Software architecture for modular, extensible and reusable signal processing components

RALPH MASCHOTTA, SIMON BOYMANN, DUNJA STEUER
Institute of Biomedical Engineering and Medical Informatics (BMTI)
Ilmenau Technical University
POB100565, D-98693 Ilmenau
GERMANY

Abstract: A common way to develop new signal processing strategies is to build libraries with various signal processing components to combine these components. The requirements for the software architecture of those components are quite extensive and sometimes conflicting. With the help of object oriented methods and common architectural and design patterns we create a software architecture to fulfill these requirements. The software architecture are modeled in Unified Modeling Language (UML).

Key-Words: Software architecture, Architectural pattern, Design pattern, Pipes and Filters, Reflection, Microkernel, Signal processing, Object oriented technology, UML

1 Introduction

The requirements concerning signal processing software often include fast processing of huge amount of data, fast implementation of new algorithms, real time processing and so on.

Some tools like MATLAB® allow the user to create and test new algorithms easily. To use these experimental algorithms it is necessary to transform these algorithms into professional software. But professional software have some more requirements. After all you have to fulfill all of them.

There are a lot of signal processing algorithms at the Institute of Biomedical Engineering and Medical Informatics (BMTI), especially in the field of adaptive recursive signal processing. The algorithms are implemented in different programming languages, in different ways of software design and after all without adequate documentation. Thus it is ineffective to reuse the existing algorithms.

Over the last 20 years, the BMTI has developed the signal processing tool ATISA (adaptive time series analysis). The main aim of this tool was the implementation of new signal processing algorithms and the use of this algorithms in different projects. But there was no consistent maintenance of the software architecture. Meanwhile various people were implementing several parts of the tool with different software techniques in different programming languages and with insufficient documentation. There are also different versions based on different operating systems. The result of this expansion is a “Big Ball of Mud”[1].

Therefore we want to create a software architecture which uses the newest software development techniques to prevent such a expansion.

2 Problem Description

The power and flexibility of modern signal and image processing tools based on the possibility to combine native algorithms is a common way to design complex signal processing strategies.

So we postulate for the new tool to be a component based signal processing tool with the following attributes:

- to have a library of native signal processing algorithms,
- to have an interface to add new algorithms to the tool,
- to have the ability to combine these components with each other,
- to import external functions like MATLAB® for example.

The requirements for such signal processing software architectures are quite extensive and sometimes conflicting. On the one hand there are the requirements concerning signal processing. On the other hand there are the requirements concerning modern software architecture.

Signal processing software demands some important qualities: fast creation of new algorithms, fast processing of data, the ability of real-time processing and the possibility of hardware implementation.

Modern object oriented software has to fulfill requirements like reuse and extensibility of components, safe program processing, usable documentation and so on.

The required fast creation of new algorithms can be fulfilled by modern software architecture. But the performance of object oriented algorithms can be lower than the performance of other algorithms which are not based on object oriented technology. That's why it is necessary to test the real time ability of the algorithms.

Furthermore it is difficult to use object oriented algorithms for hardware implementation.

This shows the necessity to balance the several requirements of a modern signal processing software architecture.

3 Problem Solution

Because of the advantages of the object oriented paradigm we decided to create an object oriented architecture. With the Unified Modeling Language (UML)[4] we have a standard language to describe this architecture. In assistance of an UML based modeling software we have the basis to create a modern object oriented software.

Another advantage of the UML is the possibility to model the natural relations between signal processing algorithms.

Furthermore there exists a large collection of patterns which describe solutions of several architectural and design problems. So we selected and combined some appropriate patterns which fulfill our requirements.

3.1 Used architectural Patterns

We decided to use a composition of three architectural patterns. To serve the problems of combining several signal processing algorithms we use the Pipes and Filters pattern. To create an adaptive system we use the Reflection pattern. And finally to manage the system we use the Microkernel pattern. In assistance of additional design patterns we modeled the following software architecture.

3.1.1 Pipes and Filters

"The Pipes and Filters pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows you to build families of related systems."[2]

A filter may enrich, refine, or transform its input data [2]. There are two different types of filters: the active and the passive filters. Active filters are autonomous algorithms like threads. For parallel processing it is necessary to create active filters. Passive filters are implemented like normal functions or procedures.

The pipes are the connectors between a data source and the first filter, between filters, and between the last filter and a data sink. As needed, a pipe synchronizes the active elements which are connected together[2].

It is an important requirement at the signal processing software architecture to create new signal processing strategies easily. It is useful to combine several existing algorithms to new signal processing strategies or append new algorithms. The Pipes and Filters architectural pattern fulfills this requirement.

We use a special type of the Pipes and Filters architectural pattern called Tee- and Join- Pipeline (Fig. 1). Every filter has a different number of data input and output channels. The possibility to combine two filters depends on the input and output data formats of the filters. So it is possible to create switches or loops. Thereby it is possible to create complex processing pipelines.

