

Working with Business Patterns & Frameworks: A Case study for Fuzzy Logic Control

Nathalie Gaertner

Bernard Thirion

Laboratoire EEA, Groupe LSI
Université de Haute-Alsace
12, rue des frères Lumière
68093 Mulhouse Cedex, France
+33 (0)3.89.33.69.72
n.gaertner@essaim.univ-mulhouse.fr

ABSTRACT

Frameworks are generic applications, described by a set of abstract classes and the way instances of their subclasses collaborate. They allow a rapid development of new applications through customization. However, two main problems are related to frameworks. First, designing a framework is a highly complex, time-consuming work and secondly, understanding the overall architecture and how to use it is difficult. One way to improve this situation is to include business and design patterns in the framework's architecture. Indeed, each pattern provides a concise and useful architectural guidance to a related problem. Moreover, the (re)use of patterns in software developments allows the integration of flexible modular adaptable well-engineered solutions at a higher level than classes. Business patterns are domain-specific patterns. Integrating these patterns into frameworks – both related to the same business – makes it possible to exploit the generic architecture of frameworks along with the high level abstractions, business knowledge and documentation of the patterns. A fuzzy logic control framework is outlined in the present paper to demonstrate the synergetic approaches of business patterns and frameworks.

Keywords

Business patterns, framework, fuzzy logic, design patterns, reuse.

1 INTRODUCTION

The synthesis of recurrent solutions through patterns originated from the pioneering work of architect Christopher Alexander at the end of the seventies [2]. The software community borrowed Alexander's idea of using patterns and pattern languages and applied it to software development. Software patterns are a way to transfer and to share expert knowledge by providing solutions to general software problems in particular contexts and also by documenting the solutions. Their associated documentation is the key element of the reuse process. Indeed, a pattern's documentation not only explains a reliable solution, but also the pattern's intent, the contexts in which the pattern can be applied, the reasons which lead to this solution, the related benefits, some drawbacks and trade-offs, some

examples found in real systems, some pitfalls and hints, etc. Thus, patterns are a way to record, document and communicate knowledge of software experts. Design patterns are the precursors of software patterns [5, 6, 11, 21]. They improve productivity and, above all, quality by recording experiences and knowledge of object-oriented designers.

Expert designers do not solve problems from scratch; they rather reuse architectural solutions that have demonstrated their efficiency in the past. Design patterns synthesize the iterative design of those well-engineered solutions, which have been continually improved to become more flexible and adaptable. As they are independent of any application or domain and are not implemented, their usefulness and applicability are extended. However, these patterns are only applied during design and they offer no support during the important and complex analysis phase. Business patterns [9, 12, 13, 19] are a way to capture and communicate domain specific expert knowledge. They offer analysis and design reuse within their domain and take advantage of the pattern concept to document solutions to recurrent problems.

Another way to reuse analysis and design is to utilize a framework. A framework is an abstraction of several applications related to the same particular domain. The claimed benefits of domain-specific frameworks are the synthesis of reliable analysis, design and code in order to reduce maintenance and development costs and to improve software quality [8]. As they are composed of lots of classes, customization requires the understanding of the static and dynamic structure. Therefore, using business patterns and their above-mentioned advantages facilitates the creation and the documentation of frameworks.

This paper first presents the concept of business patterns and frameworks. An example of a fuzzy logic business pattern illustrates the concept of business patterns, which is not known so well. The paper then compares business patterns to frameworks according to different criteria, such as flexibility, granularity or maintenance facilities. This comparison aims to demonstrate the benefits of their joint use. Finally, the overall design of an object-oriented

framework related to fuzzy logic control (FLC), that we had developed, is used to demonstrate how business patterns facilitate the development, maintenance, documentation and understanding of the framework.

2 BUSINESS PATTERNS VS FRAMEWORKS

In this section, business patterns are compared to frameworks. This comparison highlights the benefits of their complementary integration.

Business patterns

A business pattern is a reliable domain-dependent abstraction that solves a specific problem. It documents and communicates expert knowledge, independently of any

programming language.

