Analyzing Problem Frames together with Solution Patterns

Ellen Souza, Maria Lencastre, Renata Melo, Lilian Ramires, and Keldjan Alves
Departamento de Computação, Universidade de Pernambuco
Rua Benfica, 455 Madalena, 50750-410, Recife – PE, Brazil
{eprs, maria, rpllfm, lor, kao}@dsc.upe.br

Abstract

The Problem Frames approach defines identifiable problem classes based on, among other things, their context and the characteristics of their domains, interfaces and requirements, without going deeply into the solution. Other software engineering approaches deal with the concept of patterns that present well-known solutions, such as archetype, analysis and design patterns. We can say, for instance, that patterns are about solutions and problem frames are about problems. This paper attempts to make an analysis of the integration of problem classes, that is problem frames, and solutions, by analyzing a set of different kinds of patterns together within problem frames. The relationship, between these approaches, seems to have a good chance of improving software development.

1. Introduction

The Problem Frame approach [7] gathers system requirements focusing on the problem, that is, it describes the operational context in which the system has to be developed. This approach has been regarded a good way to investigate identifiable problem classes based on, among other things, their context and the characteristics of their domains, interfaces and requirements, without going deeply into the solution.

On the other hand, patterns [5] describe solutions, at different levels of abstraction, for problems based on one's experience. A pattern is a way of describing best practices, good designs, and of capturing experience in such a way that it is possible for others to reuse the solution.

During the software development process, the use of problem frames can help in the identification of well-known classes of problems and their main characteristics. In addition, the use of patterns can improve software quality, as the proposed solution has already been tested and proved. These approaches can also decrease the product's time-to-market, by reusing problem and its solutions, without the need to think about how the problem can be completely solved.

The connection between PF and patterns is a good way to help refine the phases of problem solving by starting with an instantiation of problem domains considering appropriate problem class, that come from well structured archetype and analysis patterns, going further to available problem solutions at and design levels.

It is important to notice that we use Problem Frames (PF), in upper case, to refer to the approach and problem frames, and in lower case, to refer to the basic problem classes.

This paper is organized in the following way: Section 2 presents the background to clarify the main concepts involved in this work; Section 3 explains the association proposed between solution patterns and PF; Section 4 describes the case study, taken from [4], and shows the associations identified; Section 5, presents some related work; finally, Section 6 draws some conclusions and points the way to further studies. An Appendix containing the pattern models used in this paper is provided on the last page.

2. Background

In software engineering, a pattern is a general repeatable solution to a commonly occurring problem in software design, analysis or any other software development phase [2]. Its main purpose, within the software community, is to create a body of literature to help software developers and analysts to solve recurring problems encountered throughout the software life cycle. Patterns at business and analysis levels can be employed at early development phases, while design patterns are related to the design phase, giving support to implementation.

The objective of this section is to clarify a number of concepts that will be used later in this paper, i.e.,

problem frames, archetype, analysis and design patterns. We also try to identify the underlying objective of each of these concepts.

2.1. Problem Frames

The main concept of the PF approach is the problem frame concept, since it represents a kind of pattern that captures and defines a commonly found class of simple subproblems [7]. In Figure 1, an example of a problem frame is presented. It involves a machine to be built in order to meet a requirement, and a set of domains and their interactions.

In Problem Frames notation, see Figure 1, the domains are represented by rectangles; the one with double stripes on the left is the machine. The other domains are identified by a letter in their lower right corner, describing the kind of each domain: "C" is causal, which means its properties are predictable. "B" is biddable, that is, its main characteristic is the lack of predictable behavior, and usually consists of people; and "X" is lexical, which means is a physical representation of data. The dashed oval is the requirement. The lines connecting the domains are the interfaces of shared phenomena. The CM!C1 notation signifies that the phenomenon C1 is controlled by the CM - Control Machine domain.

Jackson [3] supplies a repertoire of recognized problem classes - *problem frames* - with associated characteristics, difficulties and solution methods. It includes the following problem frames: Required Behavior, Commanded Behavior, Information Display, Workpieces and Transformation.

As shown in Figure 1, each problem frame has a concern that must be addressed. The concern identifies the descriptions one must fit together properly in a correctness argument: requirement, specification and domain description. Each description, explained in notes format (from 1 to 5), has a defined order in the central frame concern [9]; for example in Figure 1 the first step of the frame concern is "When the operation issues that command…". In conjunction with the characteristics of problem domains, the frame concern gives rise to the particular concerns that distinguish the problem classes. If one tries to fit a problem into an inappropriate class, the resulting development will probably be unsuccessful.

Examples of PF concepts and diagrams can be seen in Section 4, where this approach is used to model the case study.

