DEVELOPMENT OF MODERN APPLICATION FOR IN-VITRO DIAGNOSTICS

Petr Fiala, Michal Rost, Vladimír Španihel, Miroslav Virius
Czech technical university in Prague, Faculty of nuclear sciences and physical engineering

ABSTRACT:

This paper presents a software application for manipulating new generation of a product line of thermocyclers. A device called thermocycler is used for performing DNA amplification using PCR [4] method. As time passes by, thermocyclers are getting smaller and more complicated. Therefore, also a need of more sophisticated control software arises. This is a real-time application that must be able to run smoothly under various operating systems. Among other requirements, it has to be also robust (able to run without break-ins), must log significant events and send log files to be processed on distributor's master server. Last but not least, the application must provide modern and cross-platform GUI that adjusts for every user role. We present the design of the architecture and some implementation results.

KEYWORDS:
In-Vitro diagnostics, thermocycler, C++, cmake, Qt framework

INTRODUCTION

Thermocyclers are widely used in biomedicine or food industry. Their main purpose is to detect whether given DNA sequence contains desired subsequence or not. The process of this detection consists of three elementary steps: *DNA isolation*, *DNA amplification* and *spectroscopy analysis*. Many present-day thermocyclers are capable of performing last two steps, but not all three together. Our company is preparing new generation of device, which will be able to do all three steps in a single run.

Competitive software products for manipulation with thermocyclers are made-to-measure for existing devices. Each device producer has its own communication protocol, so there is a need to develop our own control software that will support all required features. Among all there is a strong demand for software that will run under various operating systems.

1. ACTUAL SITUATION

Present-day thermocyclers are developed by several biomedical companies like *Bioterm/Biotech* or *Corbett*. Each company distributes its own devices with its own specialized control software. Corbett's thermocyclers, which are widely used in Czech laboratories, are provided with Rotor-Gene control software.

1.1 Rotor-Gene application

Rotor-Gene 1.7.87 is a GUI application from Corbett Corporation, which is used for manipulating the *Rotor-Gene 6000* thermocycler device. Application allows processing and displaying of real time data obtained from the device. This application is written in Visual Basic language and can be used only under MS Windows operating system.

Rotor-Gene software detects thermocycler device on COM port and instructs it with time/temperature series called profiles. It is collecting the temperature and the fluorescence of the samples during the run of the device thermocycler. These data are processed with the Rotor-Gene software and the diagnosis predictions are made.

The main advantage of the Rotor-Gene software is that it is capable of running in the virtual

mode (without thermocycler connected) and providing user with the generated testing data. The user interface is complex and very intuitive despite it is organized as MDI application. All profiles can be graphically edited and data that are being collected during the run of the device are plotted in sophisticated graphs in real time.

1.2 TManager and TProfManager

Thermocycler manager 4.11 (TManager) is a software from Bioterm company for manipulation with *T3000*, *TPersonal*, *T1 plus* and *TRobot* thermocyclers. The functionality of this application is similar to Rotor-Gene, but unlike Rotor-Gene, this application does not perform the real time data analysis. Data are collected from device and stored in specified file only. The user interface of this application is very brief and not well-arranged. Profiles can be edited only as a list of text values.

Bioterm provides also a *TProfManager* application for their *TProfessional* thermocyclers. This software allows to control up to 5 thermocyclers simultaneously. Both these products are capable to run under MS Windows only.

2. TECHNOLOGIES USED

The discussed application is developed with usage of waterfall model methodology. Requirements were collected, specification and design were prepared using of UML 2.0 modeling language in the VisualParadigm suite. Application is being implemented in C++ [1] language (CEO's request). Since the C++ does not provide native GUI, threads, thread-safe containers and other required features, Qt4 library [6] is used to support them. For the user interface development, OpenGL and Qwt libraries are also utilized, because the application must provide an interactive GUI supported with the animated 3D models.

The application is being developed under QtCreator IDE and for automated makefile generation *cmake build system* [3, 5] is used. Although qmake is native choice for Qt application, we experienced some serious problem with library linking in qmake. Due to that, we decided to switch to cmake.

