Design and implementation of a real-time embedded application

Thomas Baron       Philippe Jean       Gaël Mercier

December 8, 2006
Abstract

This report illustrates the development of a real-time embedded application using the Object-Oriented Analysis & Design.
## Contents

1 Introduction ........................................... 1  
   1.1 Method ............................................. 1  
   1.2 Related work ........................................ 1  
   1.3 Contribution ....................................... 1  

2 Problem Domain Analysis .............................. 2  
   2.1 Context ............................................. 2  
   2.2 System definition ................................... 3  
   2.3 The FACTOR criteria ................................. 4  
   2.4 Problem domain model ............................... 5  

3 Application Domain Analysis ......................... 10  
   3.1 Use cases .......................................... 10  
   3.2 Functionalities ..................................... 11  
      3.2.1 Turn On/Off .................................... 11  
      3.2.2 Acquirement .................................... 12  
      3.2.3 Temperature regulation ......................... 13  
      3.2.4 Monitoring ..................................... 13  
   3.3 Interfaces .......................................... 14  
      3.3.1 MirrorEngine .................................... 14  
      3.3.2 InfraRed ......................................... 14  
      3.3.3 InfraRedDetector ................................. 14  
      3.3.4 TemperatureSensor ............................... 15  
      3.3.5 Thermostat ...................................... 15  
      3.3.6 ExternalCommunicationDevice .................. 15  

4 Design .................................................. 17  
   4.1 Well-design criteria ................................ 17  
   4.2 Architecture ....................................... 18  
   4.3 Detailed Design .................................... 20  

5 Schedulability .......................................... 35  
   5.1 Assessment .......................................... 35  
      5.1.1 Scheduling mechanism ............................ 35  
      5.1.2 Scheduling policy ................................ 35  
      5.1.3 Schedulability analysis ......................... 37  
   5.2 Conclusion .......................................... 38
6 Implementation

6.1 Software usage .................................................. 41
6.2 Source code .................................................... 41

7 Testing

7.1 Unit Testing ....................................................... 46
  7.1.1 Protocol ...................................................... 47
  7.1.2 Example ..................................................... 47
7.2 Integration Testing .............................................. 48
  7.2.1 Protocol ...................................................... 48
  7.2.2 Example ..................................................... 48
7.3 Acceptance Testing ............................................. 48

8 Conclusion .......................................................... 50

Bibliography .......................................................... 51
List of Figures

2.1 Instrument for analysis ...................................... 2
2.2 FTIR system rich picture .................................... 2
2.3 Interferometer principle .................................... 3
2.4 FTIR physical module ....................................... 4
2.5 Overview of the classes in the system ....................... 5
2.6 The TemperatureSensor Statechart Diagram ................. 6
2.7 The Thermostat Statechart Diagram ........................ 6
2.8 The Interferometer Statechart Diagram ...................... 7
2.9 The MirrorEngine Statechart Diagram ...................... 7
2.10 The InfraRedDetector Statechart Diagram .................. 8
2.11 The InfraRed Statechart Diagram .......................... 8
2.12 The ExternalCommunicationDevice Statechart Diagram .... 9
3.1 Use cases .................................................. 10
3.2 System states ............................................. 11
3.3 Acquire .................................................. 12
4.1 The layered Architecture ................................... 19
4.2 Class Diagram of the device layer .......................... 20
4.3 Class Diagram of the logic layer ........................... 22
4.4 TemperatureReading Activity Diagram ...................... 22
4.5 TemperatureRegulation Activity Diagram ................... 23
4.6 Monitoring Activity Diagram ............................... 24
4.7 Acquisition Activity Diagram ............................... 25
4.8 Scanning process Activity Diagram ......................... 26
4.9 Finalisation process Activity Diagram ....................... 26
4.10 Interrupting process Activity Diagram ...................... 27
4.11 InputCommunication Activity Diagram ..................... 28
4.12 OutputCommunication Activity Diagram ..................... 29
4.13 Class Diagram of the check layer ......................... 30
4.14 Watchdog Activity Diagram ............................... 30
4.15 Class Diagram of the data layer ........................... 32
4.16 Class Diagram of the communication layer ................. 34
7.1 Classes Logger and LoggerTest ............................. 47
7.2 LoggerTest standard output ................................ 47
7.3 Scenario Temperature Regulation ........................... 48
7.4 Trace command line ....................................... 49
7.5 TemperatureRegulationTest standard output ................ 49
# List of Tables

3.1 MirrorEngine operations ........................................... 14  
3.2 InfraRed operations .............................................. 14  
3.3 InfraRedDetector operations .................................... 14  
3.4 TemperatureSensor operations ................................. 15  
3.5 Thermostat operations ........................................... 15  
3.6 ExternalCommunicationDevice operations .................. 15  
4.1 Priority of design criteria ........................................ 18  
5.1 table of priorities .................................................. 37  
5.2 Worst-case execution time for each shared resource ...... 38  
5.3 blocking time for processes ...................................... 39  
5.4 Response Time of the processes ............................... 40
Chapter 1

Introduction

The work discussed in this report appears in the Ravenscar-Java Development project. The aim of this project is to analyse whether Ravenscar-Java profile is a realistic Java profile for industrial-time systems, to show how Real-Time UML is used a design tool and to compare a Ravenscar-Java solution with a C++ solution.

The first part of the Ravenscar-Java Development project was to implement the Ravenscar-Java profile in the way that it can be suitable for industrial cases. In order to meet with the project goal, the Ravenscar-Java profile was implemented on a Java processor, the aj-100. This processor is developed by aJile Systems.

The aj-100 processor from aJile Systems is a 100 Mhz direct execution Java processor. It is a pure Java microcontroller that uses Java bytecode as its native instruction set. The implementation of the Ravenscar-Java profile only uses one of two JVM units.

1.1 Method

The work illustrates in this report follows the method describes in [9].

1.2 Related work

Related work.

