Proceedings of the 1st WSEAS / IASME Int. Conf. on EDUCATIONAL TECHNOLOGIES, Tenerife, Canary Islands, Spain, December 16-18, 2005 (pp64-82)

A Versatile Software Framework for Rapid Development of Signal and Image Processing Algorithms

STEPHAN RUPP, CHRISTIAN WINTER
Image Processing And Medical Engineering Department
Fraunhofer-Institute for Integrated Circuits IIS
Am Wolfsmantel 33, 91058 Erlangen
GERMANY

{ruppsn,wnt}@iis.fraunhofer.de http://www.iis.fraunhofer.de

Abstract: - Digital signal and image processing are at the forefront of information technology and therefore frequently integrated into the syllabus of technical studies. The student's ambition is to come to terms with the content of the curriculum and gain experience how to apply theory when practicing on lab classes. In this contribution, we present a framework supporting students in focussing on the functional aspect of the solution when working on the lectures' excercises. Within the framework, processing steps are modelled as software components encapsulating a problem-specific algorithm. Secondary aspects of algorithm development – such as configuration dialog programming or memory management – are kept off the student and treated by the framework. Our approach is designed to be independent from any graphical user interface, however a virtual programming environment is provided, allowing the student to rapidly develop programs as being assembled as chains of the components. The user simply defines the dataflow as connections between the software components visually and performs the components' configuration by means of automatically rendered settings dialogs.

Key-Words: - Component-based Software Framework, Visual Programming Environment, Rapid Algorithm Development, Signal and Image Processing, Evaluation Platform

1 Introduction and Problem Statement

Digital signal and image processing are at the forefront of information technology and are used in a wide variety of modern electronic devices and information systems. The multifaceted and continously growing application fields demand for well-trained computer scientists and engineers being equipped with profound knowledge concerning technique and methodology in this specific domain.

For this reason, lectures in digital signal and image processing are frequently integrated into the syllabus of technical studies either to teach fundamentals or as graduate course – or they act as the substantial part of specialized studies. Thus, they have a high significance for university education with increasing impact.

Applications of computer vision and signal processing are often based on a set of basic and commonly accepted ideas and algorithms. Thus, when developing signal processing software, reuse plays a decisive role. Powerful tools such as software architecture and design patterns [1][2, 3] aiming at the development of reusable software (systems) have been evolved from the software engineering community, however they are very abstract due to their catholicity and require certain experience in order to be chosen and applied right.

On the other hand, the student's ambition is to come to terms with the content of the curriculum presented in lectures and gain experience how to apply these when practicing on the related lab class. In order to attain this, the students generally do not spend much interest in writing aesthetic, highly reusable code, nor are they willing to invest much time on secondary aspects like configuration dialog programming when working on the lab courses' excercises. Instead, they would like to concentrate on the functional aspect of the problem's solution allowing to deepen the theory's relationships accordingly. In addition, students emanating from fields like physics,

electrical or mechanical engineering are rarely familiar with the powerful concepts of software engineering nor do they have the necessary experience to apply these right, but they surely do have a need for writing reusable software components in their educational projects.

2 Related Work

Application frameworks for special domains have been intensivly proposed in the past years [4, 5, 6, 7]. Such approaches support the user in writing applications for the specific domain they were designed for and therefore they lack of catholicity. Thus, their use in education seems to be very limited.

A more promising approach is faced with the ImageJ environment [8]. It it is designed with an open architecture that provides extensibility via Java plugins. Custom acquisition, analysis and processing plugins can be developed using ImageJ's class library in conjunction with a Java compiler. Despite its powerful concept, it is restricted to image-based processing steps and lacks of the ability to process arbitrary type of data as required within signal processing and certain computer vision applications. Finally, neither a simple user interface for the creation of new plugins is provided nor constructing programs as chains of the developed algorithms is supported. However, as expected, ImageJ has recently been used within education [9] justified by its powerful concept when dealing with image-to-image algorithms.

Numerous tools supporting rapid algorithm development are commercially available. As representants, we would like to consider Cantata [10] within the Khoros environment [11] and MATLAB [12]. Cantata provides lots of features and supports distribution of processing steps. The richness of features induce a complexity that is reflected by the graphical user interface and is propably hard to handle for students in a lab class. Furthermore, Cantata is rather a *visual programming language* (VPL) as it provides visual control structures and state variables rather than a lightweight graphical user interface for simply chaining algorithmic steps.

