A pattern-based approach for information systems specification and developement

Christine Choppy
LIPN—Université Paris 13
99 Av. J.-B. Clément
93430 Villetaneuse, France
Christine.Choppy@lipn.univ-paris13.fr

Maritta Heisel
FB Ingenieurwissenschaften
Universität Duisburg-Essen
D-47048 Duisburg, Germany
Maritta.Heisel@uni-duisburg-essen.de

Abstract : Patterns such as problem frames and architectural styles are used here to support the formal specification and the development of information systems. New problem frames specific to information systems are proposed to describe identified sub-problems and to help the formal specification task. Recomposition is achieved through a component based approach and an architectural style that puts together the different components. We propose an original method to guide this process, taking advantage of UML concepts for the first decomposition level, then using patterns. These ideas are illustrated with a case study.
Keywords : information systems specification and developement, problem frames, architectural styles, components, formal specification.

1 Motivation

It is acknowledged that the first steps of software development are essential to reach the best possible adequation between the expressed requirements and the proposed software product, and to eliminate any source of error as much as possible. Thus, to start with, emphasis is put on a precise and unambiguous expression of requirements. There are several possible approaches for this aim, that have respective advantages and drawbacks. The UML notation [UML] has the advantage of being widespread, of concepts proved satisfactory in the industrial world, of graphic notations, and the drawback of lacking a formal semantics (which could put a limit to any methodological guide . . .). Formal specifications alleviate this problem, and given their precise expression, lead one to raise the questions that yield progress in the problem understanding. The remaining pitfall is the size and/or the complexity, that makes it difficult to get a synthetic understanding, and may yield misleadings. Patterns offer families of frequently met structures that the user is invited to "try" (or even to adapt) on the problem to be solved, so as to benefit from "ready to wear" structuring concepts. Patterns may thus be viewed as an elaborated mean of reusing knowledge acquired from experience. Problem frames [Jac01] are proposed by M. Jackson to provide a global structure to problems. Architectural styles [GS93, BCK98] bring structures of a finer granularity that are often usable at the design level. The component based architecture [CD01] enables one to specify how the components (that, according to structuration concepts, should be developed independently) should eventually be integrated. It may be desirable to combine these approaches so as to benefit from their combined advantages, so our purpose here is to present the corresponding approach we developed for information systems. Our attention was drawn to this class of problems that covers numerous applications because we thought that the problem frame schemas proposed by Jackson do not present a fully convenient structuration for these. The approach that we present here may be valid in a more general case, but we finalized it particularly for this class of problems. After proposing problem frames that are specific for information systems (for queries and updates), we describe our approach (Figure 1) that offers a methodological guide and indicates how to systematically link contributions from (i) *use cases* and *scenarios* currently used to express requirements, (ii) *problem frames* that allow one to identify problem structures to which *formal specifications* of the different structure parts may be associated, (iii) *component architecture* to recompose the different developed components.

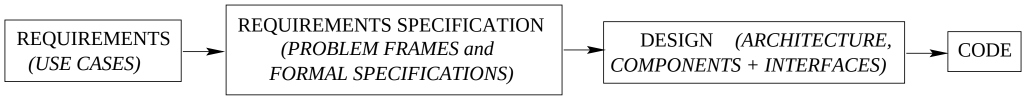

Figure 1: Development approach

After a recall (Section 2) of characteristics of the different concepts used in this work, we propose (Section 3) a methodological approach guided through a sequence of successive and combined steps. We illustrate this on a case study (Section 4) before conclusion.

2 Basic concepts

We provide here some notions on the used concepts that may be useful to understand our approach. We suppose the reader is familiar with formal specifications, and the part of the Z language used will be presented.

2.1 Use cases and scenarios

Use cases were introduced by Jacobson et al. [JCJO92] after the idea of scenarios that describe the different possibilities for a use case. Use cases were used to describe requirements relative to a software system while giving an overall view of it. UML [UML] provides a diagram for use cases (an example is shown in Figure 7) and indicates that it should be accompanied by descriptions, and that the sequence of activities in a use case should be documented by behaviour specification such as interaction diagrams (for instance, sequence diagrams in Figures 8 and 9).

2.2 Problem Frames

A problem frame [Jac01] is a schema that intuitively defines a problem class defined in terms of its context, its domains characteristics, its interfaces and its requirements. The system to be developed is represented by the "machine".

