Domain Specific Patterns: Conversions, Persons and Roles, and Documents and Roles.

Ari Schoenfeld

UACP y P del CCH

ari@servidor.unam.mx

Universidad Nacional Autónoma de México

México D.F.

Abstract

This paper exposes a proposal of three domain specific patterns. These patterns deal with abstractions from the real world that are likely to appear mainly in business and administration software systems, so that solutions to some typical problems in this kind of systems can be found. The first one is the Conversions pattern, useful for handling uniformly anything that can be converted from one scale to another. The second one, Persons and Roles, is presented as a basis for the Documents and Roles pattern, which is the third one. The Documents and Roles pattern gives a general basis for any administrative system.

Introduction

This paper presents three domain specific patterns. These patterns are intended to solve some typical problems in business and administration software systems. The first one is the Conversions pattern, useful for handling uniformly anything that can be converted from one scale to another. The second one, Persons and Roles, is presented as a basis for the Documents and Roles pattern, which is the third one. The Documents and Roles pattern gives a general basis for any administrative system.

Design patterns can deal with generic design issues applicable to almost any kind of software system (Examples of these patterns can be found in [[I]]). They usually don't deal with the mapping from the real world to a model, i.e. they use generic classes that are not from a specific domain. They mainly help us to find the best solution to a kind of design problem that could be found in any kind of system.

Domain Specific Design Patterns are mostly useful in specific domain or "kind" of systems. Even though these patterns are not useful for every software system (they are very specific), they can help to find abstractions and kinds of relationships between classes in a given domain that otherwise would take a long time and effort to find (Examples of these patterns can be found in [[II]]).

A typical problem that comes out in the first stages of design is to know whether the classes we found and their hierarchy are right, that is, if they are going to work in the future, and if they are going to be resilient to change. The patterns we present have proven to solve the problems they are intended for, and also are quite stable and change-resistant. They were used in two very different systems: one is a textiles factory administration system, and the second one is an insurance management system for insurance agents. Three patterns are presented. The first one is the Conversions pattern that helps to handle uniformly anything that can be converted from one scale to another, or from one rate to another. The second one, Persons and Roles is a solution to a problem that arises in every software system: An object can play many roles in a system and this could lead to a change in type, something that should be avoided if possible. Using a “role” class, this instance can play many roles without changing it’s type. Even though this pattern is not really new, is presented as a basis for the Documents and Roles pattern, which is the third and most important one. The Documents and Roles pattern helps to handle documents and persons in a very flexible way, useful in many software systems.

There has been a great deal of development and research in patterns. Given the enormous amount of patterns that have been identified already, there is a growing need for classifying them so this “written” experience can be located and reused. Patterns can be classified in many different ways. They can be classified as idioms, mechanisms, or frameworks[[III]]. They can also be classified depending on the phase of the software process they belong to: Analysis, Design, or Implementation. Whilst there is no consensus as for how to classify patterns, we can say that the patterns presented here, belong to the Design phase, and that they are mechanisms.

These patterns are presented following the description format of Gamma et. al.[[IV]].

I.- Conversions Pattern

Intent:

Handle and save (for precision) quantities in their original format, but use them easily in any other representation or scale in a transparent way.

Motivation:

Where conversions are needed, this pattern is very helpful. Conversions can be made between currencies, or units of measure for example. The Conversions pattern describes what classes are needed in order to save data as it was originally generated so precision is saved but easily accessed in any other scale or converted with a different rate.

Information is stored in a system so it can be retrieved and presented easily and effortlessly in different ways. With this pattern, information can be presented anytime in different scales, with no danger of making conversion mistakes, since all the conversion mechanics are encapsulated (“hidden”) in specialized objects.

Applicability:

Use this pattern when there is a need to express data differently from how it was generated, maybe to be compared, related or added (for example) to other data. This, of course, can only happen if the data can be expressed in different units or “systems of measure” and can be translated from one to another.

For example, a company that has branches in different countries, would need to see it's financial results expressed in dollars to compare them, even if they were originally stored in pesos, yens, pounds, etc.

Also, the products the same company buys or sells could be originally registered in feet. Later on the *Company* may need to consolidate the quantities in meters or inches or whatever. With this pattern, you can avoid conversion mistakes, because the classes are responsible for making all the necessary calculations to give the correct result.

