Enterprise Applications: from Requirements to Design

Christine Choppy, Denis Hatebur, Maritta Heisel and Gianna Reggio

1 Introduction

We provide a method to systematically develop enterprise application architectures from problem descriptions. The problem descriptions are based on Jackson's problem frame approach (Jackson, 2001). For enterprise applications, we have developed a specialized problem frame that takes the specifics of such applications into account (Choppy & Reggio, 2006). In particular, new domain types – such as *business worker* and *business object* – are introduced. We describe the requirements through *problem diagrams* that are instances of the enterprise application frame and of other problem frames.

In addition to Jackson's problem frame approach, we represent the business model underlying the business application by a *domain knowledge diagram*. Such a diagram identifies the domains relevant for the business process to be automated and states how they are related. It serves to analyze the business process to be automated and helps to construct the appropriate instances of the enterprise application frame, thus obtaining the problem diagrams.

From the problem diagrams, we derive two kinds of specifications: a *behavioral specification* describes how the automated business process is carried out. It can be expressed using activity or sequence diagrams. A *structural specification* describes the classes to be implemented and the operations they provide. This specification is expressed as a class diagram, where all operations are specified in OCL.

With these different models, we have described the software development problem in a detailed way, taking into account the specifics of business applications. This makes it possible to develop a suitable software architecture in a systematic way. We proceed similarly as described in earlier work (Choppy, Hatebur, & Heisel, 2011), where we derive software architectures from problem descriptions for arbitrary software. In this work, we make use of the fact that the software developement problem

C. Choppy
LIPN, University Paris 13, e-mail: Christine.Choppy@lipn.univ-paris13.fr

D. Hatebur
University Duisburg-Essen e-mail: Denis.Hatebur@uni-duisburg-essen.de

M. Heisel
University Duisburg-Essen e-mail: Maritta.Heisel@uni-duisburg-essen.de

G. Reggio
Universita di Genova e-mail: Gianna.Reggio@disi.unige.it

is decomposed in subproblems that fit to the enterprise application frame. Taking into account the identified business objects, we focus on how these objects can best be stored and accessed. In this context, questions of distributing the software to be developed and the information to be stored are addressed, too.

In a first step, we create an *initial architecture*. To obtain that architecture, we first have to decide on the responsibilities of the software to be developed (which is called *machine* in the problem frames approach). We have to inspect all domains occurring in the problem diagrams and decide if they will be part of the machine or not. Some domains may reside in the environment but still need to have an internal representation in the machine. The rules given in (Choppy & Reggio, 2006) support this task. Moreover, the initial architecture reflects the problem decomposition that was obtained by applying the problem frame approach. Each subproblem machine becomes a component in the initial architecture.

The initial architecture need not be implementable, because the interaction between the different components has not yet been taken into account. Therefore, we transform the initial architecture into an *implementable architecture*. To create the implementable architecture we have to consider technical requirements, for example that some functionality should be implemented on another computer. In business applications, there is usually a database, which may be located on a different computer than the machine we are building. In that case, we have to split the problem diagrams and create corresponding subproblem diagrams that separate the different machines accordingly. Conversely, in many cases, only one database is used to store different kinds of information, such that the components representing the different domains have to be merged into the database component. Moreover, we introduce coordinator components, considering the formal descriptions of the business model, and facade components. Finally, we allocate all machines that solve the different problems or subproblems and the considered domains to physical components of the machine to be built.

The implementable architecture that is obtained in this way does not follow a particular architectural style. If, for example, a layered architecture is wished for, the implementable architecture can further be transformed into a layered one, as is described in (Choppy et al., 2011).

All the diagrams that we present in this chapter are expressed in UML instead of the original notation used by Jackson (Jackson, 2001). This makes it possible to exploit the additional expressive power provided by UML class diagrams and also to represent software architectures and problem descriptions in the same notational framework. Carrying over the problem-frame approach to UML is achieved by defining a UML profile for problem frames (Hatebur & Heisel, 2010). That profile forms the basis for a tool called *UML4PF*, which is currently under development at the University of Duisburg-Essen. UML4PF[1] supports requirements analysis using problem frames and deriving software architectures from problem descriptions. It is based on the Eclipse development environment, extended by an EMF-based UML tool. Using UML4PF, users not only can set up diagrams in UML notation

[1] Available at http://uml4pf.org

that are translations of the original problem frame notation. Most notably, UML4PF can be used to check semantic validation conditions expressed in OCL. Such conditions concern on the one hand the semantic integrity of single models. One the other hand, also the coherence between different models can be checked.

The work presented here builds on and enhances previous work. The method for deriving architectures from problem descriptions (Choppy et al., 2011)is now tailored and elaborated, taking the specifics of developing enterprise architectures into account and integrating the business frame presented in (Choppy & Reggio, 2006).

