Materialization Patterns: A Bridge Between Software Architecture and Object-Oriented Design

Germán Vázquez Lerner[1], Marcelo Campo[1,2], Andres Díaz Pace[1,2]

[1] ISISTAN Research Institute, Fac de Cs. Exactas, UNCPBA
Campus Universitario, Paraje Arroyo Seco, Tandil, 7000, Argentina
[2] CONICET, Comisión Nacional de Investigaciones Científicas y Técnicas, Argentina
email {gvazquez, mcampo, adiaz } @exa.unicen.edu.ar

Abstract. Software architectures enable to capture early design decisions in the software design process in order to fulfill system quality attributes. In principle, many object-oriented implementations can be derived from the same architecture. However, there may be potential mismatches between the quality attributes prescribed by an architecture and those fulfilled by a derived object-oriented system. We argue that a significant step towards reducing these mismatches is the organization of a body of design knowledge and systematic reasoning procedures enabling tool support for semi-automating the materialization of software architectures. In this paper, we propose a pattern-based approach consisting of a catalog of object-oriented pattern specifications and a set of design activities to perform materializations of software architectures in such a way the quality attributes are preserved in the derived systems.

Keywords: software architecture, quality attributes, object-orientation

1 Introduction

Software architectures are a crucial abstraction in the software design process, and in the last years, they have became of great interest for the software research community. As system designs become more and more complex, the quality of the final products needs more attention. Software architecture design enables the description of the high level organization of a system, capturing early design decisions in order to fulfill quality attributes of systems such as modifiability, reusability, performance, security, among others [13][3] .

Architectures depict the gross organization of systems independently of implementation issues. Given an architecture design and a set of quality drivers, there are many possible object-oriented implementations that can be derived from it. This mapping between architecture and object-oriented designs is what we call *object-oriented materialization of software architectures* [5].

This work was partially supported by the ANPCT (PICT 11-10771)

However, there may be potential mismatches between the quality attributes prescribed by the architecture and those achieved by the derived object-oriented system. These mismatches arise due to the differences in the level of abstractions and capabilities of architecture design and object-oriented design. While, architectural models are suitable to address aspects orthogonal to system functionally (e.g. concurrency, distribution, real-time constraints, persistence and failure recovery), the object-oriented paradigm is mostly focused on modifiability and reusability issues, accordingly to its intrinsic characteristics of information hiding and encapsulation. Therefore, an object-oriented implementation of a software architecture is not necessary compliant with the quality attributes prescribed by that architecture, resulting in a gap between the software architecture and object-oriented worlds. Moreover, the derivation of object-oriented materializations requires usually developers with important background of design knowledge, expertise and experience.

Along this line, we think that the provision of a body of design knowledge and systematic reasoning procedures to aid developers in the materialization of software architecture activities is a significant step towards reducing the gap between architecture and object-oriented models. This is an evolution of the ideas developed in the ArchMatE approach [6].

In this context, we aim to enable tool development for supporting semi-automated materialization of software architectures. To do so, we propose a catalog of object-oriented pattern specifications and a set of design activities, which if adequately performed, can help developers to materialize software architectures. The use of pattern specifications enable us to systematize the representation of the mappings between the architecture and object-oriented worlds, making those mappings computationally tractable.

The rest of the paper is organized around 4 sections as follows. Section 2 describes the foundations of the proposed approach. Section 3 introduces the patterns proposed for generating object-oriented implementations of software architectures. A materialization example is presented in Section 4. In Section 5 related work is discussed. And finally, Section 6 analyzes lines of future research and rounds up the conclusions of the paper.

2 Object-oriented Materialization of Software Architectures

The software architecture of a system is a high-level specification that comprises software elements, the relationships among them and their externally visible properties [3]. Components are the primary computational elements and data stores of a system. Components interact by means of connectors. The relationships between components and connectors define the overall system configuration. Also, components have responsibilities and ports.

System functionality is realized by responsibilities distributed among components. These responsibilities are application-specific, and vary from system to system based on the particular functionality systems must implement. Consequently, the object-oriented materialization of those responsibilities is application-specific too. For this reason, they are not consider in this approach, and are left unbound to be specified at lower level design stages.

The way components interact with the environment is defined by their assigned ports. Ports define components boundaries and interfaces for services. The implementation of these ports is not dependent on system functionality, but on the contrary, is largely influenced by the quality attributes prescribed by the architecture.

On the other hand, an important contribution of software architecture is the recognition of connectors as first-class entities [2][13][14]. Connectors are the medium through which components interact with each other in order to realize their allocated responsibilities. Connectors are shared by systems in many application domains. Therefore, the implementation of connectors can be also reused across systems independently of the specific application domain.