Several business patterns are already extracted from specific domains. However, they are not systematically named that way. Business patterns exist in domains and businesses ranging from graphical user interfaces [5] to communication and distribution [5, 19, 22], banking [3], computer integrated manufacturing [1], data management [13]. To illustrate the concept of business patterns, an example taken from an interesting domain - in this case fuzzy logic - is presented in Figure 1. A short documentation is provided to simplify the illustration. The description format, called template, is similar to that used in [11] and UML [20] as the modeling language.

NAME: *LINGUISTIC VARIABLE*

INTENT

Captures imprecise and non quantifiable information.

MOTIVATION

Humans frequently use imprecise and non-accurate information. They use this information, measured by their sensors, to elaborate actions. For example, a person will feel “very hot” in a room, but will not be able to give an accurate indication of the temperature, say 101.5°F. Fuzzy Logic expresses this kind of information in a linguistic manner. The temperature can thus be defined as “rather hot”, “more or less temperate” or “very cold”.

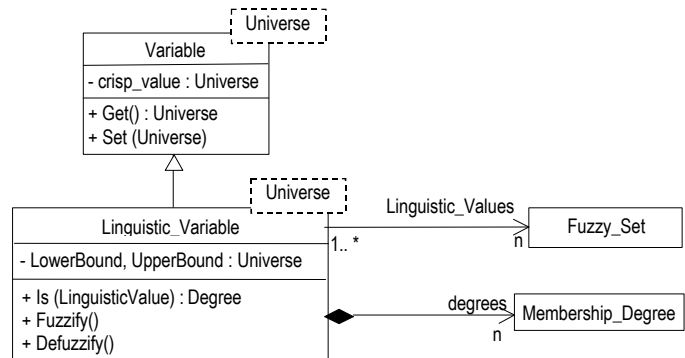
Classic sets with crisp boundaries are not suited for reflecting such imprecise information. For instance, if a temperature is defined as: cold < 55°F, hot >= 85°F and 55°F <= temperature < 85°F then, 55°F is defined as temperate; however, 54.9°F, just a tenth of a degree less than 55°F, is defined as cold. Although this is correct, the result is not the expected one, actually. Indeed, is it relevant to say that 54.9°F is cold while 55°F is temperate?

The fuzzy set theory [23] provides a way to deal with non-quantifiable information linguistically. A fuzzy set is a set with graded membership degrees (or membership values). The membership degree μ is a value in the range [0.0, 1.0], with 0.0 representing absolute falseness and 1.0 absolute truth. Membership values modulate progressively the membership of a variable to a fuzzy set. Thus, a linguistic variable, like the above temperature, is not defined with a unique fuzzy set. In fact, a temperature can be temperate **and** cold according to different membership degrees. For example, 55°F can be felt as «not really cold» and «rather temperate» thanks to the membership degrees $\mu_{\text{cold}}(55.0) = 0.3$ and $\mu_{\text{temperate}}(55.0) = 0.7$. A temperate temperature is a particular case where $\mu_{\text{cold}}(x) = 0.0$, $\mu_{\text{temperate}}(x) = 1.0$ and $\mu_{\text{hot}}(x) = 0.0$.

APPLICABILITY

This pattern should be used when a fuzzy piece of information should be recorded in a variable.

STRUCTURE



PARTICIPANTS

- ◆ **Variable**
Defines the abstraction of a variable. As the value recorded by this abstraction can be of various ‘types’, a generic type named `Universe` is used.

- ◆ **Linguistic_Variable**
Extends the `variable` class for the encapsulation of fuzzy variables. To constrain the value of a linguistic variable, two attributes `LowerBound` and `UpperBound` are added. Thus, a temperature’s value is restricted according to the kind of temperature (for example, the temperature of the human body can be limited to 97°F and up to 106°F).

The name of the linguistic variable is given by the object, an instance of the `Linguistic_Variable` class. For example,

```
Temperature: Linguistic_Variable <int>
```

- ◆ **Fuzzy_Set**
Defines more or less complex fuzzy sets.
- ◆ **Membership_Degree**
Records membership degrees limited to 0.0 and up to 1.0.