1.3 Contribution

The aim of this work show whether the implementation of the Ravenscar-Java profile is suitable for industrial applications through a case supplied by FOSS Analytical A/S. This work should prove that an application written for the Ravenscar profile is easier to write, maintain and port.
Chapter 2

Problem Domain Analysis

2.1 Context

The FTIR (Fourier Transform InfraRed [1]) module is controlled by an embedded software enclosed in an instrument for analysing and controlling the quality of agricultural production, food, pharmaceutical and chemical products. Figure 2.1 is an example of this kind of instrument.

Figure 2.1: Instrument for analysis

The FTIR system falls into 3 following modules (Figure 2.2):

Figure 2.2: FTIR system rich picture
1. The interferometer module (Figure 2.3): In order to measure a sample, a **beam of infrared** (IR) light is passed through a liquid sample and the interferometer produces a signal called an interferogram. In the liquid sample some energy for specific frequencies is absorbed, which gives a unique characteristic of the sample. The measurement is obtained by a **detector**. Because the interferogram is composed of every infrared frequencies, it needs to process a number of the scans of different lengths. Thus the interferometer has a **moveable mirror** to move to the next position in order to take a new measurement. These results are by the software to be developed averaged into an resulting interferogram.

![Figure 2.3: Interferometer principle](image)

2. The temperature controller module: The FTIR instrument is enclosed in an isolated box (Figure 2.4), a thermobox, in order to regulate the temperature of the instrument. The aim of this thermobox is to keep a constant temperature around the interferometer. The temperature inside the box is controlled at the following places: the **thermobox**, the **IR-source** (InfrRed source), the **interferometer** and the **sample cuvette**. Each place has a **temperature feeler** and a **heater/cooler** element for individual regulation. The software must implement a PID regulator for the regulation.

3. The external communication module: We know only that a measurement is started form outside. When a measurement is finished the result is sent back along with some other information. The communication follow a protocol which is not specified at the moment but the access medium is a RJ45 cable.

### 2.2 System definition

The instrument is controlled by a software running on a processor called aj-100. The instrument called FTIR system is an instrument for analysing and control-
ling the quality of industrial products. This system has real-time constraints and its aim is to analyse a sample. The different parts of the system such as the interferometer, the temperature controller and the external communication should be controlled. Thus the instrument should be able to launch a test on a sample and render the results of its analysis.

2.3 The FACTOR criteria

**Functionality for end use:** The first goal of the system is to make analysis of a sample. The system should be able to produce an interferogram or report error conditions making it impossible.

**Application Domain of end use:** It is to launch the sampling of product and collect results from the interferometer to transmit them. The software have also to report errors occurring in the system and to monitor the correct behaviour of the system.

**Conditions for success:** The system must be as dependable as the one currently produced, designed for easy maintenance and adaptation to new hardware, software and features.

**Technology to be used:** The aJ-100 real-time processor. It uses Java as its native language. The software shall be developed according to the Ravenscar-Java Profile (a subset of the Real-Time Specification for Java) and its implementation on the processor. Tools Software used for this project are Eclipse, JEM builder
and Charade. Eclipse and its plugins enable to design real-time embedded system using Object-Oriented (OO) techniques. JEM builder and charade are used to configure, load and start JVM on the aj-100 processor. We have not chosen tools to test the application but we think of using the UML 2.0 testing profile.

**Object System:** Temperature control, Interferometer, external communication, data logging modules.

**Responsibility:** To implement the functionalities ensuring the same behaviour as the existing solution but with a Java real-time processor.

### 2.4 Problem domain model

After reading Hans Sndergaard’s technical report [2], we have made out models present in this section. Models are characterized by the following objects which described the system. Statechart diagrams are also shown to extend our class definitions by adding descriptions of their behaviour.

![Figure 2.5: Overview of the classes in the system](image)

**FTIRSystem class**

The FTIRSystem class models the properties of the FTIR system states which are illustrated in Figure 2.5. It is composed of the Interferometer, Temperature-Sensor, Heater, Cooler and ExternalCommunicationDevice.
**TemperatureSensor class**
The TemperatureSensor class models the working of the temperature feeler. It just consists of reading the temperature.

![TemperatureSensor Statechart Diagram](image)

Figure 2.6: The TemperatureSensor Statechart Diagram

**Thermostat class**
The Thermostat class describes how some thermoelectric modules works. It consists of cooling or heating the place where the module is.

![Thermostat Statechart Diagram](image)

Figure 2.7: The Thermostat Statechart Diagram
**Interferometer class**
The Interferometer class models the working of the interferometer. It is composed of the MirrorEngine, InfraRed and InfraRedDetector. The behaviour of these classes are described further. The interferometer allows to scan a sample and can be interrupted.

![Interferometer Statechart Diagram](image)

**Figure 2.8: The Interferometer Statechart Diagram**

**MirrorEngine class**
The MirrorEngine class models the working of mirror. It consists of controlling the move of mirror.

![MirrorEngine Statechart Diagram](image)

**Figure 2.9: The MirrorEngine Statechart Diagram**
**InfraRedDetector class**
The InfraRedDetector class models the working of the InfraRed Detector. It consists of turning on/off the detector and taking the measurements.

![InfraRedDetector Statechart Diagram](image)

**InfraRed class**
The InfraRed class models the working of the InfraRed. It consists of turning on/off the beam.

![InfraRed Statechart Diagram](image)
**ExternalCommunicationDevice class**

The ExternalCommunicationDevice class models the working of the communication with external devices. It consists of sending messages about interferogram results and information relating to the monitoring of the system and receiving messages from outside.

![ExternalCommunicationDevice Statechart Diagram](image)

Figure 2.12: The ExternalCommunicationDevice Statechart Diagram
Chapter 3

Application Domain Analysis

3.1 Use cases

The system has only one actor which interacts with it. Behind this actor – external system –, we assume there is an operator or somebody who pilot the entire system including the FTIR system. First of all, the system just observes the power switch. Then, as its unique action, the external system can acquire an interferogram. It receives data from the monitoring function like analogical values combined with their bounds or errors that occur in the system.