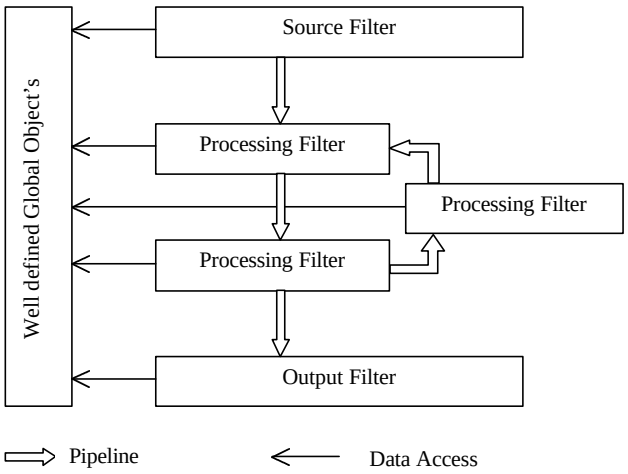

Fig. 1 Example of a Tee-and-Join-Pipeline with acess to global objects

We implemented the functionality to create push and pull pipelines to be able to start the pipeline at the beginning, at the end in between.

The signal processing algorithms can be implemented as filters. The administrative overhead to connect the filters, to call filters and so on is implemented in base classes. To create a new filter it is only necessary to inherit from these base classes.

Presently there are only passive pipes and filters implemented. But the implementation of active components is conceivable.

The Pipes and Filters architectural pattern provide the following advantages [2]:

- intermediate files unnecessary, but possible,
- flexibility by filter exchange,
- flexibility by recombination,
- reuse of filter elements,
- rapid prototyping of pipelines,
- and perhaps efficiency by parallel processing.

But there are also some disadvantages:

- sharing state information is expensive or inflexible,
- efficiency gain by parallel processing is often an illusion,
- data transformation overhead,
- error handling.

To reduce the disadvantages we use defined global singleton objects to throw messages. We have also a singleton object as an interface to store constant parameters (Fig. 1).

To connect different Pipes and Filters it is necessary to verify the compatibility between two filters.

3.1.2 Reflection

"The Reflection pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as type structures and function call mechanisms. In this pattern, an application is split into two parts. A meta level provides information about selected system properties and makes the software self-aware. A base level includes the application logic. Its implementation builds on the meta level. Changes to information kept in the meta level affect subsequent base-level behavior".[2]

All pipes and all filters consists of two components: the description as meta level component and the implementation of a signal processing algorithm as base level component (Fig. 2).

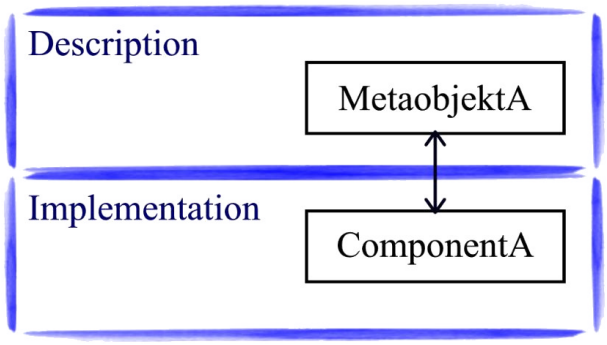

Fig. 2 Parts of the Reflection pattern

The description contains the name of the filter, the cardinality of the calculation, the number of inputs and outputs and their data types.

The processing algorithm is part of the implementation. The algorithm uses only the described input and output data. During the execution of the pipeline the description are used to identify the input and output channels.

Based on the description we can verify the connection between the filters and the functionality of the whole pipeline.

It may be useful to optimize the connection between filters by inserting a special pipe. Such pipes can be buffered pipes or synchronization pipes.

After the verification of the pipeline the pipeline can be executed. Up to now only the description of the algorithm was needed. Now the filter object with the signal processing algorithm must be created. This lazy creation can be realized with the state design pattern [3] (Fig. 3).

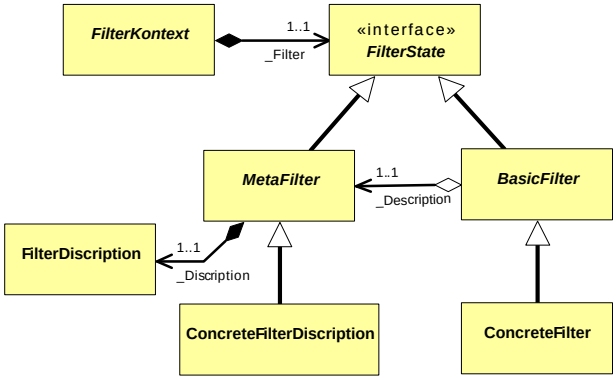

Fig. 3 The implementation of the Reflection pattern by using the State pattern.

The MetaFilter is the interface for Metaobjects and the BasicFilter is the interface for the implementation of the processing algorithms.

3.1.3 Microkernel

"The Microkernel pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration" [2] (Fig. 4).

