Design Patterns of Automated Structure-parametric Identification System

Andrey Pavlov

International Research and Training Center of Information Technologies and Systems of National Academy of Sciences of Ukraine, 03680, Kiev, av. Glushkova, 40, Ukraine

andriypavlove@gmail.com

Abstract. *The paper suggests and describes solution for two well-known problems of object-oriented software design and engineering. The first one is how to design a software to reach independence of GUI part from the rest part of the software. The second – how to design the engine of the software to make it easily to add a new and modify an existent functionality. Suggested design patterns developed by example of GMDH based modeling software and visualized using UML and C++ snippets. The GUI of the system based on the Nokia Qt library.*

Keywords

Design patterns, object-oriented design, object oriented analysis, MCV, UML, Qt, modeling software, data analysis, automated structure-parametric identification system.

1 Introduction

When making a good design of software a bunch of factors should be addressed. It's not new that 70-80% of software life cycle budget consumed by the system support. It includes adjustment and adaptation of the system to certain customer as well as making short- and long-term changes, engendered by changeability of external environment. The changeability can be characterized by announcement of a new technology, disappearance (stop of supporting) of used technology, competition on software market, and so on. All these makes system architects to pay more attention to developing a design that is quite high-level to easily introduce changes to the system. Simplicity of making changes and additions to the system called system flexibility.

2 Problem statement

System flexibility is one of the requirements to the system and determines the way of designing the software. The most important and interesting problems, which are solving here, are:

- separation of graphic user interface (GUI) from software engine, that allows independent changing any of these two parts, and total replacement of engine or GUI with a new one without making any changes to the rest of the code;
- addition of a new functionality to the system in such a way that doesn't require making changes to the basic code.

In this paper, we will describe the solution of these two problems by example of making a design of Automated Structure-parametric Identification System (ASIS) software that purposed to build forecasting, extrapolative and approximation models.

3 Design of software data model and GUI

This problem isn't a new one. One of the most known solutions, which is used in designing of web-applications, and serves as a basis for designing libraries and frameworks, is the *Model-Control-View* design pattern. However, when designing a desktop software *Control* classes are usually redundant, and event handler code is placed in View classes. So, our problem here is to create a set of *Model-View* design patterns.

The GUI is build using Nokia Qt library. All *View* classes can be divided into three categories:

– classes that read data from *Model*;
– classes that change *Model*;
– classes that read/write data to *Model*;

The idea is to introduce a `Link` class between *View* and *Model* classes (Fig. 1). Its responsibility is to update GUI when the *Model* has changed and update the *Model* when a user inputs data via GUI. GUI update performed through abstract method `modelChanged()`, which is called from the *Model* as soon as the *View* should be updated.

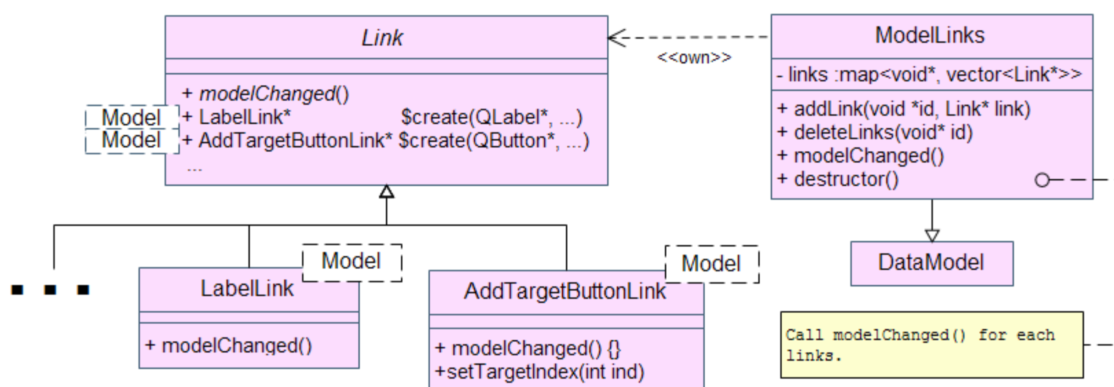

Figure 1. Class `Link`.