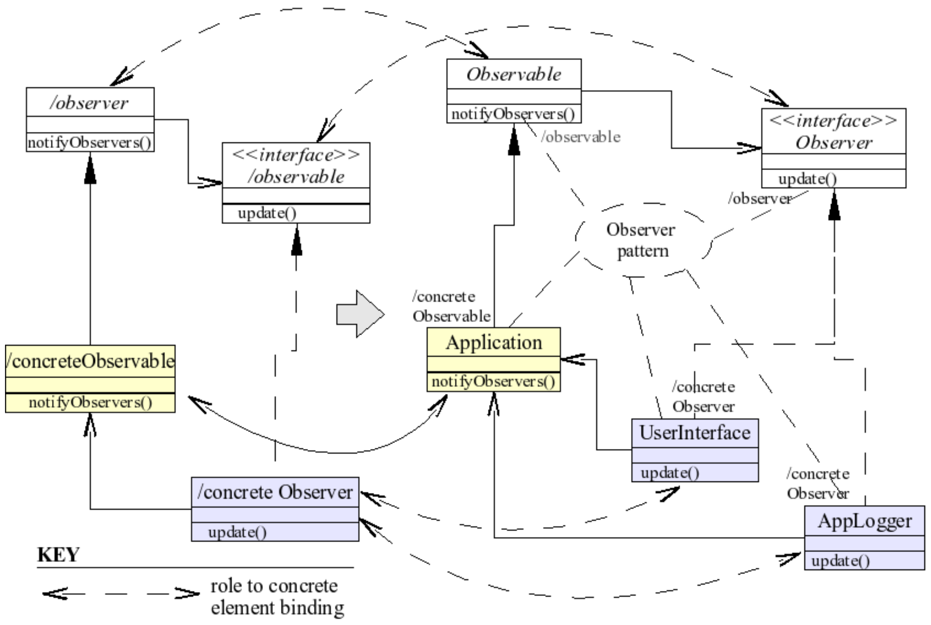

Fig. 1. An instantiation example of the observer design pattern.

In other words, ports and connectors give the underlying support for realizing component interactions in order to perform system functionality. We envision the implementation of this support as a set of recurring object-oriented design fragments with common general characteristics. Consequently, materialization of architectural elements is achieved by means of domain-independent object-oriented structures modeled as generic pattern specifications [8]. A pattern specification is a generalized interface mechanism that consists of a collection of roles to be played by predefined design elements plus a set of constraints defined on these roles . Then, a pattern can be instantiated by some software design fragment, by binding or linking the pattern roles to concrete design elements in such a way the constraints are satisfied. Figure 1 depicts an example of the instantiation of the Observer design pattern as a pattern [7].

We propose the organization of a body of design knowledge for aiding developers in the materialization of software architectures by systematizing the generic pattern specifications used for deriving implementations of architectural elements. The approach resembles the use of design patterns for solving object-oriented design problems [7]. Basically, the problem of architectural materialization is addressed by the selection of the appropriate pattern specifications according to a given architectural definition and a prescribed set of quality drivers [6]. Then, we perform the instantiations of those patterns by binding them to concrete object-oriented elements, generating finally an object-oriented system design. An outline of the proposed approach is given in figure 2.

We define a *materialization pattern* as a generic pattern specification used for deriving implementations of software architecture elements. Materialization patterns represent the bridge between architectural and object-oriented designs.

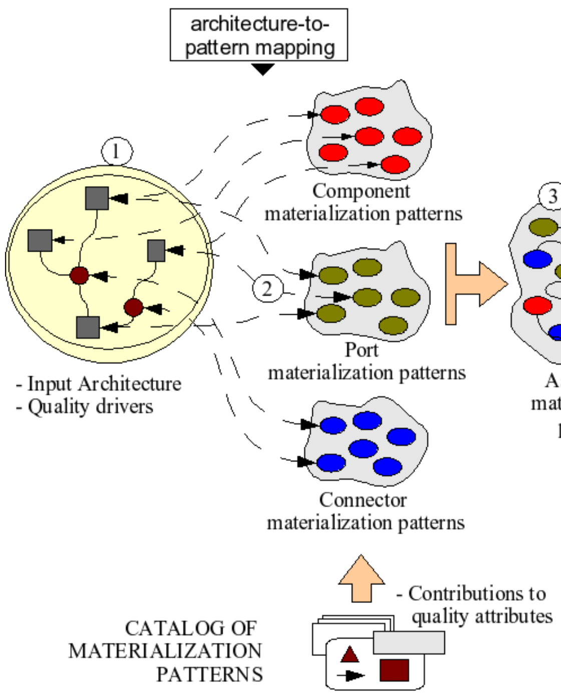
architecture-to-pattern mapping
patterns-to-object mapping
1
2
3
4
Component materialization patterns
Port materialization patterns
Connector materialization patterns
- Input Architecture
- Quality drivers
Assembled materialization patterns
Generated object-oriented system
- Contributions to quality attributes
CATALOG OF MATERIALIZATION PATTERNS

Fig. 2. Software architecture materialization approach.