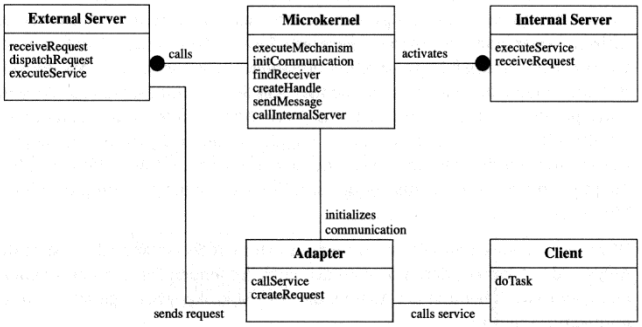

Fig. 4 The Microkernel Architecture [2]

The Microkernel is the central component. It creates pipelines based on the available pipes and filters, starts pipelines and observes them during the execution. The Microkernel is the interface between clients and the internally processing. It manages the communication with clients.

The Microkernel is implemented as Singleton [2] to ensure that only one Microkernel object is exists at runtime (Fig. 5).

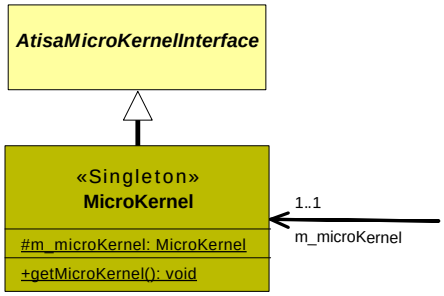

Fig. 5 The Microkernel implemented as Singleton

The internal servers are libraries of several pipes and filters. Such libraries includes various basic algorithms, signal processing algorithms picture processing algorithms or other special algorithms. All libraries are Dynamic Link Libraries (DLL). The Microkernel loads all existing libraries at runtime.

The external servers are existing tools which provide interfaces to allow the access to its internal algorithms. MATLAB® can be such an external Server. To use external functions within the Pipes and Filters system it is necessary to implement a filter. These kinds of filters are interfaces between the external server and the internal Pipes and Filters System. Those interface filters can also be a part of an internal server.

To unify the communication protocol between the Microkernel and the connected clients, the Microkernel receives all requests from an Adapter [3]. A client must implement an adapter to communicate with the Microkernel.

Another part of the Microkernel is the so called Builder [3] (Fig. 6). It is responsible for creating pipes and filters, for connecting the filters and finally for creating and verifying the pipeline. The Builder receives description of a pipeline. Based on this description the Builder creates the pipeline and allocates the constants. After that the pipe is ready to execute.

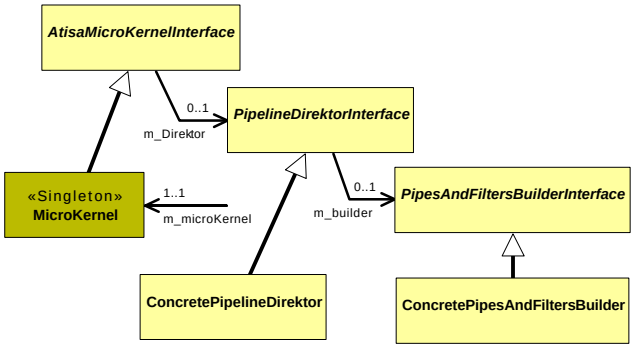

Fig. 6 The implementation of the Pipeline Builder [3]

The input and output of a pipeline depends on the operating system. To separate the calculation from the representation of the pipeline and of the results it is advisable to create a separate Microkernel. The other algorithms are located in a central Microkernel. Thereby, we get a distributed Microkernel system. The several Microkernels are connected by a Bridge [3].

3.2 Collaboration

First, the client starts the Microkernel. The Microkernel loads all existing internal servers and connects the external servers. It also connects other existing Microkernels to get the access to pipes and filters implemented there. After that the Microkernel has a list of all available pipes and filters.

The client asks the Microkernel about all available pipes and filters. Based on this information it creates a request of a pipeline to the Microkernel. The Builder translates the request and creates and verifies the pipeline based on the description of the pipes and filters. If there are errors, the Microkernel returns the messages to the client. If there are no errors the client can create a request to run the pipeline. In that case the Microkernel starts the pipeline. At this point the pipe and filter objects are

created with the implementation of the processing algorithm. Messages which are created during the execution of the pipeline will be return to the client. Please note: The input and output of the calculation are filters and part of internal servers.

4 Conclusion

The shown software architecture composed of the three described architectural patterns, the Pipes and Filters, the Reflection and the Microkernel pattern. This compositions of patterns is the basis to create modular, extensible and reusable signal processing components.

This architecture is based on the newest software modeling techniques. The object oriented model was created in assistance of UML and common architectural and design patterns.

First implementations of signal processing pipes and filters demonstrate usability of this architecture. The architecture could be an interface to build a large library of different signal processing algorithms to solve different problems in the research and in the industry.

Supported by TMWFK: B699-00011

References:

[1] Brian Foote and Joseph Yoder. *Big Ball of Mud*. Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97),Monticello, Illinois, September 1997; Technical Report #WUCS-97-34

[2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stahl; *A System of Patterns: Pattern-Oriented Software Architecture*, West Sussex, England, John Wiley & Sons, 1996

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

[4] *Unified Modeling Language;* Version 1.3. www.omg.org