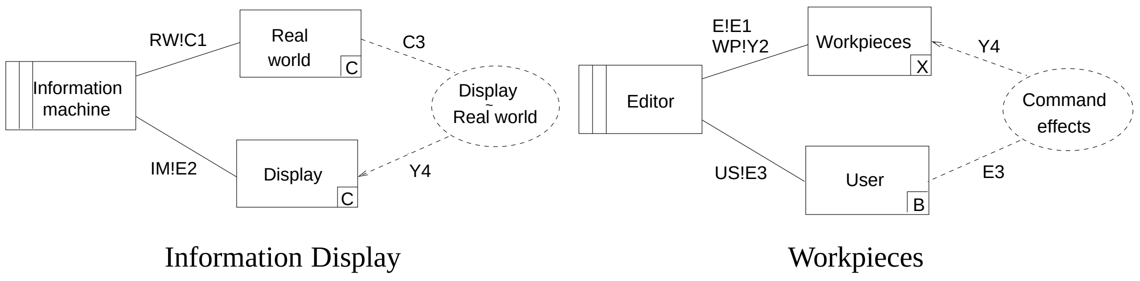

Figure 2: Problem frames diagrams

For each problem frame a diagram is set up (Figure 2). Plain rectangles denote application domains (that already exist), rectangles with a double vertical stripe denote the machine domains to be developed, and requirements are denoted with a dashed oval. They are linked together by lines that represent interfaces, also called shared phenomena. Jackson distinguishes "causal" domains that comply with some laws, lexical domains that are data physical representations, and biddable domains that are people. Jackson defines five basic problem frames, and we present here two of them (Figure 2). The "Information Display" problem frame offers a structure for applications devoted to the display of real world physical data. The "`C`" indicates that the "Real World" domain is causal, and `RW!C1` indicates that the phenomena `C1` is controlled by the Real World. The dashed line represents a reference to requirements, and the arrow indicates that it is a constraining reference. The "Workpieces" problem frame is used for tools that allow a user to create and edit a class of graphical or textual objects that may be copied or printed. The "`X`" indicates that the "Workpieces" domain is lexical. In order to use a problem frame, one should provide an instance for its domains, interfaces and requirements.

2.3 Architectural styles

Architectural styles [GS93, BCK98] are software architecture patterns that are characterized by

- a set of components (e.g., data repository, processes, . . .) that realize some functions when executed
- a topological display of the components showing their relationships when executed
- a set of semantic constraints,
- a set of connectors for communication, coordination or cooperation

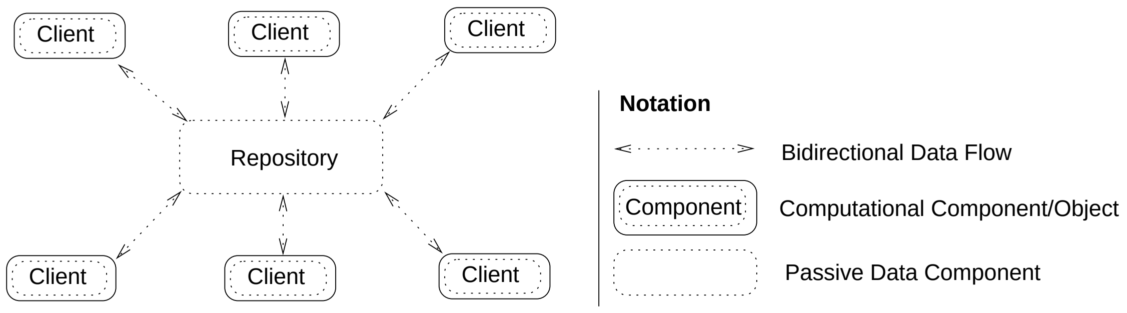

Figure 3: Data centered architectural style

Among the main architectural styles, the "repository" style (Figure 3), that is data centered and where different clients access shared data, is quite convenient for information systems.

2.4 Component based software engineering

In the architectural styles domain, the word "component" denotes a piece of software that performs computations. It is not required that this software piece should comply with any specific constraints.

However, some years ago, a new domain emerged for component based software engineering [Szy99]. The basic idea there is to build software from existing software pieces, that are embedded and somewhat independent. These software pieces are called "components", and in this context components should comply with the following requirements:

- All services given and required by a component are accessible uniquely through well defined *interfaces*.
- A component adheres to a *component model*. This model specifies among others syntactic conventions for the interface definitions and the way components communicate. It is used

to guarantee the interoperability of several components adhering to the same component model. Existing component models are *JavaBeans* [Sun97], *Enterprise Java Beans* [Sun01], *Microsoft COM*$^+$ [Mic02], and the *CORBA Component Model* [Obj02].

- Components are deployed under binary format. The source code may not be available to a component client (i.e. the agent composing systems from components). This is the reason why the component specification should contain all information necessary to use it.

In the component based software engineering approach, architectural principles play an essential role because the system composition from components is achieved according to a given architecture. The two domains thus are strongly related.

Our aproach consists in decomposing a complex information problem into subproblems that are all described using problem frames. For each subproblem, a software should be developed, and the overall system consists in an appropriate combination of these subproblem solving softwares.

We propose to realise the subproblem solving softwares as components together with interfaces derived from the interfaces derived from those defined from problem frames instances. The combination is done through *a component architecture*, as proposed for instance by Cheesman and Daniels [CD01].

3 The proposed approach description

We propose specific problem frames for information systems (queries and updates problem frames), then we describe our development approach that takes advantage both of the structuring concepts brought by the problem frames and by the architectural styles, as well as of the components integration concepts, together with a solid basis provided by formal specifications.

3.1 Problem frames for information systems

The main tasks for operation systems are the updates of the database information, and the queries to obtain some pieces of information. Usually several queries may be done simultaneously, while a lock forbids that any operation should be done at the same time as an update. In the problem frames we propose for information systems, we use a domain for the database (DBM or DataBase Model) that, when an update is done, will be constrained (that is, the values updates should comply with the requirements) relatively to the update rules, and will not be constrained when queries are performed. Thus, we shall distinguish these two cases.