The MATLAB environment is widely used within education and popular due to its high abstraction level. However, an intermediate abstraction level might be more suitable when dealing with algorithm emanating from the image processing field while keeping in mind training yielding fully-fledged graduates. In our experience, students or even graduates with a background in highly abstract, interpreting languages such as MATLAB often have enourmous problems on how to implement a certain algorithm for industrial or medical appliances in a standard programming language such as C++, especially when mathmatical techniques such as decompositions or simple matrix/vector operations are required or performance is an issue.

3 Contribution

In this contribution, we propose a framework based on a modular software architecture with distinctive processing steps being represented by software components. The *software component* itself is a *computational unit* [1] and represents "*a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected and exchanged with other components to compose a larger system*" [13].

Our approach enforces modularity due to its design and supports the student in writing reusable software components with a problem-specific algorithm encapsulated in a *plugin*. Its underlying form is textual and requires compilation, so that the student is not decoupled from problem-specific implementation details, but is still confronted with issues that are of high importance for the every-day work of professionals like code performance, memory utilization and source code documentation.

The framework is designed to be independent from any graphical user interface, but provides abstract interfaces in order to allow graphical dataflow definition, user interaction (i.e. for selection of region of interests) and component configuration in combination with a *visual programming environment* (VPE). Thus, we describe a VPE that provides a convenient way to easily and rapidly develop programs out of chains of these plugins (figure 1).

A *reflexion mechanism* is realized by the *external polymorphism* design pattern [14] enabling the

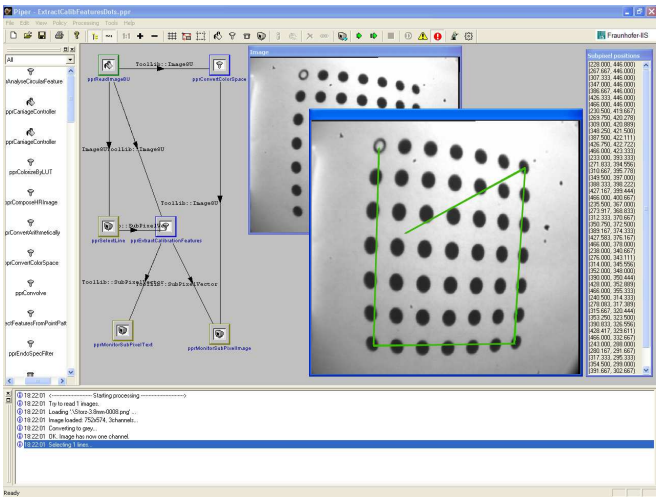

Figure 1: A screenshot of our VPE for the presented framework showing the graphical dataflow defintion as well as three types of monitors: visualization of images (partially occluded), an interactive segmentation monitor (mid) and a textual monitor presenting numeric values (right).

automatic generation of configuration dialogs along with descriptive help for the VPE user. Additionally, the same design pattern is applied to the data exchange mechanism that requires no prior knowledge of the data's type, so that every type – regardless if it belongs to third-party or an own library – can be used directly without writing any adaption code. Moreover, the data exchange has been designed to be efficient regarding performance and the system's resources with the framework being responsible for memory managment and holding off (de)allocation code from the student. Finally, our VPE is equipped with a *plugin wizard* that supports the student in generating code skeletons along with the necessary compiler project files, so that the newly created, but functionally "empty" components can be used in the framework immediately. In order not to confuse the student with code generated by the wizard, the *generation gap* design pattern [15] is applied in order to conceal framework code from the user – however the code is accessible by the student at any time.

4 Methods

In the very beginning of our work, we identified the main aspects of the software development process with respect to signal and image processing algorithms. Each of theses aspects yields a requirement that our framework needs to provide. The results are summarized in the next section whereas in the remainder of this chapter, we propose solutions how those requirements are realizable in a modular rapid development environment.

4.1 System Requirements

Applications of computer vision and signal processing often share ideas and algorithms that may already be available and – in the context of education – should be taken for granted by the student (i.e. algorithms for reading, writing or displaying images et cetera). So, **reuse of algorithms** is important and demands for a **modular design principle**.

In order to support the student in deepening the lectures' theory, the framework should prevent him from being distracted from **concentrating on the functional part** of his work. This can be achieved by certain **automatisms** provided by the framework, i.e. for rendering configuration and help dialogs from the specification of the algorithms' parameters or hiding inter-component communication and resource control by the user.