The workflow of activities developers should perform to generate materializations of software architectures is the following:

1. The developer specifies the given system architecture and the desired level of quality attributes prescribed by the architecture.
2. Based on the architecture definition and the quality attributes prescribed by the given architecture, the adequate materialization patterns are selected from the available catalog of patterns and used for materializing components, ports and connectors.
3. Afterwards, according to the architecture system configuration, the materialization patterns are assembled together conforming a single pattern specification.

4. Finally, the resulting object-oriented design outcomes by instantiating the assembled patterns by binding them to concrete object-oriented elements.

3 Materialization Patterns

A catalog of materialization patterns consisting in a set of pattern specifications is organized and used to generate object-oriented materializations of components, ports and connectors, as described below.

3.1 Component Materialization

Component types are defined in terms of their general characteristics, allocated responsibilities, assigned ports and specific features. A component type defines the implementation of a particular component instance.

Component types are materialized by one single class realizing their allocated responsibilities. Each component instance is materialized by one single object/instance derived from the class materializing the corresponding component type (every component instance must have at least one component type). Materialization patterns define also object diagrams specifying the runtime behavior of a system. Nevertheless, they are out of the scope of this paper due to space restrictions.

Specifying hierarchies of component types enables the definition of families of system architectures. In order to generate object-oriented implementations derived from component hierarchies we have defined two object-oriented design primitives modeled as generic pattern specifications: *inheritance materialization pattern* and *interface materialization pattern*. The former pattern derives hierarchies of component types by means of an object-oriented inheritance mechanism. On the other hand, interface materialization pattern is used to derive hierarchies of component types by means of a composition mechanism. This pattern is useful when components instances with multiple types are to be derived, avoiding the use of multiple inheritance in the resulting object-oriented design.

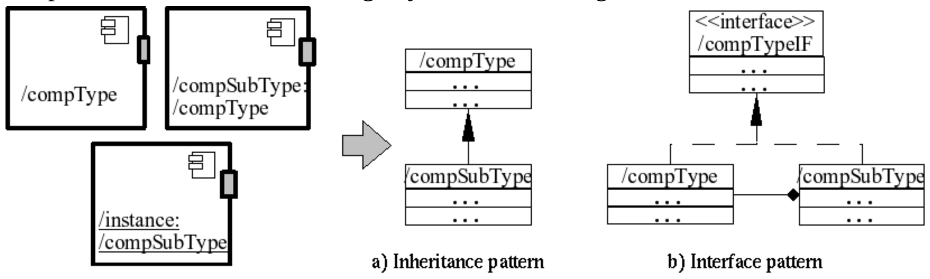

Fig. 3. Materialization patterns for deriving implementations of component hierarchies.

Figure 3 depicts both component materialization patterns. Components are modeled with UML 2.0 component diagram notation, while pattern specifications are modeled with abstract UML collaborations [12]. UML has introduced the notion of collaborations as a society of roles that work together to provide some cooperative behavior. Each element in the collaboration diagrams is shown with a back slash preceding its name, meaning that the element is a role. A role is a slot that can be bound to an actual design element (e.g., classes, objects, methods, etc.) in a system.

3.2. Port Materialization

Ports are the mean by which components interact with the environment. Ports define component service interfaces in order to allow them to require services from or to provide services to other components.

Ports are materialized by means of specific object-oriented design structures which allow objects materializing components to interact with the environment. These object-oriented structures are not application-specific, but instead they are recurrent common design structures reused across systems. Furthermore, the selection of the appropriate object-oriented designs materializing ports is driven by quality attributes, in such a way that those attributes are preserved in the generated design.

In order to derive port implementations, we have defined a set of object-oriented design primitives, in the form of generic pattern specifications, called *port materialization patterns*. Each port materialization pattern contributes to different extents to achieve specific quality attributes based on the specific object-oriented structure they define, so developers can select them consequently.