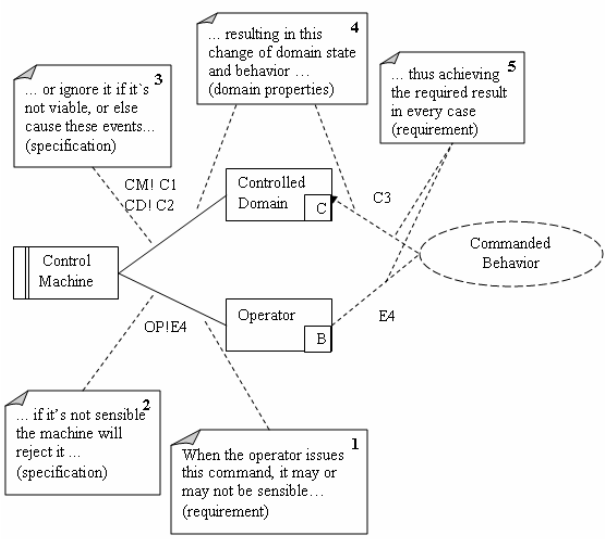

Figure 1. The Commanded Behavior problem frame concerns.

2.2. Archetype Patterns

An archetype is a primordial thing or circumstance that recurs consistently and is thought to be a universal concept or situation [3]. Because archetypes are a basic human mechanism for organizing, summarizing, and generalizing information about the world, they can reasonably have applications in the field of software development. Some examples of archetype patterns are: Party, Product, Inventory, Order, Money, etc.

The *Party* archetype pattern shown in Figure 2 describes how to represent essential information about people and organizations. As a general rule, *Parties* have no interesting behavior - they simply hold information [3]. This archetype itself is a very abstract thing, with only the most rudimentary semantics and, even when variations occur, the principal concept still holds. It only unifies the way to represent people.

Other archetype patterns models used in this paper are shown in the Appendix.

2.3. Analysis Patterns

The term analysis pattern represents patterns which capture conceptual models in an application domain in order to allow reuse across systems [2]. Analysis patterns focus on organizational, social and economical aspects of a system, since these aspects are central for the requirements analysis and the acceptance and usability of the final system.

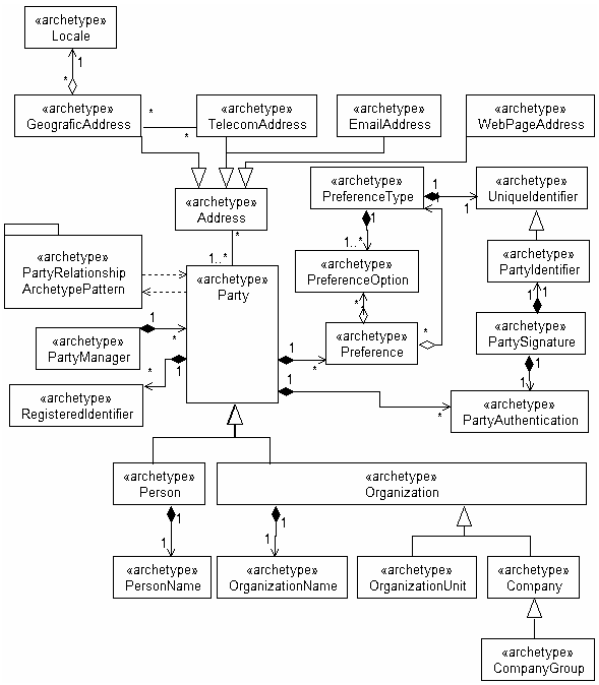

Figure 2. *Party* archetype pattern

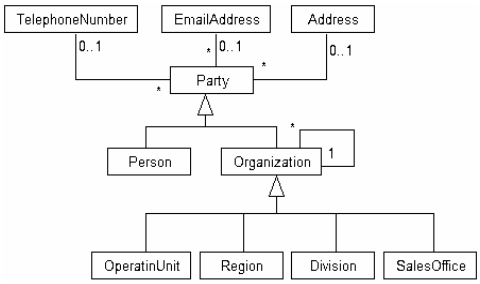

Figure 3. *Party* analysis pattern

As archetypes, analysis patterns speed up the development of abstract analysis models that capture the main requirements of the concrete problem. Figure 3 shows the analysis pattern representing *Parties*. As an archetype, it is also called *Party* and holds information about people and organizations such as address, telephone number and so on.

Notice that archetype patterns have many unique features that make them more understandable than analysis patterns because they support, among other things, Unified Modeling Language (UML) profiles, variability of model elements, pleomorphism [3].

2.4. Design Patterns

A design pattern is a description or template for solving a problem that can be used in many different situations. It is therefore not a finished design that can be transformed directly into code.

Reusing design patterns helps to prevent subtle issues that can cause major problems and to improve code readability for developers and architects familiar with the patterns. Effective software design requires a consideration issues that may not become visible until later in the implementation.

Figure 4 shows the *Composite* design pattern. It composes objects into tree structures to represent part-whole hierarchies. This pattern lets clients treat individual objects and compositions of objects uniformly [1]. The *Composite* design pattern can be applied, for example, to represent simple and compound organizations in the *Party* archetype pattern.

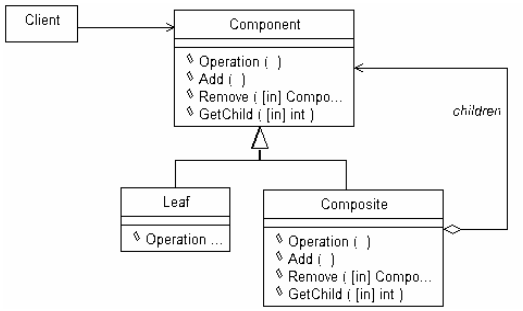

Figure 4. *Composite* design pattern

The design part of an analysis or archetype pattern contains a description of a possible realization with one or several design patterns.

