

A Reusability Concept for Process Automation Software

Wolfgang Narzt, Josef Pichler, Klaus Pirklbauer, Martin Zwinz
Business Information Systems
C. Doppler Laboratory for Software Engineering
University of Linz
Altenbergerstr. 69
A-4040 Linz, Austria
Phone +43 732 2468 9474
Fax +43 732 2468 9430
{narzt, pichler, pirklbauer, zwinz}@swe.uni-linz.ac.at

Abstract

Reusability of software is an important prerequisite for cost and time-optimized software development. This is especially true for custom software produced for only one application; process automation software in the hot rolling mill domain is a typical example of such software. This paper presents the special requirements on software in this domain and shows how a reusability concept helps to decrease overall development time. Exploiting reuse to reduce development time cannot be restricted to inner reusability as supported by object-oriented programming; furthermore, reusability of software components does not suffice to achieve reusable software. This paper introduces a combination of concepts that applies object-oriented and component-oriented techniques together with parameterization and flexible data handling. Additionally, we discuss the effect of standardization of some inner parts of the process automation software on the reusability of the whole software.

We demonstrate the application of these concepts with the example of the data handling part of the process automation software for hot rolling mills. The data handling part is the crucial part of each process automation system, whose main tasks are to receive measured values from the basic automation system, to support mathematical model calculations with these values, and to send the result values of these calculations back to the basic automation system to control the rolling mill. Therefore data handling plays an important role in making automation software reusable. Our concept involves standardizing the data items while keeping the interfaces flexible.

The concept in this paper evolved from a study of several process automation systems. This concept has been used as a basis for a process automation architecture for hot rolling mills developed as part of an ESPRIT project. A hot rolling mill automation implementation to verify these concepts is under construction.

1 Motivation

A number of quality characteristics, including reusability, describe software. Meyer [2] defines reusability as the ability of software elements to serve for the construction of many different applications. Furthermore, Meyer classifies quality according to internal and external factors. Internal factors like readability and modularity are perceptible only to computer professionals; external factors like correctness, robustness and, of course, reusability are recognizable to users too. Reusability influences directly the effort, necessary to build new applications upon existing ones. This effort is the motivation for a systematic approach to achieve reusability. Achieving a high degree of reusability necessitates designing the architecture of the software according to the requirements of the domain.

The objective of design for reusability in automation is to make software components available to various automation systems. Restricting the domain of automation, for example to hot rolling mills, facilitates recognition of common requirements and common functionality of individual automation systems. Common features can be implemented on an abstract level and later adapted to individual requirements. Through customization to the individual needs, this software core is developed further to an entire automation system. The reuse and customization of abstract parts is widespread in the object-oriented community and leads to building of object-oriented application frameworks [1,3,6]. Application frameworks for specific domains are also called domain frameworks [11].

With the aim of increasing reusability in the design and implementation of automation systems in the hot rolling mill domain, the ESPRIT project REFORM [7] was founded. Within this project an object-oriented framework is built in the programming language C++ [10] on the basis of which an individual automation system can be implemented. Although object-oriented programming contributed significantly to the reusability of the framework, object-oriented programming alone proves insufficient for achieving practical reuse.

In the framework, physical aggregates are represented by objects of special aggregate classes. Changes of attributes (position, name, ...) in such an aggregate class require modifications in the source code, which

invoke a *compile link test* phase. Admittedly, inheritance, polymorphism and dynamic binding permits leaving much of the reused code without changes, but creation of objects demands definition of the type (class) at compile time. Some techniques, like the factory method pattern [1], defer the creation of concrete objects to more specific code. The framework can be kept without concrete code, but source code has to be changed although only the type changed. Likewise, whenever types of measured values change, modification of source code is required.

This article is about an approach to eliminate the recoding deficiency by parameterization and configuration in order to increase the reusability gained from object-oriented programming. To show this, we present the design and implementation of data handling part of an automation system.

2 Requirements of the Domain

2.1 Domain of Hot Rolling Mills

To explain the specific requirements of process automation systems, in particular for hot rolling mills, we give a short overview of our software architecture. The *process automation system* (PAS) for hot rolling mills is part of the overall automation system of a production sector, which is basically divided into three automation levels [4]. The *production and planning system* (PPS) determines the rolling schedule for a hot rolling mill in advance; this mill system is controlled by the *process automation system*. Information about sensor states, measured values and so on is supplied by the *basic automation system* (BAS).

