A Pattern Language for Business Resource Management[*]

Rosana T. Vaccare Braga[1]
ICMC-Universidade de São Paulo
C. P. 668, 13560-970 - São Carlos – SP - Brazil
Phone Number: 55 16 273-9696 / 272-9238
e-mail: rtvb@icmc.sc.usp.br

Fernão S. R. Germano
ICMC-Universidade de São Paulo
E-mail: fernao@icmc.sc.usp.br

Paulo Cesar Masiero[2]
ICMC-Universidade de São Paulo
E-mail: masiero@icmc.sc.usp.br

ABSTRACT

A pattern language to deal with Business Resource Management is presented. It covers a great number of applications in business systems, including patterns for resource rental, trading and maintenance, and was designed based on practical experience in information systems development. Existing recurring patterns were applied to form patterns in our language, which were instantiated to this specific domain. The idea has been to make the language as complete as possible in order to be useful for the analysis of a wide variety of applications in this domain. Fifteen patterns are presented together with a diagram showing the precedence for checking the convenience of their utilization. Object models using UML notation are used both to present each pattern structure and a sample instantiation. The pattern language application to practical cases has shown that analysis is easily conducted, as it is a guideline for the work to be done.

INTRODUCTION

Several generic patterns, useful for information systems development, have been proposed recently, as for example Type-Object [Joh 98], Association-Object [Boy 98], Specific-Item-Transaction, Transaction-Transaction Line Item and Item-Specific Item [Coa 97]. These recurring patterns can be used in a wide range of information systems applications across different domains.

We present a pattern language to deal with resource management applications, a particular domain in information systems. This pattern language results from an evolution of more than ten years of systems development practice for medium and small business in this domain. The similarities among these systems made it worth to think about how to

[*] Copyright © 1999, Rosana T. V. Braga, Fernão R. S. Germano and Paulo C. Masiero,. Permission is granted to copy for the PLoP 1999 Conference. All other rights reserved.
[1] Financial support from FAPESP grant 98/13588-4
[2] Financial support from CNPq and FAPESP

establish a pattern language that could be used by analysts when developing such systems. The pattern language here presented, called Business Resource Management Pattern Language, is composed of several patterns, some of which are specific applications of the recurring patterns mentioned above. In fact, the pattern language stays on a higher abstraction level than those recurring patterns. It is applicable to a more specific domain and it contains the semantic value inherent to a family of applications in the domain. It provides inexperienced developers with substantial material to develop new systems, as it guides them during development, providing alternative solutions as necessary and giving orientation of the next issues to be explored.

The Business Resource Management Pattern Language was designed to help software engineers to develop applications that deal with business resource management, more specifically, applications in which transactions such as resource rental, trade or maintenance need to be logged. By transaction we mean the same as Coad [Coa 97]: a significant event to be remembered, i.e., an event that the system must remember through time. Resource rental focus primarily on the satisfaction of a certain temporary need of a good, like the use of a physician time for an attendance or the use of a videotape for watching a motion picture. Resource trade focuses on the transference of property of a good, as for example a product sale. Resource maintenance focus on the maintenance of a certain good, using labor and products to perform it, as in an electric appliance repair shop.

THE BUSINESS RESOURCE MANAGEMENT PATTERN LANGUAGE

Fifteen patterns form the presented language. Their use is elective in many cases, according to the characteristics of the application. In Figure 1 the precedence for checking the convenience of these patterns utilization is shown. Directions for this verification are supplied for each pattern but, as a whole, the precedence is observable satisfactorily in Figure 1. The main patterns in the language are RentTheResource (7), TradeTheResource (8) and MaintainTheResource (9), indicated by a thicker line. Their use is not mutually exclusive and, in fact, there are applications in which they can be fitted together. MaintainTheResource may use RentTheResource and TradeTheResource, as in a car repair shop system case, in which parts are traded and labor is rented.

The first pattern to be applied is IdentifyTheResource. Patterns 11, 12 and 13 are shown within a box denoting that they are applicable to all situations where there is an arrow ending at its border. The arrow without source ending at pattern 11 means that it is the first pattern to be verified, followed by patterns 12 and 13. The “Next patterns” section of each pattern complements and details navigation information related to Figure 1. The patterns are grouped in three sections according to their purpose, as illustrated in Figure 1. The notation used to express the patterns is UML (Unified Modeling Language) [Eri 98]. Basic class methods to create an object, to set and get attribute values, to add and remove object connections and to delete objects are not shown in the sketches, as they add more complexity to these diagrams with little gain in model effectiveness. Instead, we assume that these methods are always present in each class. To enhance overall system behavior understanding we assign system operations to class methods when necessary, choosing among existing classes those that best fit the operation functionality. Actually, system operations are more than methods, as they represent events in the real world that serve as input to the system and whose functionality is implemented by calling several methods of possibly several different classes.

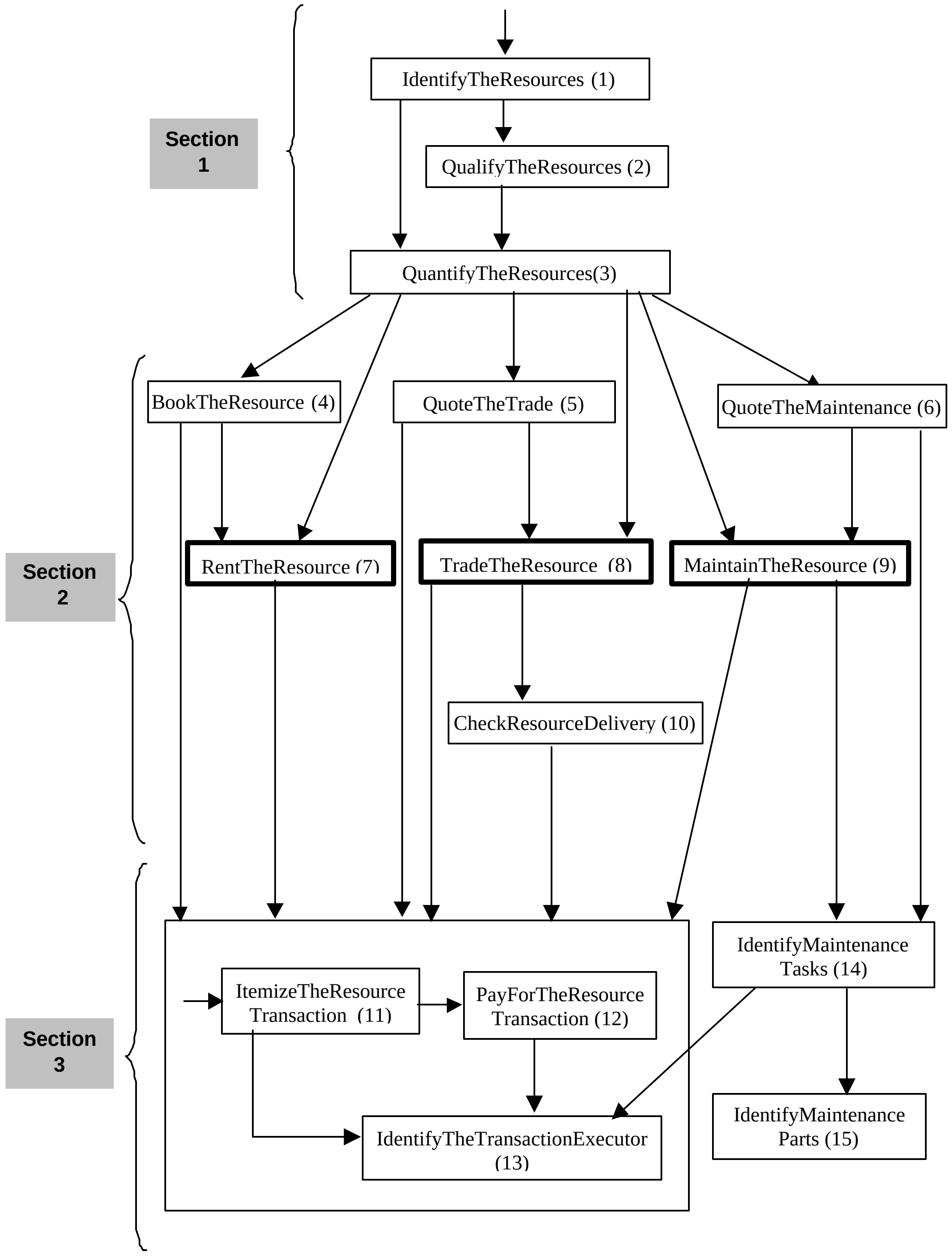

Figure 1 – Dependencies between patterns

Section 1: Initially, focus on the resources involved in your application. First you have to **IdentifyTheResources (1).** Next check the convenience to **QualifyTheResources (2).** Also, verify how the resources are quantified in the application and **QuantifyTheResources (3).**

Pattern 1 – IdentifyTheResources

Context:

Your business system deals with one or more of the following transactions: orders, sales, purchases, locations, rentals, assignments, reservations, repairs or maintenances. These transactions refer to, for example, products in a store, videotapes in a rental shop, books in a library, physician time in an attendance office, or cars in a mechanical repair shop, which are resources managed by specific systems.

Problem:

How do you represent the business resources involved in the transactions processed by the system ?

Forces:

- Business resources usually have common attributes or qualities. Keeping information about each particular resource is important for the organization that manages these resources.
- If the resource has just a few attributes they may be placed on the classes representing the transactions made by the organization. This eases the implementation, although it limits software evolution.

Therefore:

Evaluate all the subject of transactions, identifying those that can be thought of as resources being managed.

Solution:

For each resource to be managed create a “Resource” class, defining all its important attributes. An attribute “IdCode” is often provided to uniquely identify each resource and an attribute “Description” is used to describe the major resource features. Other attributes can be added according to the particular resource instance. Besides the basic class methods, not shown here as explained before, special operations are provided to list resources by IdCode and by Description, which are reports frequently required by the business manager.

Sketch:

Figure 2 shows a sketch for the **IdentifyTheResources** pattern. The “Resource” class has all the resource’s common attributes and methods.

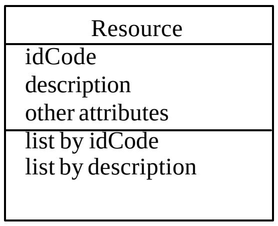

Figure 2 – IdentifyTheResource Pattern

Example:

Figure 3 shows an instantiation of the IdentifyTheResource pattern, in which "product" plays the "Resource" role in a Retail Store System.

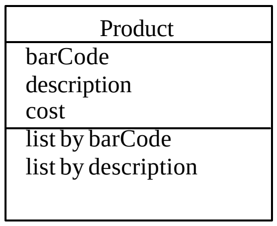

Figure 3 – Instantiation of the "IdentifyTheResource" pattern

Next Patterns:

After you IdentifyTheResource, try now to QualifyTheResource (2).

Pattern 2 – QualifyTheResource

Context:

You have identified all the resources that your application deals with and their main attributes. Some of these attributes may apply to many resource instances; for example, the same manufacturer is assigned to many parts in a retail store or thousands of vehicles have the same model.

