IMPROVING THE DATA MODEL OF PCR SOFTWARE

Petr Fiala, Michal Rost[1], Vladimír Španihel
CTU Prague, Faculty of nuclear sciences and physical engineering

ABSTRACT:
The purpose of this study is to improve an existing data model for a real-time biomedical device in order to store and manage more effectively data incoming from the device. Because the standard approach deals with collections of class instances, it is not able to store data uniformly. Moreover, adding new properties to the existing model is quite a complex problem. On the other hand, uniform data storage comes with the problem of effective selection of subsets from the data. This paper presents an architecture and implementation of our approach to solve the mentioned problems as well as the results from its benchmarks.

KEYWORDS:
thermocycler, PCR, C++, Qt framework

INTRODUCTION

Our team involves [2] in the development of an application for thermocycler device. This kind of device is used for performing Polymerase chain reaction (PCR) [4], during which the patient's DNA is being replicated. The replicated DNA is exposed to a beam of laser light, and, finally, the fluorescent light emmited from DNA is collected by photomultiplier and sent to the application for a subsequent analysis.

At present, despite the fact that original requirements for application have almost been implemented, new requirements are continuously emerging. This situation results in a continual need for change of the application's data model. Thus, we were motivated to redesign it in order to be more robust to changes in requirements.

Initial situation

Up to now, every change in the original data model comprised considerable changes in its design and implemention. If, for example, a new attribute is to be added to the existing data model; several tasks have to be performed. First, the UML model must be redesigned. Second, the source code must be reimplemented. Next, import/export of an attribute from/to all supported formats must be implemented. Finally, the newly implemented attribute must be connected to other parts of the application including the GUI.

For instance, Figure 1 shows an UML class diagram of *results*. Results are data measured by the device and sent to the application. These data comprise: current temperature inside the device, current rotation speed of the carousel, and current fluorescence of every sample. During the time results are collected from the device and stored in an appropriate data list: temperature data list, rotation data list, or fluorescence data list. As shown in Figure 1, each set of data, collected during a single run of the device (RunDataSet), contains more than one fluorescence data list. It is because fluorescence is collected for all samples currently placed in the device.

1 Corresponding author; rostmich@fjfi.cvut.cz

After implementation of the results package, new requirement has been formulated. According to this requirement, each sample can be exposed to laser beams with different wavelength (color). If the requirement was added to the original model (figure 1), there would be various things to change. At first, the RunDataSet would hold a matrix of fluorescence lists; that means one fluorescence list for each sample and each laser. Furthermore, export and import of fluorescence would have to be rewritten. And last but not least, the GUI and communication with the device would have to be updated in order to support the required feature.