According to the way the interaction of ports with the environment is implemented, two port materialization pattern types are identified : *client port materialization patterns* and *server port materialization patterns*. Client port materialization patterns are used to generate the appropriate object-oriented structures that allow objects materializing components to send messages to objects materializing the environment. Instead, server port patterns are used to generate the object-oriented structures that allow objects materializing the environment to send messages to objects materializing components.

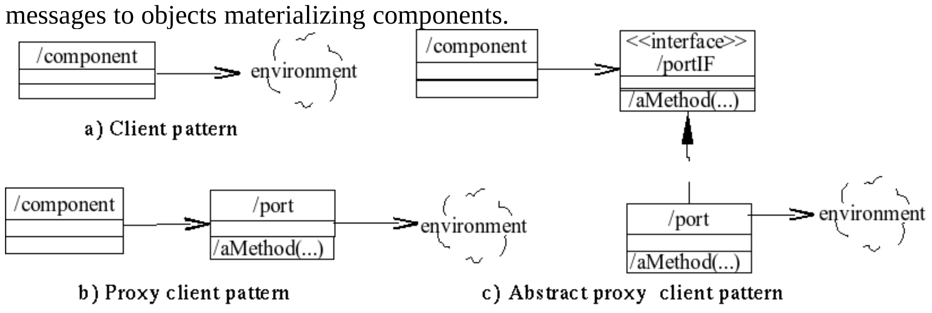

Fig. 4. Client materialization patterns for deriving port implementations.

Ports providing or requiring services interfaces can be materialized by either client or server materialization patterns depending on how those interfaces need to be implemented. The selection of the proper pattern type is given by the particular specification of the pattern materializing the connector attached to the corresponding port, as described in Section 3.3.

Analyzing different object-oriented materializations of software architectures, we have identified seven patterns used to derive component port implementations. Figure 4 and figure 5 depict client and server port materialization patterns, respectively.

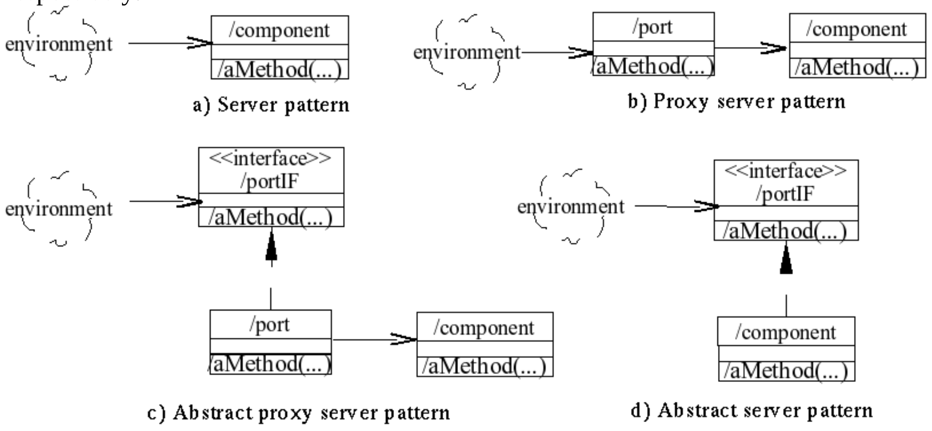

Fig. 5. Server materialization patterns for deriving port implementations.

Figure 6 presents an example of the materialization of two component types and the materialization of one of the attached ports. The component hierarchy is materialized by means of a component inheritance materialization pattern while the port is materialized by means of an abstract proxy server port pattern. The figure also shows each pattern role within its corresponding binded concrete object-oriented element.

We aim to preserve the quality attributes prescribed by an architecture in the resulting object-oriented design by selecting the proper combination of client and server materialization patterns to generate port implementations.

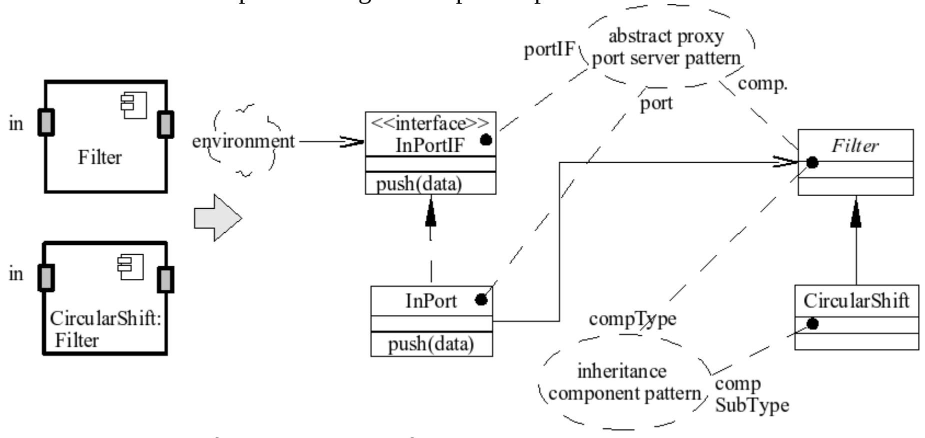

Fig. 6. An example of a materialization of components and ports.