The rest of the chapter is organized as follows. In Section 2, we introduce the basic concepts and notations we use in our work. Section 3 is devoted to the requirements analysis phase that we carry out using the enterprise application frame. As an example, we consider an online shop. Section 4 describes how we derive the architecture of the business application from the problem descriptions set up in the requirements analysis phase. Related work is discussed in Section 5, and we conclude in Section 6.

2 Background, Concepts, and Notations

In this section, we introduce the basic concepts we use in this work, problem frames that we represent using UML tools, and various diagrams (context diagrams, problem diagrams, domain knowledge diagrams, and composite diagrams), and we express properties using OCL constraints.

2.1 Context Diagrams

The different diagram types make use of the same basic notational elements. As a result, it is necessary to explicitly state the type of diagram by appropriate stereotypes. In our case, the stereotypes are $<<ContextDiagram>>$, $<<ProblemDiagram>>$, $<<ProblemFrame>>$, and $<<TechnicalContextDiagram>>$. These stereotypes extend (some of them indirectly) the meta-class *Package* in the UML meta-model.

According to the UML superstructure specification ("UML Revision Task Force", 2009), it is not possible that one UML element is part of several packages. For example a class *Client* should be in the context diagram package and also in some problem diagrams packages.[2] Nevertheless, several UML tools allow one to put the same UML element into several packages within graphical representations. We want to make use of this information from graphical representations and add it to the model (using stereotypes of the profile). Thus, we have to relate the elements inside a package explicitly to the package. This can be achieved with a dependency

[2] Alternatively, we could create several Client classes, but these would have to have different names.

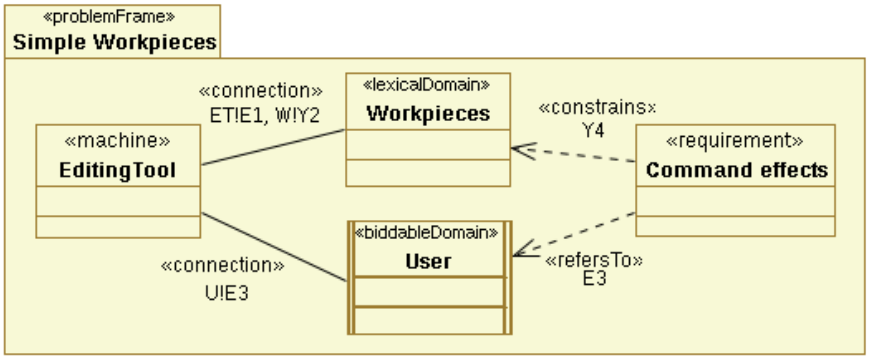

Fig. 1 *Simple Workpieces* problem frame

stereotype $<<isPart>>$ from the package to all included elements (e.g., classes, interfaces, comments, dependencies, associations).

The ***context diagram*** (see e.g., Figure 4) contains the machine domain(s), the relevant domains in the environment, and the interfaces between them. Domains are represented by classes with the stereotype $<<Domain>>$, and the machine is marked by the stereotype $<<Machine>>$. Instead of $<<Domain>>$, more specific stereotypes such as $<<BiddableDomain>>$, $<<LexicalDomain>>$ or $<<CausalDomain>>$ can be used. Since some of the domain types are not disjoint, more than one stereotype can be applied on one class.

2.2 Problem Frames

Problem frames are a means to describe software development problems. They were proposed by Michael Jackson (Jackson, 2001), who describes them as follows: *"A problem frame is a kind of pattern. It defines an intuitively identifiable problem class in terms of its context and the characteristics of its domains, interfaces and requirement."* Problem frames are described by *frame diagrams*, which basically consist of rectangles, a dashed oval, and different links between them, see Figure 1). The task is to construct a *machine* that establishes the desired behaviour of the environment (in which it is integrated) in accordance with the requirements.

Rectangles with a domain stereotype denote *domains* that already exist in the application environment. Jackson (Jackson, 2001, p. 83f) considers three main domain types:

- "*A **biddable domain** usually consists of people. The most important characteristic of a biddable domain is that it is physical but lacks positive predictable internal causality. That is, in most situations it is impossible to compel a person to initiate an event: the most that can be done is to issue instructions to be followed.*"

Biddable domains are indicated by *B* (e.g., *User* in Figure 1).

- "*A **causal domain** is one whose properties include predictable causal relationships among its causal phenomena.*"
  Often, causal domains are mechanical or electrical equipment. They are indicated with a *C* in frame diagrams. Their actions and reactions are predictable. Thus, they can be controlled by other domains.
- "*A **lexical domain** is a physical representation of data – that is, of symbolic phenomena. It combines causal and symbolic phenomena in a special way. The causal properties allow the data to be written and read.*"
  Lexical domains are indicated by *X*(e.g., *Workpieces* in Figure 1).

A rectangle with a double vertical stripe denotes the machine to be developed, and *requirements* are denoted with a dashed oval. The connecting lines between domains represent interfaces that consist of *shared phenomena*. Shared phenomena may be events, operation calls, messages, and the like. They are observable by at least two domains, but controlled by only one domain, as indicated by an exclamation mark. For example, in Figure 1 the notation *U!E3* means that the phenomena in the set *U!E3* are *controlled* by the domain *User* and *observed* by the *EditingTool*.