3. Associating Problem Frames with Solution Patterns

In order to help identifying and understanding the correlation between the 4 approaches (problem frames, archetype, analyze and design patterns), their main objectives and concepts are summarize in Table 1.

We can observe that the involved approaches deal with two different levels: the problem space and solution space. On the problem space, we have basically the Problem Frame approach, where problem frames describe classes of problems identifying, requirements, domains, phenomena, and concerns. On the solution space, the focus is on the different abstraction levels of patterns. Patterns supply a full problem description, together with known uses, motivation and, of course, the solution to the problem in different levels of abstraction.

Table 1. Concepts and Objectives

Approach	Main Objective	Concepts Applied
Problem frames	Define known classes of problems for reuse	Domain types, interaction phenomena types, requirement and concerns
Archetype patterns	Define universal concepts, organizing general information about the world.	Problem, solution, consequence, variations
Analysis patterns	Define conceptual models, that deal with business aspects	Intent, motivation, forces, solution, consequence, design, known uses.
Design patterns	Define solution to recurring problems	Problem description, solution, consequences, related patterns, known uses,

So, in the pattern space, the problem and its context is still present, but is not so emphasized as the solution. Also the involved forces, in the solution patterns, have focus on requirements, relevant to the problem being solved.

We can observe that, the solution space leads with the reuse of classes and their interaction, while problem frames leads with the reuse of problem knowledge, including requirements, domains, and interaction phenomena

3.1 Understanding Problem Space and Solution Space

As we try to make use of different levels of abstraction, from problem space to solution space, we establish a first proposal of a process to integrate them, see Figure 5. On the left-hand side of the figure we have the problem space, and on the right-hand side the solution space.

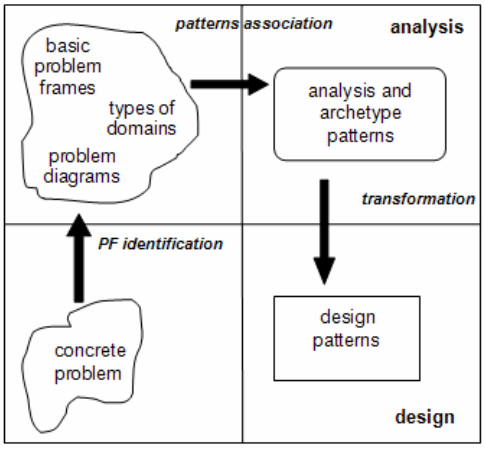

Figure 5. From problem to solution space

The process starts with a problem, which is further detailed using the PF approach. Then it goes on to the patterns, starting with the more abstract levels, presenting the correspondence of the problem domains with archetype and analysis patterns. Finally, we end with the less abstract descriptions of design patterns, that is, design problems, and how a general arrangement of elements solves them.

Archetype and analysis patterns, can improve the existing details, once a person wants to instantiate a problem frame, since they give support to universal concepts, and business domains in a generic sense. Also proposed problem frames can match many parts of behavior present in patterns at solution space.

3.2 Steps to handle solution patterns in the context of PF

The main steps needed to handle solution patterns in the context of PF are described below:

1. The first step consists in understanding the concrete problem to be solved by discovering the system main requirements.
2. After, we concentrate on drawing the PF Context diagram, in order to determine where the problem is located, and what parts of the world it concerns (see Figure 7).
3. From this point, we are already able to identify solution patterns at the analysis level that matches the existing domains. This is interesting because this association can provide more details on the problem domains.
4. From the PF Context diagram, problem diagrams are derived, and the involved requirements are more detailed. Depending on the granularity of these diagrams, basic/ available problem frames are identified and instantiated and, again, solution patterns may be employed to complete the problem domain characteristics.
5. Continuing through the solution space, after connections at the analysis level, we are able to transform those identified patterns to object-oriented design patterns, which provide a structured view of the whole system and

helps, particularly, in the definition and validation of architecture.

In this way some existing concepts, such as the universal concept form archetype patterns, can be used to improve the description of a problem frame domain, which means the description of a domain pertaining to a problem class.

In section 4 we provide associations between PF domains and patterns and between the entire problem frame and patterns.

4. Case Study

This section uses the POS case study taken from [4] to illustrate the association between PF and patterns, according to the proposed sequence of levels explained in the previous section. It also delimits the context diagram, decomposes the problem into subproblems and identifies problem frames, which match some of the identified subproblems. Patterns are subsequently associated with the domain types, problem diagrams and problem frames.

4.1. Concrete Problem

A POS system is a computerized application used in part to record sales and handle payments; it is typically used in a retail store. It includes hardware components such as a computer and bar code scanner, and software to run the system. It has interfaces to various service applications, such as a third-party tax calculator and inventory control.

The system has to support multiple and varied client-side terminals and interfaces. These include a thin-client Web browser terminal, a regular personal computer with something like a Java Swing graphical user interface, touch screen input, wireless PDAs (Personal Digital Assistant), and so forth. The POS system is a well-known case study and it is fully documented in [4].