We have analyzed several object-oriented port materializations regarding the way they influence quality attributes. The quality attributes that are mostly influenced by the way ports are materialized are performance, modifiability, integrability, security and design complexity. Table 1 resumes the level of compliance on those attributes that client port materialization patterns have. A pattern that fulfills positively a quality attribute is marked with the symbol '+', symbol '-' means that the pattern has negative implications on the corresponding quality attribute compliance level, and '+/-' means that the pattern has no influence in the corresponding quality attribute.

Table 1. Client materialization patterns and quality attribute conformance.

	Client Pattern	Proxy Client Pattern	Abstract Proxy Client Pattern
Performance	+	+/-	-
Modifiability	-	+	+
Integrability	-	+/-	+
Security	+/-	+/-	+/-
Design complexity	+	+/-	-

Finally, table 2 resumes the relationships between server port patterns and quality attributes.

Table 2. Server materialization patterns and quality attribute conformance.

	Server Pattern	Proxy Server Pattern	Abstract Proxy Server Pattern	Abstract Server Pattern
Performance	+	+/-	-	+
Modifiability	-	+	+	+
Integrability	-	+/-	+	+/-
Security	-	+	+	-
Design complexity	+	+/-	-	+

3.3 Connector Materialization

Connectors are the software entities mediating interactions among components and the glue that conforms the overall system architecture. Each connector defines roles specifying participants in the interaction of the connector. Components and connector interact together by attaching their corresponding ports and roles together, respectively.

The implementation of connectors is frequently spread among the implementation of components, so connections between objects are not explicit in the resulting design. Therefore, it is difficult to integrate implementation of connectors directly in a object-oriented design. Nevertheless, these difficulties can be addressed by performing the materialization of components and connectors separately, and then assembling them into a single design.

Connectors are identified by a type defining the connector general characteristics. There is a well- known set of standard connector types providing communication and coordination services such as procedure call, publisher/subscriber, producer/consumer or SOAP connectors, to name some of them. Additionally, services such as facilitation or adaptation are provided by more complex connectors such as load balancers or brokers [11].

The object-oriented implementation of connector types is not application-specific. Connectors of the same type are implemented by means of common object-oriented structures shared across systems. Thus, we have identified a set of common object-oriented design structures which are recurrently used for materializing connector types, modeled as pattern specifications and referred as *connector materialization patterns*.