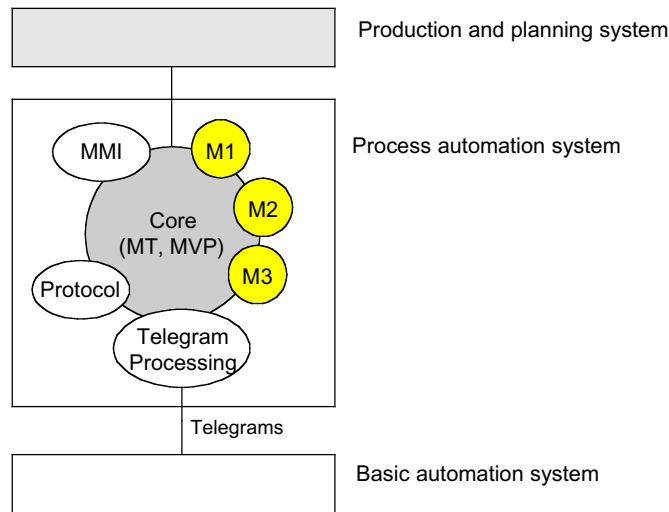


Fig. 1 Software Architecture

A process automation system (PAS, see Fig.1) always contains a core system with *material tracking* (MT) and *measured value processing* (MVP).

The *material tracking* logically follows the material through the mill in order to execute specific actions on predefined positions. The task of *measured value processing* is to collect and store the incoming measured values and to transfer them selectively to the *mathematical models* in statistically compressed form. The model calculations are used to control and optimize the production process. Most of these *mathematical models* are used before, during and after the production (precalculation, adaptation, postcalculation). Because of the mostly different protocols between level 1 and level 2 automation, a telegram handling process converts the telegrams from the basic automation into a byte stream, that is readable for the process automation system. Various *mathematical models* (M1, M2, M3), the *man machine interface* (MMI) and the *protocol mechanism* are connected to the core as shown in Fig. 1. Together with the core, they form the PAS. Usually these components are implemented in separate processes. Communication between individual components and the PAS takes place via well defined interfaces.

2.2 Requirements

To achieve a high degree of reusability in the process automation domain, the following properties have to be kept flexible:

- Configuration of an automation system depending on physical arrangement and properties of aggregates (measuring device, sensor, ...)
- Properties of measured values like type, scope and frequency
- Interface to the basic automation system (BAS), which depends on individual automation systems
- Supply of remaining components of an automation system with data.

The more the implementation is independent of these properties, the more reuse can take place.

3 Application of Reusability Techniques

3.1 Domain Modeling

In our object-oriented automation software, the physical plant is represented by objects. For every physical device category (e.g., stand, measuring device) a corresponding class can be used with or without customization for an individual physical device. All aggregate classes describing the plant configuration are derived from a base class, `Aggregate`. The construction of objects representing the plant and its aggregates and the correct connection of the aggregate objects is handled automatically by parsing a flexibly structured and extensible configuration file that includes aggregate descriptions plus parameters.

The excerpt of the configuration file depicted in the following example describes a sensor with the class name `Sensor`. The attribute `name` is initialized with `FZA` and the attribute `pos` with `14700`. The parameter `parent` describes the parent aggregate in the hierarchy.

```
[Sensor]
name    FZA
pos     14700
parent  RM
.
```

In addition to the aggregates, the material processed in the automation is also represented by objects. This object-oriented modeling of an automation system is the foundation for the implementation of data handling.

3.2 Data Handling

Measured values occur in different forms (type, scope, origin, ...). Types range from primitives like integers and floats to complex structures consisting of dozens primitives. In conventional, non-object-oriented software systems, the management of different measured value types means overloading of functional equivalent algorithms and data structures according to individual types, resulting in an immense effort whenever new types have to be introduced. Parameter types of interfaces and types of variables containing types of measured values extend over the entire software. Moreover, overloading results in duplicating of program code. This could be avoided by the means of generic algorithms (templates in C++ [10]) and abstract data types [5], but resolving them (which is done by developers) according to individual types covers the entire software again. Measured values are described by further properties which affect the implementation just as the type does.

The main distinction of measured values, type and scope is handled by one common base class and several derived subclasses. This means for every type of measured value an appropriate class was implemented which consists only of a primitive (C++) type, like integer, float or boolean. Even though values of simple types are the most frequently occurring measured values, a simple way to establish more complex types is provided by a complex measured value that is composed of several simple values (composite design pattern [1]).

A measured value is described uniquely through its name. The quantity of all descriptions results in the mill dictionary, a unique catalog of measured values and their properties. These are either related to an individual automation system or are of common validity. Therefore the entire description of measured values results in a divided mill dictionary: one part is of common validity; the other is dependent on a concrete plant. Whether the mill dictionary is implemented as a relational database or only as an structured ASCII file is of minor importance. The ability to modify this information without affecting the implementation is crucial. Changes of properties do not affect the implementation.