![Diagram of use cases](image)

Figure 3.1: Use cases
3.2 Functionalities

3.2.1 Turn On/Off

When the external system – or certainly the person behind it – turns the system on, it observes the power switch and starts by regulate the temperature. When all the temperatures are within bounds, the system is ready to acquire an interferogram. If at any time, whether the system is acquiring an interferogram or not, a physical module is not at the ideal temperature, then the current acquirement — if there is one — will be stated at failed status and the system will go back to the state where it waits for all the temperatures are within bounds. Figure 3.2 shows the system states.

![System states diagram]

Figure 3.2: System states
3.2.2 Acquirement

The acquirement use case is the only one which is directly initiated by the external system actor. It is composed of two functions. The first one is the acquire function. It is used to start an acquirement of an interferogram. The second one is the interrupt function. It is used to interrupt an acquirement.

Acquire

The acquire function is the main one of the system. We can notice that a complete measurement of a sample is composed of 32 scans where each scan has up to 3200 measurements. When the system is ready – all physical modules are at the right temperature – an interferogram can be acquired. The prerequisite is that the system is fully initialized. The type of the function is “read”: it returns a table of the interferometer results to the external system actor. Figure 3.3 shows the sequencing of the acquire activity.

![Figure 3.3: Acquire](image-url)
Interrupt

The interrupt function can be considered as a subset of the acquire function. It basically stops the interferometer during its acquisition process and reinitialize all the system. As the acquire function, this one is initiated by the external system actor. It represents a change of state in the model, thus it is an “update” function.

3.2.3 Temperature regulation

The temperature regulation use case is a parallel one of the acquirement use case. It is initiated when the system is turned on. It is composed of two functions. The first one is the temperature read function. It is aimed at reading from thermometers. The second one is the temperature regulate function. It is used to regulate the temperature with a Thermostat device according to the values read by the temperature read function.

Temperature read

The aim of the temperature read function is to take measurements periodically of four parts in the system: The thermobox, the InfraRed source, the interferometer and the sample cuvette. The type is a “read” function.

Temperature regulate

The aim of the temperature regulate function is to control the heating and the cooling of the physical modules of the system. This periodic routine use an average of five temperatures to regulate the temperature in the system. The type of this function is “signal”: it manipulates the heater and the cooler when a temperature is different (c.f. PID regulation) from its reference. It results in a direct intervention in the problem domain: the need to maintain constants temperatures (34°C in the thermobox, 42°C inside the interferometer and at the cuvette) for the interferogram acquirement.

3.2.4 Monitoring

As the temperature regulation use case, this one is initiated when the system is turned on. It is aimed at checking the state of the system in a general manner. It is composed of two functions: the monitor function which checks that the system is in the normal conditions of working and the function aliveness check function which checks that all functions of the system are still running.

Monitor

The monitor function is used to save all the analogical values such as temperatures and mirror speed in order to send them to the external system actor. It checks that each temperature value is between its bounds. If it is not the case, an error is saved which will be sent to the external system. The speed of the mirror is processed in a similar way. The monitor function also saves how many measurements has been taken and how many has failed. The type of this
function is “signal”: it results in data which will be sent to the external system actor.

**Function aliveness check**

It is the watchdog that checks the aliveness of functions. It controls that all the system functions have registered themselves to it. If it is not the case, it saves an error which will be sent to the external system actor. It is a state change in the system (alive to dead): it is a “signal” function.

### 3.3 Interfaces

Every errors that occurs when calling the methods below are handled. Such errors are saved by the functions that caught them in order to be sent to the external system.

#### 3.3.1 MirrorEngine

We assume that the mirror engine is a synchronous stepper motor [2] of which the displacement length between two steps is negligible considering the system time constraints. Figure 3.1 shows MirrorEngine operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetPosition</td>
<td>Used to get the current position of the mirror</td>
</tr>
<tr>
<td>MoveForwards</td>
<td>Used to move forwards the mirror to its next position</td>
</tr>
<tr>
<td>MoveBackwards</td>
<td>Used to move backwards the mirror to its next position</td>
</tr>
</tbody>
</table>

Table 3.1: MirrorEngine operations

#### 3.3.2 InfraRed

Figure 3.2 shows InfraRed operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetOn</td>
<td>Used to turn the InfraRed on</td>
</tr>
<tr>
<td>SetOff</td>
<td>Used to turn the InfraRed off</td>
</tr>
</tbody>
</table>

Table 3.2: InfraRed operations

#### 3.3.3 InfraRedDetector

Figure 3.3 shows InfraRedDetector operations.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SetOn</td>
<td>Used to turn the InfraRedDetector on</td>
</tr>
<tr>
<td>SetOff</td>
<td>Used to turn the InfraRedDetector off</td>
</tr>
<tr>
<td>Read</td>
<td>Used to get the value of the beam caught by the detector</td>
</tr>
</tbody>
</table>

Table 3.3: InfraRedDetector operations
3.3.4 TemperatureSensor

Figure 3.4 shows TemperatureSensor operations.

<table>
<thead>
<tr>
<th>Used to read the value from the thermometer</th>
</tr>
</thead>
</table>

Table 3.4: TemperatureSensor operations

3.3.5 Thermostat

Figure 3.5 shows Thermostat operations.

<table>
<thead>
<tr>
<th>Used to set the level of the Thermostat</th>
</tr>
</thead>
</table>

Table 3.5: Thermostat operations

3.3.6 ExternalCommunicationDevice

We assume that the protocol between the External System and the system itself is CLDC\(^1\) and the data format is UTF-8. Figure 3.6 shows ExternalCommunicationDevice operations.

<table>
<thead>
<tr>
<th>Used to receive data from the external system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Used to send data to the external system</td>
</tr>
</tbody>
</table>

Table 3.6: ExternalCommunicationDevice operations

Incoming communication

The only two incoming messages are acquire and interrupt.