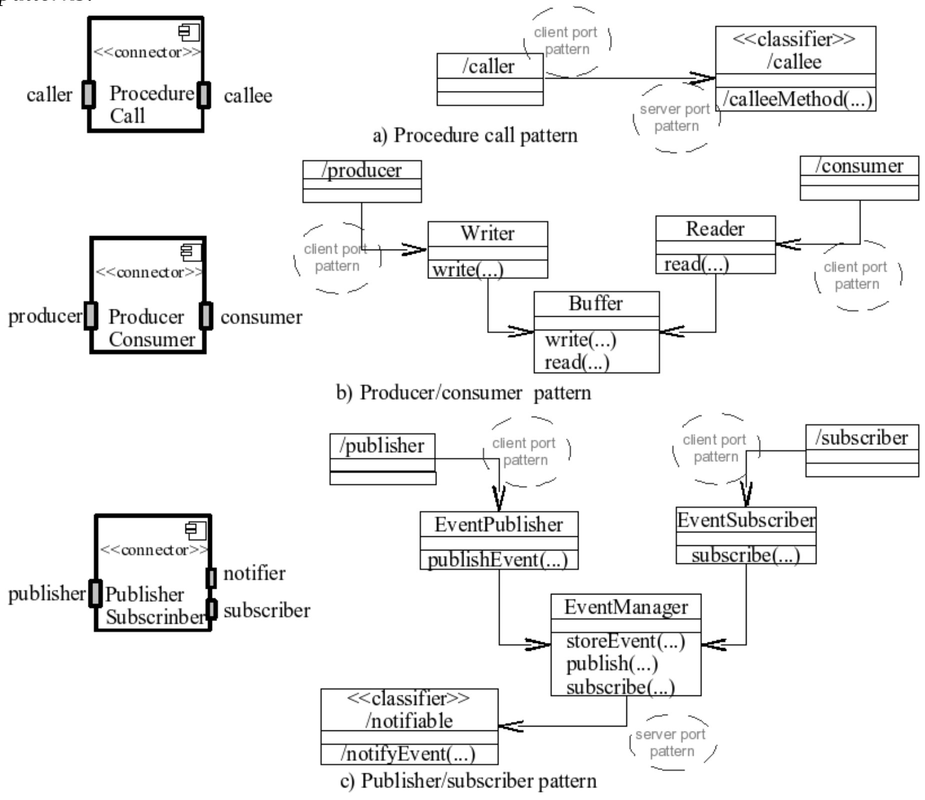

Fig. 7. Three common connector materialization patterns.

A catalog of predetermined connector materialization patterns is defined and used for deriving implementations of well-known connector types. So, for materializing a particular connector given by an architecture definition, the corresponding materialization pattern is selected from the catalog of patterns based on the specific type of the connector. Then, the pattern selected is instantiated by binding its roles to concrete object-oriented elements, generating this way the implementation of the connector.

Figure 7 shows the pattern specifications defining the static structure used for materializing procedure call, producer/consumer and publisher/subscriber connector types. Connectors are modeled as stereotyped UML2 components.

The specific implementation of each connector role determines the selection of the adequate port materialization pattern type (client or server) in order to generate the implementation of the port attached to that connector role. As shown in figure 7, an incoming association to the connector implementation prescribes the selection of a client port materialization pattern to derive the implementation of the attached port. Instead, an output association from the connector implementation prescribes the selection of a server port materialization pattern.

4 A Materialization of a Pipe & Filter System Architecture

The concepts presented in the previous sections are applied in a materialization example of an architecture based on the pipe & filter style [13]. Pipe and filter style specifies families of systems composed by filters components which are processing units transforming streams of data. Filters receive input data through their in ports, and send the processed data through their out ports. Data is passed between adjacent filters by connecting them by means of their in and out ports.

Figure 8 shows a pipe & filter architecture composed by two filters - circularShift and alphabetizer – and by two components - input and output - which serve as system data input and data output, respectively. The components are connected by a binary connector. Connectors are depicted by a stereotyped UML element with custom visualization.

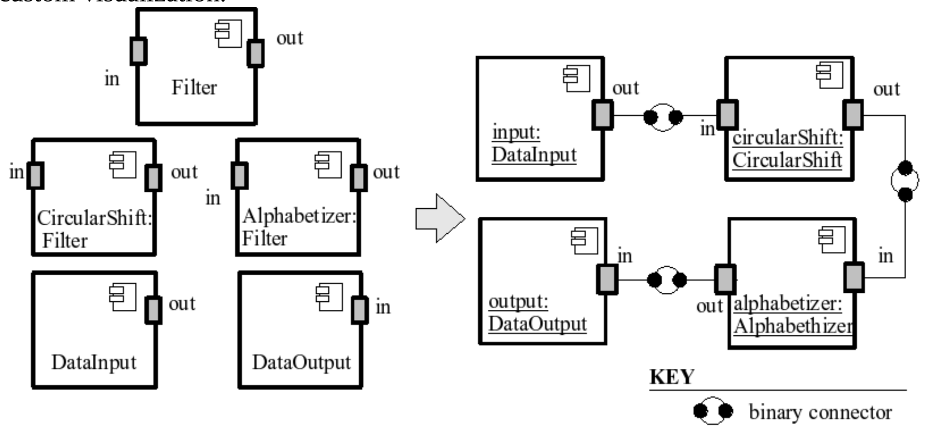

Fig. 8. A pipe and filter system.

An object-oriented materialization of the pipe & filter architecture configured with procedure call connector types is shown in figure 9. Procedure call connectors are materialized by means of the materialization pattern depicted in figure 7a. Procedure call connectors are composed by two connector role types: the caller role and the callee role. Connector caller roles are attached to component out ports, while connector callee roles are attached to component in ports.

Ports attached to connector caller roles must be materialized by client port materialization patterns. Instead, ports attached to connector callee roles are materialized by server port materialization patterns.

Figure 9 also depicts the instantiated materialization patterns along with the roles bounded to the corresponding concrete object-oriented elements. For instance, the out port of the input component is materialized using a proxy client materialization pattern. The “component” role of the pattern is bounded to the Input class in the design, and the “port” role of the pattern is bounded to the OutPort class.