Problem:

How do you identify resource qualities that deserve special treatment as they are used to categorize it?

Forces:

- Business resources are often classified into categories. In a video rental shop, for example, videos may be grouped into categories like "Adventure", "Suspense", "Romance", "Comedy", etc. This qualification is useful for obtaining meaningful reports as for instance, which type of video is preferred by customers and deserves more investment in acquisition. This categorization may be done by adding an attribute to the resource class so that the resource type is considered an attribute of the resource. This approach works well when the category is simply a group description without relevant behavior of its own.

- When there are common attributes and methods inherent to a group delimited by this attribute the separation into two classes is justified. The space necessary to keep these attributes for each resource and the consequent redundancy obtained is undesirable. However, separation may increase processing time as the reference between the two classes must be treated by the system. This must be taken into account when optimizing system performance.

Therefore:

Evaluate all the business resource attributes, identifying those that serve as a resource categorization or grouping.

Solution:

Create a "Resource Type" class for each attribute that serves to establish categories or types of resource and link it to the "Resource" class. The cardinality of the relationship must be "n to 1" as many resources share the same resource type.

Sketch:

Figure 4 shows a sketch for the **QualifyTheResource** pattern. A method was added to the "Resource" class to list the resources by type. The newly created "Resource Type" class has attributes "Code" (optional), "Description" and other attributes according to the specific case.

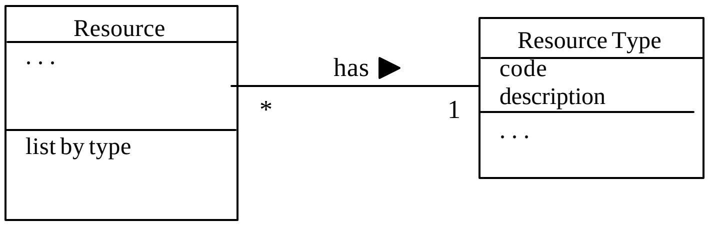

Figure 4 – QualifyTheResource Pattern

Example:

Figure 5 shows an instantiation of the QualifyTheResource pattern, in which "product" plays the "Resource" role and "Manufacturer" plays the "Resource Type" role in a Retail Store system.

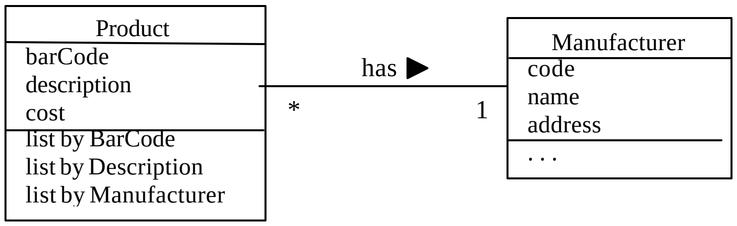

Figure 5 – Instantiation of the QualifyTheResource Pattern

Variations:

To improve even more the qualification, the resource could be classified in many grouping levels, as in an accounting system. For this, new classes could be added, according to the desired degree of qualification; for example, in a video rental, besides the movies qualification mentioned above, another qualification level could be added to group the several categories according to their rental rate ("A" to frequently rented, "B" to moderately rented and "C" to rarely rented, for example). This could allow, for example, establishing rental prices according to these levels.

Related patterns:

QualifyTheResource is an application of the "Type-Object" pattern [Joh 98] and of the Accountability Type [Fow 97].

Next Patterns:

After you QualifyTheResource (or do not), try to QuantifyTheResource (3).

Pattern 3 – QuantifyTheResource

Context:

You have identified resources that your application deals with and optionally you may have defined relevant qualities for each one. An important issue to be considered now is the form of resource quantification. There are certain applications in which it is important to trace specific instances of a resource, because they are individually transacted. For example, a book in a library can have several exemplars and each lent to a different reader. Some applications deal with a certain quantity of the resource. In these applications, it is not necessary to know what particular instance of the resource was actually transacted. For example, a certain weight of steel is sold. In other applications, the resource is dealt with as a whole, as for example a car that goes to maintenance or a doctor that attends a patient.

Problem:

How do you quantify the resource ?

Forces:

- Knowing exactly what is the form of quantification adopted by the application is important during analysis. A wrong decision at this point may compromise future evolution.
- In the case of need to trace specific instances of a resource, redundant information could be stored for the several instances of a same resource, but this redundancy would be undesirable.
- To avoid redundancy a new class could be created in which information common to all instances of a same resource could be stored only once. But a price has to be paid for dealing with two classes instead of only one, as for example, more processing time will be required.

Therefore:

Verify how the resource is dealt with and establish a form of quantification using one of the three **QuantifyTheResource** patterns.

Solution:

There are three slightly different solutions for this problem, depending on the form of quantification. When it is important to distinguish among resource instances, an additional class is created to represent the “ResourceInstance”. An attribute “Status” is added to the “Resource Instance” class, to control its life cycle individually; for example, during a book exemplar life cycle there might be four different possible status: “available”, “only booked”, “only borrowed”, and “booked and borrowed”.

When the resource is managed in a certain quantity, the attribute “Quantity in stock” should be added to the “Resource” class to deal with inventory control. In this case, an attribute “Status” is not applicable, because the system deals with larger quantities of the resource at once and so it cannot control the resource life cycle individually.

When the resource is unique the attribute “Status” should be added to the “Resource” class to control its life cycle; for example, in a car repair shop the vehicle status might be: “Working”, “Broken” and “In repair”.

Sketch:

Figure 6 shows the three different sub-patterns of **QuantifyTheResource** pattern. Use the sub-pattern shown in Figure 6 (a)**,** named “ResourceInstance”, when it is important to distinguish among resource instances; use the sub-pattern shown in Figure 6 (b), named “ResourceMeasurement”, when the resource in managed in a certain quantity; and use the sub-pattern shown in Figure 6 (c), named “SingleResource”, when the resource is unique.

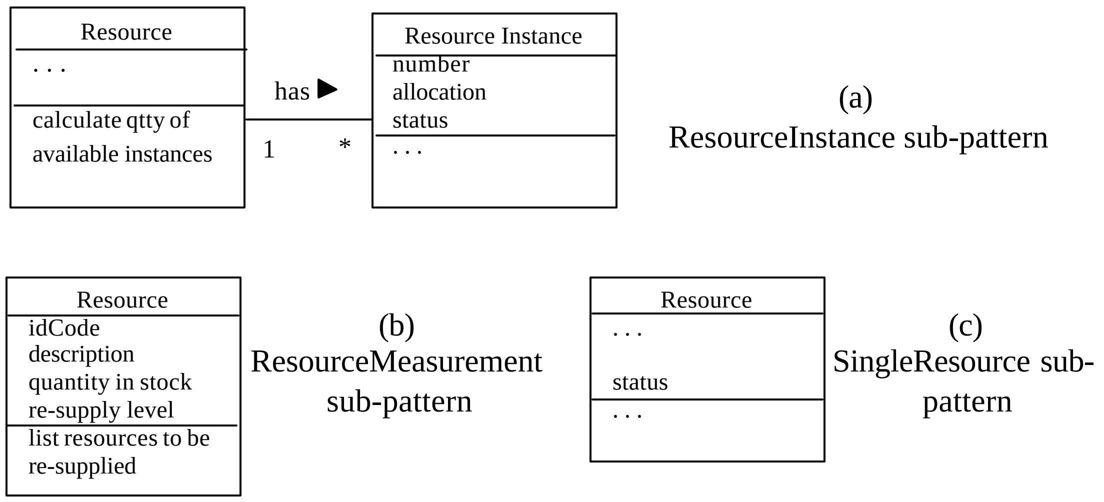

Figure 6 – QuantifyTheResource Pattern

Examples:

Figure 7 shows instantiations of the QuantifyTheResource pattern. In figure 7(a), “Video” plays the “Resource” role and “Videotape” plays the “Resource Instance” role in a Video Rental system. In figure 7(b) “Product” is the “Resource” being controlled and in figure 7(c) “Vehicle” is a “Resource” uniquely managed.

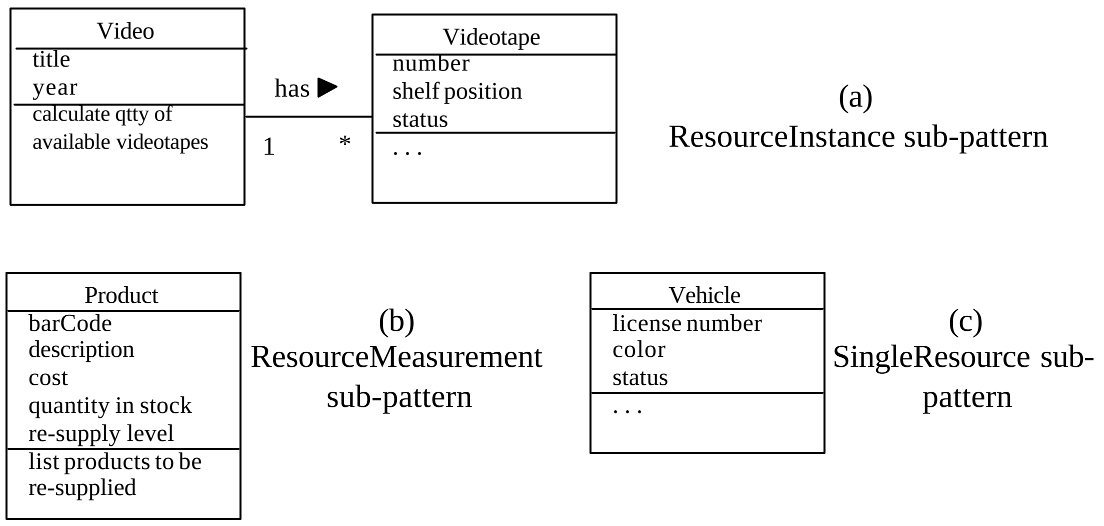

Figure 7 – Instantiation of the QuantifyTheResource Pattern

Related patterns:

ResourceInstance is an application of the "Type-Object" pattern [Joh 98], the Item-Description Pattern [Coa 92] and the Item-Specific Item Pattern [Coa 97].

Next patterns:

After you QuantifyTheResource, the application being developed must be examined to verify which kind of resource transactions are done. If the application concerns the trading of resources, that is, resource purchase or sale, you should TradeTheResource (8) and possibly QuoteTheTrade(5) and CheckResourceDelivery (10). If the application concerns the location or rental of resources, you should RentTheResource (7) and maybe BookTheResource (4). If the application deals with repairing of resources, you should MaintainTheResource (9) and maybe QuoteTheMaintenance (6). Notice, however, that there are applications where several of these patterns can be applied. For example, in a car rental system, besides the booking and the rental of the car, we may have to control acquisition, maintenance and discharge of vehicles.