3. APPLICATION REQUIREMENTS

3.1 Functional requirements

- *Profile management* - to be able to create and store different temperature profiles and send them to the device
- *Sample management* – to describe the samples and assign them to the device's rotor
- *User role management* – to work with different user roles, only the manager of the laboratory will be allowed to see the diagnosis
- *Real time data processing* - collect the data from the device in real time, transform it in the core, plot it in UI and store it in the data model
- *Service mode* - provide the service mode in which the calibration of the device should be performed

3.2 Other (non-functional) requirements

- *Robustness* - to be able to run without break-ins, all significant events must be logged
- *OS independency* - run under different operating systems
- *Eye-candiness* - support user interface with animations and visualizations, allow graphical

editing of profiles

4. ARCHITECTURE OF THE APPLICATION

During the design process, we divided the application into five modules (see Fig. 1): *common*, *datamodel*, *deviceio*, *core* and *ui*. Utility classes like logger or state machine implementation that are common for other four modules are stored in the *common* module. The *datamodel* contains all the classes that are needed for the manipulation with the profiles and measured data or for the management of physical quantities and units. The *deviceio* module allows application to communicate with the connected device, this module contains classes that wait for device messages, generate signals from these messages and send signals to the rest of the application (other modules). The *core* module is responsible for transmitting the data between the data model and *deviceio*, it also performs some calculations and data processing. The *ui* module contains the user interface of the application, beside dialogs and forms it provides graphical components for the manipulation with the time/temperature plots or for the 3D manipulation with rotor.

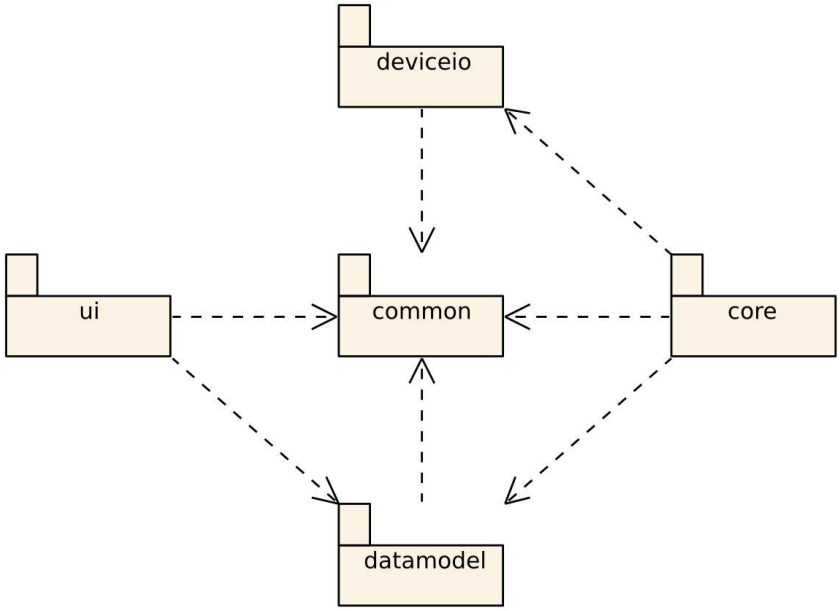

Figure 1: Package diagram - main modules of application

4.1 Architecture of datamodel

The datamodel is composed of several segments. The key part of datamodel takes responsibility for maintenance of *profiles* and *templates* (Fig. 2).

The profiles are time/temperature series that instructs device how to change the temperature as the time passes. There are three basic profiles: *hold*, *melt* and *cycling*. The *hold profile* specifies the temperature and the period, for which it must be maintained. The *melt profile* instructs device to rise the temperature from the staring value up to the end value with a given time/temperature step. Finally, the *cycling profile* instructs the device to repeat several periods, where one period is sequence of *holds*.

The template is a sequence made of various profiles. This sequence will be assigned to every run of the device.

