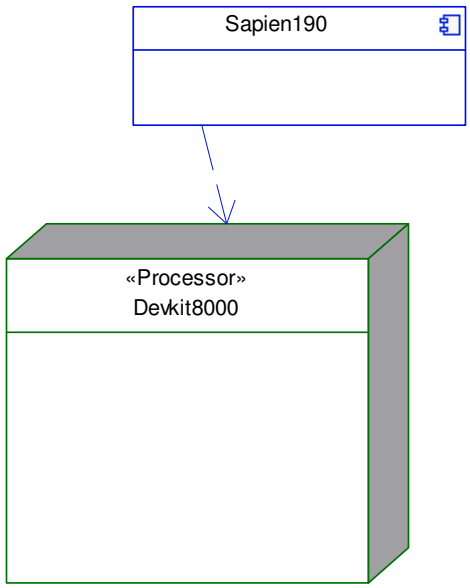


Figure 35 Sapien190 deployed on ARM Linux target platform

7.2.1 Configuration 1.

7.2.2 Configuration 2.

7.3 Node descriptions

7.3.1 Node 1. description

7.3.2 Node 2. description

8. IMPLEMENTATION VIEW

8.1 Overview

8.2 Component descriptions

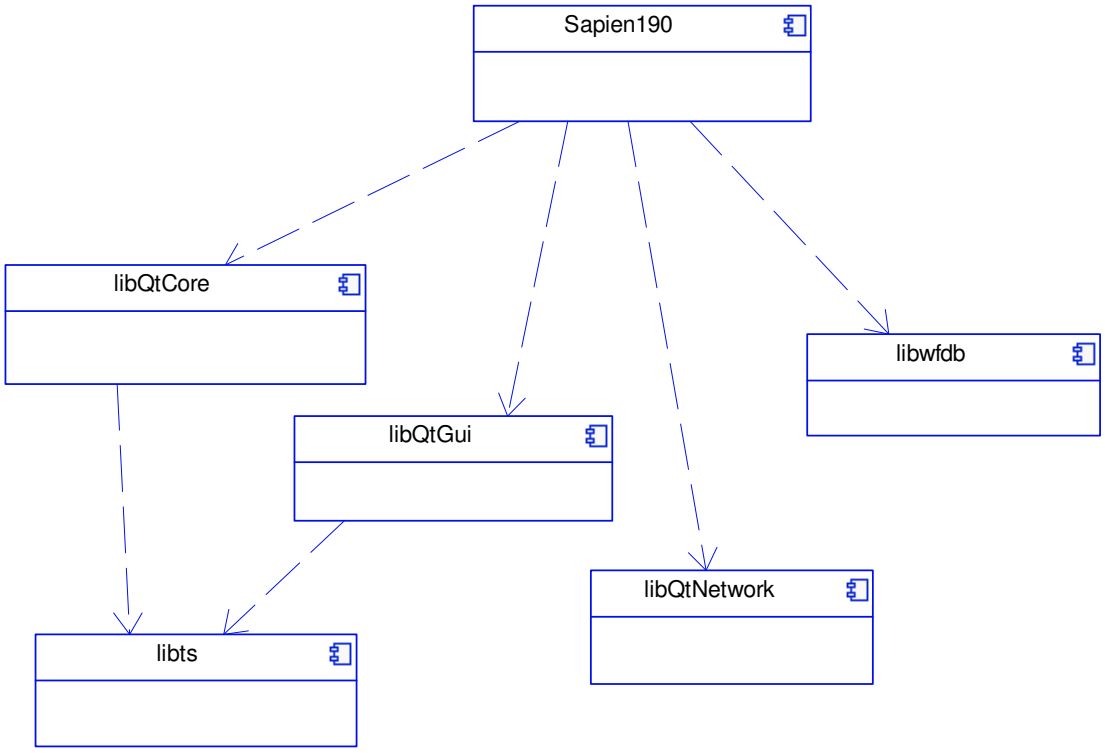


Figure 36 Component diagram for Sapien190 and libraries used from Qt and WFDB

8.2.1 Component 1

8.2.2 Component 2

9. DATA VIEW

9.1 Data model

9.2 Implementation of persistence

10. GENEREL DESIGN DECISIONS

10.1 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.

10.2 Architectural patterns

Observer Pattern:

Observer pattern help 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.

10.3 General user interface design rules

10.4 Exception and error handling

10.5 Implementation languages and tools

This section lists the chosen implementation language and tools with version numbers.
Implementation languages:
C++
Tools
Ecliplse
TrollTech QT Creator

10.6 Implementation libraries

QT-Everywhere-4.6.1

wfdb

11. SIZE AND PERFORMANCE

12. QUALITY

13. 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.