Note that `Link` class and inherited classes are templates with a parameter `Model`. Such a construction allows making independent *View* form *Model*, and replace the former without influence on *Model* and vice versa.

3.1 Pattern for reading data from the Model

Consider the `LabelLink` class for reading data (fig. 2).

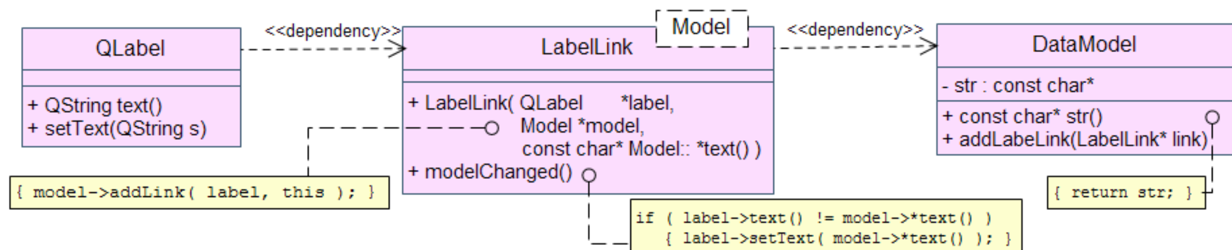

Figure 2. Pattern for reading data from the *Model*

`Qlabel` is a *View* class from Qt library that represents a label. The link is registered in the *Model* by calling `addLink` method in the constructor of the `LabelLink` class. The interface of the `Link` class is extended by the static methods where each of them creates a certain derived class. So, the usage of this pattern in *View* looks like:

```
QLabel label;
DataModel model;
Link::create( &label, &model, &DataModel::str );
```

3.2 Pattern for writing data into the Model

Consider `AddTargetButtonLink` class, that changes the *Model* (fig. 3).

Qt uses signal-slot mechanism to catch and handle user actions. For example, signal "clicked" emitted when a user presses a button. Slot is an event handler function for signal. Method `connect` designed to link any signal to slot. Out task here – is to create a `ButtonLink` class, derived from `QButton` that would handle the press button event, and more over be a template. Unfortunately, Qt doesn't allow creating derived template class due to its inner architecture. Therefore, a `TargetButtonMediator` class suggested introducing between `QButton` and `AddTargetButtonLink` classes. It must be derived from `QObject` class to be able to use signal-slot mechanism.

We can't link mediator class to link class directly, as the former must be template, therefore the link performed through abstract interface `TargetButtonLinkInterface`. Usage in *View* of this pattern looks like:

```
QListWidget  *_varsList;
QToolButton *_addTargetButton;
Link::createAddTarget
(_addTargetButton, _varsList, &_dataModel, &DataModel::setTargetIndex );
```