The other packages (submodules) of the data model are *rotor*, *profiles*, *results and model3d*. The *rotor* package is responsible for managing different types of rotors that can be used within the device. A package profile contains classes that hold information about samples (test tubes). Samples should be marked as positive control, negative control, or can be assigned to specific patient. *Results* package is responsible for holding data collected from the device (temperatures, fluorescences, rotor's revolutions). Package *model3d* contains API for manipulating with 3D models. This API is used in GUI of application to manipulate with models of rotor or other visualizations.

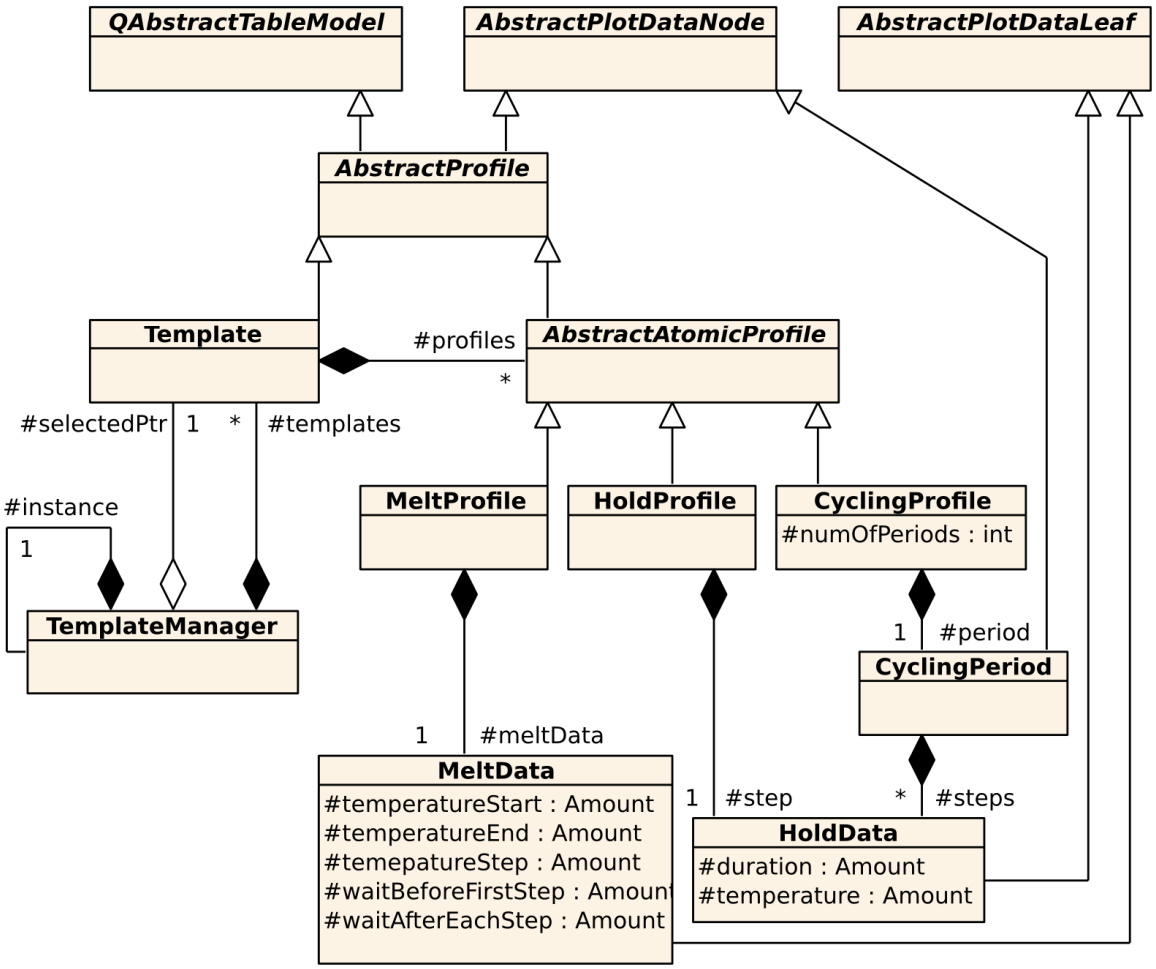

Figure 2: Class diagram for templates/profiles management

4.2 Architecture of deviceio

Since the application should be runnable under Linux and Windows operation systems it must be implemented two ways to communicate with the device. Both types of operation systems are specific but there is also possible to find some similarity.