Structure:

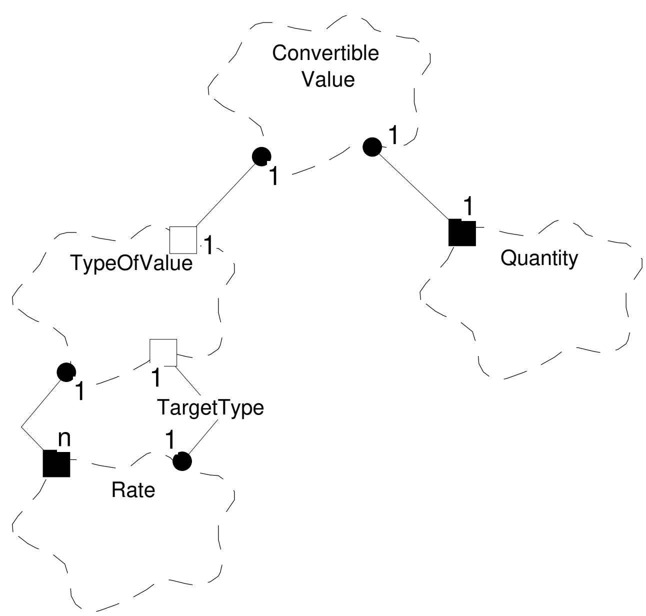

Fig. 1. - Structure of the Conversions Pattern.

Participants:

- *ConvertibleValue*: It is the class that has the value that might be converted. This is the class actually used in the system, and where the original value is stored (and will be saved just that way for later conversion, if needed).
- *Quantity*: Saves a numeric value, usually a real or alike.
- *TypeOfValue*: Saves the **type** of the value stored in *Quantity*. This class is the one responsible for calculating conversions between different *TypeOfValue*-s. The information needed to make all these conversions is stored in the *Rate* class.
- *Rate*: saves conversion information. One *TypeOfValue* can have conversion rates for all other *TypeOfValue* instances, or just for some of them.

Collaborations:

The *aConvertibleValue* object can be asked to transform it's value to a different *TypeOfValue* (targetTypeOfValue), so it sends a message to *aTypeOfValue* to make a conversion which can be done thanks to the corresponding *aRate* object (Fig. 2).

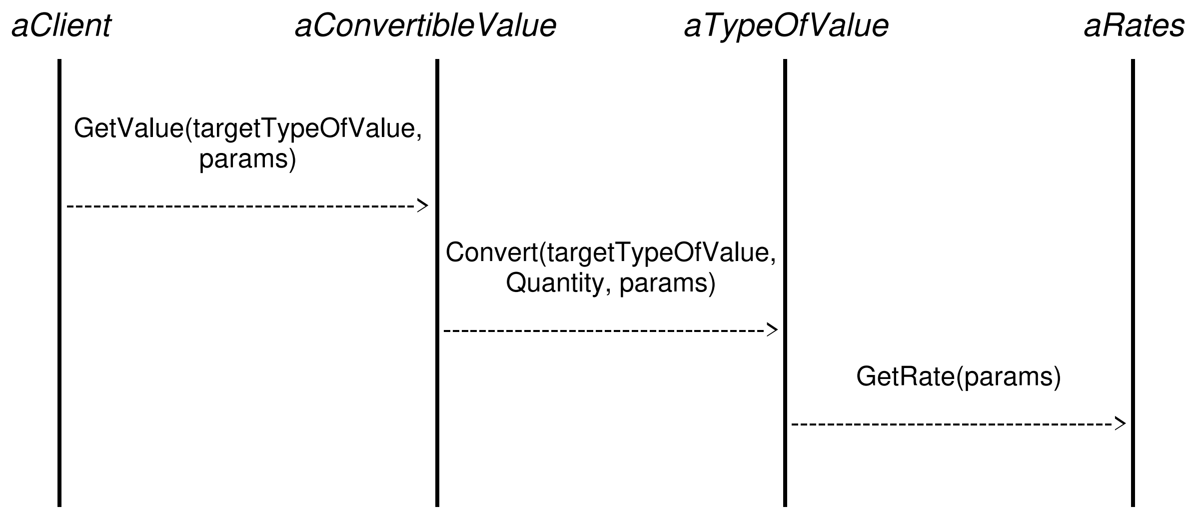

Fig. 2. - Interaction Diagram example of Conversions Pattern.

The *params* parameter can be used depending on the type of conversion. For example for currency conversions, it is necessary to define a date for the conversion. In other conversions, this could be simply not used.