In Figure 6, an example of the levels of problem and solution spaces is shown for the POS system case study, explained above in Section 3.

Notice that, in Figure 6, the concrete problem is represented by the POS system description, together with its requirements. The PF identification step consists in delimiting the problem boundaries, defining the subproblem partitioning and identifying matching problem frames. The subproblems used as examples in this paper are the *processSale*, *makePayment* and *printBalance*. These last two problems represent instances of *Commanded Behavior* and *Transformation* problem frames, respectively.

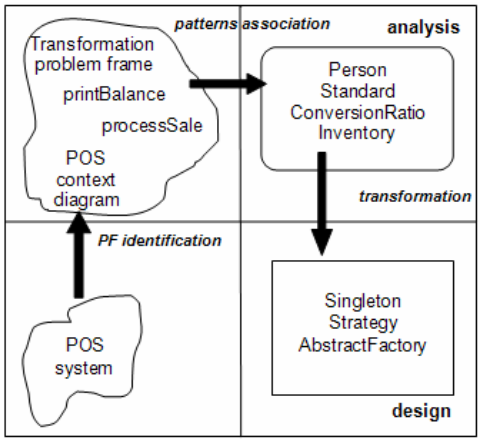

Figure 6. Example of the levels of problem and solution spaces

Patterns associated at analysis level are, for example, *Party*, *StandardConvesrion, Order* and so on; At design level, *AbtractFactory, Iterator, Composite, Strategy*, among others.

4.2. PF Identification

The concrete problem described in the previous subsection will now be analyzed and PF diagrams presented in order to understand and delimit the problem to be solved.

4.2.1. Defining the problem boundaries

The context diagram structures and delimits the problem by identifying its domains (Database, Third-part Services, POS Client, and User), together with the machine to be built (POS Server Machine). It also shows how these domains interact with each other and with the machine through the interface of shared phenomena [10], such as the ones presented in Figure 7.

For simplification, the POS Client *domain* encloses the three different interface types: Computer, PDA and Web Browser. Also, the third-party services *domain* represents for example tax calculator and inventory control.

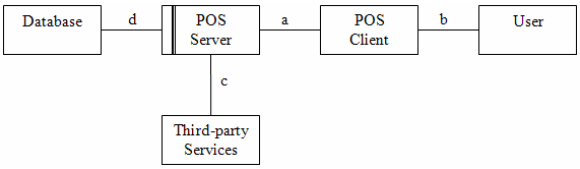

Figure 7. POS system context diagram

a: *recordSales* and *handlePayments*;
b: *purchaseItems* and *makePayments*.
c: *taxCalculator* and *inventoryControl*;
d: *returnProduct* and *returnClient information*;

4.2.2. Decomposing the problem into subproblems

In [9], the authors identify many problems and requirements. Here we illustrate the Process sale, whose main flow is as follows:

1. *Customer arrives at POS checkout with goods and/or services to purchase.*
2. *Cashier starts a new sale.*
3. *Cashier enters item identifier.*
4. *System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.*
   Cashier repeats steps 3-4 until indicates done.
5. *System presents total with taxes calculated.*
6. *Cashier tells Customer the total, and asks for payment.*
7. *Customer pays and System handles payment.*
8. *System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).*
9. *System presents receipt. Customer leaves with receipt and goods (if any).*

Figure 8 presents the problem diagram for the Process sale, as explained previously. It shows that the Customer interacts with the POS Client informing the purchased products and payment type. The POS Client collects the prices, calculates taxes, and updates the inventory and account information. Interface phenomena were not explicit in the diagram, for simplification purposes.

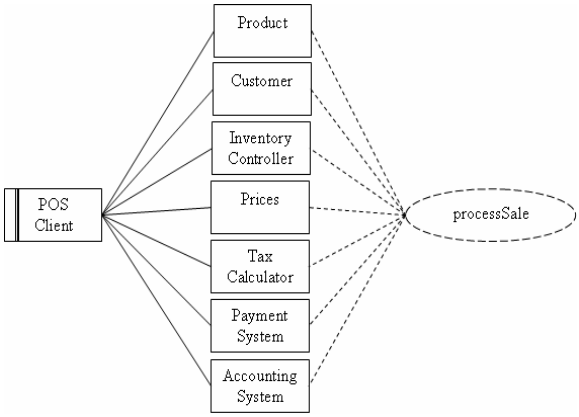

Figure 8. Process sale problem diagram

Analyzing the *processSale* problem diagram and the involved requirements, following subproblems are identified: (1) *makeNewSale* – starts a new sale; (2) *enterItem* – repeats while there are items being sold; (3) *endSale* – calculates taxes, total and payment required and print balance; and (4) *makePayment* – receives payment. In this paper we will focus only on two of them: *makePayment*, which falls into the *Commanded Behavior*, and *printBalance*, a subproblem of *endSale*, which falls into the *Transformation* problem frame.

4.2.3. Problem frame identification

The identification of problems that match existing problem frames is an important step, since it facilitates the understanding and identification of previous identified classes of problems, with associated characteristics, difficulties and solution methods.