The important requirement is the robustness. The application has to be able to find whether the device is connected or disconnected at each time. Therefore, the enumeration of device as well as the interchange of the messages is done in an infinite loop in separate thread. Fig. 4 shows the base algorithm to do one step of the loop. This base algorithm is similar for both types of operation systems. The difference is in implementation of concrete actions.

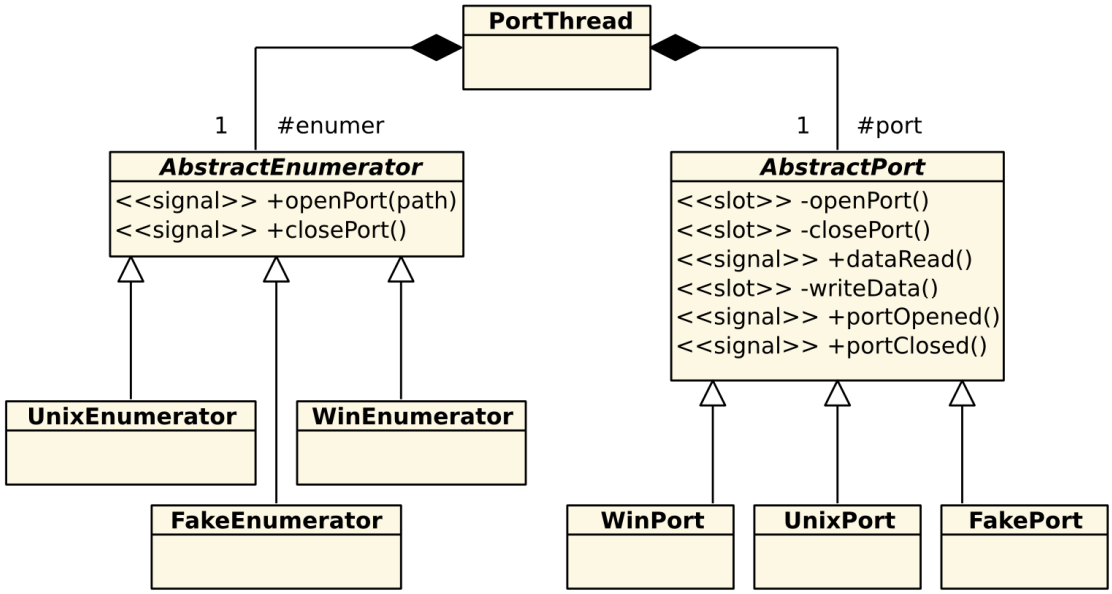

Figure 3: Class diagram for port management

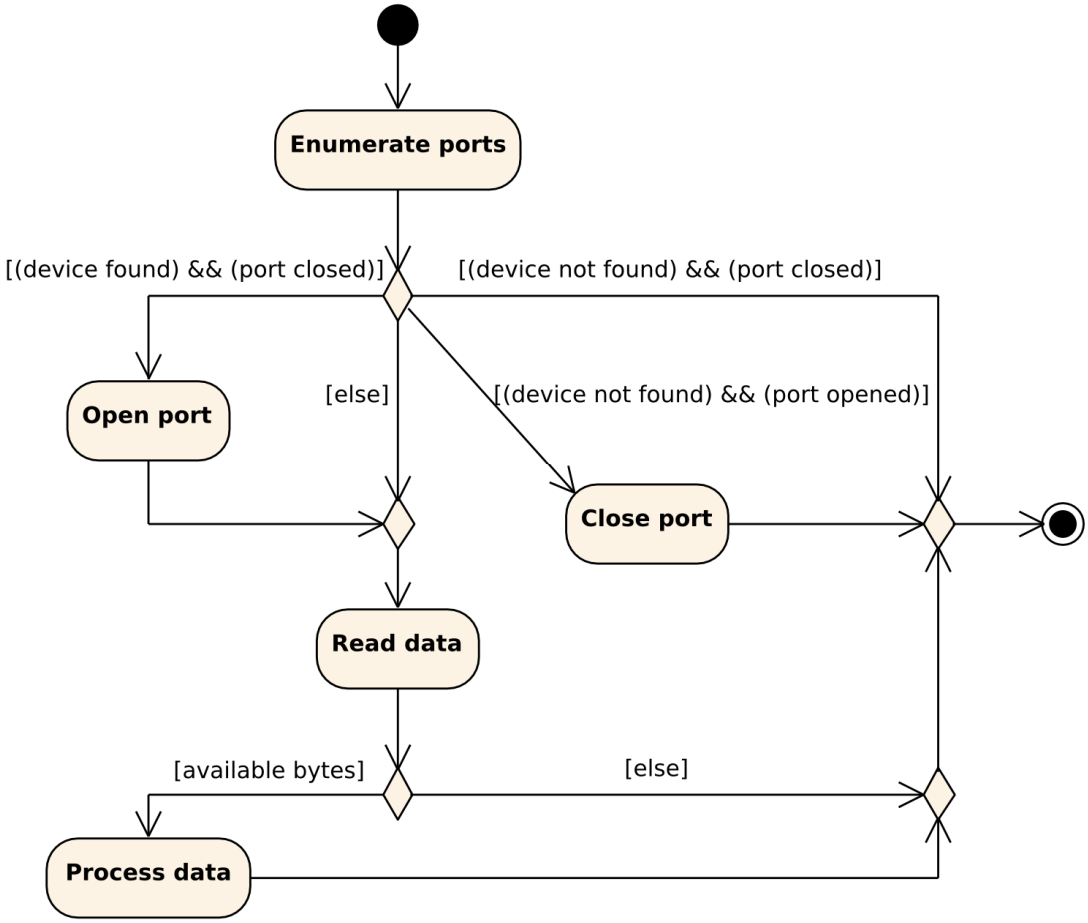

Figure 4: Activity diagram of the one step of the infinite loop

The enumeration is solved by using a Udev library on Linux systems whereas on Windows it is SetupAPI library. It is only an interim solution. The process of enumeration and communication will be replaced by an event-driven thread instead of the infinite loop in the future.

Common USB cable is used for connecting the device to PC. Microcontroller placed on the device is able to transform USB to VCP (Virtual COM Port). It is very useful because it is possible to communicate with the device using simple text communication protocol like with RS-232 port.

4.3. Architecture of common

The *common* module contains classses that can be accessed from all other modules. The most significant part of the common module is the *logger* package (shown in Fig. 5). The main reason for the implementation of the logger is to make the logging of many kinds of messages sent from different parts of the software easier. There are two possible senders of message (*application* and *device*) and four types of messages (*debug*, *warning*, *error* and *critical*).

First implementation of logger contained only one message queue. But this implementation experienced difficulties, when a large number of messages in a small amount of time was sent to the logger. These difficulties resulted in breakdown of the whole application. Therefore the *FWriterThread* class has been introduced. This class acts like a pool of the message queues. When the the currently used queue is full (reaches its maximum capacity), the *FWriterThread* is asked for an empty queue which is substituted for the current one. While an empty queue is being filled, the full one is flushed to the file. In the case that *FWriterThread* is asked for a queue and there is no empty one, new instance of the queue is created.