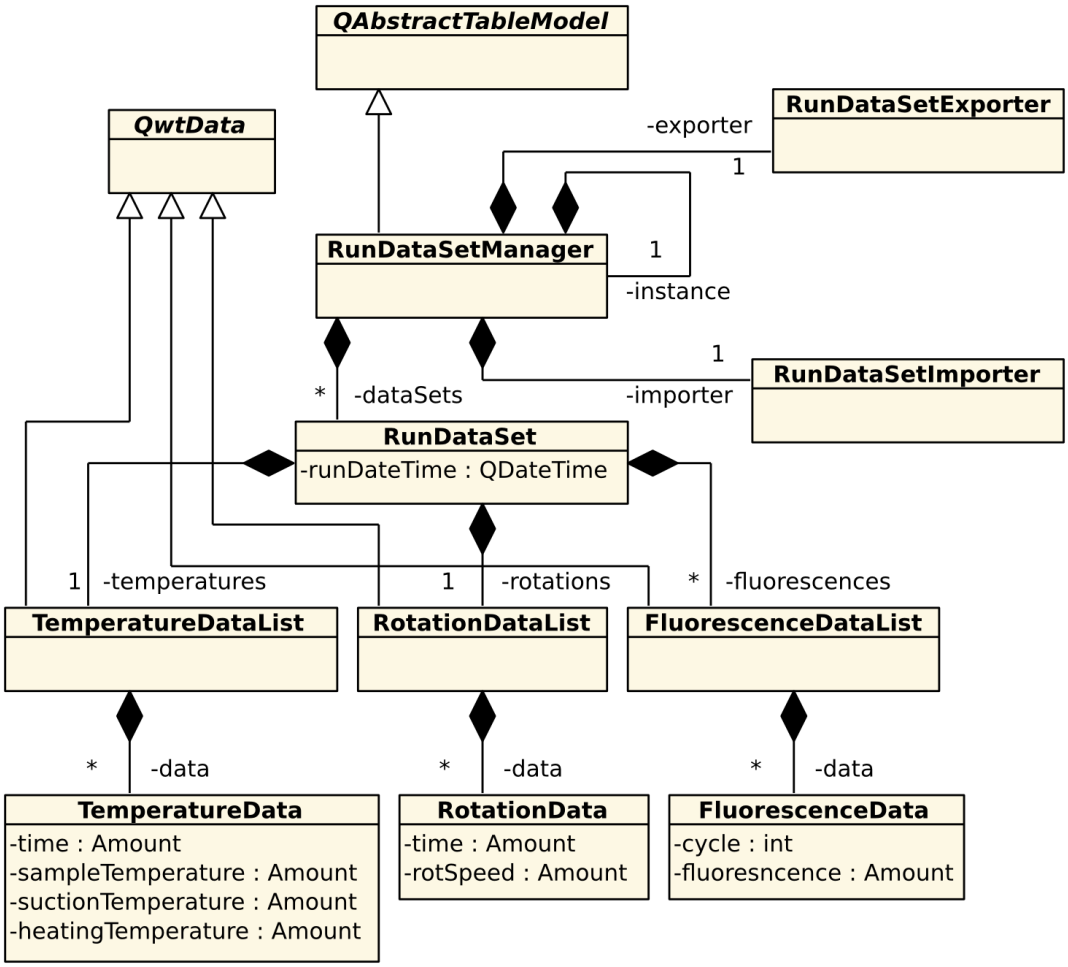

Figure 1: UML class diagram of results package

1. REQUIREMENTS FOR IMPROVEMENT OF THE DATA MODEL

After analyzing the problem, five requirements to be implemented were formulated in order to improve the original data model:

- *Robust to changes in requirements* – notable changes in requirements must produce minor changes in the implementation of the data model
- *Unity* – concentrate the data of same type at the same place
- *Unified interface* – provide all types of data with same interface in order to simplify and

consolidate their management

- *Improved metadata management* – store all necessary metadata directly with corresponding data
- *Improved accessibility* – fast access to the selected subset of the data from all parts of the application

2. ARCHITECTURE OF THE PROPERTIES PACKAGE

2.1. Analytical model

Based on the requirements an analytical model of a package have been developed referred as *properties* (Figure 2). The structure of the package follows the idea that each attribute can be stored in instance of one class: *Value*. All attribute's metadata like information about its type, its physical quantity, its valid range etc. are stored in a corresponding instance of class *Property*. An instance of class *Item* represents one observation on a certain reality, while an instance of *ItemList* represents a list of these observations. In other words, the instance of ItemList is a table where the rows represent different items, and columns represent different properties.