Section 2: When dealing with resources, several transactions are possible to manage them. The following seven patterns focuses on these Resource Management Transactions. A resource can be booked for future location (**BookTheResource (4)**), can be rented for a certain period (**RentTheResource (7)**), can have its cost quoted before being purchased (**QuoteTheTrade (5)**), can be purchased or sold (**TradeTheResource (8)**), can have its delivery checked (**CheckResourceDelivery (10)**), can have its repair quoted before being repaired (**QuoteTheMaintenance (6)**) or can be repaired (**MaintainTheResource (9)**).

Pattern 4 – BookTheResource

Context:

Your application deals with rental of resources, which may be assets lent to a customer for a certain period or services done by an expert during some time. You have already identified, qualified and quantified the resources managed by your application. Some business systems related to resource rental allow you to previously book it, to guarantee that it you will be yours at the desired schedule. For example, in a video rental shop it is usual to book a video for a specific period, mainly when it is not available when the customer tries to locate it. However, when it is available, booking becomes unnecessary.

Problem:

How do you manage resource booking performed before its actual rental ?

Forces:

- Having just a booking status attribute related to each resource would limit to one the number of customers able to book it. In real situations, several customers could form a queue of people interested in a specific resource.
- Registering booking details is essential for a good management of the available resources, as frequently booked resources are candidates to have their number increased in stock.
- Keeping information about old bookings increases storage needs and separation of booking information implies more processing time.

Therefore:

Verify if your application admits booking for its resources.

Solution:

Create a "Resource Booking" class related to the "Resource" class to represent all the details involved in booking the resource like date, period, fees, etc. Create a "Source-Party" class related to the "Resource Booking" class to represent the organization branch or department doing the booking. Create a "Destiny-Party" class to represent the customer or client that asked for the booking. The "Source-party" class is optional in this pattern, as in small systems it is the organization itself and so it is not worth to create a class to represent it. An operation to discard old bookings is also provided to avoid space wasting.

Sketch:

Figure 8 shows the **BookTheResource** pattern. The booking refers to one source-party, to one destiny-party and to one resource. Use the ItemizeTheResourceTransaction (11) pattern to book more than one resource in a single booking. There are several common attributes in a Resource Booking: the booking date, the period that the customer wants the resource to be located, the booking fee (i.e., the customer can pay in advance some portion of the rental value), and the booking situation, which controls if the booking is pending or has already been attended. There are also common methods: book the resource and cancel the booking. Methods to get bookings by source-party, by destiny-party and by resource are added to the appropriate classes, as shown in Figure 8.

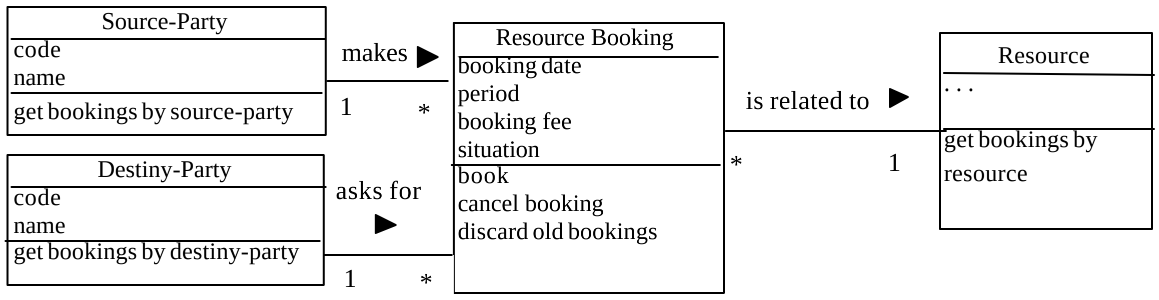

Figure 8 – BookTheResource Pattern

Example:

Figure 9 shows an instantiation of the BookTheResource pattern, in which "Video" plays the "Resource" role, "Video Booking" plays the "Resource Booking" role, "Branch" plays the "Source-party" role and "Customer" plays the "Destiny-Party" role in a Video Rental system.

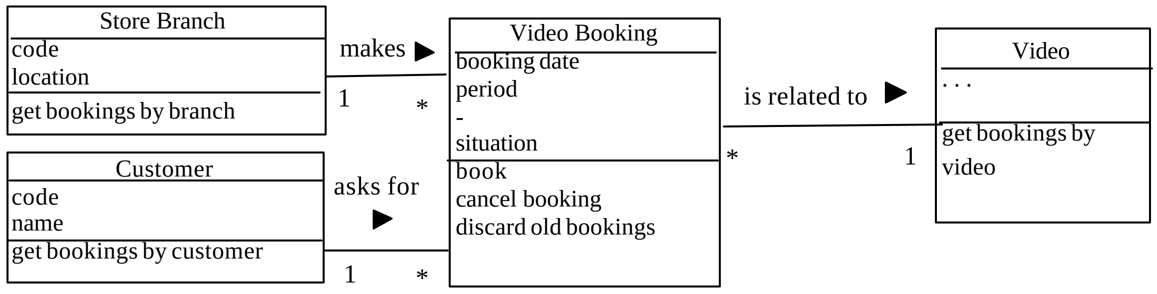

Figure 9 – Instantiation of the BookTheResource Pattern

Variations:

The booking is related to how you QuantifyTheResource (2). For example, in the case of a library, where the Resource Instance pattern has been adopted, the booking is made for the resource (the book), but what is actually located is the exemplar. However, in some cases Figure 8 has to be adapted by replacing the "Resource" class by the "Resource Instance" class. For example, in the video rental example above the customer might want to book the original language videotape instead of the version dubbed in another language. In this case, instead of booking the video itself it would be better to book the particular videotape, which is a resource instance. It is an important design decision to choose between these two solutions.

Related patterns:

BookTheResource is a particular application of the "Association-Object" pattern [Boy 98], and of the "Time-Association" pattern [Coa 92]. It is also a combined application of the "Participant-Transaction" and "Specific Item-Transaction" patterns [Coa 97].

Next patterns:

After having decided whether or not to book the resource, now RentTheResource (7). Also look at Section 3 patterns, which are useful to model other common transaction details.

Pattern 5 – QuoteTheTrade

Context:

Your application deals with resource trading, which may be products purchased or sold by the organization. You have already identified, qualified and quantified the resources managed by your application. Buying or selling products is an activity that demands planning by the party that is investing money to acquire the products. For example, organizations usually ask for a product quotation before deciding whether to trade it or not.

Problem:

How do you keep track of resource quotations performed before its actual trade?

Forces:

- Keeping information about quotations is important both when dealing with purchases and sales. In the first case, a comparative map is done for decision making support. In the second case it is interesting to offer the clients a Quotation service, in which the system could track quotations that do not turn into effective trades, giving the organization manager subsidies to investigate possible competitors.
- Additional space and processing time is needed to deal with this extra information.

Therefore:

Verify if your application admits quotation for its resource sales or purchases.

Solution:

Create a "Trade Quotation" class related to "Resource" class to represent the details involved in the resource quoting. Also create classes "Source-Party" and "Destiny-Party" as described in the solution section of BookTheResource (4) pattern (refer to that section for more details about them).

Sketch:

Figure 10 shows the **QuoteTheTrade** pattern. The quotation refers to one source-party, to one destiny-party and to one resource. Use the ItemizeTheResourceTransaction (11) pattern to quote more than one resource in a single quotation. TradeQuotation attributes are: the quotation date, the validity expiration date for that quotation and the quotation value itself. There are methods to make the quotation and to assign it to the trade, if it becomes effective. Methods to get quotations by source-party and destiny-party are added to the appropriate classes, as shown in Figure 10. A method to get quotations by resource is added to the "Resource" class.

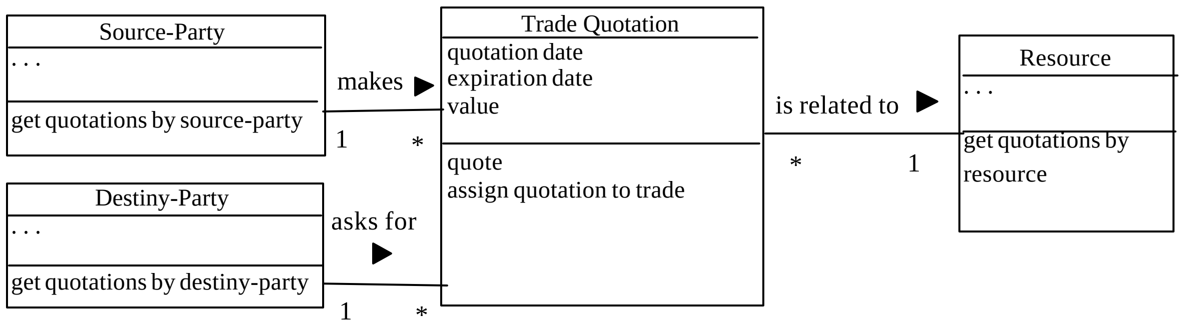

Figure 10 – QuoteTheTrade Pattern

Example:

Figure 11 shows an instantiation of the QuoteTheTrade pattern, in which “Product” plays the “Resource” role, “Purchase Quotation” plays the “Trade Quotation” role, “Supplier” plays the “Source-party” role and “Store-branch” plays the “Destiny-Party” role in a Inventory Control system.

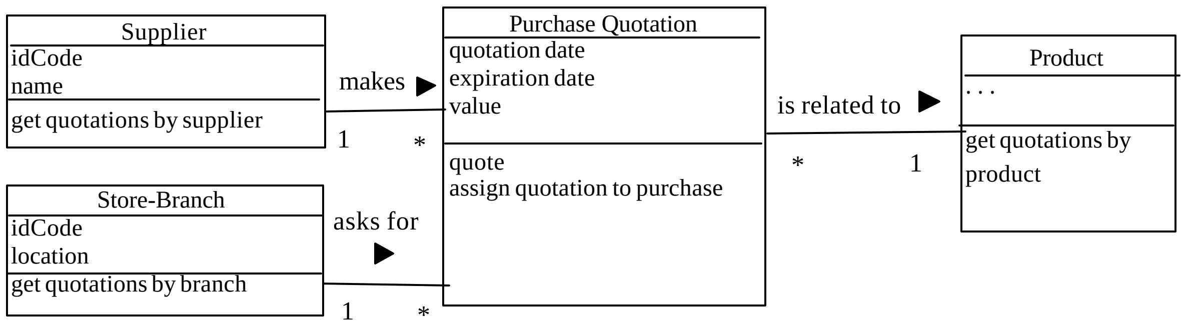

Figure 11 – Instantiation of the QuoteTheTrade Pattern

Variants:

Depending on the quantification pattern used, the “Resource” class must be replaced by the “Resource Instance” class; for example, if different instances of a resource have different costs, the quotation must be more specific.

Related patterns:

QuoteTheResource is a particular application of the “Association-Object” pattern [Boy 98], and of the “Time-Association” pattern [Coa 92]. It is also a combined application of the “Participant-Transaction” and “Specific Item-Transaction” patterns [Coa 97].

Next Patterns:

After you QuoteTheTrade you must TradeTheResource (8). Also look at Section 3 patterns, which are useful to model other common transaction details.