To describe the problem context, a connection domain between two other domains may be necessary. Connection domains establish a connection between other domains by means of technical devices. Connection domains are, e.g., video cameras, sensors, or networks.

A dashed line represents a requirement reference, and an arrow indicates that the requirement *constrains* a domain.[3] If a domain is constrained by the requirement, we must develop a machine, which controls this domain accordingly. In Figure 1, the *Workpieces* domain is constrained, because the *EditingTool* changes it on behalf of user commands to satisfy the required *Command effects*.

Jackson (Jackson, 2001) introduces five basic problem frames (*Transformation*, *Simple Workpieces*, *Information Display*, *Commanded Behaviour* and *Required Behaviour*) that can be combined and/or adapted to fit the problem studied. Research on problem frames led to define more complex problem frames corresponding to large classes of applications (e.g. geographic problem frames (Nelson, Cowan, & Alencar, 2001) for geographic information systems). In previous work, we presented a problem frame for enterprise/business applications (Choppy & Reggio, 2006), and here, we will show how we can use it to develop the system architecture.

Software development with problem frames proceeds as follows: first, the environment in which the machine will operate is represented by a *context diagram*. Like a frame diagram, a context diagram consists of domains and interfaces. However, a context diagram contains no requirements. Then, the problem is decomposed into subproblems. Whenever possible, the decomposition is done in such a way that the subproblems fit to given problem frames. To fit a subproblem to a problem frame,

[3] In the following, since we use UML tools to draw problem frame diagrams, all requirement references will be represented by dashed lines with arrows and stereotypes <<*refersTo*>>, or <<*constrains*>> when it is constraining reference.

one must instantiate its frame diagram, i.e., provide instances for its domains, interfaces, and requirement. The instantiated frame diagram is called a *problem diagram*.

Besides problem frames, there are other elaborate methods to perform requirements engineering, such as i* (Yu, 1997), Tropos (Bresciani, Perini, Giorgini, Giunchiglia, & Mylopoulos, 2004), and KAOS (Bertrand, Darimont, Delor, Massonet, & Lamsweerde, 1998). These methods are goal-oriented. Each requirement is elaborated by setting up a goal structure. Such a goal structure refines the goal into subgoals and assigns responsibilities to actors for achieving the goal. We have chosen problem frames and not one of the goal-oriented requirements engineering methods to derive architectures, because the elements of problem frames, namely domains, may be mapped to components of an architecture in a fairly straightforward way.

2.3 Problem Diagrams

In a ***problem diagram*** (see e.g., Figure 7), the knowledge about a sub-problem described by a set of requirements is represented. A problem diagram consists of sub-machines of the machines given in the context diagram, the relevant domains, the connections between these domains and a requirement (possibly composed of several related requirements), as well as of the relation between the requirement and the involved domains. A requirement refers to some domains and constrains at least one domain. This is expressed using the stereotypes $<<refersTo>>$ and $<<constrains>>$. They extend the UML meta-class *Dependency*. Domain knowledge and requirements are special statements. Furthermore, any domain knowledge is either a fact (e.g., physical law) or an assumption (usually about a user's behaviour).

2.4 Domain Knowledge Diagrams

The problem frame approach substantially supports developers in analyzing problems to be solved. It points out what domains have to be considered, and what knowledge must be described and reasoned about when analyzing a problem in depth. Developers must elicit, examine, and describe the relevant properties of each domain. These descriptions form the domain knowledge, which is represented by domain knowledge diagrams. Domain knowledge consists of assumptions and facts. Assumptions usually describe required user behavior, whereas facts describe properties of the problem environment, regardless of how the machine is built. To express mandatory behaviour of domains in the environment we have introduced ***domain knowledge diagrams*** (see e.g., Figure 3).

2.5 Composite Diagrams

Composite structure diagrams ("UML Revision Task Force", 2009) are a means to describe architectures (see e.g., Fig. 14). They contain named rectangles, called *parts*. Theses parts are components of the software. In an object-oriented implementation components are instantiated classes. Each component may contain other (sub-) components. Atomic components can be described by state machines and operations for accessing internal data. In our architectures, components for data storage are only included if the data is stored persistently. Otherwise they are assumed to be part of some other component. Parts may have *ports*, denoted by small rectangles. Ports may have interfaces associated to them. Provided interfaces are denoted using the "lollipop" notation, and required interfaces using the "socket" notation. Figure 2 shows how interfaces in problem diagrams are transformed into interfaces in composite structure diagrams. The partial problem diagram shown on the left-