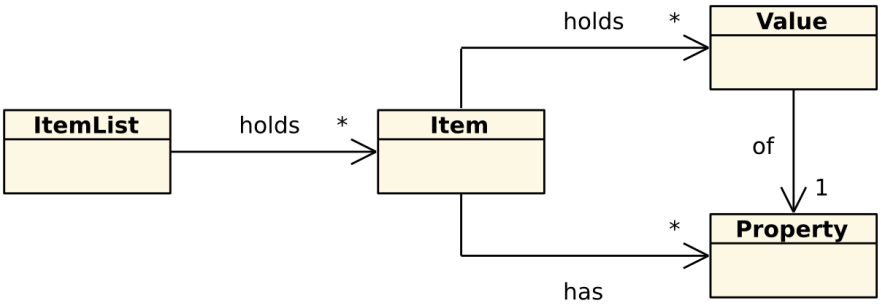

Figure 2: Analytical class diagram of properties package

2.2. Implementation model

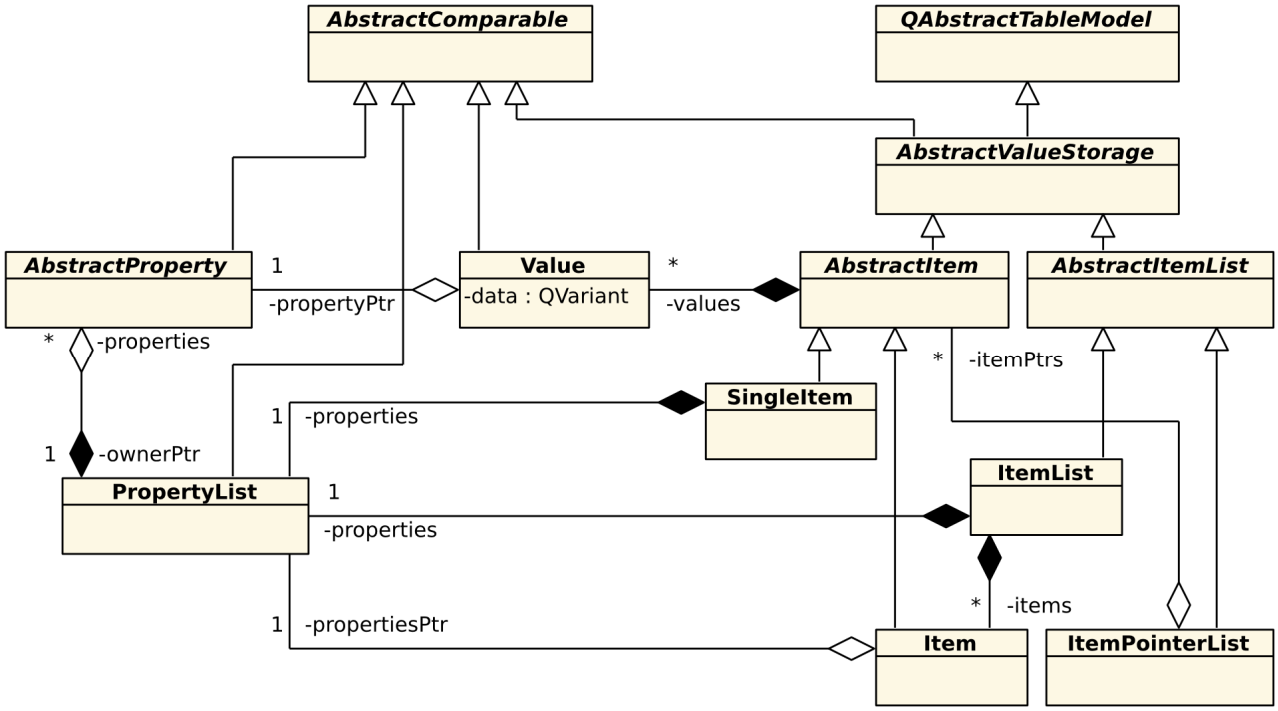

Figure 3: Implementation class diagram of properties package

After formulating the analytical model, we designed an implementation class diagram in order to prepare the module for application.

As shown in Figure 3, the analytical class ItemList is represented by an abstract class *AbstractItemList,* which has two concrete implementations: *ItemList* and *ItemPointerList*. While instances of the ItemList hold their own items, instances of the ItemPointerList hold only pointers to items of other lists. Thus, ItemPointerList is able to store results of selections from some AbstractItemList.

The analytical class Item is represented by an abstract class with two implementations: *Item* and *SingleItem*. Instances of the class Item can be owned only by instances of the class ItemList, and they share the same list of properties (an instance of *PropertyList*). On the other hand, SingleItem represents a standalone list of values of some properties.

As the application is developed in the C++ [1, 7] language and Qt framework [3, 6], we have utilized a Qt's *QVariant* class, which has been utilized for store a value of various data types in a one variable and provides conversions to standard C++ data types.

2.3. Selecting subsets from the ItemList

The *properties* package shown in Figure 3, meets first four requirements that have been formulated in the section 1. However, the last requirement still has to be applied. Since the data are stored uniformly in the instance of ItemList as instances of the Item, a requirement to selecting subsets from this data arises. An example should be made of the results package again. If all fluorescence data were stored in one instance of the ItemList, there a problem would arise with selecting only those fluorescence that correspond to the given sample and laser.

The standard approach to selection of a subset from the list comprises: iteration through all items in the list, testing a specific condition in each iteration, and passing current item to the

subset if the tested condition is met. This operation has linear complexity, but it has to be performed for every sample and every laser every time the new fluorescence is received; so the final complexity is *O(k*m*n)*, where *k* is the number of items, *m* is the number of samples, and *n* is the number of lasers. Moreover, it is necessary to split flourescence data to subsets after each measurement is performed, and since the time available between two measurements is limited, selection of subsets has to be effective. In order to speed up the selection of subsets from measurements, the following assumptions and requirements have been accepted.