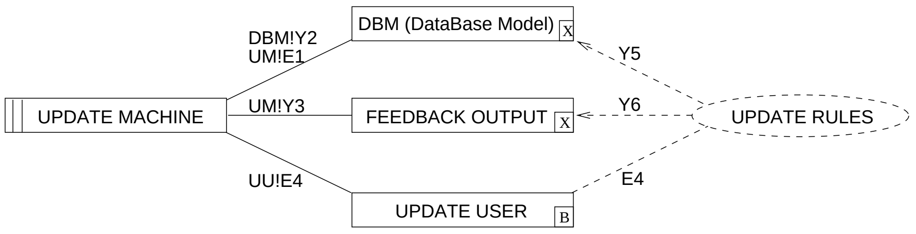

Figure 4: Update machine

Update The schema we propose for update in Figure 4 may be seen as an adaptation and an extension of the "workpieces" problem frame (Figure 2).[1] We identify domain components that

[1]Note that the schema we propose is also close to the "Commanded Information" frame that Jackson offers as a variant of the "Information Display" frame with a user where domains are causal (and not lexical), and with a simple

are a database model (“X” indicates that it is a lexical domain, that is passive), a user (“B” indicates it is a “biddable” domain) who emits update commands , and a feedback output towards the user. Requirements are expressed through update rules, and the machine that performs the updates is to be designed. The identified interfaces take into account the fact that an update operation may be preceded by a query to check that its preconditions are satisfied :
• E4 (also in UU!E4) user command with an update
• E1 (in UM!E1) query or update phenomena controlled by the machine
• Y2 (in DBM!Y2) information messages relative to the database state (and history)
• Y3 (in UM!Y3) output messages controlled by the machine and informing either of the success or of the failure reasons of the user command
• Y5 user command effects on the database, according to the update rules, and expressed in terms of data values
• Y6 user command effects on the output messages, according to the update rules, and expressed in terms of data values.
Let us note that the interfaces labelled by Y5 and Y6 display arrows that express constraints towards the database model, and the user output.

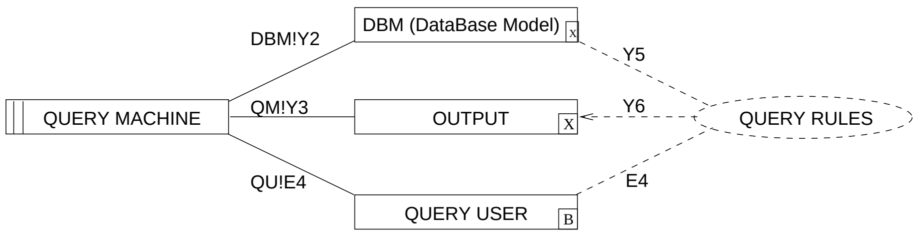

Figure 5: Query machine

Query The problem frame we propose for queries in Figure 5 may be considered as an adaptation and an extension of the “information display” problem frame (Figure 2). The domain components identified are a database model, a user emitting queries commands, and an output towards the user. Requirements are expressed through queries rules, and the machine performing the query is to be designed. The identified interfaces are the following :
• E4 (also in QU!E4) user command for the query
• Y2 (in DBM!Y2) information message on the database state (and history)
• Y3 (in QM!Y3) output message controlled by the machine, that is either an answer the user query or an error message
• Y5 information on the database state in relationship with the user command (according to the query rules), expressed by data values
• Y6 effects of the user command on data appearing on output messages (according to the query rules).
Let us note that, in this schema, only the interface labelled by Y6 exhibits a constraining arrow towards the produced output. Thus, the two schemas are very close, but we think it is important to distinguish them since only the update operation requests a lock on the database, and it is useful to have this information especially when the two schemas are composed together.

interface between the machine and the “Real World”.

3.2 Development method

For a given class of systems, we propose a methodological guide for a combined use of the concepts presented in Section 2 together with formal specifications.

Criteria for the considered systems The goal of the systems we consider is the management of data. Their environment is a “business domain”, that is an organisation created by human beings, that does not comply with physics laws. There are no sensors or actuators, nor any computer means to ensure the correspondance between the real world and its model. It is the responsibility of human beings to inform the system of the real world state changes.

Step 0 If the considered system complies with these criteria, then a use case diagram should be drawn, and a real world model should be given (e.g., through a class diagram). This model will be shared by the different machines. To this end, the usual object oriented analysis method is used, e.g., first identify the actors that communicate with the system but are not part of the system. Then identify the different use cases. Each use case should be specified by a description of the sequence of actions that pertain to it. In this paper we use sequence diagrams to describe scenarios.

Scenarios exhibit arrows from an actor towards the system. These arrows correspond to operations available for the actor. In the system realisation, each of these operations will be part of an interface corresponding to the considered use case (see step 3 description). The different use cases identification provides a natural decomposition framework supporting the next step work.

Step 1 For each use case, a problem frame instance is drawn, and if the use case implies a state change in the model, then the update machine will be used, otherwise the query machine will be used. It may be the case that additional decomposition is needed, and/or that another kind of problem frame should be used (as, for instance, the “translation frame” [Jac01] that may be applied for data processing, computations). The instance of the domains `UPDATE USER` and `QUERY USER` (see figures 4 and 5) uses the information elicited at step 0 : each available operation for an actor becomes a command and, consequently an element of the domain `UPDATE USER` or `QUERY USER`. At the end of this step, for each use case, an instance of a problem frame is established.

Step 2 Now, to each problem frame instance, the corresponding formal specification should be associated. To this end, the method (proposed in [CR00] and developed in [CH03b]) that consists in associating to each element of the problem frame diagram a specification module is applied. The data types used should also be specified. We use here the Z specification language [Spi92] so as to take advantage of its built-in state notion.