COLLABORATIONS

To convert a numerical value into a membership degree – a method called fuzzification - a crisp value is needed. Indeed, the temperature must be known, for example 55°F measured by a thermometer, so as to deduce that it is not very cold. By contrast, extracting a crisp value from a fuzzy set to get a representative value of this set is called defuzzification.

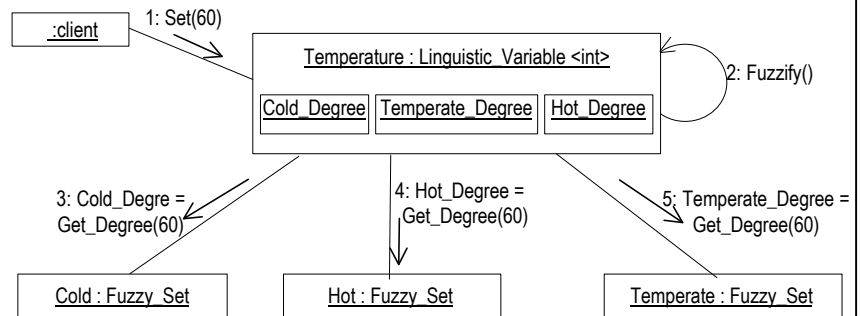
A linguistic variable is defined with different membership degrees, each related to a fuzzy set. To get each of these membership degrees, the `Linguistic_Variable` class implements a method called `Is`. This method uses a fuzzy set identifier as a parameter and returns a `Membership_Degree`. Thus, to obtain the degree of membership to the hot fuzzy set, the following instruction is used:

```
degree = temperature.Is(hot).
```

Instances of the `Linguistic_Variable` class and `Fuzzy_Set` class collaborate to obtain the membership degree of a fuzzy set. A first approach could delegate the evaluation of the membership degree to the `Fuzzy_Set` class. However, in some cases, this approach is not efficient, because the same membership degree of the same fuzzy set may be evaluated several times. This could happen, for example, during the inference of a set of fuzzy rules, when several fuzzy rules need the evaluation of the same fuzzy set's membership degree. To improve this, the `Fuzzify` method of the `Linguistic_Variable` class evaluates all the membership degrees of all the fuzzy sets and record them. These recorded degrees are then used by the `Is` method.

The `Set` method, inherited from the `Variable` class, can be overloaded to perform the fuzzification operation.

The following collaboration diagram shows how the `set` method starts fuzzification and records all the membership degrees of a temperature.



CONSEQUENCES

No complex fuzzy logic concepts, such as fuzzy rules or fuzzy inference engines, need to be defined to record a linguistic variable.

This business pattern synthesizes a simple and concise solution. However, it has different trade-offs:

- ◆ Numerous evaluations of membership degrees related to fuzzification can be avoided with a recording. This is useful when linguistic variables are used in fuzzy rules.
- ◆ When a linguistic variable is related to several fuzzy sets, and when only evaluations for one or two fuzzy sets are needed, recording all the membership degrees of all the fuzzy sets is not efficient. However this is not frequent, because rule sets often take all the possibilities into account (consistency of a rule set); thus membership degrees of all the fuzzy sets for each linguistic variable are often evaluated.
- ◆ When few evaluations of membership degrees are done, the recording could be ignored.
- ◆ The pattern can also be adapted or modified before being implemented, to take into account specific constraints or to add extensions.

RELATED PATTERNS

Fuzzy set.

Figure 1 : Short description of the Linguistic Variable business pattern

Frameworks

A framework provides a generic architecture and behavior for a family of software applications within a domain. It is made up of a set of interconnected classes, which defines the architecture, the control flow and how instances of those classes collaborate. Therefore, frameworks have complex, flexible and extensible designs. More details about frameworks can be found in [3, 8, 15].

Frameworks are related to several domains, but mainly the Graphical User Interface (GUI) domain. Smalltalk's MVC, MacApp or Microsoft Foundation Classes are some

examples of GUI frameworks. The reason why these generic applications are not widely developed is because the design of reusable highly-generic solutions is an iterative, time-consuming, complex task.