Measured values are therefore objects of measured value classes. This is the foundation for our concept to achieve reusability. Observation of the life cycle of such objects enables the definition of fundamental stages. Every individual stage holds a certain reusability potential. In the following, techniques to exploit these potential are shown according to the individual stages.

3.2.1 Creation

The interface between the basic automation system (BAS) and our PAS is the takeover of measured values from the incoming telegrams (see Fig. 2). Telegrams are data streams consisting of a sequence of individual values. The sender of the telegrams (BAS) writes all values sequentially onto a data stream, which in turn is dismantled into individual values on the receiver side (PAS). To extract individual values from the stream correctly, the order, type and length of the values are needed. Type and length are specified in the mill dictionary. The order of values within a telegram is described in a telegram database, which is part of the mill dictionary. The telegram database fixes the involved values as well as the order of values within the telegram. Again, this information is independent of the source code; i.e. modification after compile time is possible. This telegram information is always dependent on a concrete automation system (order is given by the basic automation system).

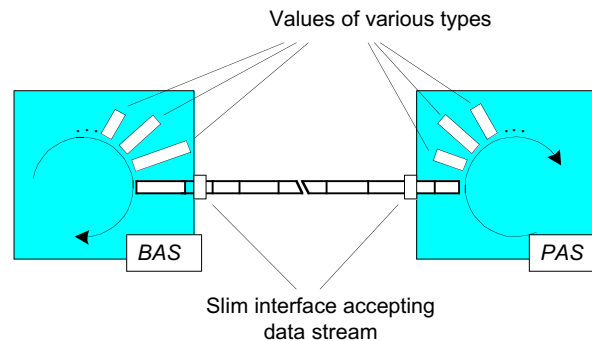


Fig. 2 Packing of measured values into telegrams

Communication of measured values in data streams enables slim interfaces (e.g., a character-sequence) instead of various interfaces with complex structures according containment and order of values. Changes of value types or even changes of telegrams necessitates no adaptation of interfaces; only the description in the mill dictionary has to be modified. Moreover, type safety is not lost as it would be as a consequence of a `void*` interface.

A measured value object is created by reading the respective number of bytes from the telegram stream and converting them into the required type. Both the number of bytes and the type are specified in the mill dictionary. For example, instead of the statement

```
Float *aMeasuredValue;  
aMeasuredValue = new Float ( initData )
```

the type of the new measured values is determined and created with

```
Value *aMeasuredValue;  
aMeasuredValue = newObject(millDict->getTypeOf("width_strip"))  
aMeasuredValue->initFrom( initData )
```

Initialization is done by the means of overwritten methods. This approach assumes that all classes implement such an initialization method which takes the telegram stream as parameter. The initialization method reads the specified number of bytes and increases the readpointer on the stream by the number of bytes read. The only restriction at compile time is the assurance that `aMeasuredValue` is at least of type `Value`. A second assumption on this approach is a "new-by-name" mechanism; which means creation of objects of a class only by specifying the class name as a string. While in languages like Java and SmallTalk such a mechanism is part of the language, in C++ this feature must be handled by the programmer. This effort, however, bears no relation to the attained benefit, which enriches our concept considerably.

3.2.2 (Inter)Storage

Value stores intended to store measured values are organized in a two-dimensional manner like tables in relational databases. One dimension is described by the name of the measured value (specified in the mill dictionary), the other by the time-stamp when the value occurred. This results in a matrix as shown in Fig.3, whereas the tuple `<name, time-stamp>` is a unique identification of a measured value within the matrix. This is a logical view on a value store only and has nothing in common with a concrete implementation of such storage structures, which are, indeed, more complex than two-dimensional matrices. A solution for the implementation of value stores can be borrowed from the implementation of sparse matrices [8]: keep the two-dimensional representation intact and avoid storage of unnecessary data.

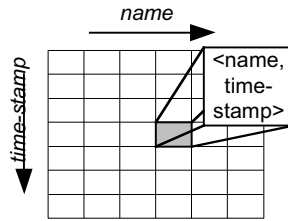


Fig. 3 Logical view of value stores

The value stores provide interfaces for managing measured values regardless of the concrete types of the measured values. Since all measured values are at least subclasses of Value, all these interfaces can be implemented as the type Value.

3.2.3 Supply

Several components of the automation systems require data (identified by names in the mill dictionary), which are stored first in the automation core. Direct access to data assumes information about where which data is located. This is a contradiction to information hiding as suggested by Pomberger [5]. Moreover, data values are uniquely specified by their names. Direct access may cause undesirable delays because the automation is blocked until the inquiry returns. Therefore our concept provides another approach. Components do not access data values directly but merely specify in advance which data is needed when (see Fig. 4 (1)). The automation system manages these requests (2) and supplies the components with the requested data (4) at the right time. The requests access (3) the data stored in the core.