At the end of this step, a Z specification, formally describing the problem frame instances (corresponding to each use case) is established. This specification describes precisely the different domains and the machine to be realised. The work achieved at this step benefits from the rigor brought by formal specifications, and from questions raised so as to specify the requested elements, thus bringing finer grain precisions on the requested realisation.

Step 3 Resulting from step, for each subproblem, the specification of a machine is given. Each machine corresponds to a use case, sharing the same database DBM, and is realised by a component. We now propose a method to integrate the components realised following their specification obtained at the preceding step. Figure 6 shows the component architecture that we propose to put together the different machines. This architecture is an instance of the “repository” architectural style (Figure 3) where the shared data are in *Database*, and the clients are the different *UseCase_JMgr* components.

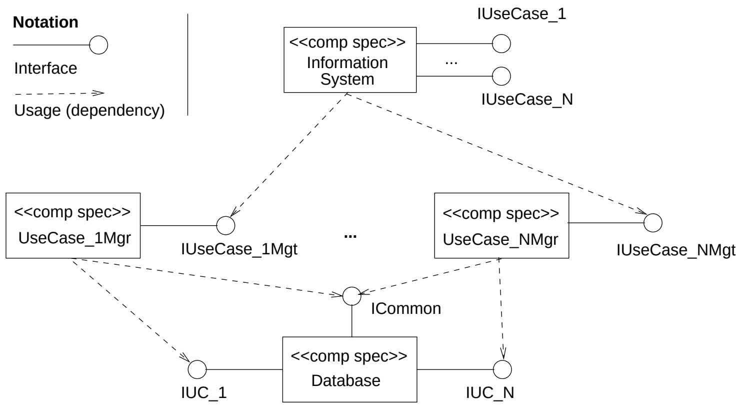

Figure 6: Component architecture for the global system

There is a component for each machine and for the global system, as well as for the database. The global system *Information System* provides an interface *IUseCase_J* for each use case *J* to its environment. Following a convention used by Cheesman et Daniels [CD01], the components that correspond to submachine problems are named *UseCase_JMgr*. In order to realise an interface *IUseCase_J*, it is necessary to use the component *UseCase_JMgr* via its interface *UseCase_JMgt*, as indicated by the arrows denoting dependencies.

All *UseCase_JMgr* components communicate with the database that puts an interface *IUC_J* at each *UseCase_JMgr* component's disposal. In order to avoid repetitions in the *IUC_J* interfaces, operations that are used by more than one *UseCase_JMgr* component are put in an *ICommon* interface.

Operations that constitute the interfaces *UseCase_JMgt* correspond to interfaces of the different *Update Machine*s and *Query Machine*s developed at Step 1. Operations of interfaces *IUseCase_J* correspond to commands available to the corresponding problem users. Interface operations *UseCase_JMgt* must provide the elements required for the realisation of the interfaces *IUseCase_J* operations. This condition checks that all subproblems solutions are enough to solve the problem started with.

In order to specify in more detail the different component communication, it is possible to use collaboration diagrams (as, for instance, in the Cheesman et Daniels approach [CD01]).

4 Case study: on line shop

An e-commerce system provides the following services:
- Clients may browse to find out whether the products they are looking for are on sale, to get information on these products, and to order them.
- Employees take care of orders management and restocking.
- The manager may ask for sales statistics, decide to stop selling some products or to start selling a new product.

4.1 Step 0 : use cases

The above description leads to identify the following different use cases (Figure 7) :

(i) for an actor who is a client of this shop ("Customer"), searching for information ("Browse"), and sending orders ("Send Order"),

(ii) for an actor who is an employee, order management ("Process order"), restocking ("Refill stock"),

(iii) for an actor who is the shop manager, taking decisions concerning the selling of products ("Take Product Decisions"), and request of reports ("Require Management Reports").

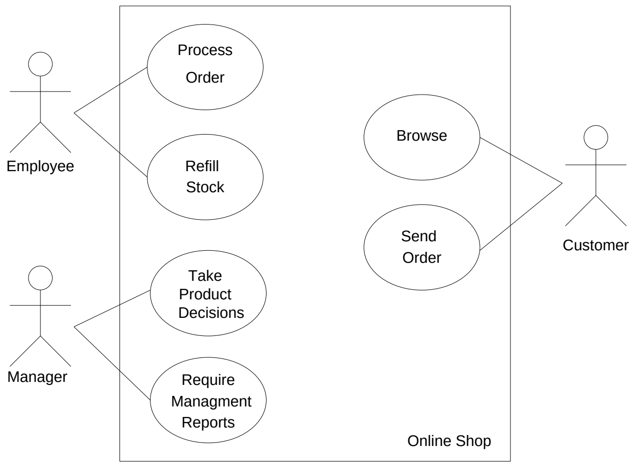

Figure 7: On-line shop use cases

The first two use cases will be dealt with in detail, and their possible scenarios expressed by sequence diagrams. A customer may (Figure 8) look for products according to given criteria `cr` or ask for the product properties `p`.