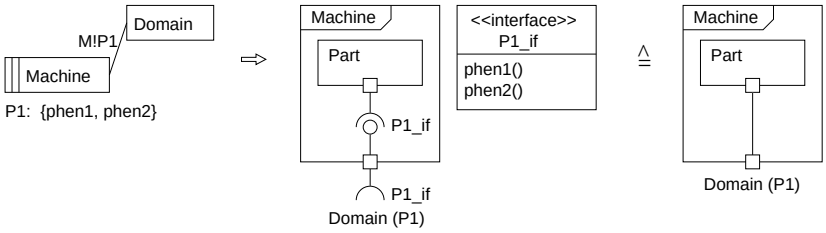

Fig. 2 Notation for Architectures

hand side of Figure 2 states that the phenomena *phen1* and *phen2* shared between the machine and a domain are controlled by the machine. In the composite structure diagram (with associated interface class) shown in the middle of Figure 2, this is expressed by a required interface *P1_if* of the *part* component of the machine, which is the same as for the whole machine. Shared phenomena controlled by a domain correspond to provided instead of required interfaces of the *part* and the machine, respectively. Because of this direct correspondence, we do not use the socket and lollipop notation in the following, but use connectors between ports as shown on the right-hand side of Figure 2. These connectors can be implemented e.g. as data streams, function calls, asynchronous messages or hardware access.

The architecture of software is multi-faceted: there exists a structural view, a process-oriented view, a function-oriented view, an object-oriented view with classes and relations, and a data flow view on a given software architecture. We use the structural view from UML 2.0 that describes the structure of the software at runtime. After that structure is fixed the interfaces need to be refined using sockets, lollipops and interface classes to describe the possible data flow. Then the corresponding active or passive class with its data and operations can be added for each component. Thereby the process-oriented and object-oriented views can be inte-

grated seamlessly into the structural view. That approach and the corresponding process are described in (Heisel & Hatebur, 2005).

2.6 OCL

The *Object Constraint Language* (OCL) (UML Revision Task Force, 2010; Warmer & Kleppe, 2003) is part of UML ("UML Revision Task Force", 2009). It is a notation to describe constraints on object-oriented modeling artifacts such as class diagrams and sequence diagrams. A constraint is a restriction on one or more elements of an object-oriented model.

OCL constraints are denoted in the UML model they belong to or in a separate document. We only use the constraint type *class invariant*, which is a constraint that must hold before and after execution of a method, but can be violated during method execution. The basic format of an OCL class invariant is as follows:

context *identifier* **inv** : *boolean expression*

where **context** is a keyword to mark the relative model element indicated by *identifier* from which other model elements can be referenced. The keyword **self** can be used within *boolean expression* to access the **context**. *identifier* is a class, attribute name, association name, operation name, or the like. The keyword **inv** describes that this constraint is an invariant and *boolean expression* is some boolean expression, often an equation.

As types we mostly use classifiers from the UML model the context refers to. We commonly use navigation paths (aka association ends or role names). Often associations are one-to-many or many-to-many, which means that constraints on a collection of objects are necessary. OCL expressions either state a fact about all objects in the collection using quantification or facts about the collection itself.

3 Enterprise Business Modelling: an approach à la problem frame

In this section we introduce the Business Frame for enterprises and the associated (UML) Business Model, illustrating both with an application to the Electronic Commerce case study.

All in all, we perform the following steps in the requirements analysis phase of business application systems:

1. Set up an enterprise business diagram (domain knowledge diagram).
2. Set up a context diagram.
3. Instantiate Enterprise Application Frame.
4. Derive behavioral specifications.

5. Derive structural specifications.
6. Set up software lifecycle.

We discuss these steps in the following sections.

3.1 Enterprise Business Diagram (Domain Knowledge Diagram)

Before starting the development of an enterprise application it is important to accurately understand and model the business in which the application will operate. This activity is called *business modelling* and will result in a so-called *business model*.

We assume that the business to be modelled consists of various interacting entities (called *business entities*) that achieve the various business-specific goals by means of dedicated cooperations called *business processes*.

We introduce a frame in the style of Jackson (cf. Sect. 2.2) to help to get abstractly the structure of the business, before modelling it in UML. That frame is shown in Fig. 5. Technically, the *business frame* is a frame with a domain for each business entity, where the business processes are described in terms of composite phenomena.

The business frame domains (examples can be found in Fig. 3), that are the business entities, can be classified as:

- *business objects*, the entities which are subject to the business (e.g., a catalogue, an insurance contract document), and are lexical domains. Business objects are passive and cannot cause any events.
- *business workers*, the human entities acting in the business (e.g., a manager, a clerk), and are biddable domains;
- *systems*, software systems or mechanical apparatus with a role in the business (e.g., Payment System for handling the payments or a scanner for paper documents). The systems are causal domains.

The business entities may be further classified in internal and *external* w.r.t. the business; the *internal* ones are those under the responsibility of the enterprise managing the business. The rule for deciding whether an entity is internal or external is the following: in case of a reorganization of the business by the managing enterprise, an internal entity may be modified, whereas an external one may not; thus Payment System is external, whereas a stock or a clerk are internal (indeed the enterprise may change the stock organization or the way the clerk works, but not the way the Payment System performs). This is displayed using the corresponding stereotypes (e.g. $<<businessObject>>$, $<<businessWorker>>$, $<<externalSystem>>$, etc. in Fig. 3). The external entities are marked with the corresponding stereotypes, and their box is left uncoloured (e.g., in Fig. 3 Payment System is an external system), while for simplicity the stereotypes for the internal ones do not include ”internal’ but their box is coloured.