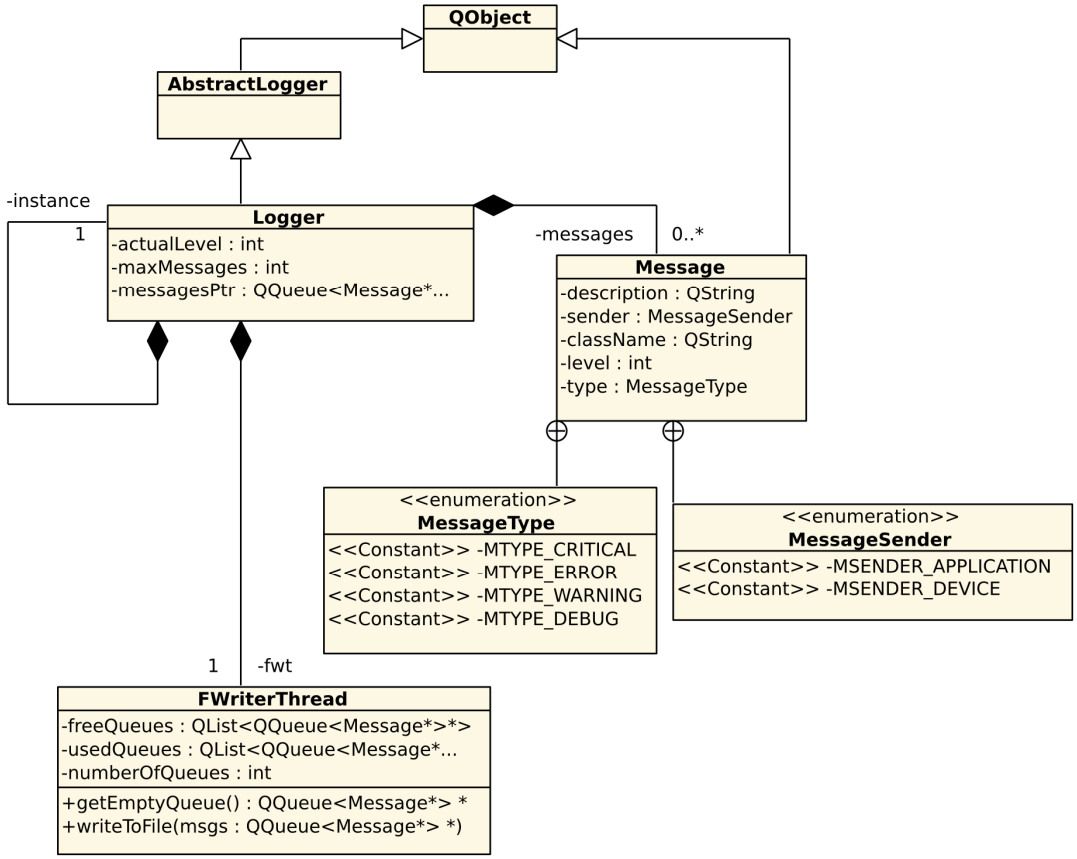

Figure 5: Class diagram of logger

5. RESULTS AND FUTURE DEVELOPMENT PLANS

The application development is so far still in implementation phase and the device is also under heavy development. Our software team is provided with only a prototype of the microcontroller. Even though this microcontroller is not connected with device, it is capable of testing data generation. Our application is capable of receiving, storing and plotting of this data. During the development, application is being tested under Archlinux, Ubuntu Linux, MS Windows XP, MS Windows 7 operating systems.

It is estimated that the whole development will take approximately another half of a year, if everything progresses smoothly. User roles management system must be implemented to fit the datamodel, widgets for controlling and monitoring of the device are still missing in the UI and only virtual mode of *deviceio* and *core* modules are fully implemented. Communication test with device prototype has yet to be performed.

Fig. 6 presents the screenshot of the application prototype on which the 3D model of the rotor with samples can be seen.