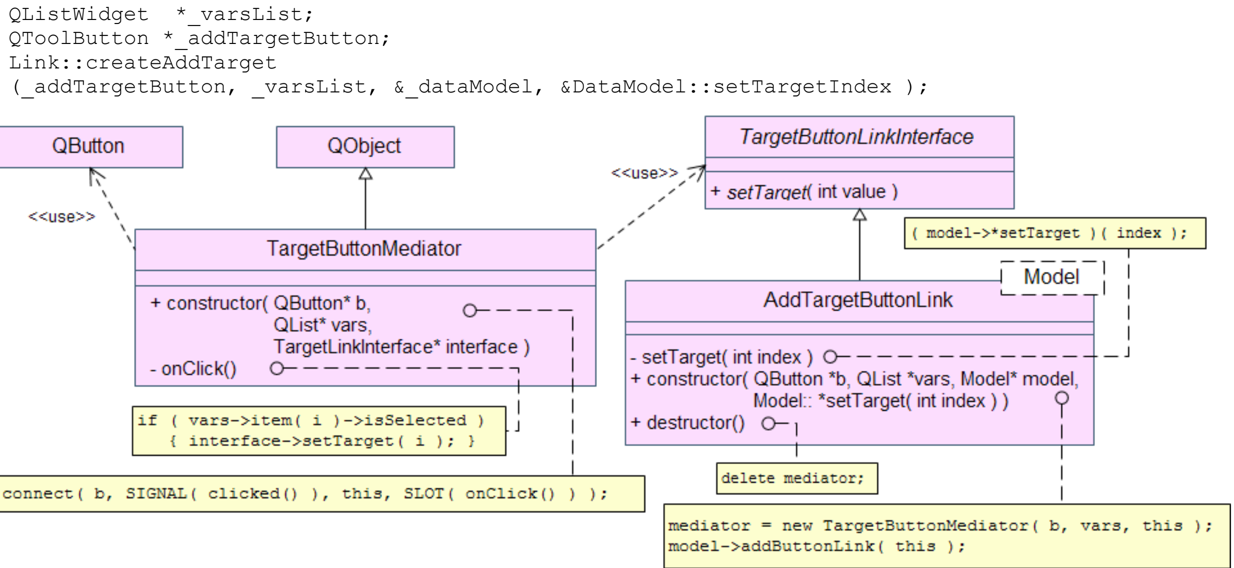

Figure 3. Pattern for writing data into the *Model*

The only drawback of the pattern is necessity to create separate link class for each UI control.

3.3 Pattern for reading/writing data

Consider `SpinLink` class. Object of `QSpin` class is an UI control that enables user to set (reset) *Model* parameter numerically. Two describe above approaches are combined in this pattern (fig. 4).

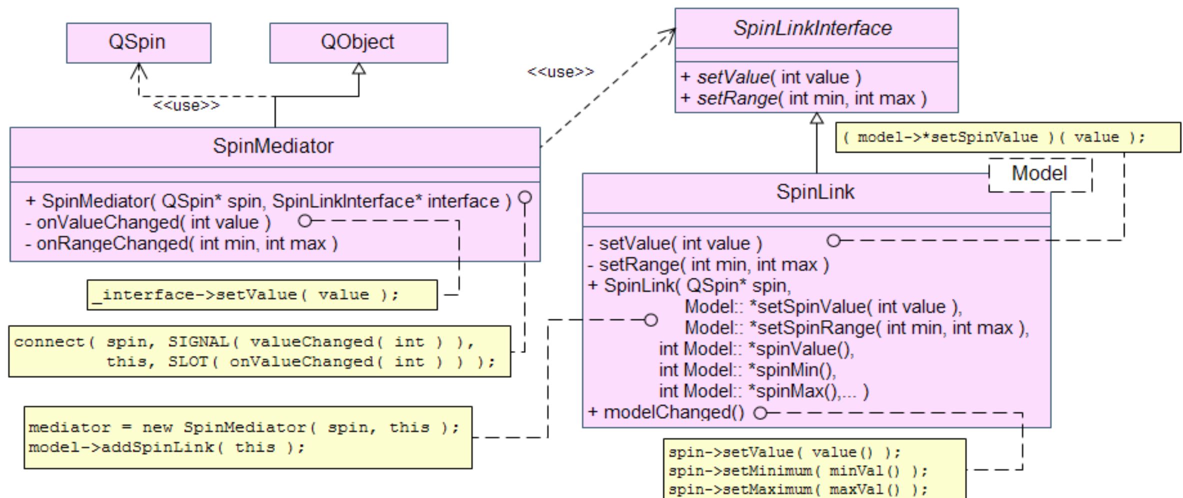

Figure 4. Pattern for reading/writing data

View is updated by the *Model* methods: `spinValue`, `spinMin`, `spinMax`. Data is written in the *Model* by the following sequence of method calls: `valueChanged` (*View*) -> `onValueChanged` (mediator) -> `setValue` (link interface) -> `setValue` (link) -> `setSpinValue` (*Model*). The usage in the *View* looks like:

```
Link::create( _forecast_spin, &_dataModel, &_dataModel,
              &DataModel::setForecastSteps, &DataModel::setForecastStepsRange,
              &DataModel::forecastSteps, &DataModel::minForecastSteps,
              &DataModel::maxForecastSteps );
```