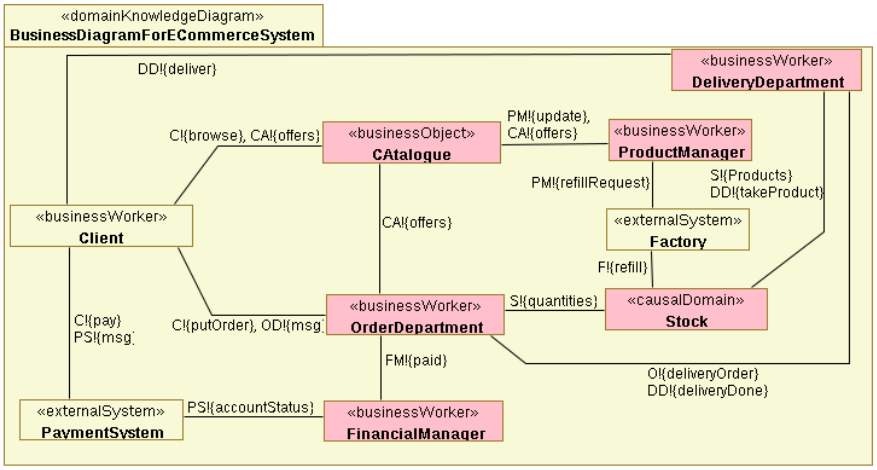

Fig. 3 Electronic Commerce Domain Knowledge Diagram

The classification in internal and external entities is completely orthogonal to the classification in business objects, workers and systems. All these domains are considered as given, and this marking is implicitly assumed.

In a Business Frame the domains are connected by composite phenomena that correspond to the *business processes*. The domains connected in a business process are called the *participants* of the process. We consider a business process as a cooperation between business workers, business objects and systems (both internal and external).

Electronic Commerce: Domain Knowledge Diagram

We illustrate our approach on the Electronic Commerce example from (Choppy & Reggio, 2006, 2005), that is a system for selling products via the internet. We consider the following requirements for this system:

R1 A Client can browse the offers in the Catalogue and check the availability.
R2 When a Client puts an order, if the product is available in the stock and the debit request is granted, then the product is taken from the stock (reducing the quantity) and delivered to the client; otherwise, an error message is returned.
R3 The ProductManager can update the Catalogue (add, remove, change entries).

Fig. 3 shows the Domain Knowledge Diagram for our Electronic Commerce example. This frame exhibits internal business workers (e.g., Manager), internal business objects (e.g., CAtalogue[4]), and external systems (e.g., Factory and Payment System).

[4] CAtalogue controls phenomena CA!{offers}, hence its name with a capital A.

The activity diagram in Fig. 9 shows the behavior of the business workers for the selling process. For the description of the activity before any machine is developed, we have no *StockRepresentation* but the *Stock* itself and the activity *UpdateStock-Representation* is not necessary.

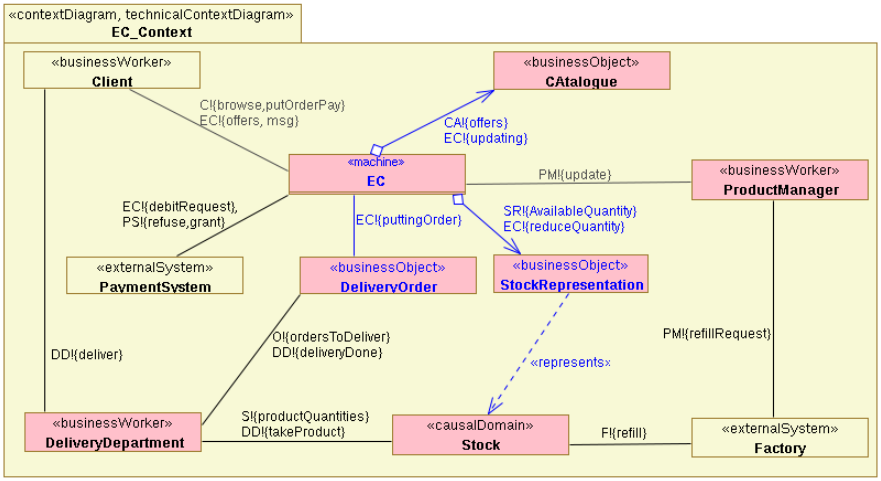

Fig. 4 Electronic Commerce context diagram

3.2 Context Diagram

Given the understanding provided in the domain knowledge diagram, the context diagram is then established by introducing the enterprise application to be developed represented by a machine (in Fig. 4, machine EC) and its corresponding interfaces to handle the requirements. Note that now the machine *EC* is responsible to handle the client order, which was previously under the responsibility of the *OrderDepartment*.