Pattern 6 – QuoteTheMaintenance

Context:

Your application deals with resource maintenance or repair. You have already identified, qualified and quantified the resources managed by your application. In this case, the resource is basically a customer asset that presents faults or needs periodical maintenance. It must be fixed to be used again or to prevent it from failing within a time interval. For example, cars, television sets, electric appliances and computers are resources that often have problems during their life cycle. Most customers would like to have an estimate cost before doing the repair, because sometimes it is not worth to do it due to its high cost compared to the resource value.

Problem:

How do you keep track of maintenance quotations asked by customers ?

Forces:

- Sometimes quotation information is not important unless a concrete maintenance follows the quotation. In this case, quotation attributes may be placed together with maintenance attributes. For example, when you take your television set to be fixed you may be informed about the costs involved before deciding to discard or repair it. The television repair shop may want or not to log the quotation that was not followed by a repair.
- But if you go to a car repair shop to ask for a quotation you will probably be charged for it and the employee responsible will be paid for doing it. So the car repair shop system needs to register the quotation even if a repair does not follow it. Additional storage space and processing time are required when you choose to record each quotation.

Therefore:

Verify if your application admits maintenance quotation before the actual maintenance.

Solution:

Create a "Maintenance Quotation" class related to the "Resource" class to represent the details involved in quoting the maintenance like date, value, etc. Also create classes "Source-Party" and "Destiny-Party" as described in the solution section of BookTheResource (4) pattern (refer to that section for more details about them).

Sketch:

Figure 12 shows the **QuoteTheMaintenance** pattern. The quotation refers to one source-party, to one destiny-party and to one resource. Use the ItemizeTheResource Transaction (11) pattern to quote more than one resource in a single quotation. There are several common attributes in a Maintenance Quotation: the quotation date, the validity expiration date for that quotation and the quotation value itself. There are also common methods: make the quotation and assign it to the maintenance, if it becomes effective. Methods to get quotations by source-party and destiny-party are added to the appropriate classes, as shown in Figure 12. A method to get quotations by resource is added to the "Resource" class.

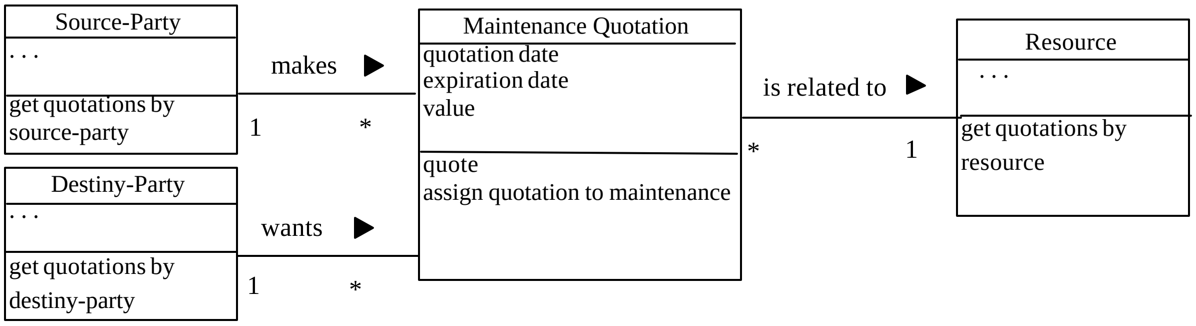

Figure 12 – QuoteTheMaintenance Pattern

Example:

Figure 13 shows an instantiation of the QuoteTheMaintenance pattern, in which "Vehicle" plays the "Resource" role, "Repair Quotation" plays the "Maintenance Quotation" role, "Repair shop branch" plays the "Source-party" role and "Customer" plays the "Destiny-Party" role in a Car Repair Shop system.

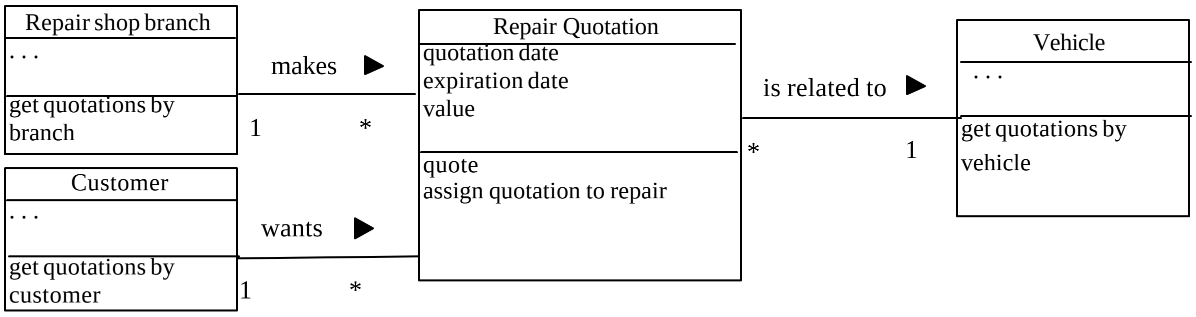

Figure 13 – Instantiation of the QuoteTheMaintenance Pattern

Related patterns:

QuoteTheMaintenance is a particular application of the "Association-Object" pattern [Boy 98], and of the "Time-Association" pattern [Coa 92]. It is also a combined application of the "Participant-Transaction" and "Specific Item-Transaction" patterns [Coa 97].

Next Patterns:

After you QuoteTheMaintenance you must MaintainTheResource (8). Also look at Section 3 patterns, which are useful to model other common transaction details.

Pattern 7 – RentTheResource

Context:

Your application deals with rental of resources, which may be assets lent to a customer for a certain period or services done by an expert during some time. You have already decided whether or not to book the resources before they are rented.

Problem:
How do you manage resource rentals made by your application ?

Forces:

- Several details are involved in renting a resource. Keeping this information is important for a good management of available and rented resources.
- Knowing about previous rentals may help managers to predict which resources deserve more investment in future acquisitions.
- Special care has to be taken so that additional storage space and information processing time are compensated by a better system functionality.

Therefore:
Verify if your application allows resource rental for a certain period of time.

Solution:
Create the "Resource Rental" class related to the "Resource" class to represent all the details involved on renting the resource like date, period, fees, etc.

Sketch:
Figure 14 shows the **RentTheResource Pattern**. The rental refers to one source-party, to one destiny-party, and to one resource. See the ItemizeTheResourceTransaction pattern for renting more than one resource in just one rental. If you have applied the "BookTheResource" pattern (4), link the "Resource Rental" class to the "Resource Booking" class with a "0..1 to 0..1" cardinality relationship, as a booking may or may not give rise to a rental and a rental may occur after a booking or may be done directly. In this case there is no need to link classes "Source-party" and "Destiny-party" to the "Resource Rental" class, as the parties involved in the booking remain the same when the rental is performed. If you have not applied pattern 4, i.e., your application does not need a booking service, also create classes "Source-Party" and "Destiny-Party" linked to the "Resource Rental" class to represent the organization branch renting the resource and the rental beneficiary (the customer, for example), respectively. There are common attributes in a Resource Rental: the starting and finishing dates and the rate rental to be paid by the customer. There are also some common methods: rent the resource, return the resource (when the customer gives it back to the system) and calculate earnings (for example the monthly earnings). Methods to get rentals by source-party, by destiny-party and by resource are added to the appropriate classes, as shown in Figure 14.

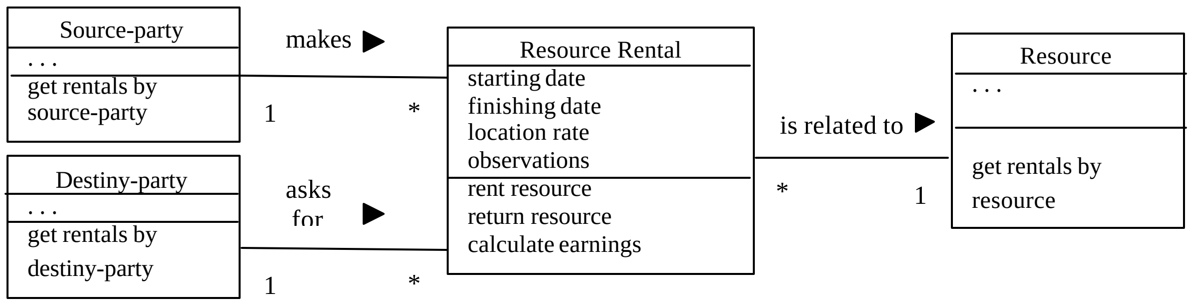

Figure 14 – RentTheResource Pattern

Example:

Figure 15 shows an instantiation of the RentTheResource pattern, in which "Videotape" plays the "Resource" role, "Video Rental" plays the "Resource Rental" role, "Rental store branch" plays the "Source-party" role and "Customer" plays the "Destiny-Party" role in a Video Rental Store system.

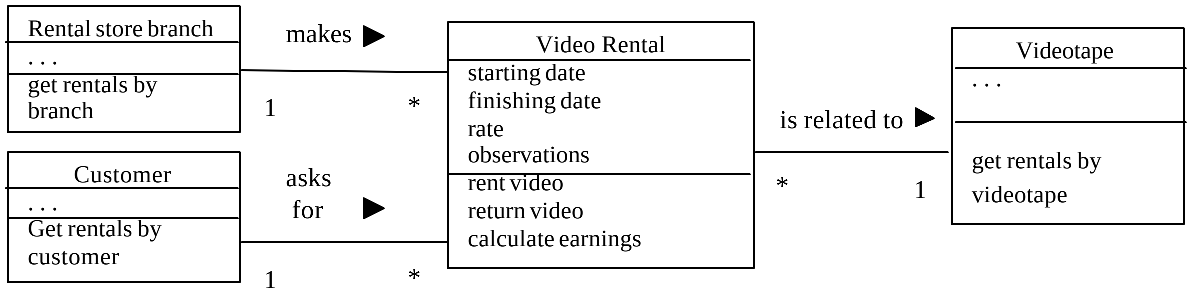

Figure 15 – Instantiation of the RentTheResource Pattern

Related patterns:

RentTheResource is a particular application of the "Association-Object" pattern [Boy 98], and of the "Time-Association" pattern [Coa 92]. It is also a combined application of the "Participant-Transaction" and "Specific Item-Transaction" patterns [Coa 97]. If you consider classes "Resource Booking" (pattern 4) and "Resource Rental" (this pattern) there is an application of the "Transaction – Subsequent Transaction" pattern [Coa 97].

Next patterns:

Now look at Section 3 patterns to specify other details about the rental transaction.

Pattern 8 – TradeTheResource

Context:

Your application deals with trade of resources, which may be resources sold and/or purchased. You have already identified, qualified and quantified the resources traded by your application and optionally applied the QuoteTheTrade pattern (5). Resource trading may be thought of as a resource property transference, in which a resource owned by one party becomes owned by another party. In a sale, if the resource is not available in stock, the customer can fill in a request that will be granted when possible. In a purchase, the system must do a request to the supplier, who delivers the resource within a certain period.