In addition, the creation of new plugins should be quickly and easily feasible – ideally without writing any line of code – however, the algorithm itself requires programming. This demands for a **reflexion mechanism** providing **meta information** and a facility that gathers information about the plugin's connection points, the algorithm's parameters and dependent third-party libraries in order to automatically generated all the necessary project files and code skeletons.

4.2 Conceptual Design

From a conceptual point of view, we identify three basic component types that are related to the fundamental tasks almost every signal or image processing application usually covers. Since an application requires certain data, there must be an entity where data is created. As soon as the data is available, the application's algorithms can start to operate on it, propably providing intermediate results but finally

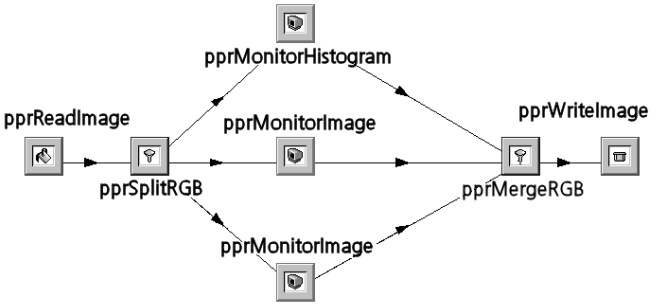

Figure 2: An exemplary topology exhibits the components of the *pipes-and-filter* architecture: source (`pprReadImage`), filter (`pprSplitRGB`, `pprMergeRGB`), sink (`pprWriteImage`) and monitor plugins (`pprMonitorImage`, `pprMonitorHistogram`).

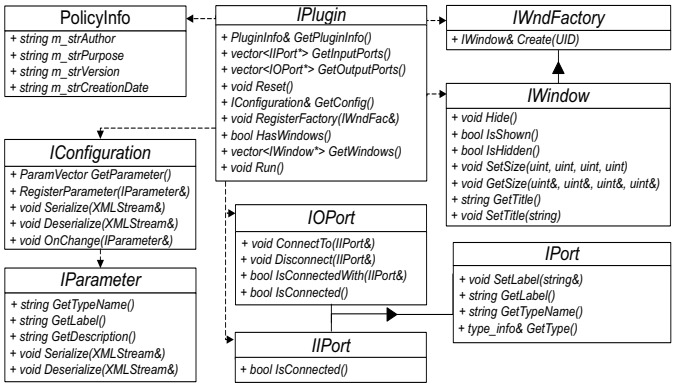

Figure 3: The framework is designed to be independent from any graphical toolkit which is attained by a set of interfaces defining fundamental services.

resulting in an output. Conceptually, the algorithms' input data is transform into output data – either intermediate or the final result. Since the application should perform a certain task, the result needs to be interpreted, visualized or – conceptually spoken – consumed. The entities being responsible for creation, transformation and consumation are denoted as **sources**, **filter** and **sinks**. Additionally, we define one more concepts in order to meeting the requirements for easy-to-use data visualization and user interaction: a **monitor** is a specialized filter being able to create, access and capturing mouse events in a window of the VPE enabling i.e. interactive selection of image regions, color picking or just displaying of textual or graphical signal representations.

Besides these *functional aspects*, the need for data exchange between adjacent components induce *topological aspects*. A link between two components is modelled as a **channel** being attached to the component's communication end-points (**ports**). It is associated with the data type that it intends to transfer. The data itself is modelled as a **token** and transmitted over the channels between adjacent processing units. The corresponding di-graph describes the *dataflow* between components and defines the topology of a program (figure 2).

4.3 Architecture

Both, the *functional* and *topological* concepts of the previous section lead directly to the architectural *pipes-and-filters* design pattern [16] that instantly determines the system's fundamental software architecture [1].

4.4 Design and Implementation Aspects

This section is concerned with fundamental design decision and implementation aspects in order to meet the requirements discussed in section 4.1. Due to the limited space, we have to restrict the coverage of design and implementation details to the fundamental key concepts of our approach.

Regarding the implemenation, we decided to use the C++ programming language due to its dominance in industrial and medical image processing projects, object-oriented character and its powerful template mechanism supporting generative programming techniques [17].

4.4.1 Interfaces

One of our major concerns when designing the framework was decoupling the frameworks business logic from any graphical user interface (GUI), so that everybody is able to use it in conjunction with his favorite graphical toolkit. In accordance with this, the framework is built on a set of interface classes, providing all the services an application may need to manage the plugins – this includes access to the parameters and the token types that a plugin requires or provides (figure 3).