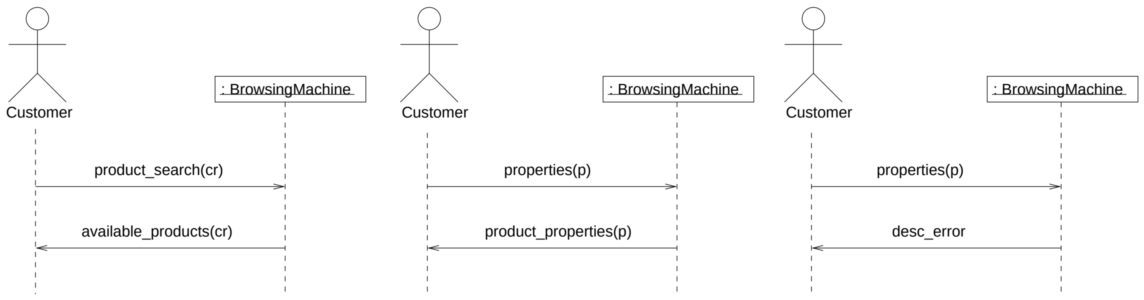

Figure 8: Scenarios for the *Browse* use case

A customer `c` send an order (Figure 9) for a given quantity `q` of a product `p`.

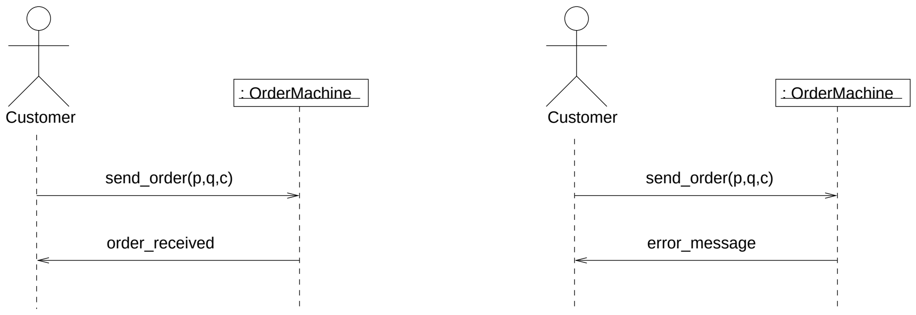

Figure 9: Scenarios for the *SendOrder* use case

4.2 Step 1 : problem frames instances

The different identified use cases yield

- update frame instances associated with the “Send Order”, “Process order”, “Refill stock”, and “Take Product Decisions” use cases
- query frame instances associated with the “Browse” and “Require Management Reports” use cases.

We provide below two examples that correspond to the “Browse” use case in Figure 10, and to the “Process order” use case in Figure 11.

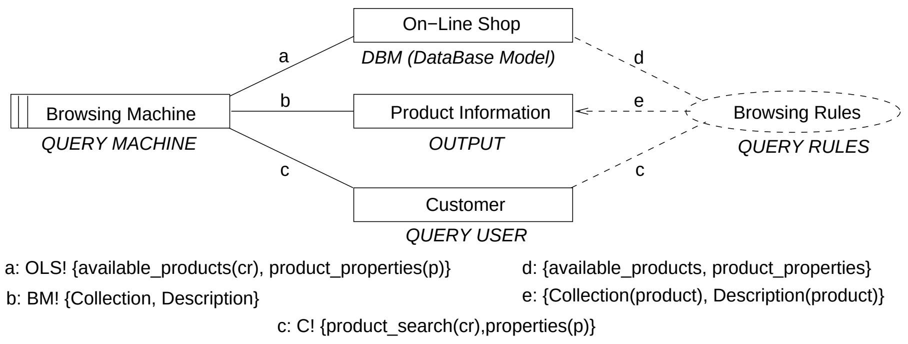
On−Line Shop
DBM (DataBase Model)
Browsing Machine
QUERY MACHINE
Product Information
OUTPUT
Customer
QUERY USER
Browsing Rules
QUERY RULES
a
b
c
d
e
c
a: OLS! {available_products(cr), product_properties(p)}
b: BM! {Collection, Description}
c: C! {product_search(cr),properties(p)}
d: {available_products, product_properties}
e: {Collection(product), Description(product)}

Figure 10: Query frame instance associated with the “Browse” use case

Figure 10 shows an instance of the problem frame proposed for queries (Figure 5), where the interfaces `c` are the customer `C!` queries for products according to some criteria `cr` or to product `p` characteristics, and interface `a` denotes information provided by the on-line shop `OLS!`. Figure

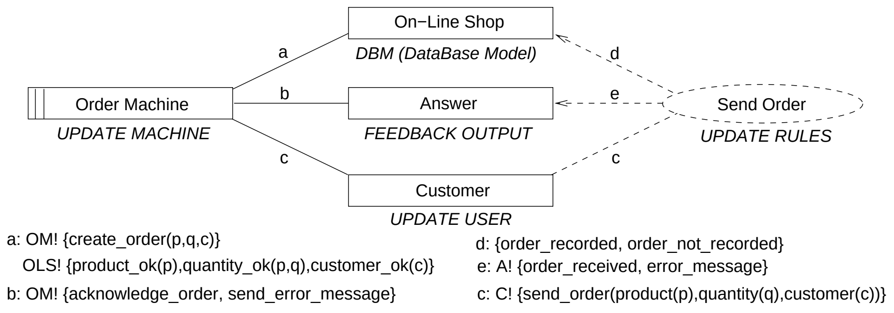
On−Line Shop
DBM (DataBase Model)
Order Machine
UPDATE MACHINE
Answer
FEEDBACK OUTPUT
Customer
UPDATE USER
Send Order
UPDATE RULES
a
b
c
d
e
c
a: OM! {create_order(p,q,c)}
OLS! {product_ok(p),quantity_ok(p,q),customer_ok(c)}
b: OM! {acknowledge_order, send_error_message}
d: {order_recorded, order_not_recorded}
e: A! {order_received, error_message}
c: C! {send_order(product(p),quantity(q),customer(c))}

Figure 11: Update frame instance associated with the “Send Order” use case

11 is an instance of the problem frame proposed for updates (Figure 4), where interfaces `c` are the customer `C!` order sendings, and interface `a` denotes the creation of an order by the machine `OM!` and checks done by the on-line shop `OLS!` on the product `product_ok(p)`, etc.