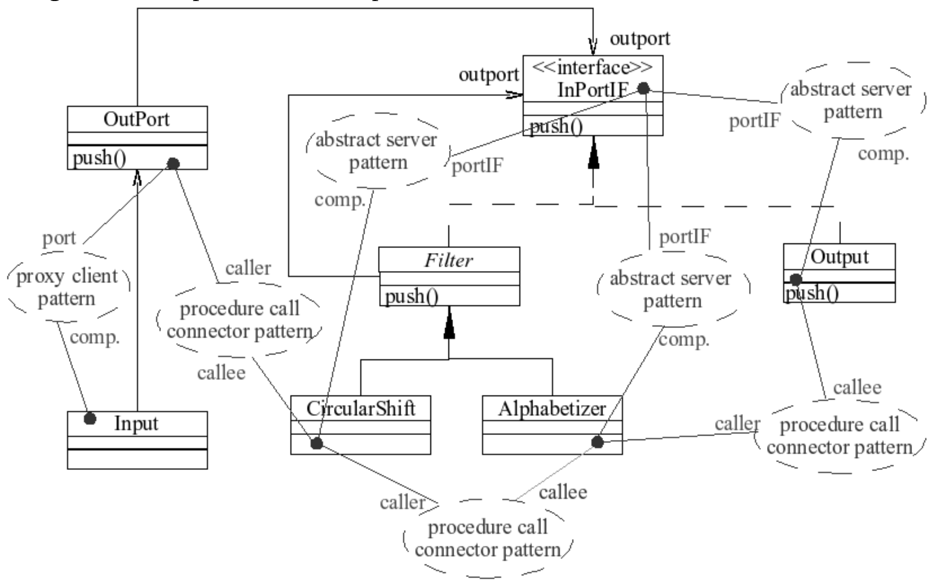

Fig. 9. A pipe and filter materialization with passive filters.

Concrete object-oriented elements can be shared by different materialization pattern roles. In the design shown in figure 9, the “port” role of the proxy client pattern and the “caller” role of the procedure call connector pattern are shared and both bounded to the OutPort class.

Another materialization example is presented by deriving the pipe & filter system configured, in this case, with producer/consumer connector types. The generated object-oriented design is depicted in figure 10. Producer roles of the producer/consumer connectors are attached to component out ports, while consumer roles are attached to component in ports. Producer/consumer connectors are materialized by instantiating the producer/consumer connector materialization pattern depicted in figure 7b.

5 Related Work

The problem of architectural materialization was historically addressed by means of application frameworks which capture architectural abstractions of related systems providing extensible templates for applications within particular domains. Different instantiations of application frameworks provide different implementations of the corresponding software architecture [9].

An approach supporting implementation of C2-based software architectures is described in [10]. The C2 style is used for building messaged-based applications, typically user interface applications. A set of object-oriented frameworks are proposed for materializing C2 systems by instantiating them so particular object-oriented implementations can be generated.

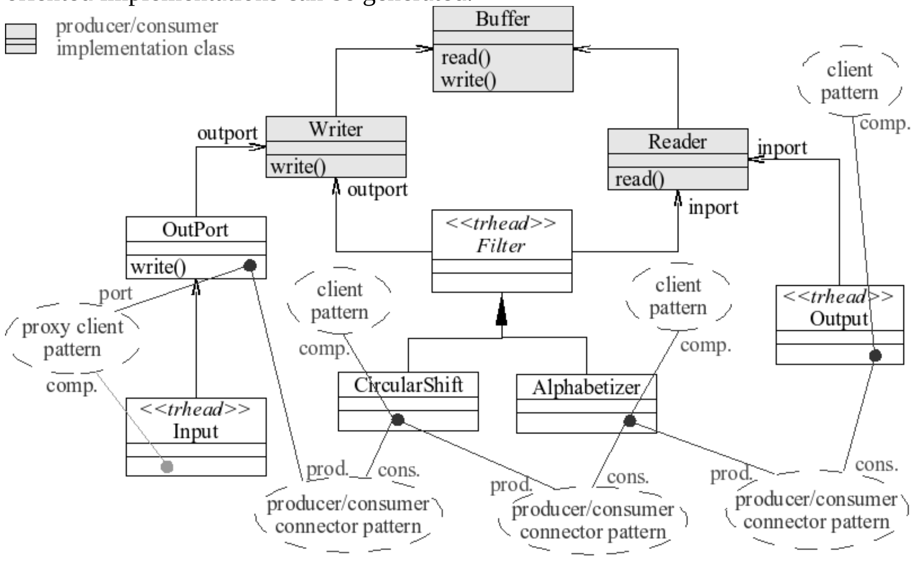

Fig. 10. A pipe and filter materialization with active filters.