ConvertibleValue is responsible for expressing it's quantity in any scale (if allowed). To make the conversion it relies on TypeOfValue which is the only class responsible for making the conversions. To make the conversion, it uses the *Rate* class which has the data needed to calculate the desired conversion. insured

Consequences:

A data can be stored as it was originally generated, what preserves accuracy. Since the conversions are handled inside ConvertibleValue, data can be accessed as if it was of the same type, in the same scale so every query or report can be viewed in the desired scale (or according to the previous example, in the desired currency).

If in the future there is a new scale included in the system, all what is needed is to specify the rates for conversions, and "old" data can be expressed in this new scale (continuing with the previous example, this is specially useful in countries like Mexico where there are changes in the currency to "eat" zeroes (5000 pesos changed to be 5 "new" pesos). In this case there is no need to change even a line of code: simply use the "new-pesos" currency, and all the queries can be expressed in "old" or "new" pesos).

Implementation:

Quantity can be any numeric type.

As it was stated before, the parameters of the *GetValue* function may vary depending on the type of conversions.

Not all the possible rates for conversions might be captured in the system, so maybe some conversions will be impossible to calculate. In these cases it is necessary to define how to handle this situation (raising an exception, giving a default value, etc.). In this case the “Meaningless Behavior” pattern from the CHECKS pattern language[[V]] is very appropriate.

There can also be illegal conversions (for example, feet can’t be translated to degrees, or ounces to meters) and appropriate actions should be taken. This is the responsibility of *TypeOfValue*.

Most conversions are symmetric and this has to be assured (i.e. a value converted to a different scale and back, must be the same).

Known uses:

The most common and typical uses of this pattern are for Currencies (fig. 3), and Units of Measure (fig. 4). For example, the *Money* class has a *Quantity* and a *Currency*. It must be very accurate in any calculation and conversion, but what’s more important is that it is capable of expressing it’s quantity in any currency at any given date (if the exchange rates are available). The conversions are actually calculated by the *Currency* class at a given date. *Currency* is responsible for saving and “knowing how” to convert between currencies at any given date.

We can send a message to *Money* like GetValue(currency2, date), that will give us the value in currency2 at a given date of the amount (quantity) this instance owns.

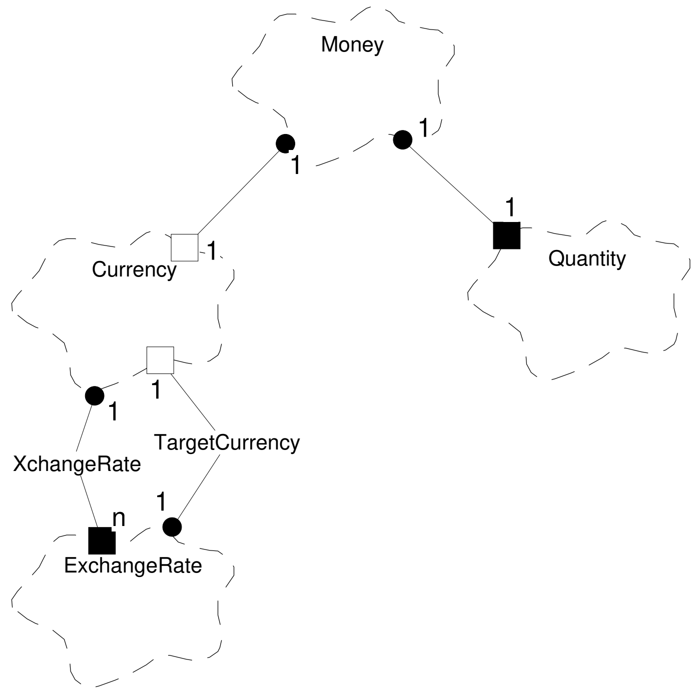

Fig. 3 -Conversions Pattern instantiated for Money.