Granularity

A first element of comparison between business patterns and frameworks is the reuse level. Each pattern gives an arrangement of classes and/or objects and defines the collaborations between the instances of these classes. Thus, they provide reusable micro-architectures at a higher level than those provided by single classes.

On the contrary, frameworks are semi-complete applications that allow the rapid prototyping of whole applications if the developer knows about the framework used. Thus, they allow large-scale reuse with a larger granularity than classes and patterns.

Reuse level

Sometimes, it is less time-consuming to write a component from scratch than to search for it in a library and understand its behavior, or than to implement a business pattern. For some applications, reusing coded or partly coded solutions, even non-flexible ones, can be attractive. But at other times, design or analysis reuse is more interesting than code reuse, because reuse starts earlier in the development process. Thus, depending on the application and the problem at hand, a developer has to choose between component reuse, pattern reuse and/or framework reuse.

On the one hand, business patterns provide a way to reuse one model of a domain’s subset. They are built by one or more specialists and do not depend on specific applications or implementations; they can be adapted and extended while being reused. As business patterns encapsulate analysis, they are only useful within their domain, contrary to design patterns. But they can make use of design patterns to integrate flexible and reliable architectural solutions into their design. Widespread design patterns like the template method, the composite pattern or the strategy pattern often appear in the design of business patterns. On the other hand, domain-specific frameworks synthesize reliable analysis, design and code in order to reduce maintenance and development costs and improve software quality [8]. So, frameworks reuse analysis, design and code to provide high reuse rates.

Intelligibility

Intelligibility is the key to reuse enhancement. The documentation of patterns provides a great amount of information to make them understandable and reusable. The knowledge of the experts who defined the pattern is written down so as to capture and communicate their expertise. The static architectural solution, dynamic information and instructions on how and when to use the pattern are recorded. This knowledge is useful for developers, especially less experienced ones. Indeed, the object-oriented technology offers several design alternatives for solving a software problem. As patterns synthesize and document appropriate and judicious solutions, they act as pedagogical materials.

To reuse generic applications, such as frameworks, it is necessary to understand the intent and the behavior of the framework’s classes. Business patterns, as well as design patterns, can enhance the intelligibility of such large complex assembly of classes.

Flexibility and extensibility

Modification and extension capabilities depend on how the framework or the pattern were designed [15]. But as frameworks are implemented to allow the rapid prototyping of applications, they depend on a programming language. As a result of this code reuse, flexibility and adaptability are restricted. By contrast, business patterns provide reusable, language independent solutions that are more extensible and adaptable; but a coding effort is needed.

As the patterns must be coded to be integrated into an application during the implementation phase, they are less useful for non-software engineers. However, the documentation of problem resolutions that can be adapted, interconnected, or extended is valuable for software engineers.

Maintenance

Software maintenance is simplified through the integration of patterns as they document the solution provided. In some cases, even maintenance information can be added to the pattern’s documentation.

Each framework provides a common architecture. When maintenance modifications are made to the framework, all the newly created applications based upon this framework will benefit from the modifications. The maintenance of already specialized applications is also facilitated, because the modifications are similar from one application to another.

Using patterns within frameworks

To synthesize the previous argumentation, all the characteristics of patterns, components and frameworks are summarized in Figure 2.

The complexity and the size of frameworks are the main problem for their reuse. To customize a framework, the developer must be familiar with the overall structure, how the classes and objects collaborate, when the framework invokes a specific method, etc. However, if a programmer knows about the framework, the benefits are manifold.