The ArchJava approach extends the Java language by incorporating to it architectural features [1]. Given an ArchJava program specification, a special compiler maps the architectural features defined in the program into standard Java language, generating a running system and ensuring that the resulting implementation conforms the prescriptions given by the architectural features specified.

6 Conclusions and Future Work

We have described a pattern-based approach based on a body of design knowledge and a set of predetermined activities that developers need to perform in order to derive materializations of software architectures. We aim to enhance system quality by preserving the attributes prescribed by the architecture in the final system implementation. The approach represents a step towards a systematic and comprehensive design process, which involves and links both architecture and object-oriented design worlds.

Furthermore, architectural conceptual integrity can be preserved in the final object-oriented design by restricting developers in the use of predetermined materialization patterns, defined in the catalog, for generating the implementations of the elements defined by architectures [3] [4].

We are developing a rule-based Java prototype that allows experts to organize a catalog of materialization patterns based on specific application needs. The tool aids developers in the materialization of architectures activities. Thus, given a system architecture definition and based on the quality attributes prescribed, the tool suggest developers the adequate materialization patterns and aids in the instantiation of those patterns for generating the concrete object-oriented materialization of the architecture.

A line of future work is the organization of a catalog of semantic features characterizing the static and dynamics structures of components. These features involve the characterization of filters, knowledge sources and other predefined component types [13]. Also, object-oriented materializations of architectural mechanisms such us priority managers or caches can also be explored.

Finally, the high level architecture of an implemented system can be inferred by matching the object-oriented implementation of the system with the materialization patterns defined in the catalog of patterns. Hence, the approach enables the reconstruction of system architectures from object-oriented implementation designs by means of system reverse engineering capabilities.

References

1. Aldrich, J., Chambers, C., and Notkin, D.: ArchJava: Connecting Software Architecture to Implementation. In proceedings ICSE 2002 Pp. 187-197. May 2002.
2. Allen, R., Garlan, D.: Formalizing architectural connection. In Proceedings of the 16th International Conference on Software Engineering, pages 71-80, Sorrento, Italy, May, 1994.
3. Bass, L., Clements, P., and Kazman, R.: Software Architecture in Practice. 2nd Edition. Addison-Wesley. 2003.
4. Brooks, F.: The Mythical Man-Month: Essays on Software Engineering. Addison-Wesley, 1975.
5. Campo, M., Díaz Pace, A., and Zito, M.: Developing Object-oriented Enterprise Quality Frameworks using Proto-frameworks. In. Software: Practice and Experience,Vol 32 No 8, Pp. 837 – 843. Wiley. 2002.
6. Díaz Pace, A., Campo, M.: ArchMatE: From Architectural Styles to Object-oriented Models through Exploratory Tool Support. Proceedings OOPSLA 2005, Pp. 177-132. International Conference on Object-oriented Programming, Systems, Languages and Applications. October 16-20th, San Diego, CA, USA. 2005USA, October 2005.
7. Gamma, E., Helm, R., Johnson, R., and Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. Massachussetts, 1995.
8. Hakala, M., Hautamäki, J., Koskimies, K., Paakki, J., Viljamaa, A., and Viljamaa, J.: Generating application development environments for Java frameworks. In proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE'01), Erfurt, Germany, September 2001, Springer, LNCS2186, 163-176.
9. Johnson, R.: Frameworks = (Components + Patterns). Communications of the ACM, pages 39 - 42, October 1997.
10. Medvidovic, N., Mehta, N., and Mikic-Rakic, M.: A Family of Software Architecture Implementation Frameworks. In proceedings 3rd IFIP WICSA, 2002.

11. Mehta, N., Medvidovic N., and Phadke, S.: Towards a taxonomy of software connectors. Proceedings of the 22nd international conference on Software engineering. Pp. 178 - 187. 2000.
12. Object Management Group, Unified Modeling Language Specification. Version 2.0 WWW site available at http://www.uml.org/.
13. Shaw, M., and Garlan, D.: Software Architecture, Perspectives on an Emerging Discipline. Prentice-Hall. 1996.
14. Spitznagel, B., Garlan, D.: A Compositional Approach for Constructing Connectors. Submitted to The Working IEEE/IFIP Conference on Software Architecture, The Netherlands, August 28-31, 2001.