4 Design of the system engine

Any GMDH algorithm consist of four components [1]: model structure generator, parameters estimation algorithm, selection criterion and model class. Work [2] notes that the model class component is not necessary for defining GMDH iterative algorithm, as generation of model class is a king of initial vectors processing procedure that is run before model building process actually starts. Generalized Relaxational Iterative Algorithm (GRIA) [2] has two model structure generators: directed search generator and exhaustive search generator. The iterative GMDH algorithm utilizing exhaustive search generator is quite simple:

1. Generate structure of the models of current iteration.
2. Estimate parameters of these models.
3. Calculate the selection criterion for these models.
4. Select a subset of models that will be passed on the next iteration.
5. Check the stopping rule. If it isn't satisfied, go to the next iteration and repeat steps 1-5.

Performed object oriented analysis of the algorithm separates out the following five components: structure generator, estimation parameters algorithm, selection criterion calculation algorithm, model selection rule, algorithm stopping rule. These components can be represented by corresponding abstract classes, and the whole algorithm is taken as the basic multilayered GMDH algorithm (the term "multilayered algorithm" unions iterative and multistaged GMDH algorithm classes). Depending on the specific conditions, the algorithm can be specified with concrete derived classes. This design pattern is called strategy [3].

Let us concretize the basic algorithm. In our case, the exhaustive search generator of the GRIA is equivalent to the generator of simplified GMDH algorithm [4], the parameters estimation algorithm is the recurrent algorithm [5]. In GRIA three selection criteria are available. One of them is combined and makes a combination of the rest two in such a way, that it can be turned in one of them just by giving certain coefficients. So that is not necessary to create an abstract class for selection criterion, we manage with one class that implements combined criterion.

However, the GMDH algorithm using directed search generator doesn't fall into the structure of the basic multilayered algorithm, as its model structure generation process (step 1 of the algorithm) performs parameters estimation, selection criterion calculation and model selection procedures. But the specifics of the five algorithm components in this algorithm is analogues to the basic one.

The class diagram describing the algorithms of the GRIA and their components represented on the fig. 6.

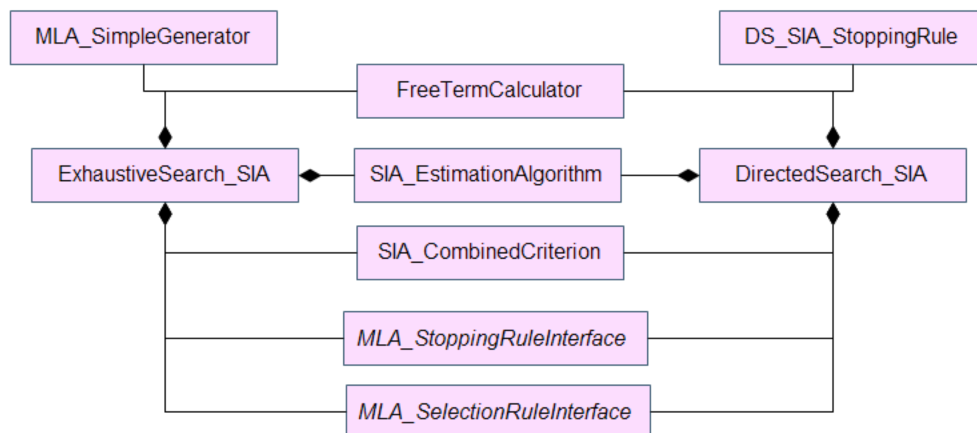

Figure 6. Algorithms of GRIA and their components.