13.1 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.

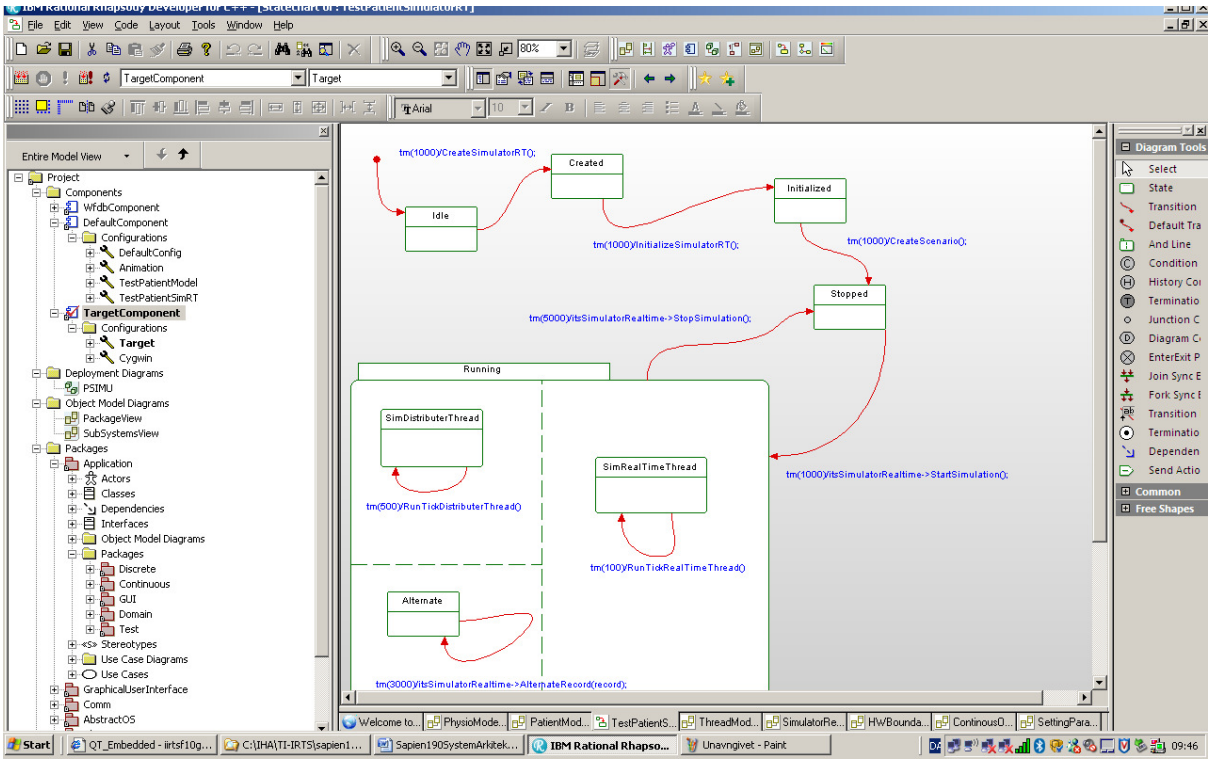


Figure 37 Rhapsody UML model for Sapien 190 used for simulation and test

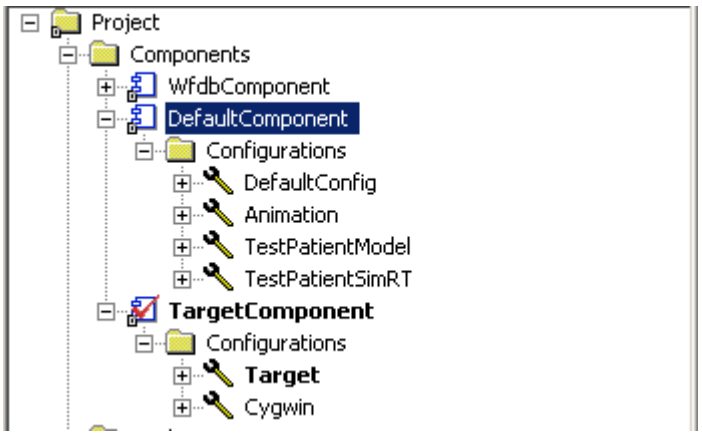


Figure 38 Rhapsody components for testing and source code generation

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 configurated 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.

13.2 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

13.3 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".

13.3.1 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)

14. 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.