This same pattern instantiated for Units of Measure could be applied in a stock tracking system, or any system dealing with products or merchandise. As we can see in Fig. 4,

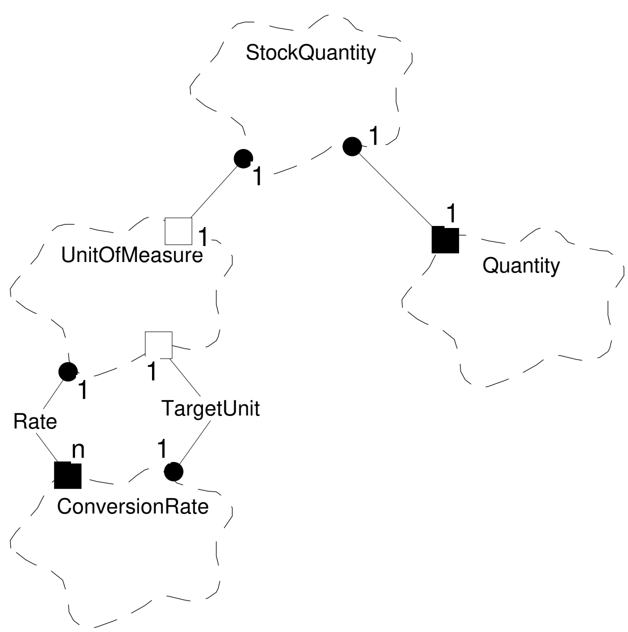

Fig. 4 - Conversions Pattern instantiated for StockQuantity.

we could track a given textile (for example) so we have a StockQuantity instance that tells us that we have 1000 ft. We could send a message to this object GetValue('meters') and it would use UnitOfMeasure to calculate the corresponding meters. In this manner we can calculate the total stock in meters, feet or inches, as desired, even if part of the received merchandise was in feet, part in meters, and part in inches.

2.- Persons and Roles

Intent:

Persons in a system participate under different roles. Persons can freely change their roles at any time and this must not mean that a person object changes it's type. Also, a person can have many roles simultaneously and increase or decrease them as needed.

Motivation:

Whenever a software system deals with persons (could be users, employees, clients, students, companies, etc.) there is a very adaptive pattern that could be used in almost any of them, and solves a problem that has been discussed a lot. A *Person* can get a job, so he is an *Employee*, in the mean time he starts a degree in the university, so he is a *Student*, then since he is always sleeping at work they fire him so he is *Unemployed*, and so on. The problem here is that if we define a hierarchy where for example, *Employee* inherits from *Person*, just as *Student* or any other role (well yes, an *Employee is_a Person* and also a *Student* is_a *Person* so it's understandable why it's so common to model it like this), how can we express in our system that a *Person* is an *Employee*, and also a *Student*? The problem is that inheritance is a way of **classifying** that implies a partition, i.e. an instance of one type can't be at the same time one of the other types. Even a change of class (something every programmer want's to avoid) does not help because somebody *is_a Student* at the same time he *is_a Employee*.

What we see is that persons play many different **roles**, and most of the times, at the very same time. We need another class (*Role*) which will help us to model this. Being a *Student*, an *Employee*, a *User*, a *Company*, are only *Roles* that the same *Person* could play at the same time. A *Person* has to be aware of all the roles it plays, and also a *Role* knows which *Person* it is about.

Applicability:

This pattern can be applied to any system that deals with persons and roles persons play. In almost any software system for an organization, persons and their roles are modeled in a way or another.

Structure:

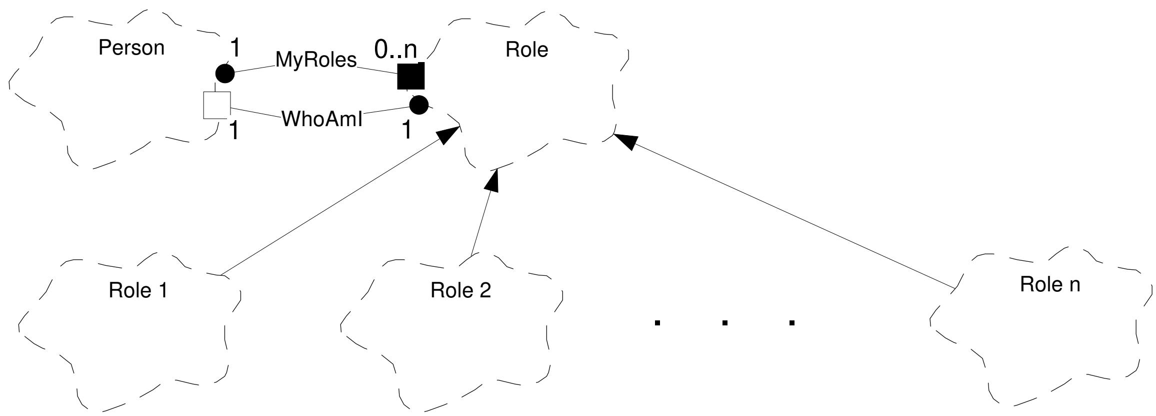

Fig.5. -Structure of the Roles Pattern.