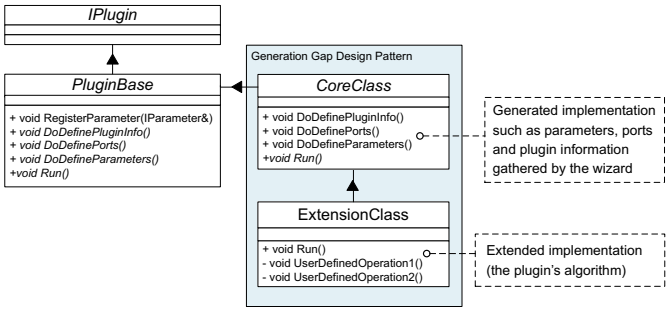

Figure 4: Generation gap design pattern for encapsulating code that has been automatically generated by the plugin wizard. The plugin's specific algorithm is implemented in the `Run()` routine that is automatically called by the framework.

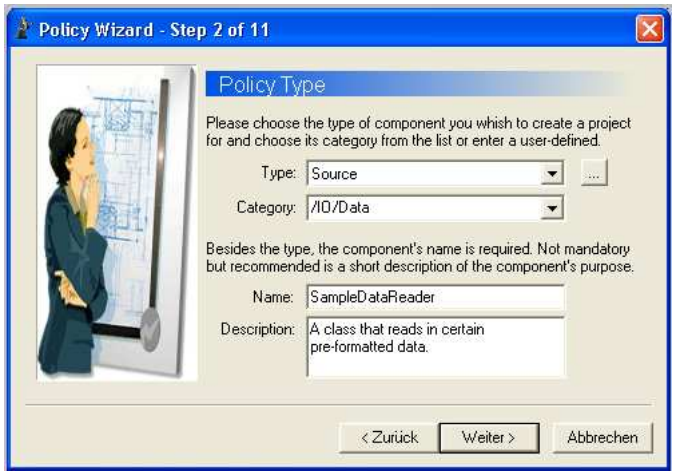

Figure 5: The plugin wizard supports the user when creating new plugins guaranteeing ease-of-use.

4.4.2 Generation Gap

Surprisingly, some framework designers still make use of macros in order to provide shortcuts for framework code. The virtual intend is to hide implemenation details from the user and let the preprocessor expand the macro definition on compilation [18]. We think that there are quite better ways to realize this, since macros are error-prone and hard to debug due to their lack of type-safety and non-evaluating, textual replacement of arguments. Especially when the code is created by a code generator, a more convenient approach will make use of the generation gap design pattern [15] (figure 4). Powerful tools like the QT designer [19] or *ibuild* [20] do apply the generation gap pattern in conjunction with their code generation facility - as we do, too.

As already mentioned, we provide a *plugin wizard* which supports the user when setting up new processing units (figure 5). Nevertheless, the user can set up a new policy by inheriting from the appropriate framework's core plugin class and customize the subclass to the specific needs by overriding/overwriting and calling the frameworks service methods. Since no special preprocessor statements have to be placed, creating a new plugin is as straightforward as creating a new C++ class with full control by the programmer.

To speed up the creation, the wizard collects the plugin's name, its communication ports, the parameters it is supposed to have as well as pathes for dependent libraries. Optionally, a description of the algorithm's purpose can be obtained. From this meta information all the necessary code is generated by the wizard and placed in the **core class** of the generation gap pattern. In contrast, the user's code is placed in the inherited **extension class**, which is created by the wizard too. By this, the details of the framework are hidden from the user's eyes so that he is only concerned with 'his' extension class. Anyhow, the code is accessible at any time – even on compiler errors or warnings – and provides type-awareness and maintainable code. Moreover, it is on the user's behalf to decide if he is interested in the frameworks details or not. Due to the fact that the framework specific, generated code is hidden in the core class, the programmer can concentrate on the functional aspect meeting the corresponding requirement identified in section 4.1.

4.4.3 Data Exchange and Automatic Resource Control

As one of the requirements, data exchange should be hidden from the user so that he is not messed up with writing allocation or deallocation code in order to pass tokens. However, inter-component communication should still be efficently realized and provide versality. In order to support the user, the framework takes responsibility for managing all the resources that are required in order to transport tokens from

one plugin to another one. Since one-to-many connections are supported, dispatching the tokens to all subsequent entities is required. For performance reasons, the tokens being passed to consecutive processing units are not copied, but passed by reference instead. Additionally, the framework possesses the token's object for its complete life-cycle and though is responsible for deallocating its resources.