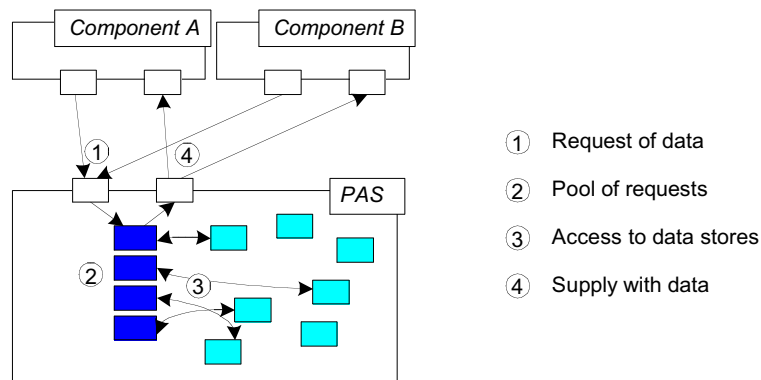


Fig. 4 Supply of data

The registration of requested data as the return of results takes place via a defined interface between components and the core. For example, in our framework interprocess communication is implemented with the Common Object Request Broker Architecture (CORBA; see [9]); therefore interfaces are defined in the Interface Definition Language (IDL; see [9]).

Insertion of requests between components and the automation core enables separation of implementation details of the storage and the interfaces in the form of requests. Furthermore, the internal storage is protected against manipulation by the remaining components.

3.2.4 Examination

Examination means access to the actual value, wrapped by the measured value object (see code excerpt below). This task cannot be implemented in a generic (abstract) way.

```
RFloat *f = dynamic_cast<RFloat*>(aMeasuredValue);
float ff = f->asFloat();
```

Here we leave the abstract level; also casts from abstract to concrete types require dynamic type-checks: This makes time demands at run time. Fortunately, examination usually takes in components and thus does not encumber the core of the automation.

3.2.5 Data Disposal

As long the material (strip) is in the automation system, the respective data is held in the memory and thus is accessible. Some subsystems of the process automation system (for example, the report generator) need access to material-related data even long after the material is taken from the plant. This period of time can

extend from hours to days, weeks and even months. This necessitates a mechanism to provide material-related data even after the material has left the plant.

The material object and all related data values are „passivated“ on a permanent storage medium (file) through the serialization mechanism. The name of the file must be unique over time; therefore the ID of the material is best suited for this task. Through „activation“ from the file, the material data can be restored and used in the same way as when the material object was still in memory.

4 Conclusion

In this article, we showed a concept to build a flexible architecture for automation software. This concept is not an independent, self-contained solution but results from the combination of various techniques. Object-oriented programming is the foundation. In combination with components, parameterization and flexible configuration, we are able to implement a flexible software core that can be used to implement various automation systems. This increases the reusability which in turn facilitates cost and time-optimized software development.

References

- [1] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesely, Professional Computing Series, Reading, Massachusetts, 1994.
- [2] Meyer B.: *Object-Oriented Software Construction*, Second Edition, Prentice Hall PTR, New Jersey, 1997
- [3] Johnson R.: *Documenting Frameworks Using Patterns*, Object-Oriented Programming Systems, Languages, and Applications conference proceedings, pp. 63-76, Vancouver, British Columbia, Canada, ACM Press, October 1992.
- [4] Pirklbauer K., Plösch R., Weinreich R.: *Object-Oriented and Conventional Process Automation Systems*, Proceedings of 39th International Scientific Colloquium at TU Ilmenau, Germany, September 27-30, 1994, pp. 566-571, Bd. 3, ISSN 0943-7207.
- [5] Pomberger G., Blaschek G.: *Software Engineering*, Carl Hanser Verlag, 1996
- [6] Pree W.: *Design Patterns for Object-Oriented Software Development*, Addison-Wesely, 1995
- [7] REFORM: *A Reusable Framework for Rolling Mills*, online at <http://www.ssw.uni-linz.ac.at/REFORM/home.html>, accessed May 1998.
- [8] Sedgewick R.: *Algorithms - Second Edition*, Addison-Wesley, Publishing Company, Inc. Reading, Massachusetts, 1988
- [9] Siegel J.: *CORBA Fundamentals and Programming*, John Wiley & Sons, Inc. 1996
- [10] Stroustrup B.: *The C++ Programming Language*, Third Edition, Addison-Wesley 1997
- [11] Taligent: *Building Object-Oriented Frameworks*, online at <http://www.ibm.com/java/education/oobuilding/index.html>, accessed September 1998