	Design pattern	Business pattern	Business object	Framework
Programming language independence	<i>Yes</i>	<i>Yes</i>	<i>No</i>	<i>No</i>
Domain dependence	<i>No</i>	<i>Yes</i>	<i>Yes</i>	<i>Yes</i>
Granularity	<i>Medium / Low</i>	<i>Medium</i>	<i>Low / Medium</i>	<i>High</i>
Flexibility	<i>High</i>	<i>High</i>	<i>Medium</i>	<i>Good</i>
Comprehension	<i>Easy</i>	<i>Easy</i>	<i>Medium</i>	<i>Complex</i>
Integration facilities	<i>High</i>	<i>High</i>	<i>Medium</i>	<i>Low</i>
Reuse phase	<i>Design</i>	<i>Analysis & Design</i>	<i>Coding</i>	<i>Analysis, Design & Coding</i>

Figure 2 : Characteristics of design pattern, business patterns, business objects and frameworks

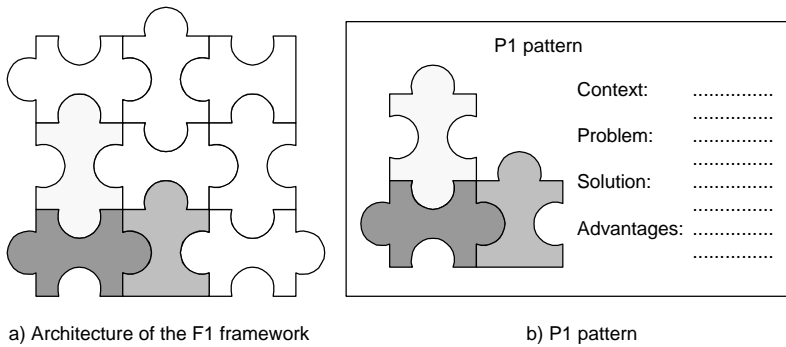


Figure 3 : Documentation of a framework with a pattern

The use of business patterns (as well as design patterns) in frameworks is a promising research area [4, 16, 17, 18]. They provide higher abstraction levels than those provided by classes and objects. They provide a well-defined and unified vocabulary and reduce the complexity of the framework and improve its understandability (cf. Figure 3). Of course, the user needs to know about the concept of patterns and uses this concept in an effective manner (understanding a framework only through the development language would be harder for him).

The benefits of integrating business patterns or design patterns into frameworks differ according to the family of the patterns. Business patterns provide abstractions from the domain and can therefore be integrated during analysis and design. On the other hand, design patterns provide domain independent architectural solutions. Thus, design patterns help to create and understand the design of frameworks, whereas business patterns analyze and also document part of the domain.

The idea of associating business patterns and design patterns with frameworks and business components is a way to exploit all the advantages of these abstractions. Reliable, flexible, documented and easily understandable artifacts are thus created. For example, it is possible to take advantage of the implementation provided by a framework linked with the documentation of business patterns. Thus, the choices made during analysis and design are explained and references to other related business patterns are given. This simplifies the understanding and use of the framework.

3 FUZZY KNOWLEDGE BASED CONTROL FRAMEWORK AND BUSINESS PATTERNS

Fuzzy knowledge based control (FKBC) [7, 10, 14] is an original interesting research area based on fuzzy logic. It is well suited for modeling human reasoning in a simple neat manner, using a knowledge based system and an inference procedure. In order to reuse the inference system and the fuzzy knowledge database structure in various applications, the design of the controller should be extensible, adaptable, easily reusable and understandable. The creation of a framework

for fuzzy knowledge based control is a way to achieve this aim and allows rapid prototyping of new applications.

Purpose

Fuzzy knowledge based control is used in different businesses or domains. It integrates human expertise, using fuzzy logic and knowledge based systems. It allows the modeling of human reasoning, using linguistic variables and fuzzy “If-Then” rules. For example, the control strategies of operators can easily be modeled, because knowledge can be expressed in a natural, understandable way. In control theory, FKBCs are generally used when the process to be controlled is non-linear or when conventional control methods offer no clear solutions.

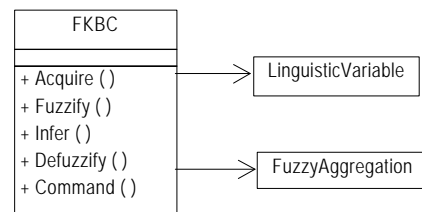
An object-oriented black-box FKBC framework has been developed [10]. It is a flexible generic architecture that can be used to improve the development of supervision control or diagnosis, but also for planning and scheduling.