Problem:

How do you manage the resource trades made by your application ?

Forces:

- It is essential to log trade information, as it can be used to generate important reports on resource demand and organization gains (most systems in this domain are concerned with profits).

- The additional storage space and processing time required to register trade information has to be confronted with possible gains in system functionality when evaluating costs versus benefits.

Therefore:
Verify if your application admits resource trade.

Solution:
Create a “Resource Trade” class related to the “Resource” class to represent all the details involved in resource trade. If the “QuoteTheTrade” pattern (5) was applied, link the “Resource Trade” class to the “Trade Quotation” class with a “0..1 to 0..1” cardinality relationship, as a trade quotation may or may not give rise to a trade and a trade may occur after a quotation or may be done directly. “Source-Party” class always represents the original resource owner and “Destiny-Party” class always represents the final resource owner. Thus, in the case of a sale, the “Source-Party” is the organization department or branch that sells the resource and the “Destiny-Party” is the customer buying it. In the case of a purchase, the “Source-Party” is the supplier organization and “Destiny-Party” is the organization department or branch buying the resource. Even if you have applied pattern 5 you must evaluate the need of linking the “Resource Rental” class to “Source-Party” and “Destiny-Party” classes, because many quotations may be done but just one of them will turn into a trade.

Sketch:
Figure 16 shows the **TradeTheResource** pattern. The trade refers to one source-party, to one destiny-party and to one unitary resource. See the <u>ItemizeTheResourceTransaction</u> pattern for trading more than one resource in just one trade. The “Resource Trade” attribute Situation denotes the trade stage: if it is pending, partially fulfilled or fully fulfilled. A “quantity” attribute is added to class “Resource Trade” when, in the “QuantifyTheResource” pattern application, the “Resource Measurement” sub-pattern has been used, to denote a non-unitary resource trade. Besides the “Resource Trade” methods to make a trade, cancel a trade, calculate earnings and get non-delivered trades, methods to get trades by source-party, destiny-party and resource are placed in the appropriate classes, as shown in Figure 16.

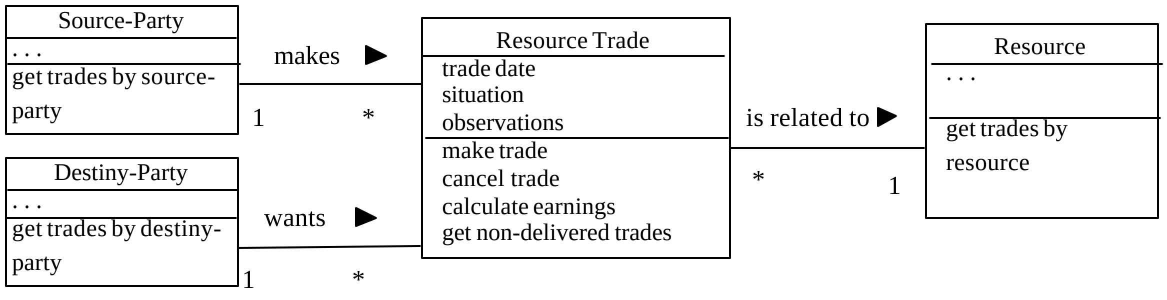

Figure 16 – TradeTheResource Pattern

Example:

Figure 17 shows an instantiation of the TradeTheResource pattern, in which "Product" plays the "Resource" role, "Purchase" plays the "Resource Trade" role, "Supplier" plays the "Source-party" role and "Store-Branch" plays the "Destiny-Party" role in a Inventory Control system.

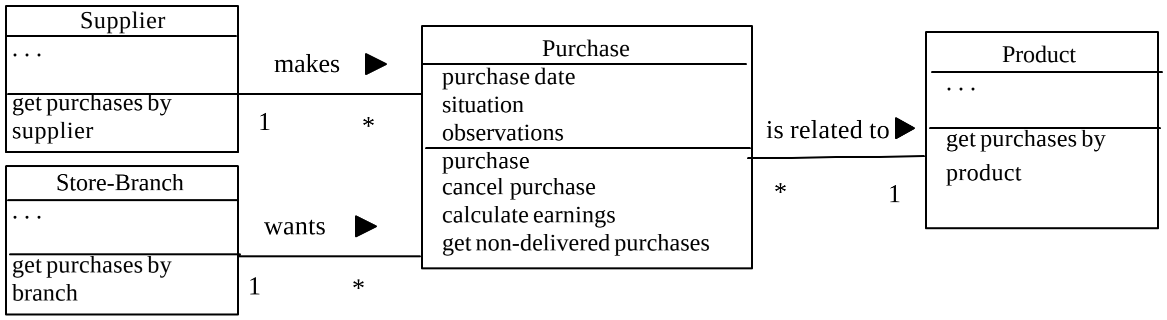

Figure 17 – Instantiation of the TradeTheResource Pattern

Related patterns:

TradeTheResource is a particular application of the "Association-Object" pattern [Boy 98], and of the "Time-Association" pattern [Coa 92]. It is also a combined application of the "Participant-Transaction" and "Specific Item-Transaction" patterns [Coa 97].

Next Patterns:

The Trade is followed by a delivery. So, now, use the CheckResourceDelivery Pattern (10). Also look at Section 3 patterns, which are useful to model other common transaction details.

Pattern 9 – MaintainTheResource

Context:

Your application deals with resource maintenance or repair, as explained in pattern 6 context section. You have already identified, qualified and quantified the resources managed by your application and optionally quoted the maintenance.

Problem:

How do you control resource maintenance performed by your application ?

Forces:

- Keeping records about maintenance performed is important both to customers and to organizations that do maintenance. Customers have the right to complain if the maintenance is not satisfactory. Parties usually need this information for financial purposes. A simple alternative when this information does not need to be kept is to have a resource attribute containing the last maintenance date.
- A lot of extra space is needed to store maintenance information and having several maintenance records related to each resource implies more processing time to retrieve the last maintenance.

Therefore:

Verify if your application admits the repairing or maintenance of resources.

Solution:

Create a “Resource Maintenance” class to control the maintenance made by your application and link it to the “Resource” class. If the “QuoteTheMaintenance” pattern (6) was applied, link the “Resource Maintenance” class to the “Maintenance Quotation” class with a “0..1 to 0..1” cardinality relationship, as a maintenance quotation may or may not give rise to a maintenance and a maintenance may occur after a quotation or may be done directly. If pattern 6 was not applied, create classes “Source-Party” and “Destiny-Party” with the same meaning as described in BookTheResource (4) pattern.

Sketch:

Figure 18 shows the **MaintainTheResource** pattern. The maintenance refers to one source-party, to one destiny-party, and to one resource. See the ItemizeTheResourceTransaction pattern for maintaining more than one resource in just one maintenance transaction. Attributes like “Entry Date” and “Exit Date” are necessary in the “Resource Maintenance” class, as well as “Faults presented”, that takes care of what is wrong with the resource. Some of its methods are: Open Maintenance, to register the resource to be repaired; Close Maintenance, to finish the maintenance; and Get Pending Maintenances, to retrieve those maintenances that are not yet finished. Methods to get maintenances by source-party, by destiny-party and by resource are placed in the appropriate classes, as shown in Figure 18.

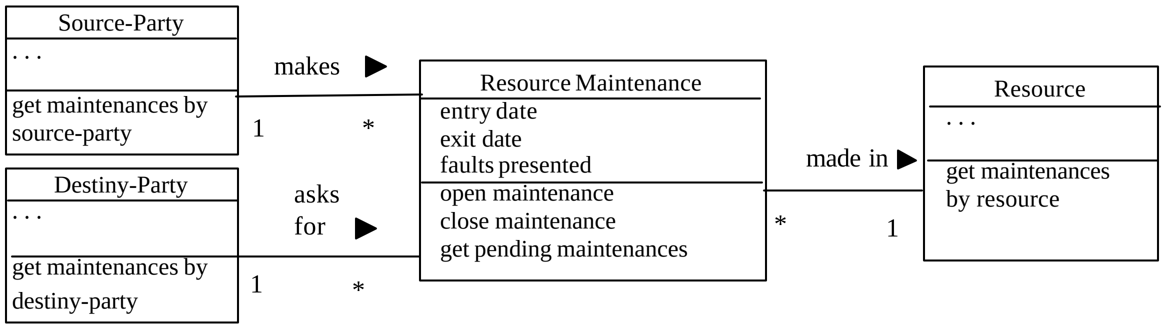

Figure 18 – MaintainTheResource Pattern

Example:

Figure 19 shows an instantiation of the MaintainTheResource pattern, in which “Vehicle” plays the “Resource” role, “Repair log” plays the “Resource Maintenance” role, “Repair shop branch” plays the “Source-party” role and “Customer” plays the “Destiny-Party” role in a Car Repair Shop system.

Related patterns:

MaintainTheResource is a particular application of the “Association-Object” pattern [Boy 98], and of the “Time-Association” pattern [Coa 92]. It is also a combined application of the “Participant-Transaction” and “Specific Item-Transaction” patterns [Coa 97]. If you consider classes “Maintenance Quotation” (pattern 6) and “Resource Maintenance” (this pattern) there is an application of the “Transaction-Subsequent Transaction” pattern [Coa 97].

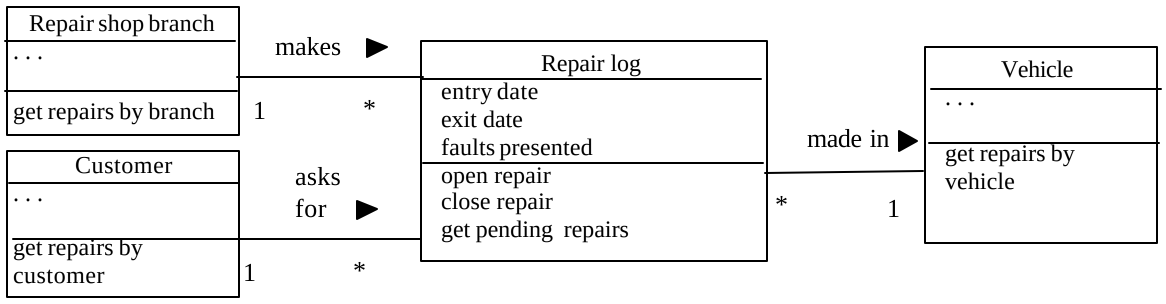

Figure 19 – Instantiation of the MaintainTheResource Pattern

Next patterns:

Now look at Section 3 patterns to specify other details about the maintenance transaction.

Pattern 10 – CheckResourceDelivery

Context:

Your application deals with trade of resources, which may be resources sold and/or purchased. You have already identified, qualified and quantified the resources traded by your application, optionally applied the QuoteTheTrade pattern (5) and applied the TradeTheResource pattern (8). In some systems, a mechanism has to be provided to check the delivery against the trade. For example, when you make a purchase it is not guaranteed that you will receive exactly what you have purchased. A mechanism has to be created to check if the delivery corresponds to the purchase, before you increase the stock level. Similarly, when you make a sale, your delivery department must ensure that the products are correctly sent to the customer, before you decrease the stock level.