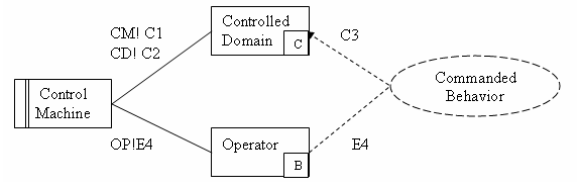

Figure 9. The Commanded Behavior problem frame

Figure 9 shows the *Commanded Behavior* problem frame. It represents the idea that there is some part of the physical world whose behavior is to be controlled so that it satisfies certain conditions. Thus, the problem is to build a machine that will impose such a control.

Figure 10 presents a problem diagram for the *makePayment* requirement explained in steps 6 and 7 from the Process sale main flow. This is an example of the *Commanded Behavior* problem frame. The Customer purchases an item and informs its identification and the payment type. The POS Client machine is responsible for effecting the payment and issuing the paper receipt. The Customer is a biddable domain while the Payment system and POS Client machine are causal.

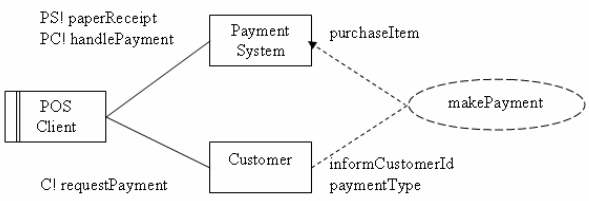

Figure 10. makePayment diagram.

Figure 11 shows the *Transformation* problem frame diagram. Like *Commanded Behavior*, it has a Machine domain, other domains, a requirement and shared phenomena. The Input is a given domain, which is informed by the user, while the Output is to be processed by the Machine. Both domains are lexical.

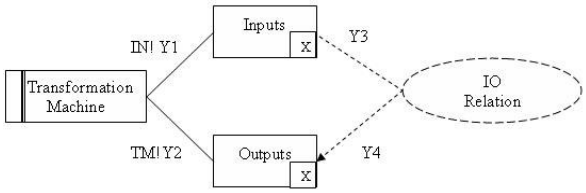

Figure 11. The Transformation problem frame

The *Transformation* problem frame can be used to print the sale balance, as shown in Figure 12. In the POS example, the Printer Machine domain is responsible for the transformation of the sale's line data into a printed receipt containing a description of the items and total.

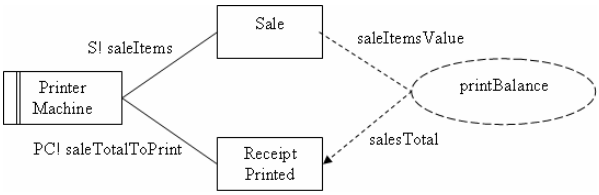

Figure 12. printBalance diagram

Next, object-oriented solutions, based on patterns, are proposed for the identified problem diagrams presented in this section. The *Customer/User* domains from Figures 7, 8 and 10 are associated with the *Party* archetype patterns explained in Section 2.

We also present patterns which deal with conversions that are specializations of the *Transformation* problem frame.

For the problem diagram shown in Figure 8, several patterns can be associated. We chose the *Product, Inventory* and *Order* archetype patterns as they represent important causal domains of the POS system and they also have a closed relationship.

4.3. Pattern association at Analysis level

The archetypes were chosen for the associations rather than analysis patterns because they produce a more detailed model and can be easily adjusted to the system requirements. Therefore, in this paper, we also consider the analysis patterns as they represent another important level of abstraction.

4.3.1. *Party* archetype pattern

Almost every business is concerned to some degree with maintaining information about parties and the roles these parties play in the various relationships between them [3].

An example of the Party archetype pattern can represent a Biddable domain at a business level. This relationship was very intuitive, even obvious. The Biddable domain can be an Operator, a User, a Customer, and the connection still holds true. They all have unpredictable behavior even when having a procedure to follow. As a general rule, Parties have no interesting behavior - they simply hold information. Figure 9 shows an example of the *Operator* domain in the *Commanded Behavior* problem frame and Figure 2 presents the *Party* archetype model composition.

The corresponding pattern in the analysis phase is also called the *Party* analysis pattern. It is more simplified than the archetype, as shown in Figure 4.

The main advantage of the use of the *Party* archetype pattern is that it has a unified way of representing information about parties and eliminates redundancy in systems, poor data quality, lost business opportunities and other problems [3].

4.3.2. *StandardConversion* and *UnitConverter* archetype pattern

Conversion operations are available in many different types of software. The *StandardConversion* archetype defines a *conversionFactor* that can be used to convert a *Quantity* from a source *Unit* (*Input domain*) into a *Quantity* from a target *Unit* (*Output domain*).

Also, the *UnitConverter* archetype is responsible for converting a *Quantity* from a source *Unit* (*Input domain*) into a Quantity from a target *Unit* (*Output domain*) [3]. Both patterns are shown in Figure 13.