By providing a simple pointer to a successive processing unit, the tokens integrity needs to be secured, so that intended or even unintended deleting of a token's data can be prevented. For this reason, a template-based guard class is introduced in order to control the access to the encapsulated data. Since the data's type is provided as template argument any arbitrary type can be used for inter-component data exchange:

```
template <class T> class Token {
public:
    Token();
    Token(const Token<T>& ptr);
    explicit Token(T*& data);
    virtual ~Token();

    bool operator==(const Token<T>& ptr);
    bool operator!=(const Token<T>& ptr);

    const T& operator*();
    const T* operator->();
    T GetWriteAccess();

private:
    T*   m_pData;
    int  m_iCount;
};
```

Whenever the user needs read access to the token's data, the `*` (or `->`) operator obtains a constant reference (or pointer) to the token's object. Since an address is returned no time-consuming deep copying of objects takes place and so, access is gained without any loss of performance.

Recalling the dispatching mechanism, tokens are not passed *by value*, but *by reference* (pointer) instead. Without any protection mechansims (guard), a write access to the referenced object would not only affect the current processing unit, but all the processing units the token is dispatched to. In order to prevent this *sideeffect*, the user has to announce explicitly write access by calling the `GetWriteAccess` method of the templated `Token` class resulting in a deep copy of the contained object's data.

Finally, a simple reference counting mechanism tracks the references to a particular token and indicates the framework that a token's resources can be freed as soon as the counter is equal to zero.

4.4.4 Reflexion

In order to focus on the essentials of an algorithm secondary aspects have to be automated. A reflexion mechanism gathers and provides relevant information about a plugin that is used by the framework to generate configuration dialogs or for performing semantic checks on connection of plugins.

▷ **Parameters and Dialog Rendering**

The parameter's interface class provides standardised services for getting and setting the parameter's value as well as methods for associating its type with a name and a description of its purpose. This information is accessed by the proposed VPE in order to render the dialogs with mapping the types to GUI elements or utilizing the purpose description for displaying help (figure 6). All access is realized by type-safe method calls so that no parsing and interpretation of additional textual description is necessary. In order to equip a plugin with a parameter, the user simply has to pass an appropriate parameter object to the plugin's base class service routine `RegisterParameter(IParameter& )` – or he makes use of the graphical plugin wizard.

▷ **Persistency**

In addition to the former paragraph, the frameworks implemenation of the parameter's interface realizes a generic mechanism for reading and writing a plugin's state. For this, the plugin's base class implementa-

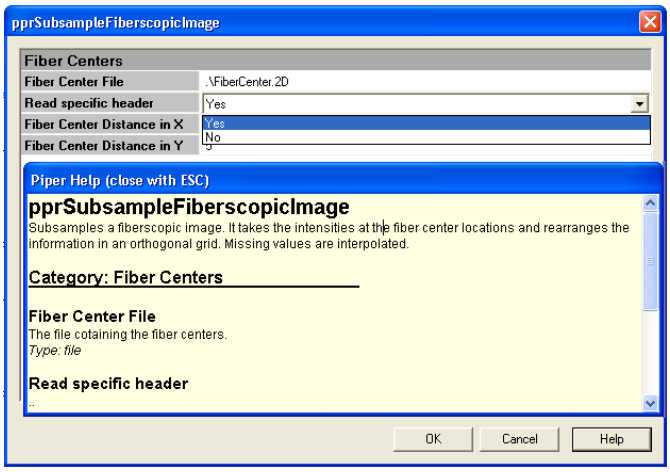

Figure 6: Configuration and help dialogs are automatically rendered by the framework enabling the user to simply adjust the algorithms parameters without touching any line of source code.

tion offers a service method, that iterates through all the parameters associated with the plugin and calls their (de)serialization method depending on wheter to read or to write. By this, the plugin's state can easily transformed into or recoverd from a XML stream.

5 Discussion And Conclusion

The presented approach proved its worth its applicability in numerous internships, thesis and research projects at our department. Many of our students who wrote their diploma or masterthesis were using the platform in order to make use of existing I/O and visualization plugins which allowed to concentrate on their theme and prevented from wasting time with implementation effort that does not contribute to their theses.

Due to its direct access and minimal overhead, our approach can be gainfully used in lectures as well and hence, is currently evaluated as demonstration tool and as development platform for the associated lab classes.