Outgoing communication

The outgoing messages are:

- errors
- speed and its bounds
- temperature regulation informations:
  - temperatures and their bounds
  - difference with the reference temperature
  - result of PID regulation and details of its computation
- interferometer:
  - how many time the FTIR module has been on

\(^1\)Connected Limited Device Configuration
- how many measurements has been taken
- how many measurements failed

- watchdog
- interferogram
Chapter 4

Design

The purpose of the design is to specify a solution which can be easily converted into programming code. The process design can be divide into three categories: criteria, architectural and detailed design. Criteria design specifies criteria for object-oriented and good design. Architectural design details the largest software structures, such as subsystems, packages and tasks. Detailed design specifies attributes and methods within individual classes for each functionality of our application.

4.1 Well-design criteria

Object Oriented Analysis & Design emphasizes three general criteria which are usability, flexibility, comprehensibility. There are also many quality criteria proposed by researchers over the past years (usable, secure, efficient, correct, reliable, maintainable, testable, ...). Of course, we do not use all these criteria because criteria are often in conflict or not relevant to the specific project. Design have to accentuate essential criteria.

Having considered the general object-oriented criteria, we can give a priority for each criteria:

Table 4.1 shows the priority of design criteria. We placed special emphasis on correctness and temporal correctness, because without these characteristics, the system cannot be used at all. The first one concerns the fulfilment of requirements. Analysis defined to be the specification of the essential aspects of the system required by customers. That is these aspects without which the system is considered incorrect or incomplete. The second one is essential to real-time systems. In order to verify whether the system violates its specified timing constraints, we will make a schedulability test. Of course dynamic testing is also a good method to verify temporal behavior of the software. We also put some emphasis on usability, maintainability, testability and comprehensibility because these characteristics are much-valued in Object Oriented programming. These criteria allow the system’s adaptability, a low cost to locate and fix system defects, an easiness to ensure that the deployed system performs its intended function, and an easy understanding of the system. We gave all other characteristics lower priority, except for reliability and portability which
<table>
<thead>
<tr>
<th>Criterion</th>
<th>Very important</th>
<th>Important</th>
<th>Less important</th>
<th>Irrelevant</th>
<th>Easily fulfilled</th>
</tr>
</thead>
<tbody>
<tr>
<td>Usable</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Secure</td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Efficient</td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Correct</td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
<td></td>
</tr>
<tr>
<td>Reliable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Maintainable</td>
<td></td>
<td>✗</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Testable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Flexible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Comprehensible</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Reusable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Portable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Interoperable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
<tr>
<td>Temporal correctness</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 4.1: Priority of design criteria

are irrelevant because we design a real-time application bound to a specific technical platform.

### 4.2 Architecture

Software architecture is a process of designing the global organization of a software system. The system should model the problem domain and thus implement all functional requirements. The aim of this part is to create a comprehensible and flexible system structure with connections between each component. A component architecture is a structural system view makes the system easier to understand and organize the design work.

**Architecture design**

The component architecture defines the overall system structure and models relationships between components as dependencies. These dependencies can be realized in different ways (aggregation, specialization, ...). We defined a layered architecture with 5 components as described below:
The layers in the embedded software is shown in Figure 4.1.

**Device (System interface):** This layer takes care about the control of the different devices in system.

**Logic Check (Function):** This layer takes care of the control of the system in order to implement all functional requirements (temperature regulation, interferometer measurement, monitoring, ...).

**Data (Model):** This layer takes of the storing the state of the problem domain, used by and generated by the functional layer.

**Communication (Technical platform):** This layer implements the system interface to communicate with other network devices.
4.3 Detailed Design

Afterwards, we designed the very high level structure of the system. We can work down to detailed decisions. A component is any piece of software or hardware which has a clear role in the system. It can be isolated, allowing to replace it with a different component that has equivalent functionality. In figures below, we show the details of the different classes (components) with their specifications and interfaces. Following its our specification of the classes that contain essential attributes and non-trivial operations.

- Device layer

Figure 4.2: Class Diagram of the device layer

TemperatureSensor
- Purpose: Control the temperature sensor.
- Operations: read the temperature

Thermostat
- Purpose: Control a thermostat to heat/cool.
- Operations: update the level of the thermostat

Mirror Engine
- Purpose: Control the motion of the mirror.
- Operations: update/read the position of the mirror

InfraRed
- Purpose: Control the infrared.
- Operations: turn on/off infrared
InfraRedDetector
- Purpose: Control the infrared detector.
- Operations: turn on/off infrared detector, read the measurement

ExternalCommunicationDevice
- Purpose: Control the communication device for all external communications.
- Operations: send/receive data
- Logic layer

Figure 4.3: Class Diagram of the logic layer

**TemperatureReading**
- Purpose: Read and store the temperature.
- Attributes: thread id, temperature sensor, temperature buffer, message.

Figure 4.4: TemperatureReading Activity Diagram
TemperatureRegulation

- Purpose: Regulate the temperature according to the stored temperatures.

- Attributes: thread id, heater, cooler, mode, temperature buffer, temperature reference, pid to compute the new temperature, message sent to the logger.

Figure 4.5: TemperatureRegulation Activity Diagram
Monitoring

- Purpose: Check temperatures and mirror speed in order to send errors if values are not between its bounds.
- Attributes: temperature buffer, message.

Figure 4.6: Monitoring Activity Diagram
**Acquirement**

- **Purpose:** Start an acquirement of an interferogram.
- **Attributes:** mirror engine, infrared, infrared detector, message.

![Acquirement Activity Diagram](image)

*Figure 4.7: Acquirement Activity Diagram*
Figure 4.8 shows the scanning process:

![Scanning process Activity Diagram](image)

Figure 4.8: Scanning process Activity Diagram

Figure 4.9 shows the finalisation process:

![Finalisation process Activity Diagram](image)

Figure 4.9: Finalisation process Activity Diagram
Figure 4.10 shows the interrupting process:

Figure 4.10: Interrupting process Activity Diagram
InputCommunication

- Purpose: Receive message from the external communication device.
- Attributes: service which provides the communication, byte buffer to receive data, message sent to the logger or command received.

Figure 4.11: InputCommunication Activity Diagram
OutputCommunication

- Purpose: Send message (error, warning, information or interferogram) to the external communication device.
- Attributes: service, byte buffer to send data, log to send, message.