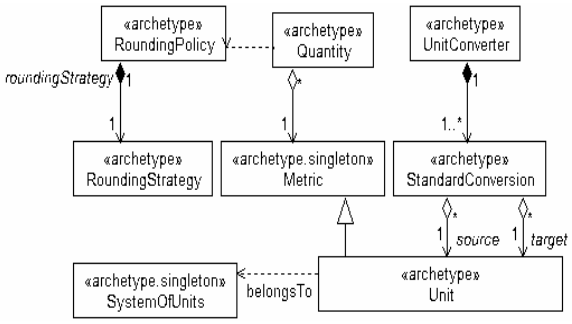

Figure 13. *StandardConversion* and *UnitConverter* archetype patterns

Conversions themselves are transformations and their behavior can be represented by the *Transformation* problem frames, whose machine domain type is *causal*. They receive computer-readable input files whose data have to be transformed to produce certain required output files. Figure 11 shows the *Transformation* problem frame. Both *Input* and *Output* domains are lexical, which means they have a physical representation of data.

At level of analysis, quantity conversion can be represented by an example of the *ConversionRatio* analysis pattern that formally converts quantities from one unit to another, as shown in Figure 14.

4.3.3. *Product* archetype patterns

The *Product* archetype pattern provides a general abstraction for representing information about a company's goods and services from a selling perspective. It contains a complete model that will lead to flexible business systems that are easy to adapt to new business opportunities.

The pattern model shows how to represent types of products; the Figure 15 in the appendix section shows the *Product* archetype patterns and the involved subpatterns.

In Figure 8 the *Product* and *Price* domains were identified within the problem description. The first one of these contains information about the products that can be sold in the POS system and the second contains all the pricing rules for the products.

For these two domains, the *Product* archetype patterns can be used to identify the information necessary for the solution to and understanding of the problem and also to give a more precise specification of the problem.

The generalized *Product* archetype pattern provides a flexible model, and is what we expect to be used in most cases. Whith simple modifications, the *Product* archetype pattern represents the POS product types very well.

4.3.4. *Order* archetype patterns

When a customer decides to purchase a product, we need to have some way of recording exactly what is required. This is known as an order. It is a request made by a customer to deliver some goods or services. In return the seller normally receives some payment or other compensation.

Figure 16 shows the *Order* archetype patterns together with parties, products and services provided. The *Order* archetype provides a complete model for the action executed by the POS Client Machine for the *processSale* requirement, as shown in Figure 8. It includes information about taxes, discounts, payment strategies, customers, products and so on.

4.3.5. *Inventory* archetype patterns

An inventory is a store of goods, but it can also be used to manage the delivery of services.

Even having the *Inventory* as a third-party system, as shown in Figures 7 and 8, the pattern model helps one to understand and identify the information needed for storage and it also helps to define a common interface between the POS and third-party systems .

Figure 17, in the Appendix, presents the *Inventory* pattern and its relationships with other patterns.

4.4. Transformation to patterns at design level

Considering the transformations presented in [8], the following examples of object-oriented design patterns can be used for the *Party* archetype and analysis patterns, enriching the whole transition process:

- For the creation of *Person* and *Organization* classes the *Abstract Factory* design pattern is suitable as it provides a common interface.
- The *Composite* design pattern can be applied to represent simple and complex organizations. It is used to represent part-whole hierarchies of objects where clients treat individual objects and compositions of objects uniformly [1]. It is presented in Figure 4.
- If *Persons* and *Organizations* share the same storage data structure, the *Iterator* design pattern provides a uniform interface for traversing different aggregate structures [1].

A design pattern solution for conversions from one unit to another combining archetype and analysis is shown in Figure 13 and explained bellow:

- *SystemOfUnit* represents a set of units and is unique for all conversion operations. The same applies to *Metrics*. Thus, for both classes the *Singleton* design pattern is appropriate. It ensures a class has only one instance, and provides a global point of access to it.
- Rounding operations have different strategies according to specific quantities. The *Strategy* design pattern lets the algorithm vary in respective of the clients that use it [2].

4.5. Discussion

In PF there are several problem frames which use biddable domains, so we believe that archetype patterns are good candidates to specialize the biddable domains in problem frame context. So, they can be applied to give support to problem frames instantiation, and domain descriptions. The person archetype pattern and analysis pattern are examples of this.

Table 2. Instantiation of several patterns

PF concepts	Archetype pattern	Analysis pattern	Design pattern
Biddable Domain	*Person*	*Person*	*Abstract Factory, Composite, Iterator*
Lexical Domain	*Quantities* in different units	*Quantities* in different units	
Causal Domain	*Unit Converter Order, Product Inventory*	*Conversion Ratio*	
Basic Problem Frame	*Standard Convertion*	*Conversion Ratio*	*Singleton, Strategy*

More over, we can also see a similar situation in case of lexical and causal domains. In the Transformation problem frame, proposed by Jackson, the Transformation Machine is a causal domain and both *Input* and *Output* domains are lexical. Associations with Lexical and causal domains have also been proposed lie for example the quantities, in case of lexical domains, and Unit converter in case of causal domains. Table 2 presents the different instances of solution patterns.

A major advantage of all the patterns presented in this paper is the level of abstraction provided by each approach, in this case Archetype and Analysis, and also the reuse supported by the object-oriented approach. However the use of patterns can affect the flexibility and reusability of the resulting system. Notice that building too much flexibility into a system can also make it too complex. Engineering demands a trade-off between the cost of building and maintaining artifacts and the feature it will provide [2].

In conclusion we can state that, for the illustrated example, the integration between patterns and PF permits a deeper analysis of the problem and a more complete solution.