The FKBC framework has a Facade [11] called FKBC, which has the following interface:

Overall design

An overview of the fuzzy knowledge based control framework is presented in Figure 4. Some of the business patterns integrated in the architecture – such as the linguistic variable presented before - are highlighted.

The FKBC framework uses the Infer method to define the values of the output variables that will be sent to the controlled process.



Starting with this Facade class, a user can consult the FuzzyAggregation and LinguisticVariable patterns to understand the usefulness and the behavior of their

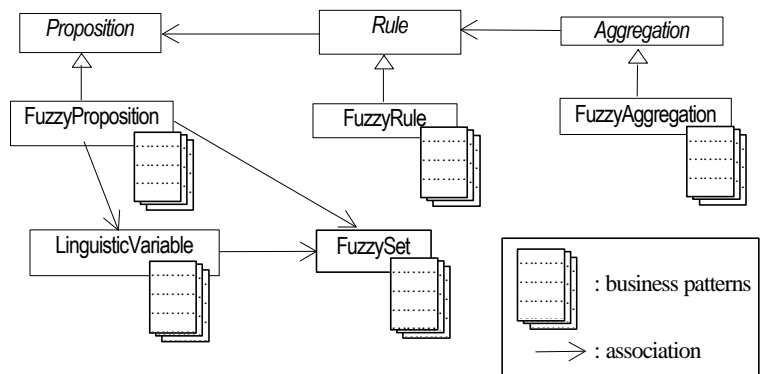


Figure 4 : FKBC design

associated classes. The related business patterns of these two patterns further describe the framework, and the new related patterns describe it further, etc.

The developed business patterns define and describe in detail the FKBC framework classes. Here is just a short explanation of the main framework's abstractions:

➤ Several rules can be validated at the same time, because a linguistic variable has a membership degree for each fuzzy set. For example, take the two following aggregated rules:

```
Heating = IF temperature==Temperate
          THEN maintain
          OR  IF temperature==Cold
          THEN increase
```

The output command for heating will be calculated by combining the results of the two rules. This combination is called fuzzy aggregation.

➤ Each fuzzy rule has the following format: "IF premise THEN consequence" where premise and consequence are fuzzy propositions. When evaluated, a fuzzy rule defines an output fuzzy set.

➤ Propositions can be simple or complex and connect linguistic variables to linguistic values (represented by fuzzy sets). For instance, "Age is young and Height is tall" is a proposition. The evaluation of a proposition returns a value, which is used in a rule's premise; whereas the assignment of a proposition is used in a rule's consequence.

Implementation and testing

The fuzzy knowledge based control framework was developed in C++ and tested. An example of simulation was the implementation of power regulation for traffic lights to increase road safety and save energy [10]. The reuse of the FKBC framework for real process control is currently being tested.

A graphical user interface (GUI) can easily be integrated into the instantiated framework thanks to the unified approach of object-oriented technology. On-line modifications can also be carried out to improve the FKBC system. Indeed, flexibility through dynamic binding has been introduced to modify the fuzzy controller without recompilation. This is a relevant contribution of this framework as no general guidelines for the tuning of a fuzzy controller exist.

4 CONCLUSION

Business patterns provide the expected features and benefits of patterns: programming language independence, documentation of the solutions, flexibility, adaptability, etc. However, in contrast to design patterns, business patterns are closely related to a specific domain. An example extracted from the fuzzy logic domain has been presented to illustrate this business pattern concept.

Business patterns, frameworks and business objects have been compared so as to highlight the differences between these domain-dependent abstractions. They provide, among

other things, different kinds of reuse (analysis reuse, design reuse and/or code reuse), more or less flexibility and different granularities. Each domain-dependent abstraction has its own advantages and disadvantages; but the synergetic use of these domain-dependent artifacts takes advantage of the benefits of each artifact and compensates for some of their drawbacks.