Participants:

- *Person* is the basis of all the possible roles in the system. A *Person* has all the data that does not change from role to role.
- *Role* is an abstract class from which any possible role in the system inherits. The specific roles have all the attributes related to that role.

Collaborations:

A *Person* has a list of all it's *Role*-s. On the other hand a *Role* has a reference to the *Person* it is about. A *Role* could see the attributes that the *Person* owns as his, for example a role *Employee* could use the birth date (a *Person* attribute) to calculate the age for calculations or validations.

Persons are normally used all around in any administration system. They have a list of attributes like name, date of birth, addresses, phones etc. Note that a Corporate Entity could also be modeled as a *Person* (with the same attributes). With this pattern, *Role*-s are the ones to be used all around, and the *Person* will be accessed through the *Role*.

A *Role* actually is like an envelope for the class *Person* plus, it adds its own behavior. If the attribute needed is specific to the *Role*, the *Role* answers directly. If not, it asks *Person* for that attribute:

For example:

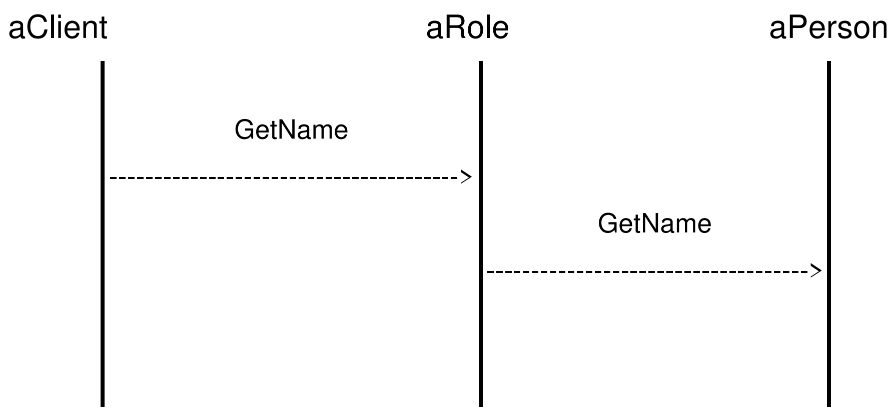

Fig. 6 - Interaction Diagram example of Roles Pattern.

When a *Role* is created, one of the parameters of the constructor is the *Person* this *Role* is for. Inside the constructor, the attachment is made, and that is the only place is made. No changes are allowed to the reference of a *Role* to a *Person*. When the attachment is made, all the mutual references are updated.

Consequences:

This pattern avoids the need of changing the type of an instance whenever is a change in the role a Person plays. We only need to create or destroy a Role object associated to that person.

All it's data like name, birth date, addresses, phones, etc. is accessible from all it's roles and could be seen as part of the roles themselves. Any *Role* could be created and destroyed as needed. A *Person* could even have two or more *Role*-s of the same kind at the same time (like two jobs, something not so strange nowadays).

Implementation:

There are some validations that should be made to avoid contradictory *Role*-s of a *Person*, like *Employee* and *Unemployed*, or *Mother* when de attribute sex of the *Person* is male. The *Role*-s are responsible for making these validations whenever they are associated with the corresponding person. Also there are *Role*-s only for individual persons, only for Corporate Entities, or for both.

In the system, almost all the references should be to *Role*, and not to *Person*. The *Person* can be accessed via the *Role*.

Known Uses:

In the following diagram we can see some examples of *Role*-s.