Figure 4.12: OutputCommunication Activity Diagram
• Check layer

Figure 4.13: Class Diagram of the check layer

Watchdog

- Purpose: Check if all threads are always alive and have not missed their deadlines and log an error whether a thread didn’t notify its aliveness.

Figure 4.14: Watchdog Activity Diagram
AlivenessRegister

- Purpose: Register information about the aliveness of all thread.
- Attributes: list which contains how many notification has been made by each thread.
- Operations: update the notification count of a thread, read the list of threads state
- Data layer

**AcquirementRegister**
- Purpose: Register information about the acquirement (elapsed time, number of measurements, min and max of the mirror speed, ...).
- Attributes: initial time, measurement count, failed measurement count, minimal and maximum mirror speed.
- Operations: read elapsed time, update/read measurements count, update/read min and max of the mirror speed

**AcquirementMode**
- Purpose: Register the mode of the acquirement.
- Attributes: mode.
- Operations: update/read the mode of the acquirement

**Interferogram**
- Purpose: Register all measurements taken by the acquirement and compute the final interferogram.
- Attributes: state, table of all measurements, table of the interferogram's values.
- Operations: update the interferogram's state, compute the interferogram, add a measurement, initialize the interferogram, give the interferogram values in byte

**Logger**

Figure 4.15: Class Diagram of the data layer
– Purpose: Register all information which have to be sent by the external communication device.
– Attributes: a table of all logs.
– Operations: add a new log, read and remove the next log to send

Log
– Purpose: Register any information.
– Attributes: status of the message (error, warning, information,...), message.
– Operations: initialize a log, read status/message

TemperatureBuffer
– Purpose: Register temperature values read by the sensor
– Attributes: buffer to store temperature values.
– Operations: add a new temperature value, read the average of the temperature values

PID
– Purpose: Compute the Proportional Integral Derivative.

MonitorRegister
– Purpose: Register the environment state.
– Attributes: ready (true/false according to environment ready/not ready).
– Operations: read/update the environment state
Communication layer

- **CDLCService**
  - Purpose: Send and receive data on a socket.
  - Attributes: input and output stream to send data.
  - Operations: send/receive data

Figure 4.16: Class Diagram of the communication layer
Chapter 5

Schedulability

The development of an embedded real-time application supposes that it needs to be scheduled. First, we know that scheduling is based on three components:

- an algorithm for allocating the resources (scheduling mechanism)
- an algorithm for ordering access to resources (scheduling policy)
- a means of predicting the worst-case behavior of the system when the policy and mechanism are applied (schedulability analysis)

5.1 Assessment

5.1.1 Scheduling mechanism

We will use the Fixed Priority Scheduling (FPS) mechanism in a preemptive scheme for this purpose, a common way to schedule real-time systems. A static-priority algorithm assigns all priorities at design time, and those priorities remain constants for the lifetime of the task: we are in this case [8].

5.1.2 Scheduling policy

Theoretical approach

We have to follow five assumptions [6] in order to make the schedulability analysis possible. These assumptions are as follows [8], p. 48; we can underline that “Not all of these assumptions are absolutely necessary, and the effects of relaxing them will be discussed in a later section.”

1. The requests for all tasks for which hard deadlines exist are periodic, with constant interval between requests.
2. Deadlines consist of run-ability constraints only – i.e. each task must be completed before the next request for it occurs.
3. The tasks are independent in that requests for a certain task do not depend on the initiation or the completion of requests for other tasks.
4. Run-time for each task is a constant for that task and does not vary with time. Run-time here refers to the time which is taken by a processor to execute the task without interruption.

5. Any aperiodic tasks in the system are special; they are initialization or failure recovery routines; they displace periodic tasks while they themselves are being run, and do not themselves have hard, critical deadlines.

The deadline monotonic conditions

It exists some scheduling algorithms, as Rate Monotonic (RM) policy. We can try ([5] p.474) to use the Deadline Monotonic policy [4]. We cannot use RM for instance because it needs deterministic deadlines exactly equal to periods, and we have some tasks with a deadline different from the period. Both (RM and DM) could be supported on the implementation of the Ravenscar Java Profile as said in section 6.1 of [11]. We chose to use a Wikipedia article as a start to verify some conditions of the Deadline monotonic scheduling. From http://en.wikipedia.org/wiki/Deadline-monotonic,

1. All tasks have deadlines less than or equal to their minimum inter-arrival times (or periods). This condition is satisfied, according to the next section.

2. All tasks have worst case execution times that are less than or equal to their deadlines. We can assume this condition is satisfied.

3. All tasks are independent and so do not block each others execution (for example by accessing mutually exclusive shared resources). We have to consider this condition in our Response Time Analysis by using a “Blocking” member in the equations.

4. No task voluntarily suspends itself. We do not have this kind of action in the system. This condition is satisfied.

5. There is some point in time, referred to as a critical instant, where all of the tasks become ready to execute simultaneously. Yes, we have this critical point in the system, at the start. This condition is satisfied.

6. Scheduling overheads (switching from one task to another) are zero. According to the manual of aJ-100 [3], Thread to thread yield in less than 1 μs. In our system, the smallest deadline is 200 μs. We can assume that the Scheduling overhead is negligible, so can be compared to zero. This condition is satisfied.

7. All tasks have zero release jitter (the time from the task arriving to it becoming ready to execute). We can assume this condition is satisfied.

We will use DM [5], p.484 (Scheduling) to give priorities to our tasks, because when \( D < T \), Deadline Monotonic priority ordering is optimal. The general principle is the fixed priority is decided the following way [7]:

\[
D_i < D_j; P_i > P_j
\]
Priorities of processes

See Table 5.1 for an example using the DMS. Let’s assume that T is the period or inter-arrival minimal time, D the deadline, C the worst case computation time, P the priority and U the Utilization.

Note: If there was no deadlines specified for one task we assumed that it was equal to his period (for a periodic task) or minimal inter-arrival time (for a sporadic task) according to [10]. Moreover we decided when deadlines were equals between tasks to assign a arbitrary priority.