5. Related Work

After considerable research, we found no papers relating PF to archetype, analysis or even design patterns. However, we did find some papers proposing approaches or other patterns for problem frame-oriented software development. In [12] the authors propose software architectural patterns corresponding to the basic problem frames that may serve as a starting point for the construction of the software solving the given problem. Also in [13][14][15] an *ad hoc* UML-based development method for some of the most relevant problem frames is provided showing how problem frames may be used upstream of a development method. In [10] the crosscutting nature of some properties of a problem is explored to analyze the impact on the modularization of concepts and, therefore, the evolution of the system.

In [8] a study on the use of patterns within the RUP software development process is presented. Through the study of design patterns, analysis patterns and archetype patterns the author makes a transition between models. A model that describes a problem solution using archetype patterns, for example, is described using analysis patterns and design patterns.

6. Conclusion and Future Work

Our analysis shows that studies involving an integrated approach, using PF and other patterns, are promising. Their conceptual resemblances allow associations at different levels, as explained in Section 3. Some of the associations were proposed for problem frames; the association, at this level of abstraction, that is not considering an instantiation, is very helpful for binding the specification and design phases.

In this paper, we presented an overview of the transitions, which starts with PF diagrams, passing through archetype, analysis and, finally design patterns. However, we did not going deeply into any of them. We consider the whole process a big step, which must be further explored. In future studies, we will consider: (1) making other case studies, in order to refine and validate the proposed integrated process; (2) exploring the transition of basic problem frames deeply; (3) analyzing other PF concepts which may be associated with patterns, such as the phenomena concept, in order to formalize the study; (4) defining specific archetype patterns that represent in a complete and suitable fashion the existing PF domain types.

As the idea of software patterns is not restricted to the object-oriented community, we will also consider including other kinds of patterns, maintaining a flexibility of approach, to the development process.

7. References

[1] Gamma, Erich, et. al. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[2] Fowler, Martin. Analysis Patterns – Reusable Object Models. Addison-Wesley, 1997.

[3] Arlow, Jim; Neustad, Ila. Enterprise Patterns and MDA – Building Better Software with Archetype Patterns and UML. Addison-Wesley, 2003.

[4] Larman, Craig. Applying UML and Patterns – An Introduction to Object-Oriented Analysis and Design and Iterative Development. Prentice Hall PTR, 2005.

[6] Geyer-Schulz, Andreas; Hahsler, Michael – Software Engineering with Analysis Patterns. Wien, 1991.

[7] Jackson, Michael; Problem Frames: Analyzing and structuring software development problems. Addison-Wesley, 2001.

[8] Moraes, Tiago; Aplicação de padrões ao processo de desenvolvimento de software RUP. UPE, 2006.

[9] M. Lencastre, J Botelho, P. Clericuzzi, J. Araújo, A Meta-model for the Problem Frames Approach, Workshop in Software Model Engineering, at Models, Jamaica, 2005.

[10] M. Lencastre, J. Araujo, A. Moreira, J. Castro, Analyzing Crosscutting in Problem Frames Approach. Proceedings of 2nd International Workshop on Advances and Applications of Problem Frames - IWAAPF´06, Shanghai, China, 2006, ACM Press, pp 59-64.

[11] Jackson, M. A.: Problem Analysis Using Small Problem Frames, Proceedings of the South African Computer Journal 22; Special Issue on WOFACS'98 (1999) pp 47-60.

[12] C. Choppy, D. Hatebur and M. Heisel: Architectural patterns for problem frames; IEE Proc.-Softw., 2005, Vol. 152, No. 4.

[13] Choppy, C., Reggio, G.: A UML-Based method for the commanded behavior frame. In: IWAAPF'04, Karl Cox and Jon G. Hall and Lucia Rapanotti Eds., 2004, IEE pp. 27-34.

[14] Choppy, C., Reggio, G.: A UML-Based approach for Problem Frame oriented software development. In: Elsevier eds, Information and software technology 47, 2005, pp 929-954.

[15] Lavazza, L., Del Bianco V.: Combining Problem Frames and UML in the description of software requirements. Fundamental Approaches to Software Engineering (FASE06), Vienna, Austria, 2006.

[16] J.G. Hall and L. Rapanotti, Towards a Semantics of Problem Frames, Technical Report 2003/05, Department of Computing, The Open University, Milton Keynes UK.

Appendix

The patterns presented in this paper are detailed in this appendix, with the aim of clarifying and facilitate their understanding. They include the *ConversionRatio* and *Measurement* analysis patterns and the *Product*, *Order* and *Inventory* archetype patterns.

The *Measurement* analysis pattern considers various things that can be measured as objects and introduces the object type *Phenomenon Type* as shown in Figure 14. The *Phenomenon Types* are things that can be measured [2].