3.3 Enterprise Application problem frame

The Enterprise Application problem frame in Fig. 5 displays the various kinds of domains to be taken into account (<<*businessObject*>>, <<*businessWorker*>>, <<*externalSystem*>>) in an enterprise application. These domains are interfaced with the requirements through a constraining reference and the relevant phenomena. They are also interfaced with the *EnterpriseApplication* (that is the <<*machine*>>

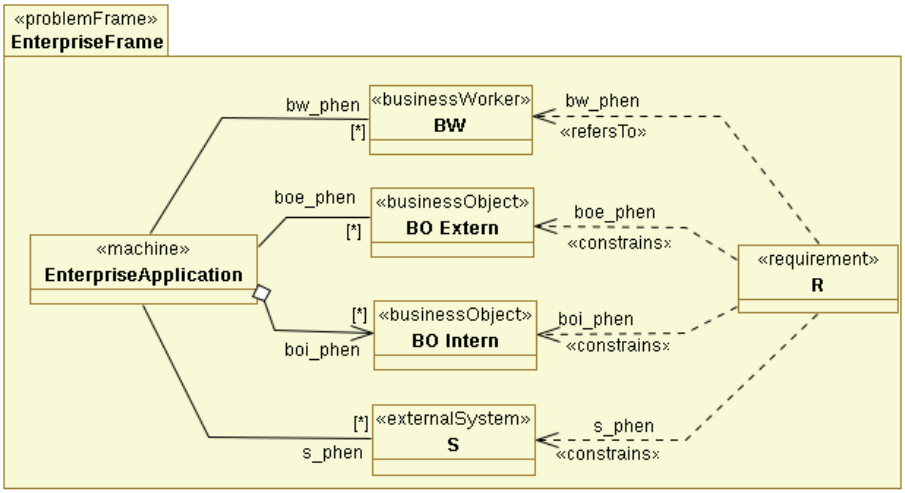

Fig. 5 Enterprise Application Problem Frame (EA Frame)

to be developed), and the relevant multiplicity applies. Notice that internal business objects are part of the enterprise application.

In the following, the Enterprise Application problem frame will be instantiated into problem diagrams describing the various subproblems.

Electronic Commerce: Initial subproblem description and problem diagrams

At this stage, the initial subproblems are identified and described using problem diagrams (instances of the Enterprise Application problem frame).

In our Electronic Commerce case study, the initial subproblems are EC_browse (Fig. 6), EC_sell (Fig. 7), and EC_upCat (update Catalogue) (Fig. 8).

Problem EC_browse (cf. Fig. 6) refers to a causal *Stock* domain with which the machine has no interface. Our approach is to introduce a lexical domain that is a representation of the causal domain with an interface to the machine. We add an assumption (or requirement) stating that the state of the lexical domain corresponds to the state of the causal domain. The rationale is that our business software is not embedded – therefore we do not introduce an interface between the machine and a causal domain being not a computer. We also add new representation domains to the context diagram, and a stereotype for dependency: $<<represents>>$.

The lower part of Fig. 6 contains *mapping diagrams* that show how the problem diagram is related to the context diagram. Such mapping diagrams are needed to automatically check the coherence between the different diagrams. The machine