Assumptions:

- All measurements are stored in an ordered list; the last measurement is in the bottom of the list
- All measurements should be split to *m* different subsets by *m* different values of one ordinal property
- All measurements should be split to $m_1 \cdot m_2 \cdot \ldots \cdot m_n$ different subsets by m_1, m_2, ..., m_n different values of *n* ordinal properties

Requirements:

- All yhr split subsets must be disjunct
- Each measurement should belong to exactly one subset
- Each measurement should be processed at once
- Finding the right subset for a given measurement must be effective

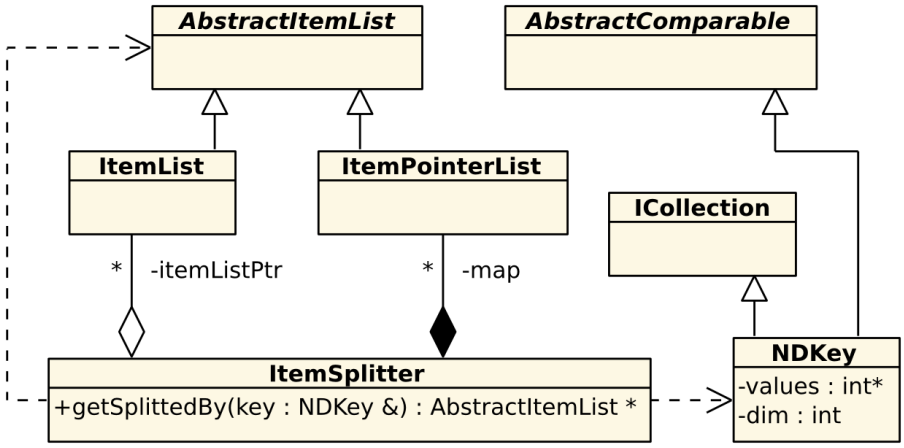

Figure 4: Implementation class diagram with ItemSplitter class

Based on assumptions and requirements a class *ItemSplitter* has been designed (Figure 4). ItemSplitter holds an instance of *QMap* [6] in which individual subsets (instances of ItemPointerList) are stored. Keys for the mentioned QMap are instances of class *NDKey* which represents an *n*-dimensional key.

When a new instance of ItemSplitter is created, a pointer to the original ItemList object has to be specified as well as indices (or names) of properties that will be used for splitting the original ItemList. Then the ItemList object is iterated for once and its items are sorted to appropriate subsets.

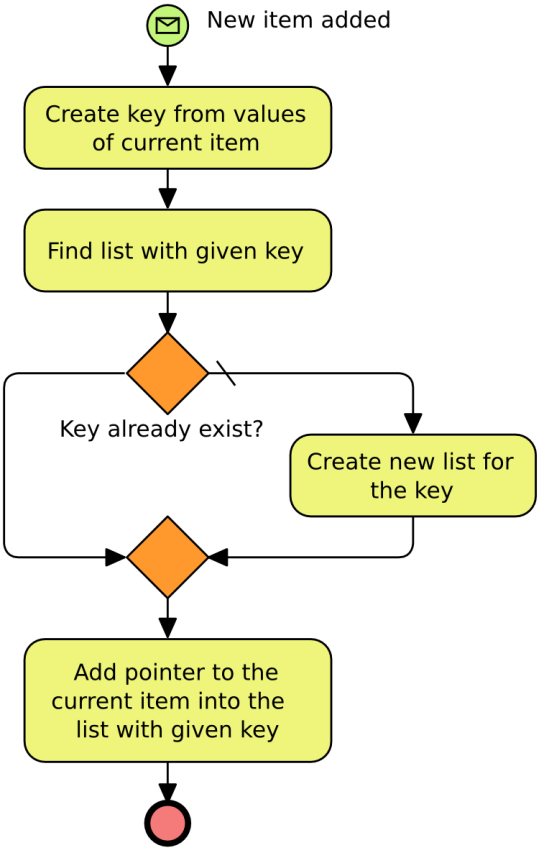

Figure 5: Process of finding of correct subset for an item

After a new measurement is performed by the device, the result is passed as the new item into the corresponding instance of the ItemList, and the instance of ItemSplitter finds the right subset and adds a pointer to the mentioned item to it. This process is described in Figure 5. Since the complexity of finding key in QMap with n items is $O(log(n))$ [6], the complexity of the previously mentioned process is $O(n + log(m_1 \cdot m_2 \cdot \ldots \cdot m_n))$ where n is the number of properties used for splitting, and m_i is the number of different values of the property i.