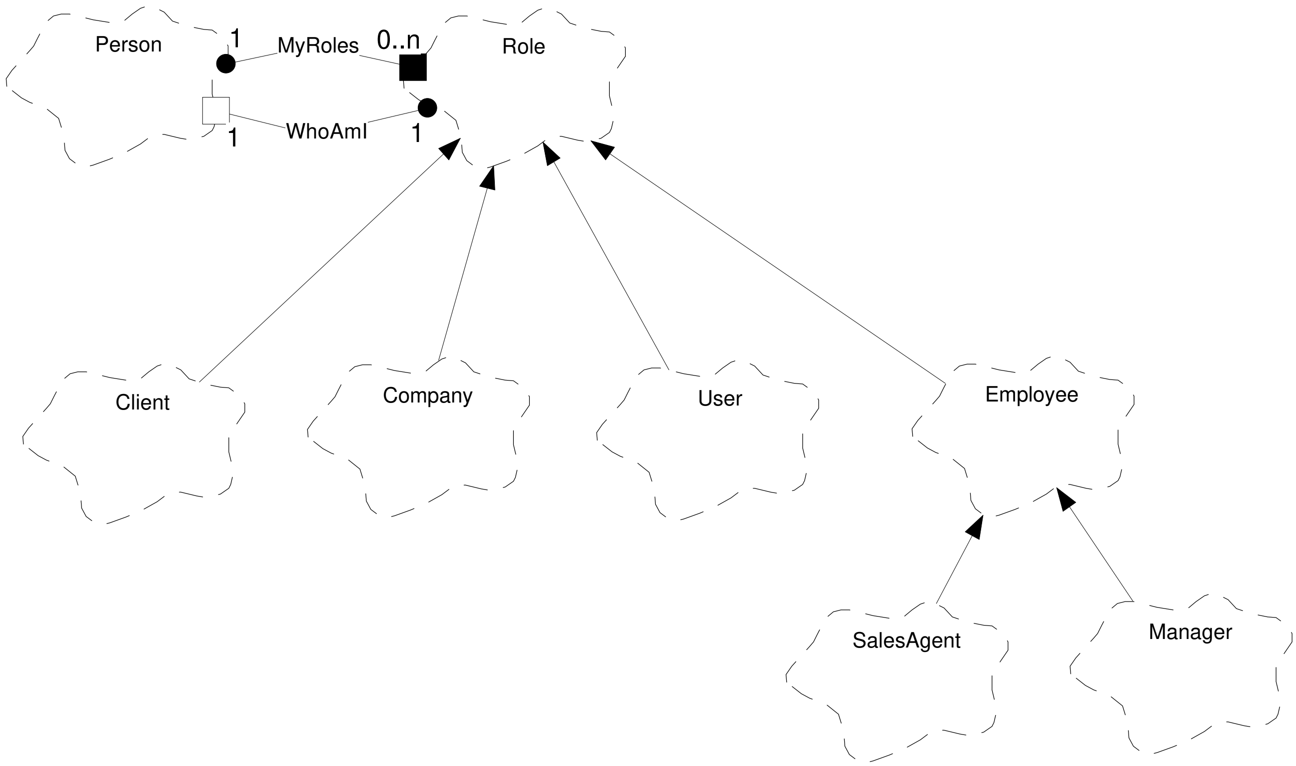

Fig. 7. - Example of the Persons and Roles Pattern.

This pattern was used in two systems: one for a textiles factory and one for an insurance management system for insurance agents. In the first one, the company decided to sell at special prices some of the products it manufactured to their employees. Since all the employees were already in the system, all what was needed was to create the *Client* roles for them and capture a few attributes for this particular *Role* (it is a big factory, so this was very neat).

In the second one, a *Person* in the system can become *Insured* and if he/she pays the insurance, it will also be a *Client*, and this can be applied anytime to anyone in the system, something very practical when you talk about many persons that could acquire new roles very often (Insured, Client, Beneficiary, Guarantor, etc. for example).

Related Patterns:

Coplien's **Envelope-Letter** idiom[[VI]] is a technique for changing an object's class at run-time. This is not exactly what we are doing here since we don't only want to change the class of an object, we also want to approach it in different ways (even many of the same kind) at the same time.

The **State** pattern in Gamma's book also makes it possible for an object to appear to change it's class, but here too, the object changes it's class but only one at a time.

Note that this pattern could actually be generalized so any entity could have many roles. For example, an *Invoice* plays many roles in a company. Amongst others roles, it's the exit permit for the merchandise to leave the warehouse; It's the document that states the amount the client has to pay.

The OOram Method has a deep coverage of roles of entities but with a different approach[VII].

The Perspective Application method can be used to find roles of persons[VIII].

3.- Documents and Roles

Intent:

Express any relationship of documents and roles of persons in a system.

Motivation:

In every organization, most of the work that has to be tracked, managed, followed up, saved for future queries, etc., deals with documents, and people. Actually, documents are always related to people, and people to documents. By expressing this relationships it is possible to navigate through them and easily find anything related to a *Document* or a *Person*.