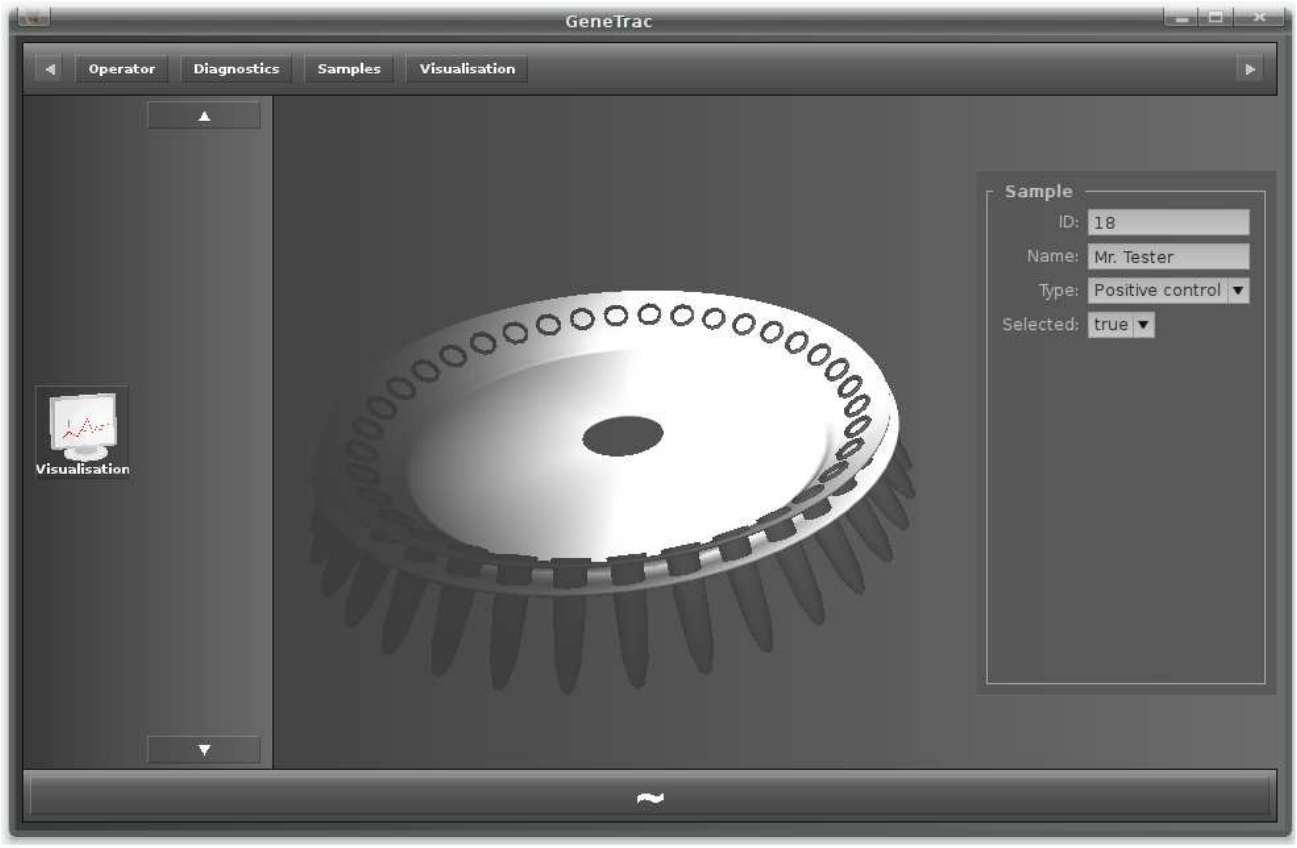

Figure 6: Screenshot of the application

ACKNOWLEDGEMENTS:
This work is supported from grants: MŠMT LA08015 and SGS 11/167.

REFERENCES:

[1] Dirk L.; Mejzlík P.; Virius M. *Jazyky C a C++ podle normy ANSI/ISO*. Praha: Grada Publishing 1999. ISBN 80-7169-631-5

[2] Fries R. C.; King P. H. *Design of Biomedical Devices and Systems - second edition*. CRC Press 2009. ISBN 9781420061796

[3] Hoffman B.; Martin K. *Mastering CMake*. Kitware 2010. ISBN 978-1-930934-22-1

[4] Hunt M. *Real Time PCR*. [online]. 2010-07-01, [cit. 2011–03–27]. Available on WWW: <http://pathmicro.med.sc.edu/pcr/realtime-home.htm>.

[5] Kitware Incorporated. *CMake documentation*. [online]. 2010-12-09, [cit. 2011–03–27]. Available on WWW: <http://www.cmake.org/cmake/help/documentation.html>.

[6] Nokia Corporation. *Qt Reference Documentation*. [online]. 2010-06-09, [cit. 2011–03–27]. Available on WWW: <http://doc.trolltech.com/4.7/>.

[7] Pecinovský R. *Návrhové vzory*. Brno: Computer Press 2007. ISBN 978-80-251-1582-4