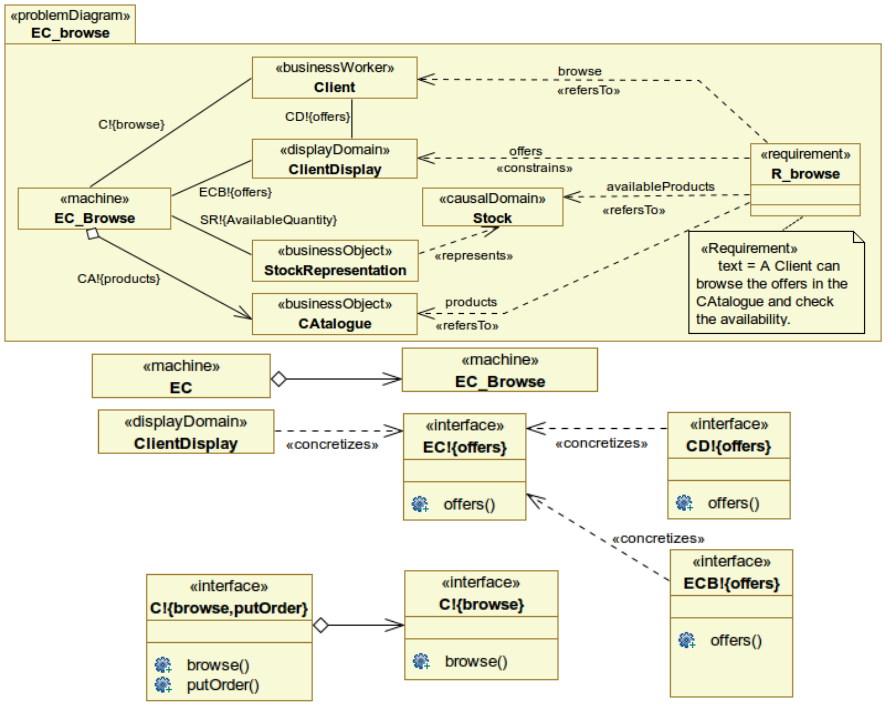

Fig. 6 Problem Diagram for EC_browse

EC_browse is a part of the machine *EC*, and the *ClientDisplay* is a display domain used to concretize interface between the *Client* and the machine. Furthermore, the suproblem *EC_browse* uses only a part of the interface between the *Client* and the machine.

For problem *EC_sell* (cf. Fig. 7), let us note that we again – as in EC_browse – distinguish between the stock, that is a causal domain, and the stock representation that is an internal business object (with the dependency relation $<<represents>>$). We also show shared phenomena between problem domains, such as *deliver* between the *DeliveryDepartment* and the *Client*.

The problem diagram for EC_upCat is an instance of the simple workpieces problem frame (Jackson, 2001). The *ProductManager* should be able to add, remove, or change products in the *CAtalogue*. Therefore, the *CAtalogue* is constrained and the requirement refers to the *ProductManager*.

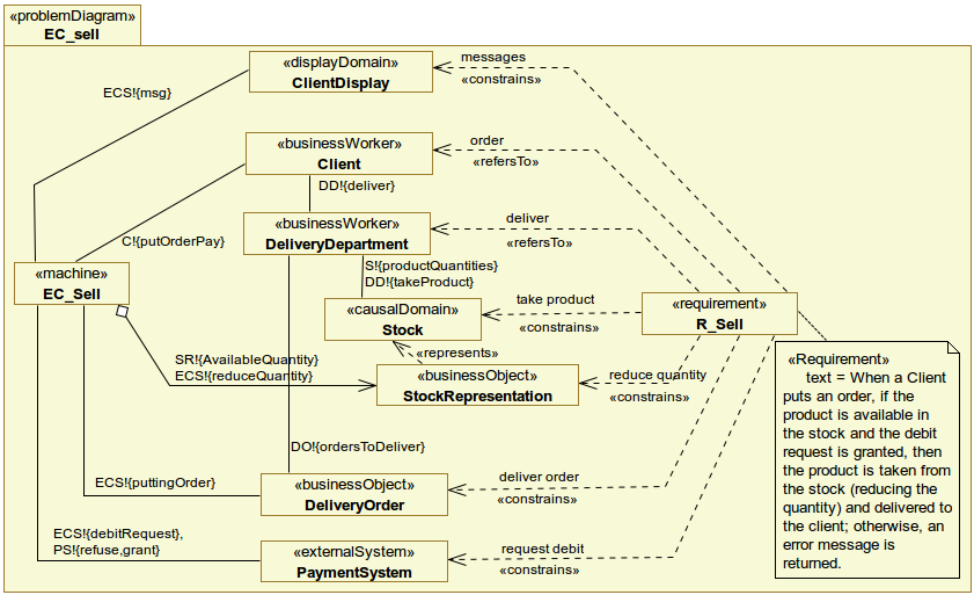

Fig. 7 Problem Diagram for EC_sell

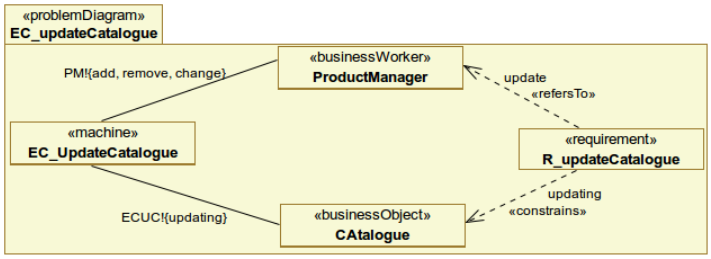

Fig. 8 Problem Diagram for EC_upCat

3.4 Derive Behaviour Specifications

Until now, we only have considered *requirements*. According to the problem frame approach, requirements refer to problem domains. Descriptions describing the behaviour of the machine are called *specifications*. Specifications are implementable requirements. They are derived from requirements by using domain knowledge. Specifications treat the machine as a black box and describe how the machine reacts to external stimuli and how it behaves in order to achieve the requirements. For more details, see (Jackson & Zave, 1995).

The behaviour of the machine can be specified using either activity diagrams or sequence diagrams. Here we present an activity diagram for problem EC_sell in Fig. 9. Swimlanes are used to structure the diagram with respect to the various

actors. As already stated in Section 3.2, the machine *EC_Sell* takes the roles of the *OrderDepartment* and *FinancialManager*. Therefore, the swimlanes of the activity diagram given in Fig. 9 annotated with *OrderDepartment* and *FinancialManager* represent the behaviour of the machine in interacting with its environment.