14.1 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:

$ 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/

14.2 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

14.3 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`

14.4 Execution-control (start, stop and restart)

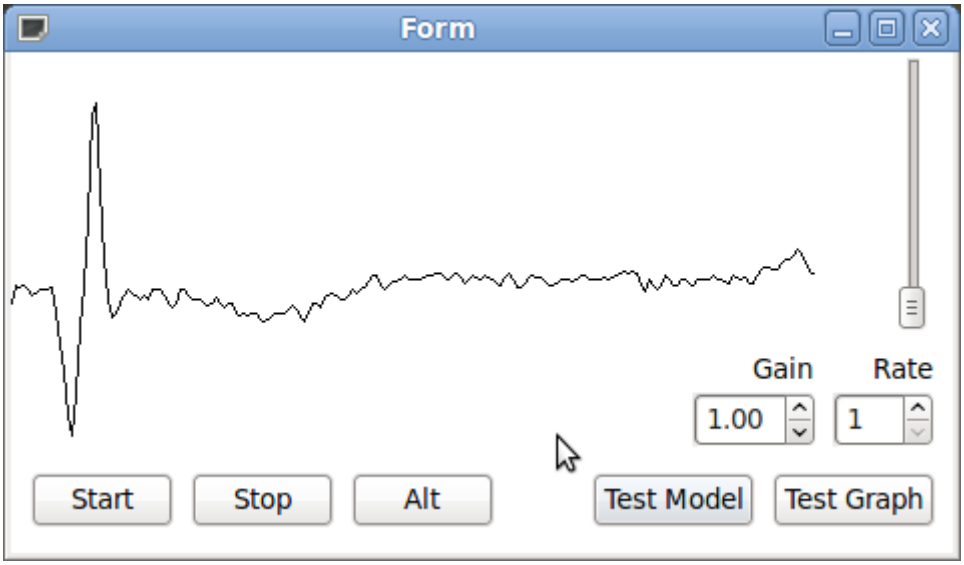


Figure 39 Sapien190 test version

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 gain and rate of playing the record file can be adjusted during simulation.

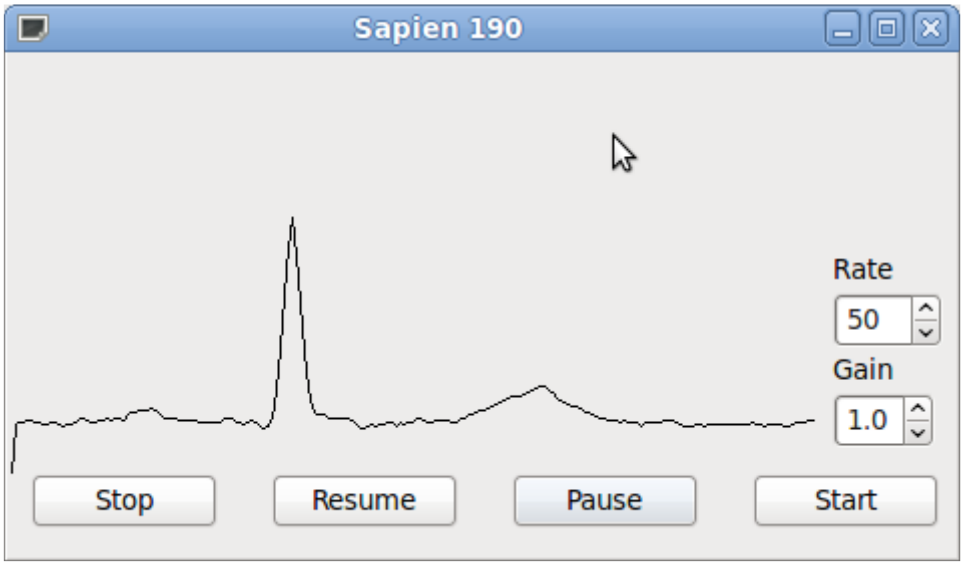


Figure 40 Sapien 190 Final prototype

14.5 Error messages

15. APPENDICES

Test: Compute time without DAC.

Ch1 Sample 49
Pulse 99
Compute 122
Ch0 Sample 100
Ch1 Sample 50
Compute 91
Ch0 Sample 103
Ch1 Sample 51
Compute 91
Ch0 Sample 104
Ch1 Sample 52
Compute 92
Ch0 Sample 106
Ch1 Sample 53
Compute 214
Ch0 Sample 107
Ch1 Sample 53
Compute 213
Kill Thread DistributerThread
Kill Thread RealTimeThread
Ch0 Sample 108
Ch1 Sample 54
Compute 183