<table>
<thead>
<tr>
<th>Task</th>
<th>Type</th>
<th>T (ms)</th>
<th>D (ms)</th>
<th>C (ms)</th>
<th>P</th>
<th>U</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquirement</td>
<td>sporadic</td>
<td>0.333</td>
<td>0.2</td>
<td>0.04</td>
<td>13</td>
<td>0.2</td>
</tr>
<tr>
<td>Temperature</td>
<td>periodic</td>
<td>1000</td>
<td>100</td>
<td>2.5</td>
<td>9..12</td>
<td>0.025x4</td>
</tr>
<tr>
<td>regulation (×4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Temperature</td>
<td>periodic</td>
<td>200</td>
<td>200</td>
<td>1</td>
<td>5.8</td>
<td>0.01x4</td>
</tr>
<tr>
<td>reading (×4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitoring</td>
<td>periodic</td>
<td>333</td>
<td>333</td>
<td>30</td>
<td>4</td>
<td>0.09</td>
</tr>
<tr>
<td>Output</td>
<td>periodic</td>
<td>333</td>
<td>333</td>
<td>15</td>
<td>3</td>
<td>0.045</td>
</tr>
<tr>
<td>communication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input</td>
<td>periodic</td>
<td>500</td>
<td>500</td>
<td>25</td>
<td>2</td>
<td>0.045</td>
</tr>
<tr>
<td>communication</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Watchdog</td>
<td>periodic</td>
<td>8000</td>
<td>8000</td>
<td>80</td>
<td>1</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Table 5.1: table of priorities

5.1.3 Schedulability analysis

Utilisation-based schedulability test

The utilisation schedulability test for DMS Similar to RMS utilisation-based schedulability test, except \( C_i/D_i \) used instead of \( C_i/P_i \) can be done in DM by this formula:

\[
\sum_{i=1}^{n} \left( \frac{C_i}{D_i} \right) \leq n(\sqrt{n} - 1)
\]

If we apply it to our test case:

\[
(0.2 + (0.025 \times 4) + (0.01 \times 4) + 0.09 + 0.045 + 0.045 + 0.01) \leq 0.71
\]

\[
0.53 \leq 0.71
\]

This first step told us that we can still use the DM policy to order our tasks, but we still have to make the response time analysis regarding blocking and shared resources. There is a note about considering blocking in the utilisation-based tests in [5], p.489: “Blocking can also be incorporated in the utilization-based tests, but now each process must be considered individually.” Nevertheless, we can go on the response time analysis.
Response Time Analysis

These are the shared resources we have: TemperatureBuffer(1 to 4), Logger, AlivenessRegister, AcquirementRegister, Interferogram, AcquirementMode. So, we need to compute $B_i$, which is the maximum blocking time that process $i$ can suffer, using the priority inheritance model (to avoid priority inversion). $K$ is the number of critical sections (resources).

$$B_i = \sum_{k=1}^{K} \text{usage}(k,i)C(k)$$

See Table 5.2 for $C(k)$.

<table>
<thead>
<tr>
<th>Name of the shared resource</th>
<th>C (worst-case execution time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TemperatureBuffer4</td>
<td>0,1 ms</td>
</tr>
<tr>
<td>TemperatureBuffer3</td>
<td>0,1 ms</td>
</tr>
<tr>
<td>TemperatureBuffer2</td>
<td>0,1 ms</td>
</tr>
<tr>
<td>TemperatureBuffer1</td>
<td>0,1 ms</td>
</tr>
<tr>
<td>Logger</td>
<td>0,1 ms</td>
</tr>
<tr>
<td>AlivenessRegister</td>
<td>0,1 ms</td>
</tr>
<tr>
<td>Interferogram</td>
<td>0,01 ms</td>
</tr>
<tr>
<td>AcquirementRegister</td>
<td>0,04 ms</td>
</tr>
<tr>
<td>AcquirementMode</td>
<td>0,01 ms</td>
</tr>
</tbody>
</table>

Table 5.2: Worst-case execution time for each shared resource

The worst blocking time ($B_i$) for each process is in Table 5.3. We followed the basic model to compute $B_i$ proposed in [5], p.489.

We will use this formula to compute the response time of each tasks (we have to consider the blocking $B_i$):

$$R_i = C_i + B_i + \sum_{j=1}^{i-1} \frac{R_j}{T_j} C_j$$

Applied to our example we can have the results (all results are in milliseconds) in Table 5.4.

5.2 Conclusion

All the results from the assessments (Utilisation test, Response time analysis) prove that the system is schedulable, or at least, seems to be a priori. The worst response time of each process is always less or equal to its deadline, so that it indicates us that it is impossible to miss a deadline, with all the values we have defined. We considered that we have chosen a quite pessimistic computation time for each process; in the real case we expect to find lower values. Anyway, the utilisation and response time analysis will be also done a posteriori in the implementing and testing process.
<table>
<thead>
<tr>
<th>Process</th>
<th>Shared resources</th>
<th>Worst-case blocking time $B_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acquirement</td>
<td>AcquirementMode&lt;br&gt;AcquirementRegister&lt;br&gt;AlivenessRegister&lt;br&gt;Interferogram&lt;br&gt;Logger</td>
<td>0,16 ms</td>
</tr>
<tr>
<td>Temperature regulation #4</td>
<td>AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#4</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Temperature regulation #3</td>
<td>AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#3</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Temperature regulation #2</td>
<td>AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#2</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Temperature regulation #1</td>
<td>AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#1</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Temperature reading #4</td>
<td>AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#4</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Temperature reading #3</td>
<td>AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#3</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Temperature reading #2</td>
<td>AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#2</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Temperature reading #1</td>
<td>AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#1</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Monitoring</td>
<td>AcquirementMode&lt;br&gt;AcquirementRegister&lt;br&gt;AlivenessRegister&lt;br&gt;Logger&lt;br&gt;TemperatureBuffer#1&lt;br&gt;TemperatureBuffer#2&lt;br&gt;TemperatureBuffer#3&lt;br&gt;TemperatureBuffer#4</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Output Communication</td>
<td>AlivenessRegister&lt;br&gt;Interferogram&lt;br&gt;Logger</td>
<td>0,2 ms</td>
</tr>
<tr>
<td>Input Communication</td>
<td>AcquirementMode&lt;br&gt;AlivenessRegister&lt;br&gt;Logger</td>
<td>0,1 ms</td>
</tr>
<tr>
<td>Watchdog</td>
<td>AlivenessRegister&lt;br&gt;Logger</td>
<td>0 ms</td>
</tr>
</tbody>
</table>