This pattern makes the expression and representation of all the possible relationships between documents and roles very flexible. It also makes it very easy to change whenever needed. When this associations are "hard-coded", they are very rigid and difficult to change (increment or decrement relationships). With this pattern any *Document* can be associated to any *Role.* The only changes needed, are in the validation of the allowable relationships.

This pattern enables us to attach any number of documents to a document, any number of *Role*-s to a *Role* (of *Person-s*), and any number of documents to a *Role* of a *Person* (and vice versa).

Applicability:

Use this pattern in any administrative system when you need flexibility and resilience to change in the relationships of *Document-Document*, *Role-Role* (of *Person*-s), and *Document-Role*s.

Structure:

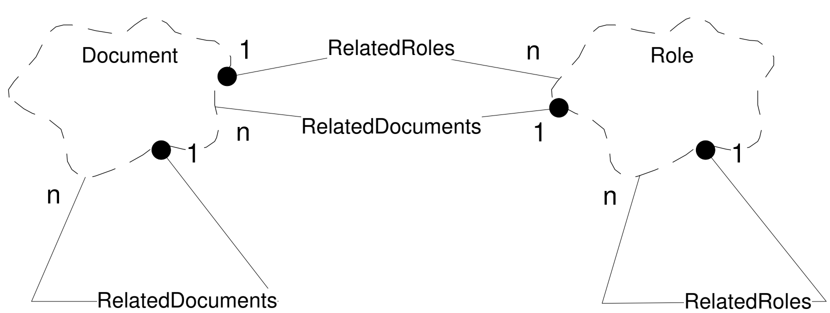

Fig. 8. - Roles and Documents Pattern.

Participants:

- *Documents* are very common in administrative systems. Every document in the system should inherit from *Document*.

This class (and it's descendants) map every real-life document to the system. Every administration system deals with lot's of classes of documents.

- *Role* is a role of a *Person* and is explained in the Persons and Roles Pattern.

Collaborations:

The structure is what is important in this pattern. There are six basic methods for both *Document* and *Role*: AddDocument, AddRole, GetDocument, GetRole, DeleteDocument and DeleteRole,. These methods perform the basic operations on the lists of *Document* and *Role*.

The collaboration between classes depends on the actual classes and the specific scenarios they are involved in. For what this pattern is concerned, there will be multiple calls to the previous methods. Every *Document* and *Role* is responsible of checking if the requested additions or deletions are valid.

With this structure, it is possible to navigate through the documents that are related and through the people involved. For example, if there is a problem with a given *Client*, we can look at all the documents related to him, and let's say we find the one with the problem. From that document we can find who contacted the client, what letters were sent, who transported the goods, who received them in the client's side, and everything to find out what caused the problem and what *Person* we have to talk to. The collaborations here are between all this related *Documents* and *Roles* through their links.

Consequences:

This pattern allows much flexibility, adaptability and resilience to change of any relationship of *Document-Document*, *Role*-*Role* (of *Person*-s), and *Document*-*Role*s. Most of the time changes in an organization relate to changes in these relationships. In this case, this pattern will save us from making deep changes in the classes, we will only need to allow, disallow or require the addition of another kind of relationship.

Implementation:

It's important to validate the allowed relationships.

Whenever a *Document* is inserted in the *Role* RelatedDocuments list, this *Role* should appear automatically in the RelatedRoles list of the *Document* (and vice versa)

Note that the *Role*-s themselves describe the type of relationship. For example, a *Client* role related to an *Invoice* tells us that he is the one the *Invoice* is made for. A *Client* related to a check, tells us who made the check. The same *Person* with the *Supplier* role related to a check, tells us to whom this check was made for. So, as we can see, the meaning of the relationship between a *Role* and a *Document* is in the type of role. There is no need to specify somewhere else the type of the defined relationships.

Known Uses:

For example, a *Letter is_a Document*, that has a *Sender*, and a *Addressee*, which are *Person*-s (*Role*s of *Person*). On the other hand, these two people have this *Document* related to them (for the first one as a *Letter* that he sent and the second one as a *Letter* he received). Any

Letter related to a *Sender* role is a *Letter* this *Person* wrote. Just as any *Letter* related to an *Addressee* role is a *Letter* this *Person* received.