4.3 Etape 2 : Z specification

In the following, we formally specify the online shop in Z, that is, according to step 2 of our development method in Section 3.2, we specify the different domains (together with their operations) and the machine to be realized (here for the *Browse* and *SendOrder* use cases). We chose the Z language because the online shop has a state that is altered by customer orders and other actions.

4.3.1 Auxiliary definitions

The following (self-explanatory) basic types are needed:

$[ProductCode, CustomerCode, OrderCode, ProductDescription]$

For the Type *ProductDescription* we need an error element (cf. Figure 8).

$| \; desc\_error : ProductDescription$

The function *max_quantity* records what quantity of a given product may be ordered at one time. For example, our online shop would not be able to deliver 2 million copies of the book "Problem Frames" by Michael Jackson. The function is partial, because new products may occur on the market for which we cannot give a quantity yet.

$| \; max\_quantity : ProductCode \rightarrowtail\!\!\!\!\!\!\!+ \;\; \mathbb{N}$

Our online shop has to keep track of products, customers, and orders. For easy reference, each such item has a unique code. The uniqueness requirement is expressed in the state invariant of the schema *OnLineShop* given in Section 4.3.2.

Product

$$\begin{array}{l} pcode : ProductCode \\ quantity : \mathbb{N} \\ descr : ProductDescription \\ \hline descr \neq desc\_error \end{array}$$

Customer

$$\begin{array}{l} ccode : CustomerCode \end{array}$$

Order

$$\begin{array}{l} ocode : OrderCode \\ product : ProductCode \\ quantity : \mathbb{N} \\ cus : CustomerCode \end{array}$$

4.3.2 State Definition for the Domain *OnlineShop*

The state of the online shop is defined by the following schema, which has a slightly different name than the name of the domain. The invariant expresses to code unicity as well as the constraints defined by the *max_quantity* function for each product available, and the constraints that, for each order, there exists a registered customer and a registered product in the database.

OnLineShop

$$\begin{array}{l} products : \mathbb{P}\, Product \\ customers : \mathbb{P}\, Customer \\ orders : \mathbb{P}\, Order \\ \hline \forall p_1, p_2 : products \mid p_1 \neq p_2 \bullet p_1.pcode \neq p_2.pcode \\ \forall c_1, c_2 : customers \mid c_1 \neq c_2 \bullet c_1.ccode \neq c_2.ccode \\ \forall o_1, o_2 : orders \mid o_1 \neq o_2 \bullet o_1.ocode \neq o_2.ocode \\ \forall p : products \bullet p.pcode \in \mathrm{dom}\, max\_quantity \\ \forall o : orders \bullet (\exists c : customers \bullet o.cus = c.ccode) \wedge (\exists p : products \bullet o.product = p.pcode) \end{array}$$

4.3.3 *OnlineShop* domain basic operations

The basic operations correspond to the interface phenomena of the domain *OnlineShop* of the instantiated frame diagrams (Section 4.2).

The predicates *ProductOK*, *CustomerOK*, and *QuantityOK* (cf. interface a in Figure 11) below are intended to be used by other operations. Therefore, they allow the state to be changed (we use Δ*OnLineShop* instead of Ξ*OnLineShop*) but they do not specify how this state is changed.

$$\begin{array}{|l}\multicolumn{1}{l}{\textit{ProductOK}}\\ \hline \Delta \mathit{OnLineShop}\\ pc? : \mathit{ProductCode}\\ \hline \exists\, p : \mathit{products} \bullet\\ \qquad p.\mathit{pcode} = pc?\\ \hline\end{array}
\qquad
\begin{array}{|l}\multicolumn{1}{l}{\textit{CustomerOK}}\\ \hline \Delta \mathit{OnLineShop}\\ cc? : \mathit{CustomerCode}\\ \hline \exists\, c : \mathit{customers} \bullet\\ \qquad c.\mathit{ccode} = cc?\\ \hline\end{array}
\qquad
\begin{array}{|l}\multicolumn{1}{l}{\textit{QuantityOK}}\\ \hline \Delta \mathit{OnLineShop}\\ pc? : \mathit{ProductCode}\\ \mathit{quant}? : \mathbb{N}\\ \hline \mathit{ProductOK}\\ \mathit{quant}? \leq \mathit{max\_quantity}\; pc?\\ \hline\end{array}$$

The following operation belongs to the use case *SendOrder*. It is a partial operation that works only if the previously defined predicates are true.

$$\begin{array}{|l}\multicolumn{1}{l}{\textit{CreateOrder}}\\ \hline \Delta \mathit{OnLineShop}\\ pc? : \mathit{ProductCode}\\ \mathit{quant}? : \mathbb{N}\\ cc? : \mathit{CustomerCode}\\ \hline \mathit{ProductOK}\\ \mathit{CustomerOK}\\ \mathit{QuantityOK}\\ \exists\, o : \mathit{Order} \mid\ (\forall\, oo : \mathit{orders} \bullet oo.\mathit{ocode} \neq o.\mathit{ocode}) \wedge o.\mathit{product} = pc? \wedge\\ \qquad\qquad\quad o.\mathit{quantity} = \mathit{quant}? \wedge o.\mathit{cus} = cc?\\ \qquad \bullet\ \mathit{orders}' = \mathit{orders} \cup \{o\}\\ \mathit{products}' = \mathit{products}\\ \mathit{customers}' = \mathit{customers}\\ \hline\end{array}$$

4.3.4 Specifying the Domains *Answer* and *Customer*

The *Answer* domain contains just two elements.