Table 5.3: blocking time for processes
<table>
<thead>
<tr>
<th>Task</th>
<th>Priority</th>
<th>T</th>
<th>C</th>
<th>D</th>
<th>R</th>
<th>B</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Watchdog</td>
<td>1</td>
<td>8000</td>
<td>80</td>
<td>8000</td>
<td>188,1</td>
<td>0</td>
<td>108,1</td>
</tr>
<tr>
<td>Input communication</td>
<td>2</td>
<td>500</td>
<td>25</td>
<td>500</td>
<td>97,32</td>
<td>0,1</td>
<td>72,22</td>
</tr>
<tr>
<td>Output communication</td>
<td>3</td>
<td>333</td>
<td>15</td>
<td>333</td>
<td>69,02</td>
<td>0,2</td>
<td>53,82</td>
</tr>
<tr>
<td>Monitoring</td>
<td>4</td>
<td>333</td>
<td>30</td>
<td>333</td>
<td>51,94</td>
<td>0,2</td>
<td>21,74</td>
</tr>
<tr>
<td>Temperature reading #1</td>
<td>5</td>
<td>200</td>
<td>1</td>
<td>200</td>
<td>10</td>
<td>0,2</td>
<td>15,74</td>
</tr>
<tr>
<td>Temperature reading #2</td>
<td>6</td>
<td>200</td>
<td>1</td>
<td>200</td>
<td>16,74</td>
<td>0,2</td>
<td>15,54</td>
</tr>
<tr>
<td>Temperature reading #3</td>
<td>7</td>
<td>200</td>
<td>1</td>
<td>200</td>
<td>15,58</td>
<td>0,2</td>
<td>14,38</td>
</tr>
<tr>
<td>Temperature reading #4</td>
<td>8</td>
<td>200</td>
<td>1</td>
<td>200</td>
<td>12,76</td>
<td>0,2</td>
<td>11,56</td>
</tr>
<tr>
<td>Temperature regulation #1</td>
<td>9</td>
<td>1000</td>
<td>2,5</td>
<td>100</td>
<td>11,6</td>
<td>0,2</td>
<td>8,9</td>
</tr>
<tr>
<td>Temperature regulation #2</td>
<td>10</td>
<td>1000</td>
<td>2,5</td>
<td>100</td>
<td>11</td>
<td>0,2</td>
<td>6,36</td>
</tr>
<tr>
<td>Temperature regulation #3</td>
<td>11</td>
<td>1000</td>
<td>2,5</td>
<td>100</td>
<td>5,92</td>
<td>0,2</td>
<td>3,22</td>
</tr>
<tr>
<td>Temperature regulation #4</td>
<td>12</td>
<td>1000</td>
<td>2,5</td>
<td>100</td>
<td>3,1</td>
<td>0,2</td>
<td>0,4</td>
</tr>
<tr>
<td>Acquirement</td>
<td>13</td>
<td>0,333</td>
<td>0,04</td>
<td>0,2</td>
<td>0,16</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.4: Response Time of the processes
Chapter 6

Implementation

This part discusses implementation.

6.1 Software usage

We use Charade and JEMBuilder.

Charade  Charade is the software provided by aJile Systems to communicate with the board.

6.2 Source code

/*
 * Logger.java
 */

package org.ftir.data;

/**
 * Defines a <code>Logger</code>.
 *
 * @version 1.0.0 26/10/2006
 * @author Thomas Baron
 * @author Gael Mercier
 * @author Philippe Jean
 */
public class Logger {

/**
 * Defines the default count of <code>Log</code> contained by a
 * <code>Logger</code> instance.
 */

41
static final public int DEFAULT_CAPACITY = 128;

static private Logger instance;

private Log[] logs;
private int logCount;
private int readIndex;
private int writeIndex;
private StringBuilder buffer;

/**
 * Constructs a new <code>Logger</code> with a default size.
 * @see org.ftir.data.Logger#DEFAULT_CAPACITY
 */
public Logger() {
    this(DEFAULT_CAPACITY);
}

/**
 * Constructs a new <code>Logger</code> with the specified size.
 * @param capacity a positive integer.
 */
public Logger(int capacity){
    Log[] logs = new Log[capacity];
    for(int i = capacity;--i >= 0;) {
        logs[i] = new Log();
    } // for i
    this.logs = logs;
    buffer = new StringBuilder(Log.DEFAULT_MESSAGE_CAPACITY);
    instance = this;
}

public String toString() {
    return null;
}

/**
 * Returns the current instance of this one.
 * @return a <code>Logger</code> instance.
 */
static public Logger getInstance() {
    return instance;
}
public int getCapacity() {
    return logs.length;
}

public synchronized int getLogCount() {
    return logCount;
}

public synchronized boolean hasLog() {
    return (logCount > 0);
}

public synchronized boolean addLog(Log log) {
    if(log == null) {
        return false;
    } // if
    log.getMessage(buffer);
    return addLog(log.getStatus(),buffer);
}

/**
 * Adds a <code>Log</code> into this one. The <code>Log</code> is decomposed into a status and a message.
 */
   * @param status a status.
   * @param message a message.
   * @return true if the specified <code>Log</code> is added into this one, false otherwise.
   * @see org.ftir.data.Status
   */
public synchronized boolean addLog(int status, StringBuilder message) {
    if ((message == null) || (logCount >= logs.length)) {
        return false;
    } // if
    logs[writeIndex].initialize(status, message);
    writeIndex = (writeIndex + 1) % logs.length;
    logCount++;
    return true;
}