The same happens for example with an *Invoice*. The people (or *Role*s) involved are: The *Company* that issues the *Invoice, the Person* or *Company* the *Invoice* is made for (*Client*), the *Person-s* (*Employees*) who made the sale (*Agents*, *Customer_Support*, etc.), the *Person* that has to be called to know when the payment will be ready, and any other *Person* that could become related, can be easily added with this pattern. Notice that the *Role* itself describes the relationship with the *Document* (i.e. the *Role Agent* related to an invoice tells us who was the agent that made that sale). Also, a document has a list of it's related documents. From the last example, the *Invoice* could have the *Check* that paid it, the *Shipment* or *Shipping-Document* it was sent with, the client-satisfaction *Questionnaire*, or if it was canceled, the *Credit-Note* that replaced it.

Conclusion

Although Object Oriented Programming solves many problems of Procedural Programming, there is still a complexity in software systems that shouldn't be overlooked. Even small Object Oriented Systems have more objects than can be handled by a (normal) human mind. The use of patterns can so greatly reduce the complexity of the design and implementation, that they should be used whenever possible. Moreover, they can help us (and so they did in the two systems mentioned above) to achieve a great deal of reuse.

Notation

Fig. 9 presents an example of the Booch notation:

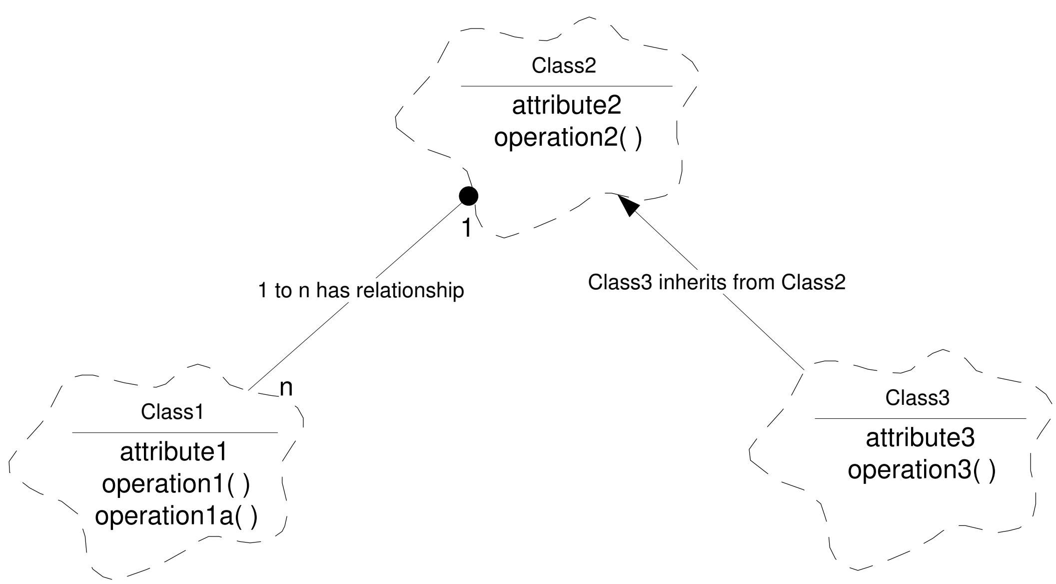

Fig. 9. - Booch notation.

[I] Gamma Erich, ... “Design Patterns - Elements of Reusable Object Oriented Software”, Addison Wesley, 1995.

[II] P Coad, D. North & M. Mayfield, “Object Models Strategies, Patterns and Applications”, Yourdon Press, Prentice Hall, 1995.

[III] Booch, G. Object Solutions “Managing the Object-Oriented Project”, Addison-Wesley, Redwood City, CA, 1996.

[IV] Gamma Erich, ... “Design Patterns - Elements of Reusable Object Oriented Software”, Addison Wesley, 1995, pp 6-7.

[V] J.O. Coplien, D. Schmidt, “Pattern Languages of Program Design”, Addison Wesley, 1995.

[VI] J.O. Coplien “Generative Pattern Languages: An emerging direction of software design”, C++ Report, Jul-Aug 1994.

[VII] Reenskaug, T., P. Wold and O.A. Lehne “Working with Objects; The OORam Software Engineering Method”, Prentice Hall, 1995

[VIII] S.Y. Liao, "Perspective Application: An Efficient Tool to Identify Roles and Subclasses in Object-Oriented Modeling", ROAD Volume 2, No. 5, January-February 1996