Problem:

How do you manage the resource deliveries made by your application ?

Forces:

- In some applications the trade is registered by the system only after the correct delivery. Consequently, nothing is known about the trade in the meantime. This can cause lack of precision in several reports. This decision, although economic, does not reflect the real situation.
- If checking is done just to confirm the trade, an attribute may be added to the trade containing the delivery date. But if there is more information about the delivery that is important for system effectiveness then this information needs to be recorded by the system.
- Knowing about previous deliveries may be important when choosing among several suppliers.
- Storage space and processing time are increased when deliveries are registered individually.

Therefore:

Verify if your application admits delivery checking.

Solution:

Create a “Resource Delivery” class related to the “Resource” class to control the verification done at the resource delivery. As a delivery is associated to a trade, link the “Resource Delivery” class to the “Resource Trade” class with a “1 to 1” cardinality relationship. This cardinality may change if you have used the sub-pattern “Resource Measurement” or applied the ItemizeTheResourceTransaction pattern, as will be seen ahead.

Sketch:

Figure 20 shows the **CheckResourceDelivery** pattern. The delivery must refer to a trade, and a trade is always followed by a delivery. The delivery refers to one resource. See the ItemizeTheResourceTransaction pattern for delivering more than one resource in just one delivery. Besides the “Resource Delivery” methods to make or to cancel a delivery, a method to get deliveries by resource is placed in “Resource” class, a method to assign a delivery to a trade is place in “Resource Trade” class, and methods to get deliveries by source-party and by destiny-party should be placed in “Source-party” and “Destiny-party”, respectively.

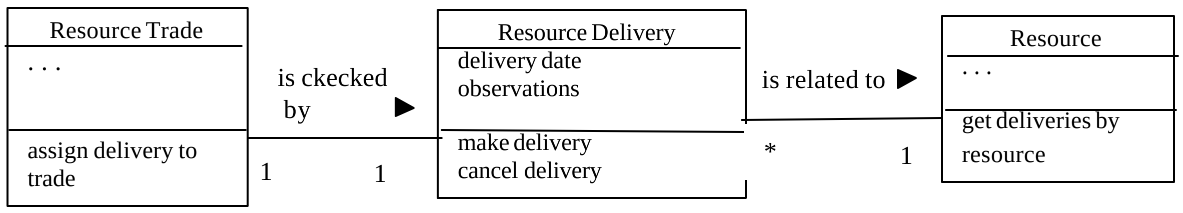

Figure 20 – CheckResourceDelivery Pattern

Example:

Figure 21 shows an instantiation of the CheckResourceDelivery pattern, in which “Product” plays the “Resource” role, “Delivery” plays the “Resource Delivery” role and “Purchase” plays the “Resource Trade” role.

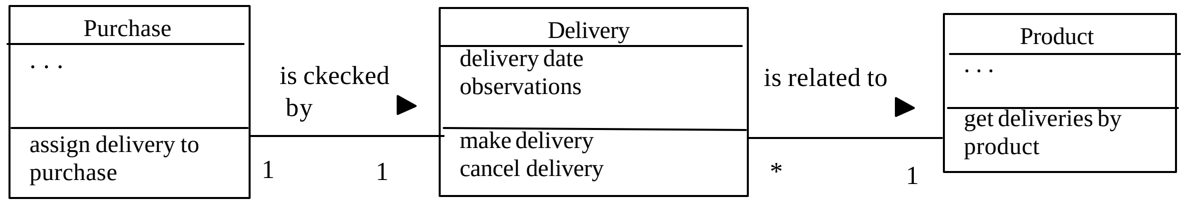

Figure 21 – Instantiation of the CheckResourceDelivery Pattern

Related patterns:

CheckResourceDelivery is a particular application of the “Association-Object” pattern [Boy 98], and of the “Time-Association” pattern [Coa 92]. It is also a combined application of the “Participant-Transaction” and “Specific Item-Transaction” patterns [Coa 97]. If you consider classes “Resource Trade” (pattern 8) and “Resource Delivery” (this pattern) there is an application of the "Transaction-Subsequent Transaction" pattern [Coa 97].

Next Patterns:
Use Section 3 patterns to model other delivery transaction details.

Section 3: There are several common actions related to most of the Resource Transactions discussed in the previous section. The same transaction can have a number of items, each of which related to a different resource (**ItemizeTheResourceTransaction (11)**).The transaction can generate some kind of payment (**PayForTheResourceTransaction (12)**). Maybe the transaction is performed by a person or a team, and this is important for the system (**IdentifyThe TransactionExecutor (13)**). When a resource needs to be maintained we have to care for the labor service that is necessary to make the maintenance (**IdentifyMaintenanceTasks (14)**) and also about the parts used during it (**IdentifyMaintenanceParts (15)**). In fact, both the labor service and the parts are resources when seen in the context of maintenance, whereas the resource to be maintained is actually a good owned by the customer.

Pattern 11 – ItemizeTheResourceTransaction

Context:

Your application manages resources and you have applied one or more of Section 2 patterns. In some applications, it would be desirable to have more than one resource being managed in just one resource transaction. For example, when a customer goes to a video rental store, he or she will probably rent more than one videotape in the same visit. Or when a request is made to a supplier, usually several different products are requested at the same time. Thus, it is convenient to allow a single transaction to have many items. The classes that represent resource transactions, obtained after the application of Section 2 patterns, are indicated in Table 1.

Table 1 – Possible Resource Transactions

Pattern	Resource Transaction class
BookTheResource (4)	Resource Booking
QuoteTheTrade (5)	Trade Quotation
QuoteTheMaintenance (6)	Maintenance Quotation
RentTheResource (7)	Resource Rental
TradeTheResource (8)	Resource Trade
MaintainTheResource (9)	Resource Maintenance
CheckResourceDelivery (10)	Resource Delivery

Problem:
How can you allow several resources being managed in a single transaction ?

Forces:

- Having just a "quantity" attribute in the class that represents the transaction is not a valid solution when different resources are managed in the same transaction: this would only solve the problem of managing more than one unit of the same resource in the same transaction.
- Allowing multiple items managed in the same transaction is not essential in some applications: for example, in a car repair shop the customer almost always takes only one car to be repaired. In this case, possible exceptions could be treated by creating two or more transactions, as the cost to add this functionality would not compensate its rare utilization.

Therefore:

Verify if one transaction can manage several different resources.

Solution:

Create a "Transaction Item" class aggregated to the "Resource Transaction" class, to control each item managed by that transaction.

Sketch:

Figure 22 shows the **ItemizeTheResourceTransaction** pattern. The link between the "Resource Transaction" class and the "Resource" class (or "Resource Instance" class in some cases) has to be removed and a link from the "Transaction Item" class to the "Resource" class (or "Resource Instance") has to be established, as shown in figure 22. The "Transaction Item" class represents each different resource managed by the transaction. It has optional attributes "Quantity" and "Value", applicable only if the Resource Measurement Pattern (3b) was used. A method to totalize transaction items is added to the "Resource Transaction" class.

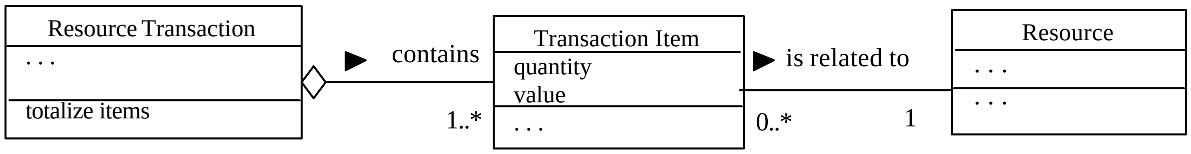

Figure 22 – ItemizeTheResourceTransaction Pattern

Example:

Figure 23 shows an instantiation of the ItemizeTheResourceTransaction pattern, in which "Product" plays the "Resource" role, "Purchase" plays the "Resource Transaction" role and "Purchase Item" plays the "Transaction Item" role.

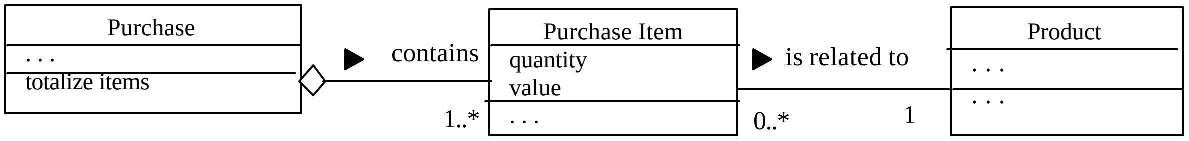

Figure 23 – Instantiation of the ItemizeTheResourceTransaction Pattern

Related patterns:

ItemizeTheResourceTransaction is a particular application of the "Behavior across a Collection" pattern and "State across a collection" pattern [Coa 92]. If you consider classes

"Resource Transaction" and "Transaction Item" there is an application of the "Transaction-Transaction Line Item" pattern [Coa 97]. If you consider classes "Transaction Item" and "Resource" there is an application of the "Item - Line Item" pattern [Coa 97].

Next Patterns:

Now verify whether or not it is necessary to PayForTheResourceTransaction (12) and to IdentifyTheTransactionExecutor (13).

Pattern 12 – PayForTheResourceTransaction

Context:

Your application manages resources and you have applied one or more of Section 2 patterns. Most resource transactions have a cost to be paid by one of the parties involved. For example, to locate a house, to buy a refrigerator or to have their car repaired customers have to pay a certain amount. If the party does not have all the money available at once, some applications offer the possibility of paying in installments. This results in a more complex management, involving incoming and overdue installments control.

Problem:

How can you control the payments associated with resource transactions ?

Forces:

- Having just a few attributes in the "Resource Transaction" class to control the payments done is not always enough, as more accurate information is necessary for a good management of incoming and overdue installments.
- Having customer's historical information about old payments gives the organization better support when they ask for more credit.
- Treating each installment individually adds complexity to the system. This must be considered, for example, when all customers pay cash.

Therefore:

Verify if the Resource Transaction can be paid in more than one installment.

Solution:

If the transaction is paid in only one installment, just add the attributes "Payment Date" and "Value" to the corresponding "Resource Transaction" class (see Table 1). Otherwise, create a "Payment" class related to the "Resource Transaction" class to keep track of each installment to be paid by the appropriate party.

Sketch:

Figure 24 shows the **PayForTheResourceTransaction** pattern. A method "Decide payment conditions" is added to the "Resource Transaction" class to establish how many installments the customer wants and the interest rates involved. The "Payment" class has attributes "Due Date", "Payment Day", "Installment number", "Value" and "Situation". This last attribute controls the possible states of an installment, for example, it can be an incoming, overdue or already paid installment. The method "Coming installments" lists all the installments with dates in the future. The method “Overdue Payments” lists all the installments with dates in the past that are not yet paid. The method “Register Payment” takes care of quitting an installment paid by the customer. The method “Payments done” lists all the installments paid in a certain period.