We have tested this synergetic use with a fuzzy knowledge based control (FKBC) framework. The design of this framework includes several business patterns to facilitate its creation and understanding. The created framework offers analysis, design and code reuse, whereas fuzzy logic business patterns – with their associated documentation – facilitate the understanding and reuse of the generic architecture. The overall design of this FKBC framework has been outlined to show the integrated fuzzy logic business patterns.

5 REFERENCES

1. Aarsten A., Elia G. and Menga G. G++: A Pattern Language for Computer Integrated Manufacturing in Pattern Languages of Program Design, Addison-Wesley, NY, 1995.
2. Alexander C., Ishikawa S., Silverstein M., Jacobson M., Fiksdahl-King I. and Angel S. A Pattern Language, Oxford University Press, New York, 1977.
3. Bäumer D., Gryczan G., Knoll R., Lilienthal C., Riehle D. and Züllighoven H. Framework Development for Large Systems, Communication of the ACM, 40(10), pp. 52-59, October 1997.
4. Beck K. and Johnson R., Patterns generate architectures, ECOOP'94 (European Conference on Object-Oriented Programming), Bologna, Italy, pp. 139-149, July 1994.
5. Bushmann F., Meunier R., Rohnert H., Sommerlad P. and Stal M. Pattern-oriented software architecture: a system of patterns, Wiley, 1996.
6. Coad P. Object-oriented patterns. Communications of the ACM, 35(9) pp. 152-159, September 1992.
7. Driankov D., Hellendoorn H. and Reinfrank M. An Introduction to Fuzzy Control, Springer-Verlag, Berlin, 1993.
8. Fayad M. E., Schmidt, D. C. and Johnson, R. E. Object-oriented Application Framework: Problems and Perspectives, Wiley, NY, 1997.
9. Fowler M. Analysis Patterns: Reusable Object Models, Addison-Wesley, Reading MA, 1997.
10. Gaertner N. and Thirion B. Object-Oriented fuzzy control: An integrated and flexible approach. International Conference on Computational Intelligence for Modelling Control and Automation'99 (CIMCA'99), Vienna, Austria, 1999.
11. Gamma E., Helm R., Johnson R. and Vlissides J. Design patterns: elements of reusable object-oriented software, Addison-Wesley, Reading MA, 1995.

12. Haugen R. Dependent Demand – a Business Pattern for Balancing Supply and Demand in the 4th Pattern Languages of Programming Conference, Washington University Technical Report 97-34.
13. Hay D.C. Data Model Patterns: conventions of Thought, Dorset House Publishing, New York, 1996.
14. Jager R. Fuzzy Logic in Control, Thesis Technische Universiteit Delft, 1995.
15. Johnson R. E. and Foote B. Designing reusable classes, Journal of Object-Oriented Programming, 1(2):22-35, June/July 1988.
16. Johnson R.E. Documenting Frameworks Using Patterns, OOPSLA '92 Proceedings, SIGPLAN Notices, 27(10): 63-76, Vancouver BC, October 1992.
17. Johnson R. E. Frameworks = (Components + Patterns), Communication of the ACM, Vol. 40, No. 10, pp. 39-42, October 1997.
18. Lajoie R. and Keller R. Design and reuse in object-oriented frameworks: patterns, contracts, and motifs in concert, Proceedings of the 62nd congress of the Association Canadienne Française pour l'Avancement des Sciences (ACFAS), Montreal, Canada, pp. 94-105, May 1994.
19. Mowbray T. J. and Malveau R. C. Corba Design Patterns, Wiley Computer Publishing, 1997.
20. Object Management Group. Unified Modeling Language – UML Notation Guide Version 1.1. Ad/97-08-05. Framingham, Mass. November 1997.
21. Pree W. Design Patterns for Object-Oriented Software Development, Addison-Wesley, 1995.
22. Schmidt D. C. Using Design Patterns to Develop Reusable Object-Oriented Communication Software, Communication of the ACM 38(10), pp. 65-74, October 1995.
23. Lofti A. Zadeh. Fuzzy sets. Information and Control, vol. 8, 1965, pp. 338-353.