$\mathit{Answer} ::= \mathit{order\_received} \mid \mathit{error\_message}$

Biddable domains such as customers are specified by the commands they may give.

$\mathit{OrderCustomer} ::= \mathit{order}\langle\langle \mathit{ProductCode} \times \mathbb{N} \times \mathit{CustomerCode} \rangle\rangle$

4.3.5 Specifying the domain *OrderMachine* and taking the *SendOrder* use case into account

The operations of Section 4.3.3 are used by the *OrderMachine* (cf. Figure 11) either to generate a new order or to give an error message, leaving the state of the online shop unchanged.

$$\begin{array}{|l}\multicolumn{1}{l}{\textit{SendSuccess}}\\ \hline \mathit{answ}! : \mathit{Answer}\\ \hline \mathit{answ}! = \mathit{order\_received}\\ \hline\end{array}
\qquad
\begin{array}{|l}\multicolumn{1}{l}{\textit{SendError}}\\ \hline \mathit{answ}! : \mathit{Answer}\\ \hline \mathit{answ}! = \mathit{error\_message}\\ \hline\end{array}$$

$$\begin{array}{l}\mathit{SendOrder} \mathrel{\widehat=} (\mathit{ProductOK} \wedge \mathit{CustomerOK} \wedge \mathit{QuantityOK} \wedge \mathit{CreateOrder} \wedge \mathit{SendSuccess})\\ \qquad\qquad\quad \vee\ (\Xi \mathit{OnLineShop} \wedge \mathit{SendError})\end{array}$$

4.3.6 Taking into Account the Use Case *Browse*

Taking into account another use case based on the same information system leads us to specify new operations on the domain *OnlineShop*.

A *BrowseCustomer* can either look for products fullfilling certain criteria[2], or s/he can inspect the description of a given product.

$$BrowseCustomer ::= product\_search\langle\langle \mathbb{P}\, Product\rangle\rangle \mid properties\langle\langle ProductCode\rangle\rangle$$

Accordingly, two new basic operations for querying the state of the system have to be added to the definition of the domain *OnlineShop*.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ProductSearch}} \\
\Xi OnLineShop \\
cr? : \mathbb{P}\, Product \\
available\_products! : \mathbb{P}\, ProductCode \\
\hline
available\_products! = \\
\qquad \{p : products \mid p \in cr? \bullet p.pcode\}
\end{array}$$

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ProductProperties}} \\
\Xi OnLineShop \\
pc? : ProductCode \\
info! : ProductDescription \\
\hline
\exists\, p : products \bullet p.pcode = pc? \land info! = p.descr
\end{array}$$

To specify the domain *BrowsingMachine*, we first define an error schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\textit{ProductNotPresent}} \\
\Xi OnLineShop \\
pc? : ProductCode \\
info! : ProductDescription \\
\hline
\forall\, p : products \bullet p.pcode \neq pc? \\
info! = desc\_error
\end{array}$$

The browsing machine interface consists in the operations *ProductSearch* (cf. above) and *Properties* defined as follows:

$$Properties \mathrel{\widehat{=}} ProductProperties \lor ProductNotPresent$$

4.4 Etape 3 : recomposition

Figure 12 shows the component architecture obtained for the on-line shop when building an instance of the architecture in figure 6.

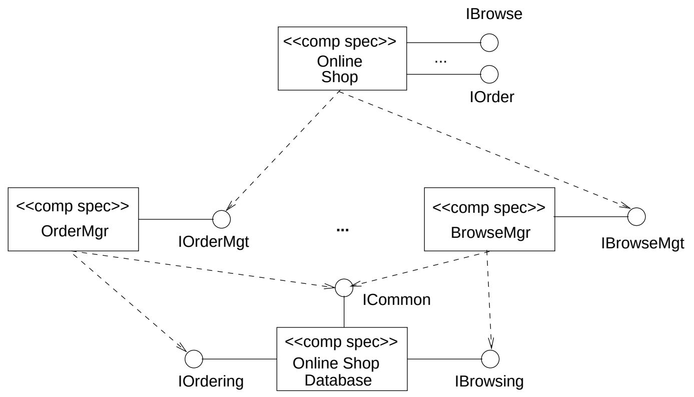

Figure 12: On-line shop component architecture

The component *OnlineShopDatabase* is specified by the schema *OnLineShop* given in paragraph 4.3.2 and by the operations defined in this schema, that is *ProductOK*, *CustomerOK*, *QuantityOK*,

[2]In Z, predicates are identified with their extension, i.e. the set of items for which the predicate is true. Hence, a criterion on products is identified with a set of products, which has type $\mathbb{P}\, Product$.

CreateOrder, *AvailableProducts*, *ProductProperties*.

The interface *ICommon* contains the operation *ProductOK*, because this operation is used by the two components *OrderMgr* and *BrowseMgr*.