◇

In this paper, we presented a framework based on a modular software architecture with distinctive processing steps being represented by software components that can be gainfully used in education. Our approach enforces modularity due to its design and supports students in writing reusable software components with a problem-specific algorithm encapsulated in a plugin. The framework is designed to be independent from any graphical user interface. It provides abstract interfaces in order to allow graphical dataflow definition, user interaction (i.e. for selection of ROIs) and component configuration.

Additonally, we present a visual programming environment that supports easy creation and configuration of new – functionally empty – plugins. Secondary aspects like UI programming or resource control for inter-component communication are kept away from the user and are automatized by the framework. Fundamental plugins for reading, displaying or writing images or – in general – any kind of signal processing step can be taken for granted by the students on a lab class, whereas a particular solution for an excercise has to be implemented and tested in a separate plugin.

This all supports students in concentrating on the functional part of their work and allows them to deepen the content of the curriculum as we experience with our students. In the end, we hope that such an approach will strengthen the educational effect of lectures and hopefully result in fully-fledged graduates with extensive theoretic and practical skills.

Acknowledgements: The research was supported by the *Research Training Group 244* of the *German Research Foundation (DFG)*. In case of the second author within the current research subproject *A7* of the *Collaborative Research Centre (SFB) 603*.

References

[1] Shaw M. and Garland D. *Software Architecture. Perspectives on an Emerging Discipline*. 1 ed. Prentice Hall, 1996.

[2] Gamma E., Helm R., Johnson R., and Vlissides J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[3] Buschmann F., Meunier R., Sommerlad P., and Stahl M. *Pattern-Oriented Software Architecture, Vol.1 : A System of Patterns*. 1 ed. John Wiley and Sons, 1996.

[4] Vogelgsang C., Scholz I., Greiner G., and Niemann H. lgf3 - A Versatile Framework for Vision and Image-Based Rendering Applications. In Greiner G., Niemann H., Ertl T., Girod B., and Seidel H.P. (eds.), *Vision, Modeling, and Visualization 2002*. Infix, -, 2002.

[5] Müller T.O., Stotzka R., Beller M., Ruiter N.V., and Hartmann V. Schneller Aufbau medizinischer Diagnosesysteme mit ICE. In *Bildverarbeitung für die Medizin*, 2004.

[6] Duret-Lutz A. Olena: a Component-based Platform for Image Processing, mixing Generic, Generative and OO Programming. *In Proc. of the Generative and Component-based Software-Engineering (GCSE '00)*, October 2000, pp. 13–19.

[7] Rupp S. Generische 3-D-Rekonstruktion von Hohlräumen aus monokularen endoskopischen Bildfolgen. Master's thesis, Technical University of Hamburg-Harburg, Germany, 5 2003.

[8] ImageJ - Image Processing and Analysis in Java. URL `rsb.info.nih.gov/ij`.

[9] Burger W. and Burger M.J. *Digitale Bildverarbeitung - Eine Einführung mit Java und ImageJ*. Springer, 2005.

[10] Khoral Research. URL `www.khoral.com`.

[11] K.Konstantinides and Rasure J. The Khoros software development environment for image and signal processing. *IEEE Transactions on image processing*, vol. 3(3), 1994, pp. 243–252.

[12] MATLAB - The Language of Technical Computing. URL `www.mathworks.com`.

[13] D'Souza D.F. and Wills A.C. *Objects, Components and Frameworks with UML - the Catalysis Approach*. Addison-Wesley, 1998.

[14] Cleeland C., Schmidt D.C., and Harrison T.H. External Polymorphism. *In Proc. of the 3rd Pattrn Languages of Programming Conference*, September 1996.

[15] Vlissides J. *Pattern Hatching - Design Patterns Applied*. 1 ed. Addison-Wesley, 1998.

[16] Vermeulen A., Beged-Dov G., and Thompson P. The pipeline design pattern. *In OOPSLA'95 Workshop on Design Patterns for Concurrent, Parallel and Distributed Object-Oriented Systems*, Oktober 1995.

[17] Alexandrescu A. *Modern C++ Design*. Addison-Wesley, 2001.

[18] Wenzel F. and Grigat R.R. A Framework for Developing Image Processing Algorithms with Minimal Overhead. *5th WSEAS Int. Conf. on Signal, Speech and Image Processing (SSIP'06)*, August 2005, pp. 185–190.

[19] The QT Toolkit. URL `www.trolltec.com`.

[20] Vlissides J. and Tang S. A Unidraw-based user interface builder. *Proc. of the ACM SIGGRAPH 4th Annual Symposium on User Interface Software and Technology*, vol. 3, November 1991, pp. 202–210.