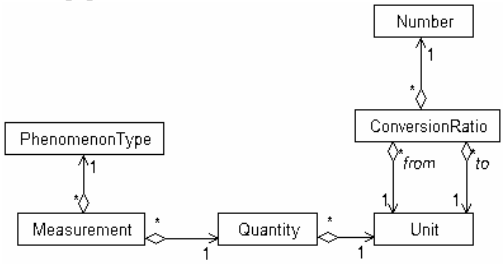

Figure 14. *ConversionRatio* and Measurement analysis patterns

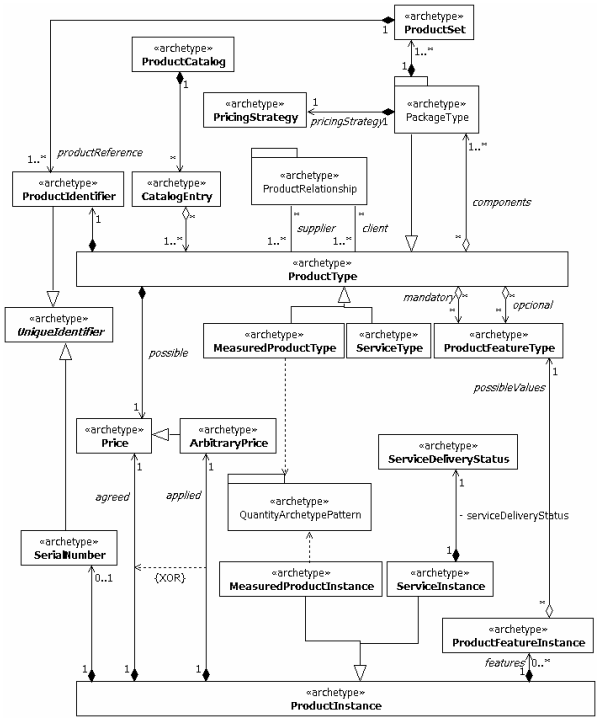

Figure 15. Product archetype patterns

The *Product* pattern model shows how to represent types of products, product specification, persistent storage of product information, amount of money that must be paid in order to purchase good or services, products sold by measure and so on. Figure 15 shows

the *Product* archetype patterns and the sub patterns involved.

Figure 16 shows the *Order* archetype patterns together with parties, products and services provided.

The pattern is made up of other important patterns such as *Party* and *Product* required for implementing an Order.

Figure 17 shows the *Inventory* archetype patterns, which require a prior understanding of the *Product* archetype. Inventory always tends to be handled in a similar way - a stock of items is created and maintained at an appropriate level or, in the case of services, the capacity to deliver the service is maintained [3].

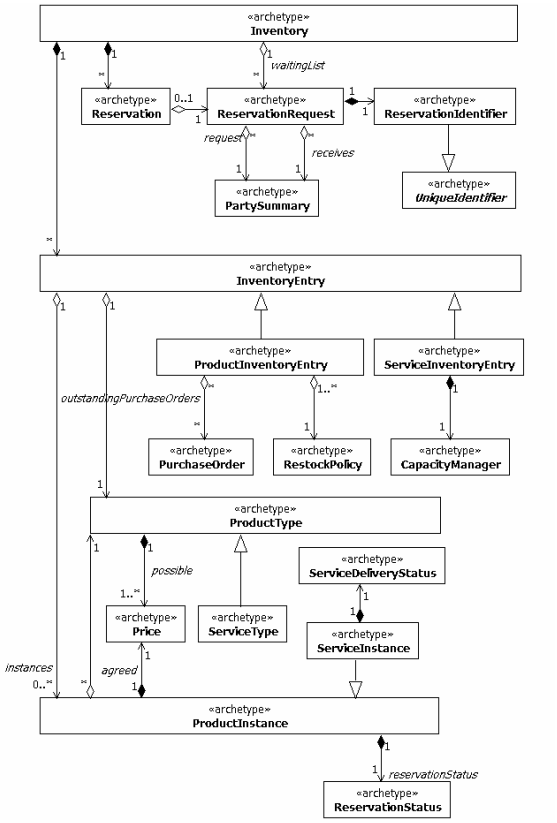

Figure 16. *Order* archetype patterns

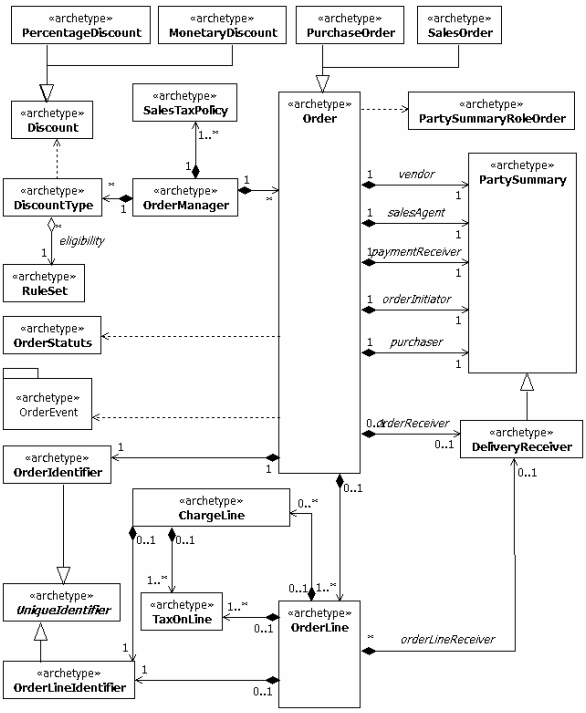

Figure 17. *Inventory* archetype patterns