/**
 * Extracts and removes a <code>Log</code> from this one.
 * @param log a <code>Log</code> used as returned value.
 * @return true if the argument is updated, false otherwise.
 */
public synchronized boolean removeLog(Log log) {
    if ((log == null) || (logCount == 0)) {
        return false;
    } // if
    log.initialize(logs[readIndex]);
    readIndex = (readIndex + 1) % logs.length;
    logCount--;
    return true;
}

/**
 * Removes a <code>Log</code> from this one.
 * @param log the result <code>Log</code>.
 * @deprecated you should use <code>Log.removeLog(Log)</code> instead.
 * @see org.ftir.data.Logger#removeLog(Log)
 */
public void getNextLog(Log log) {
    removeLog(log);
}
} // class ------------------------------- Logger
Chapter 7

Testing

This part discusses the testing phase. Testing is the process used to check the correctness, completeness, security and quality of developed application. In our case, we have developed a set of tests in order to verify the robustness of the embedded real-time application.

There are two major approaches to testing. The first one is called black box testing\(^1\) and the second one is called white box testing\(^2\).

**Black box testing**  Black box testing alludes to tests that are conducted at the software interface. Test cases demonstrate that software functions are operational, that input is properly accepted and output is correctly produced. Black box testing requires the software engineer to derive sets of input conditions that will fully exercise all functional requirements for a program.

**White box testing**  White box testing alludes to tests that are conducted with knowledge of the program structure and it exercises every logical path of the software. Using white box testing methods, the software engineer produces test cases that guarantee that all independent paths within a module have been exercised at least once, exercise all logical decisions on their true and false sides, execute all loops at their boundaries and within their operational bounds and exercise internal data structures to assure their validity. All these tests should give expected results.

### 7.1 Unit Testing

Unit testing aims at finding defects in individual classes. Unit test is procedure used to validate that a particular module of source code is working properly. Ideally, each test case is independent from the others; auxiliary objects can be used to assist in testing a module in isolation. Besides, unit testing provides a sort of living document. Clients and other developers looking to learn how to use a module can look at its unit test to determine how to use it to fit their needs and gain a basic understanding of the interface.

---

\(^1\)Term for functional testing

\(^2\)Term for structural testing
7.1.1 Protocol

For our application, unit tests were developed for every relevant module. Unit testing were not conducted for the functional part because that is covered by the first level of integration testing. For the rest of the application, unit testing were conducted using the library JUnit. For each tested class, an other is created named in the same way with the suffix Test. Each test class can be run independently from the others.

7.1.2 Example

Class Logger were tested as well as other classes of package org.ftir.data. First of all, a class called LoggerTest were created. This class derives class TestCase of library JUnit. Then, three test methods were developed in order to validate the whole class by passing through all logical paths. Figure 7.1 shows classes Logger and LoggerTest. Figure 7.2 shows the execution trace of class LoggerTest.

![Logger and LoggerTest classes](image)

Figure 7.1: Classes Logger and LoggerTest

```plaintext
Time: 0.11
OK (3 tests)
```

Figure 7.2: LoggerTest standard output

---

3Package org.ftir.logic and class org.ftir.check.Watchdog
7.2 Integration Testing

Integration testing is the phase in which individual software modules are combined and tested as a group.

7.2.1 Protocol

In our application, integration testing were conducted for every function\(^4\). Unit testing has already covered the model layer. First of all, in order to test a function, a sequence diagram were drawn to describe expected message transfers between the function and other classes. Then, a test class is created to run once the function. Finally, the test class is run with a trace library to get the sequence of method calls. To validate the test, the sequence diagram and the method call trace have to be compared manually. No automatic test is used for integration testing.

7.2.2 Example

Figure 7.3 shows the sequence diagram of function TemperatureRegulation. Figure 7.4 shows the command line to get the method call trace. Figure 7.5 shows the trace of method calls. That is relevant at the level of method run. Indeed, we can see expected messages: getAverage, compute, setLevel, setLevel and notify.

Figure 7.3: Scenario Temperature Regulation

7.3 Acceptance Testing

Something about acceptance testing.

\(^4\)Package org.ftir.logic and class org.ftir.check.Watchdog
java -jar ..\lib\trace.jar -classpath ..\classes
-noID -exclude StringBuilder -exclude DeviceException
org.ftir.logic.TemperatureRegulationTest

Figure 7.4: Trace command line

|org.ftir.logic.TemperatureRegulationTest.main([Ljava.lang.String;)
| |org.ftir.data.Reference.<clinit>()
| |org.ftir.data.Reference.<clinit>
| |getHeater(org.ftir.device.impl.TemperatureDevices)
| |getHeater=org.ftir.device.impl.TemperatureDevices$HeaterCoolerImpl
| |getCooler(org.ftir.device.impl.TemperatureDevices)
| |getCooler=org.ftir.device.impl.TemperatureDevices$HeaterCoolerImpl
| |getValue(org.ftir.data.Reference)
| |getValue=34.0
| |run(org.ftir.logic.TemperatureRegulation)
| |getAverage(org.ftir.data.TemperatureBuffer)
| |getAverage=0.0
| |compute(org.ftir.data.PID, -34.0)
| |compute=-68.0
| |setLevel(org.ftir.device.impl.TemperatureDevices$HeaterCoolerImpl, 0)
| |setLevel
| |setLevel(org.ftir.device.impl.TemperatureDevices$HeaterCoolerImpl, 3)
| |setLevel
| |org.ftir.check.AlivenessRegister.getInstance()
| |org.ftir.check.AlivenessRegister.getInstance=org.ftir.check.AlivenessRegister
| |notify(org.ftir.check.AlivenessRegister, 8)
| |notify
| |run
|org.ftir.logic.TemperatureRegulationTest.main

Figure 7.5: TemperatureRegulationTest standard output
Chapter 8

Conclusion

This is the conclusion.
Bibliography