Abbreviations of SIA and MLA stand for Simplified Iterative Algorithm and Multi-Layered Algorithm correspondently. Moreover, the last is wider term than the former. `FreeTermCalculator` class is used to calculate free term of the model, when the model building process done. `DS_SIA_StoppingRule` implements a special stopping rule in the algorithm utilizing directed search generator. The algorithms comprise two abstract classes: `MLA_StoppingRuleInterface` and `MLA_SelectionRuleInterface`, that define the interfaces to work with any rules of multilayered algorithms. The diagram of derived classes for these interfaces depicted on the fig. 7.

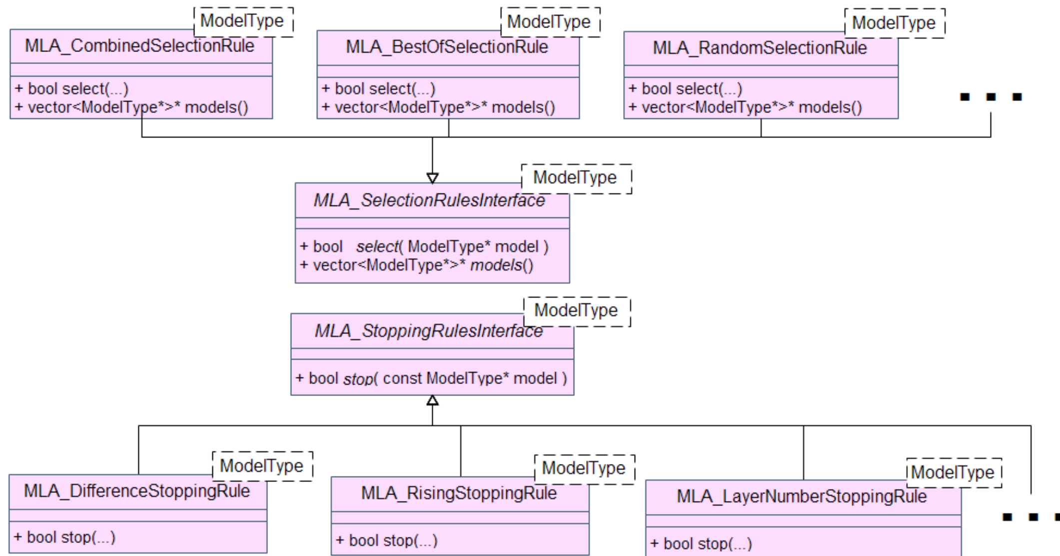

Figure 7. Stopping and selection rules of the GRIA

5 Conclusions

In the paper solved two the most popular and important problems of software design:

– created a set of design patterns that make the GUI classes totally independent form the rest of the system;
– the design of the engine of the GMDH based modeling software allows adding a new and modifying an existent functionality very easily and quickly.

References

[1] Stepashko V.S. GMDH algorithms as a basis of automatization of the modeling process using experimental data. / Avtomatica. – № 4. – P. 44-55. (in Russian)

[2] Pavlov A. V. Generalized relaxational iterative algorithm of GMDH // Indutive modeling of complex systems. Col. scien. papers, iss. 2. – K.: IRTCITS NASU, 2011. – P. 95-108 (in Russian)

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software // Addison-Wesley. 1994. P. 395 ISBN 0-201-63361-2.

[4] Sheludko O.I. Self-organization of mathematical models for solving some problems of reliability and control: Diss. cand. tech. sc.: 05.13.01 / Sheludko Oleg Ivanovich – K., 1975. – 166 P. (in Russian)

[5] Pavlov A. V., Stepashko V.S. Recurrent algorithms for calculation of coefficients and selection criterion in relaxational GMDH algorithm. // Cybernetics and computing technique. – 2011. – Iss. 165. – P. 72-82 (in Russian)