In case the instance of ItemSplitter is no longer needed, it can be deleted without affecting the original ItemList.

DISCUSSION

Implementing *properties* for the *results* package

After the properties package has been created, it has been used for the reimplementation of the *result* package; its new class diagram can be seen in Figure 6.

All data are now stored in instances of ItemList, so they share the same interface. All functionalities like XML import/export or reporting are written at once for ItemList. Futhermore, if a new functionality were required, it would be written also at once.

If, for example, a new attribute is to be stored together with fluorescence, the new property can be added to the *fluorescence* list, and there is no need to change the design, or

reimplement functionalities like import/export of fluorescence.

All metada data like physical quantities or valid ranges for values are stored together with the properties in the ItemList at once for all items.

Before the device is started, an instance of ItemSplitter is created in order to split the measured fluorescence by indices of samples and indices of lasers. When a new measurement is performed, the only operation that has to be carried out is finding the corresponding subset for the measured item. So, the complexity of splitting is $O(log(n \cdot m))$ where n is the number of samples and m is the number of lasers.

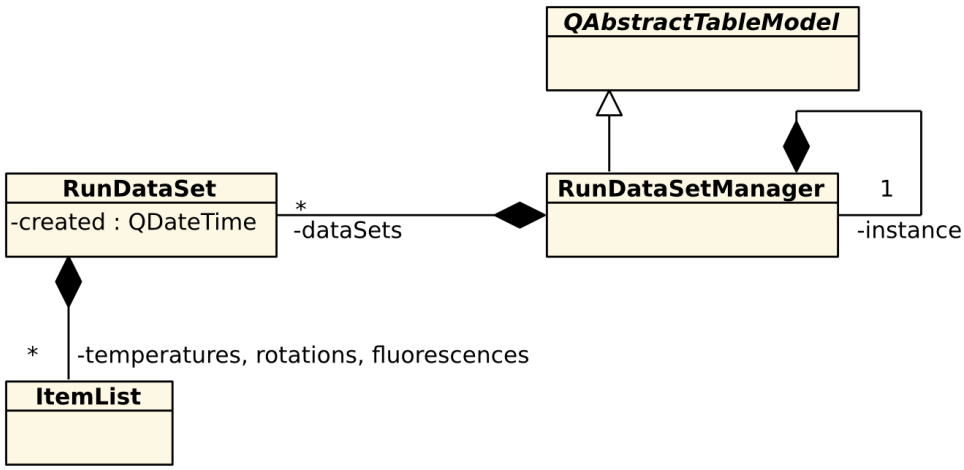

Figure 6: Class diagram of reimplemented results package

Testing the *result* package

The effectivity of message processing was tested with QTestLib [6]. Using this library a benchmark with 1024 iterations was performed for the prepared set of fluorescence data. The average time needed for parsing message from the device, storing it in the ItemList, and splitting it to the appropriate subset was calculated as 0.111 milliseconds. This result is acceptable for our purposes.

CONCLUSION

We have designed and implemented a package *properties* which has allowed us to simplify the original data model of PCR application. Now, all data can be stored and managed in a more uniform way and can be accessed in reasonable time. This approach can be utilized in development of other applications that are designed in our company.

ACKNOWLEDGEMENT

Creation of this paper was partially supported by grants: MŠMT LA08015 and SGS 11/167.

LITERATURE

[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] Fiala P., Rost M., Španihel V., Virius M. *Development of Modern Application for In-Vitro diagnostics*. Ostrava, 2011.

[3] Fiala P., Rost M., Španihel V., Virius M. *Knihovna Qt4, prostředí QtCreator a možnosti*

jejich využití. Ostrava, 2011.
[4] Hunt M. *Real Time PCR*. [online]. 2010-07-01, [cit. 2011–03–30]. Available on WWW: <http://pathmicro.med.sc.edu/pcr/realtime-home.htm>.
[5] Pecinovský R. *Návrhové vzory*. Brno: Computer Press 2007. ISBN 978-80-251-1582-4
[6] Qt Project. *Qt Reference Documentation*. [online]. 2012-03-06, [cit. 2012–03–30]. Available on WWW: <http://qt-project.org/doc/qt-4.8/>.
[7] Virius M. *Pasti a propasti jazyka C++*. Brno: Computer Press 2005. ISBN 80-251-0509-1