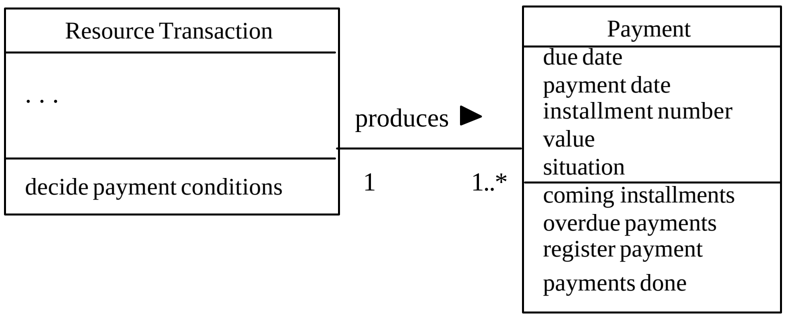

Figure 24 – PayForTheResourceTransaction Pattern

Example:

Figure 25 shows an instantiation of the PayForTheResourceTransaction pattern, in which “Sale” plays the “Resource Transaction” role and “Accounts Receivable” plays the “Payment” role.

Related patterns:

PayForTheResourceTransaction is a particular application of the “Transaction – Subsequent transaction” pattern [Coa 97].

Next Patterns:

Now verify if it is necessary to IdentifyTheTransactionExecutor (13).

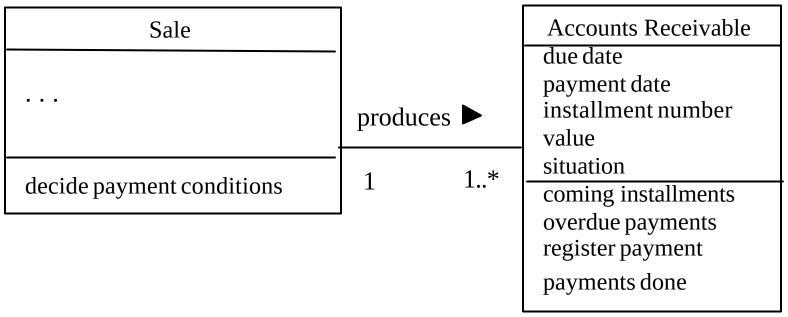

Figure 25 – Instantiation of the PayForTheResourceTransaction Pattern

Pattern 13 – IdentifyTheTransactionExecutor

Context:

Your application manages resources and you have applied one or more of Section 2 patterns. Knowing the person or team who executed a resource transaction is useful in certain applications; for example, in a computer store, the salesman who sells computers and peripherals receives commission for the sales, weekly or monthly. Thus, the system will need this information to generate commission reports.

Problem:
How do you identify the person or entity responsible for the transaction execution ?

Forces:

- Having just an "executor" attribute in the "Resource Transaction" class may be a good solution for small systems where nothing except the executor name is available. But in some systems the executor has other attributes that are necessary for appropriate management as, for example, a settled salary, a special commission percentage, a minimum sales value, etc.
- Storing each executor separately demands more space and processing time that will be worth only if the application needs so.

Therefore: Verify if the transaction executor is important for the system.

Solution:
Create a "Transaction Executor" class related to the "Resource Transaction" class (see Table 1) to represent the possible persons or teams responsible for the transaction.

Sketch:
Figure 26 shows the **IdentifyTheTransactionExecutor** pattern. A method is added to the "Resource Transaction" class to provide the commission payment related to a specific transaction. The "Transaction Executor" class has attributes Code, Name, Specialty, commission percentage, minimum sales value and salary and methods to get transactions by executor and to list commissions paid.

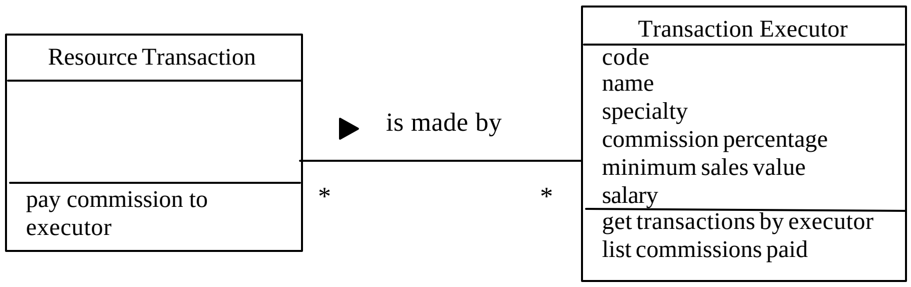

Figure 26 – IdentifyTheTransactionExecutor Pattern

Example:
Figure 27 shows an instantiation of the IdentifyTheTransactionExecutor pattern, in which "Sale" plays the "Resource Transaction" role and "Salesman" plays the "Transaction Executor" role.

Related patterns:
IdentifyTheTransactionExecutor is a particular application of the "Participant - Transaction" pattern [Coa 97].

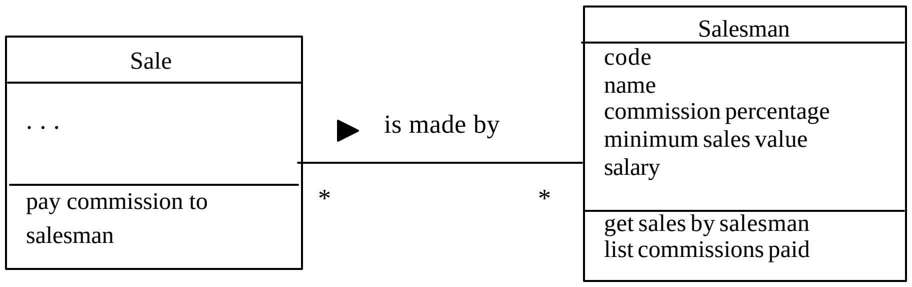

Figure 27 – Instantiation of the IdentifyTheTransactionExecutor Pattern

Variations:

In large applications, the executor could be a team, so it would be necessary to include one more class, related to Transaction Executor, to control the sharing of commission among team members.

Next Patterns:

If your transaction is a Maintenance, now verify if it is necessary to IdentifyMaintenanceTasks (14) and to IdentifyMaintenanceParts (15). Otherwise, verify in Table 1 if your application has other transactions for which this section patterns have not yet been applied.

Pattern 14 – IdentifyMaintenanceTasks

Context:

Your application deals with resource maintenance and you have applied the MaintainTheResource (9) pattern and optionally patterns 6, 11, 12 and 13 (or any combination of them). When a resource presents a fault and needs to be maintained, usually there are some labor services involved in repairing it. For example, a car with a brake problem could need to have a change of the brake shoes, an adjustment of the brake cable and a lubrication of the pedal support. In some cases, each of these labor services is done by a different person. Thus, it is important to specify the several tasks performed during the Resource Maintenance.

Problem:

How can you identify the tasks involved in a maintenance transaction or in a maintenance quotation ?

Forces:

- If little information about maintenance activities is necessary then this information might be logged as maintenance attributes. However, all maintenance cases would be limited to a certain number of tasks. If this number is low certain cases would not be covered and if this number is high space would be wasted.
- In most systems it is desirable to control each maintenance activity individually, as this allows its repetition in different maintenance cases to be better controlled

and compared. So the availability of this information to prepare quotations for new cases and for production scheduling would be improved.

Therefore:

Verify if repairing the resource involves several tasks, possibly executed by different people.

Solution:

Create an aggregate class “Maintenance Task” for the “Resource Maintenance” class, with attributes “Problem to solve”, “Labor description”, “Hours spent” and “Cost”.

Sketch:

Figure 28 shows the **IdentifyMaintenanceTasks** pattern. A maintenance may have several tasks. The “Maintenance Executor” class is optional and is equivalent to “Transaction Executor” class, shown in Figure 26. The decision to use it or not is based on the same subsidies related on the IdentififyTheResourceExecutor pattern. If the decision is to use it, link it either to the “Maintenance Task” class (if each task may be performed by a different executor), as shown in Figure 28, or to the “Resource Maintenance” class (if the whole maintenance is performed by only one executor).

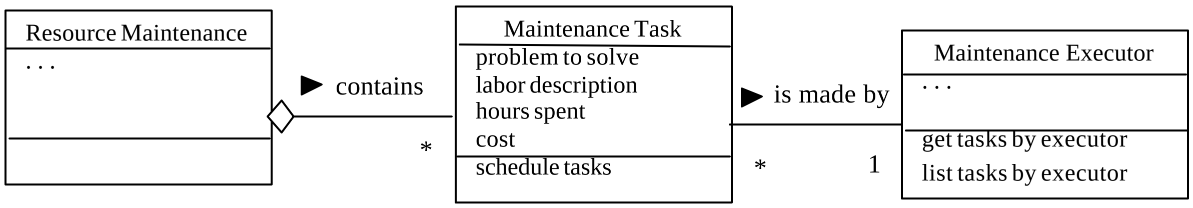

Figure 28 – IdentifyMaintenanceTasks Pattern

Example:

Figure 29 shows an instantiation of the IdentifyMaintenanceTasks pattern, in which “Vehicle repair” plays the “Resource Maintenance” role, “Labor task” plays the “Maintenance task” role and “Repairman” plays the “Maintenance Executor” role.

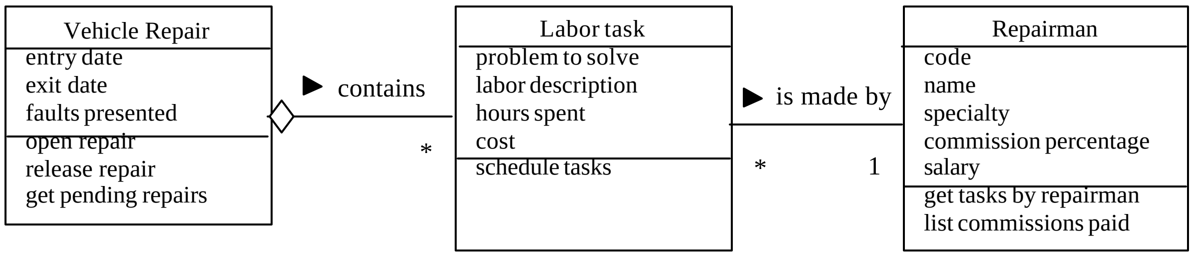

Figure 29 – Instantiation of the IdentifyMaintenanceTasks Pattern

Related patterns:

If you consider classes “Resource Maintenance” and “Maintenance Task” there is an application of the “Transaction-Transaction Line Item” pattern [Coa 97].

Next Patterns:

Now verify if it is necessary to IdentifyMaintenanceParts (15). Otherwise, verify in Table 1 if your application has other transactions for which this section patterns have not yet been applied.

Pattern 15 – IdentifyMaintenanceParts

Context:

Your application deals with resource maintenance and you have applied the MaintainTheResource (9) pattern and optionally the patterns 6, 11, 12, 13 and 14 (or any combination of them). During a resource maintenance, probably some parts that belong to the resource need to be changed, either because they have failed or because they are likely to fail soon. For example, if a car has a brake problem then the break shoe has to be substituted by a new one and grease has to be provided to lubricate the pedal support. In such cases it is important to specify which parts are used in a Resource Maintenance.