The interface *IOrdering* contains the operations *CustomerOK*, *QuantityOK* and *CreateOrder*. The interface *IBrowsing* contains the operations *AvailableProducts* and *ProductProperties*.

The component *OrderMgr* (corresponding to *OrderMachine* in Figure 11) provides to the environment the operation *SendOrder*, that is defined in paragraph 4.3.5. This operation uses the operations of the interfaces *ICommon* et *IOrdering*.

The component *BrowseMgr* (corresponding to *BrowsingMachine* in Figure 10) provides to the environment the operations *ProductSearch* et *Properties*, that are defined in paragraph 4.3.6. These operations use the operations of the interfaces *ICommon* and *IBrowsing*.

The operations of the interfaces *IBrowse* and *IOrder* correspond to those of the interfaces *IBrowsing* and *IOrdering*. Realising an operation of the interfaces *IBrowse* et *IOrder* merely consists in a call of the operation bearing the same name in the interface *IBrowsing* or *IOrdering*.

5 Conclusions and perspectives

We propose here a method for the specification and the development of information systems. This method is based on several kinds of patterns. The contributions of our approach are the following:

- We provide criteria to identify the systems for which our method is applicable (section 3.2). To achieve this, we identify a new kind of domain, that is “business domain”. Our criteria are clear and easy to determine.
- We conceive two new problem frames that are well suited to information systems. These systems are not dealt with by the problem frames given by Jackson [Jac01]. Information systems are decomposed into subsystems that deal either with updates or with queries. This decomposition is done after use cases, which is new as regards Jackson’s work.
- Thus, our method establishes a link between object oriented analysis and problem frames, which was not shown up to now.
- The method described here integrates the method described in [CR00, CH03a] where other formal specification languages were used (CASL [BM04], CASL-LTL [RAC03], and LOTOS). Here this new experience shows that this method applies as well to another language, Z.[3] Each part of the problem frame instance is formally specified. The method described in [CH03a] provides the validation conditions for the produced specification.
- Each identified subproblem is solved quite independently from the others (taking into account the fact that the database is shared by all subproblems). This leads us to the question of the solutions recomposition. In order to solve this problem we propose a component architecture that is an instance of the repository architectural style.
- Altogether, our method establishes links between object oriented analysis, problem frames, formal specifications, architectural styles, and the component based approach. Up to now these different techniques were used in a more or less isolated manner.
- Our approach provides a substantial methodological guide that leads to develop related documents.

In the future, we intend to specify in more detail the communication between the different architecture components, for instance using communication patterns.

[3]Thus, our subject here is not to promote the use of a given language for some kinds of applications.

Moreover, such a method aiming at a guided and integrated use of several techniques and several patterns, while presented here for information systems development, should be widely applicable when defined for other kinds of problems characterised by other problem frames.

References

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, 1998.

[BM04] Michel Bidoit and Peter D. Mosses. *CASL, The Common Algebraic Specification Language - User Manual*. Number 2900 in Lecture Notes in Computer Science, Tutorial. Springer-Verlag, 2004.

[CD01] John Cheesman and John Daniels. *UML Components – A Simple Process for Specifying Component-Based Software*. Addison-Wesley, 2001.

[CH03a] Christine Choppy and Maritta Heisel. Systematic transition from problems to architectural designs. Technical Report LIPN-2003-05, Université Paris XIII, France, 2003. 34 pages.

[CH03b] Christine Choppy and Maritta Heisel. Use of patterns in formal development: Systematic transition from problems to architectural designs. In M. Wirsing, R. Hennicker, and D. Pattinson, editors, *Recent Trends in Algebraic Development Techniques, 16th WADT, Selected Papers*, LNCS 2755, pages 205–220. Springer Verlag, 2003.

[CR00] Christine Choppy and Gianna Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert, C. Choppy, and P. D. Mosses, editors, *Recent Trends in Algebraic Development Techniques, 14th WADT, Selected Papers*, LNCS 1827, pages 104–123. Springer Verlag, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.

[GS93] David Garlan and Mary Shaw. An introduction to software architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, volume 1. World Scientific Publishing Company, 1993.

[Jac01] Michael Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[JCJO92] I. Jacobson, M. Christerson, P. Jonnson, and G. Overgaard. *Object-Oriented Software Engineering: A Use-Case Driven Approach*. Addison-Wesley, 1992.

[Mic02] Microsoft Corporation. *COM*$^+$, 2002. http://www.microsoft.com/com/tech/COMPlus.asp.

[Obj02] The Object Management Group (OMG). *Corba Component Model, v3.0*, 2002. http://omg.org/technology/documents/formal/components.htm.

[RAC03] Gianna Reggio, Egidio Astesiano, and Christine Choppy. CASL-LTL: A CASL extension for dynamic reactive systems – version 1.0 – summary. Technical Report DISI-TR-03-36, Università di Genova, Italy, 2003.

[Spi92] J. M. Spivey. *The Z Notation – A Reference Manual*. Prentice Hall, 2nd edition, 1992.

[Sun97] Sun Microsystems. *JavaBeans Specification, Version 1.01*, 1997. http://java.sun.com/products/javabeans/docs/spec.html.

[Sun01] Sun Microsystems. *Enterprise JavaBeans Specification, Version 2.0*, 2001. http://java.sun.com/products/ejb/docs.html.

[Szy99] Clemens Szyperski. *Component Software - Beyond object oriented programming*. Addison Wesley, 1999.

[UML] UML Revision Task Force. *OMG UML Specification*. `http://www.uml.org`.