Forces:

- In applications where there is an inventory control sub-system it is desirable to discriminate the parts used in a maintenance, as this information can be used by the inventory control to decrease stock level, thus linking the two sub-systems. It is also important to treat parts individually to ease warranty control, although space and processing time requirements are increased.
- On the other hand, if parts are not logged by any sub-system then information about parts used in maintenance might be logged as attributes of the maintenance. Using this approach would either limit the number of parts used in each maintenance to a pre-defined value or establish a certain space to textually list the parts used.

Therefore: Verify if it is necessary to know which parts were used in the resource maintenance.

Solution:

Create a "Part used in maintenance" class aggregated to the "Resource Maintenance" class. Also create a "Part" class to contain all parts available in the organization.

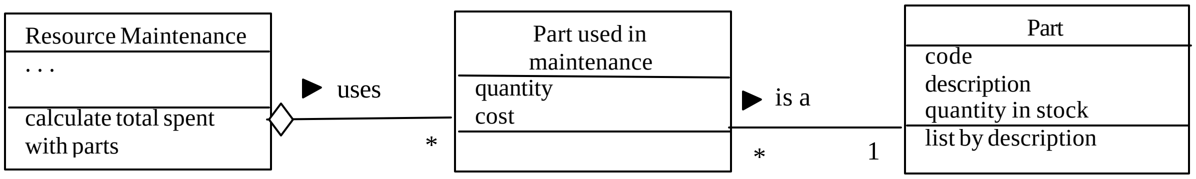

Figure 30 –IdentifyMaintenanceParts Pattern

Sketch:

Figure 30 shows the **IdentifyMaintenanceParts** pattern. Each resource maintenance may use several parts and each part used in maintenance refers to a specific part in stock. A method to calculate the total amount spent with parts must be added to the "Resource Maintenance" class.

Example:

Figure 31 shows an instantiation of the IdentifyMaintenanceParts pattern, in which "Vehicle repair" plays the "Resource Maintenance" role, "Part used" plays the "Part used in maintenance" role and "Part" plays the "Part" role.

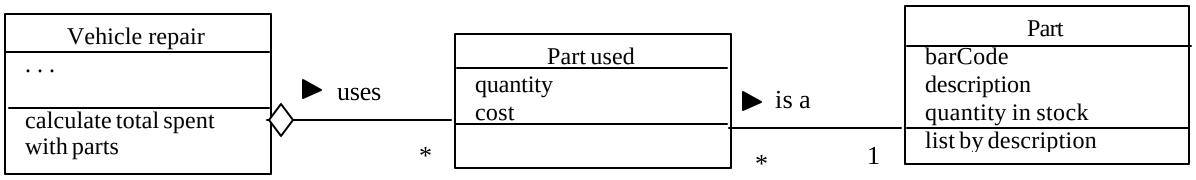

Figure 31 – Instantiation of the IdentifyMaintenanceParts Pattern

Related patterns:

If you consider classes "Resource Maintenance" and "Part used in maintenance" there is an application of the "Transaction-Transaction Line Item" pattern [Coa 97]. If you consider classes "Part used in maintenance" and "Part" there is an application of the "Item - Line Item" pattern [Coa 97].

Next Patterns:

Now verify in Table 1 if your application has other transactions for which this section patterns have not yet been applied.

An Application Example

Applying the Pattern Language to model a small Car Repair Shop, patterns (1), (2), (3b), (3c), (8), (9), (11), (12), (13), (14) and (15) were used. The final object model produced is shown in Figure 32. Table 2 summarizes the result of the pattern language application. Reading the columns named "Repair subsystem" and "Purchase subsystem" from top to bottom, the sequence that was followed in the application of the pattern language to these subsystems can be seen. The final object model may be enhanced finding possible class generalizations as, for example for "Accounts Payable"/"Accounts Receivable" and for "Purchased part"/"Requested part".

Concluding Remarks

The presented pattern language reflects ten years of professional practice developing systems in the resource management domain. Its application to new cases has made analysis much easier, as it supplies a guideline for more disciplined analysis with the assurance that the main aspects of the systems in this domain are taken care of.

We plan to extend this language to include store of resources and a better treatment for payments. A framework based on this pattern language will also be developed.

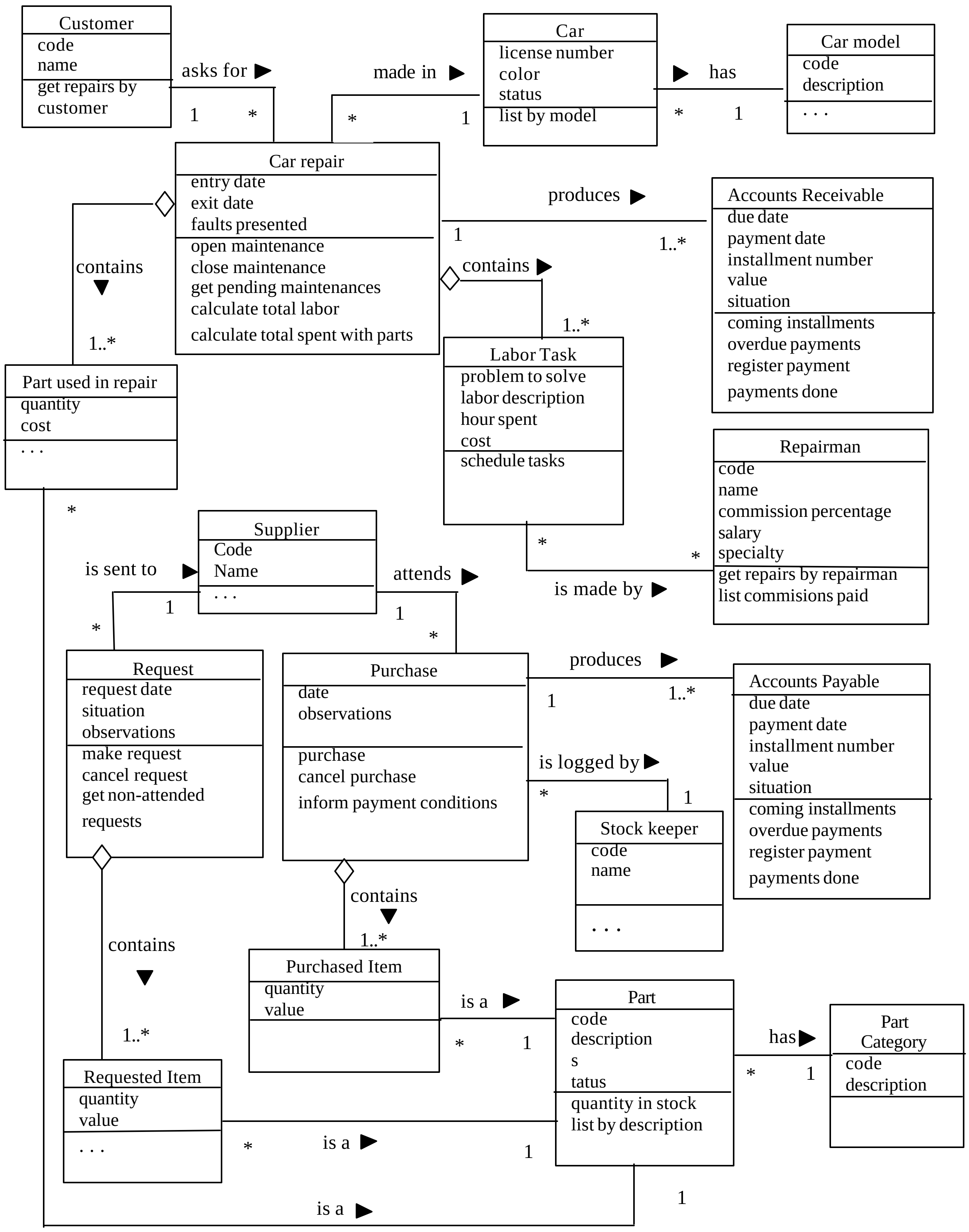

Figure 32 – Application of the Pattern Language to a simple Car Repair Shop

Table 2 – Summary of the pattern language application results

Pattern	Class Participant	Repair subsystem	Purchase subsystem	
1 – IdentifyTheResource	Resource	Car	Part	
2 – QualifyTheResource	Resource type	Car model	Part category	
3b – QuantifyTheResource			No classes added, only attributes	
3c – QuantifyTheResource		No classes added, only attributes		
8 – TradeTheResource	Source-Party Destiny-Party Resource Trade		Supplier - Request	
9 – MaintainTheResource	Source-Party Destiny-Party Resource Maintenance	- Customer Car repair		
10 – CheckTheDelivery	Resource delivery			Purchase
11 – ItemizeTheResource Transaction	Transaction item		Requested item	Purchased item
12 – PayForTheResource Transaction	Payment	Accounts receivable		Accounts payable
13 – IdentifyTheTransaction Executor	Transaction executor	Repairman		Storekeeper
14 – IdentifyMaintenanceTasks	Maintenance task Maintenance executor	Labor task Repairman		
15 – IdentifyMaintenanceParts	Part used in maintenance Part	Part used in repair Part		

Acknowledgments

This work has begun during the shepherding process of another paper, workshoped at PLOP'98 [Braga 98]. We are grateful to Norm Kerth who, in that process, suggested that the patterns of that paper could be broken down into smaller patterns to form the pattern language here presented. We also thank Bruce Whitenack, who is the shepherd of this paper, for his support.

References

[Boy 98] Boyd, Lorraine. *Business Patterns of Association Objects*. In "Martin, Robert C. (ed.); Riehle, Dirk (ed.) and Buschmann, Frank (ed.) Pattern Languages of Program Design 3", Addison-Wesley, pp. 395-408, 1998.

[Braga 98] Braga, Rosana T. V.; Germano, Fernão, S.R.; Masiero, PauloC. *A Confederation of Patterns for Resource Management*. Workshoped in the 5th Conference on Pattern Languages of Programs (PLoP'98), Monticello-Illlinois, August 1998.

[Coa 92] Coad, Peter. *Object-Oriented Patterns*. Communications of the ACM, V. 35, nº9, pp. 152-159, September 1992.

[Coa 97] Coad, P.; North, D.; Mayfield, M. *Object Models: Strategies, Patterns and Applications*, Yourdon Press, 2nd edition, 1997.

[Eri 98] Eriksson, Hans-Erik and Penker, Magnus. *UML Toolkit*, Wiley Computer Publishing, 1998.

[Fow 97] Fowler, Martin. *Analysis Patterns*. Addison-Wesley, 1997.

[Joh 98] Johnson, Ralph and Woolf, Bobby. *Type Object*. In "Martin, Robert C. (ed.); Riehle, Dirk (ed.) and Buschmann, Frank (ed.) Pattern Languages of Program Design 3", Addison-Wesley, pp. 47-65, 1998.