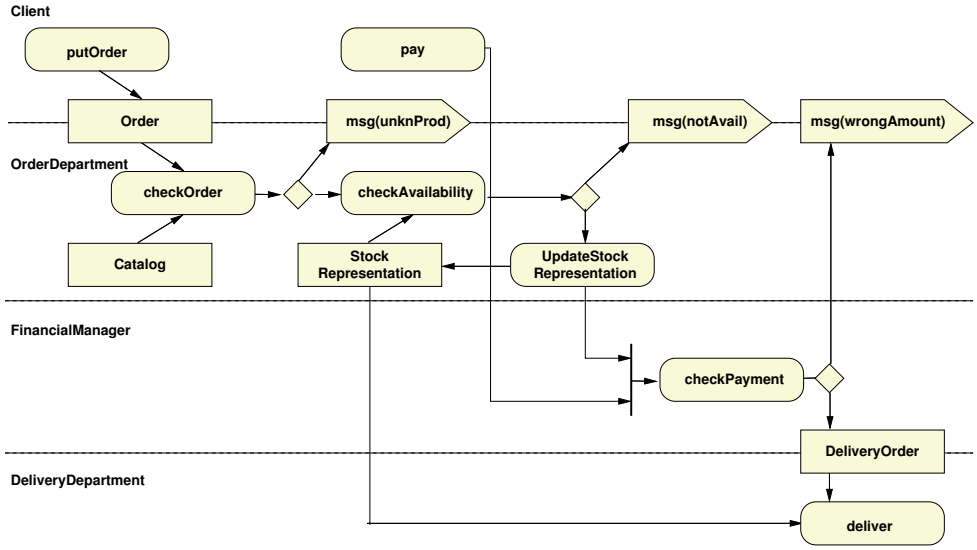

Fig. 9 Activity diagram

3.5 Structural Specification / Business class model and formal requirements

The next step is to set up the business class model together with OCL contracts that will be a formal expression of the requirements (pre and post) with the initial class diagram. We need to specify the operation provided by the machines in the sub-problems. For our example, the class diagram associated with subproblem EC_sell is given in Fig. 10. The classes represent the data to be implemented by the machine in order to be able to exhibit the behavior specified in the behavioural specification.

The OCL contract for the operation *putOrderPay* of the machine *EC_Sell* is given below. The precondition states that all input data are valid. The postcondition says that the (private) operations *putOrder* and *pay* are called.

```
1 putOrderPay(product: String [*], clientData: ClientData, accountData:
      String)
2 PRE:
3         Product.allInstances().id->includesAll(product) and
4         clientData.name<>'' and clientData.address<>''
5 POST:
```

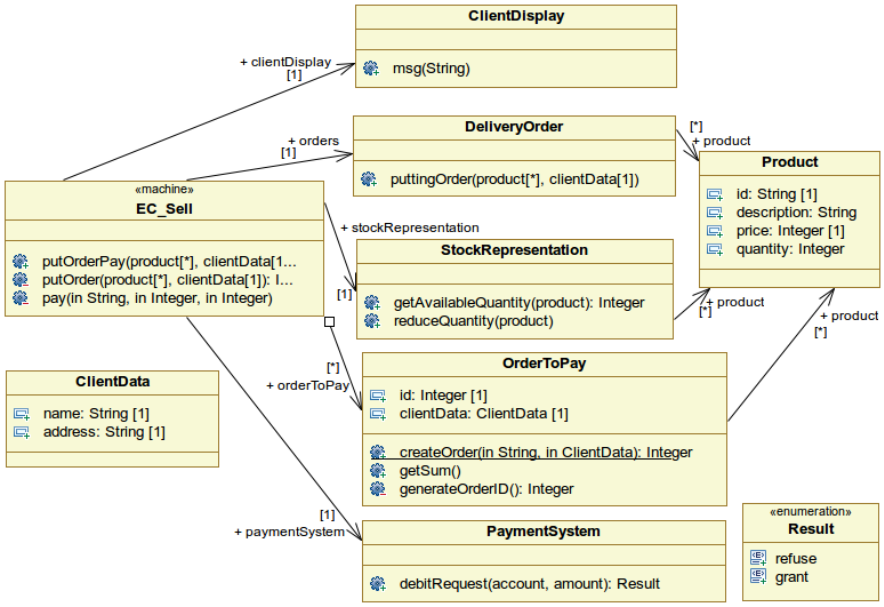

Fig. 10 Business class model of EC sell

```
let otpid: Integer =
  putOrder(product,clientData)
in
  self^pay(accountData,
           orderToPay->select(o| o.id =
                otpid)->asSequence()->first().getSum(),
           otpid)
```

The operations *putOrder* and *pay* are defined as follows. The postcondition of *putOrder* expresses that the client receives a message that the product is not available if the quantity stored in the stock representation is not strictly greater than 0. Otherwise a new order is created and the stock representation is updated by deducing the ordered quantity.

```
putOrder(product: String [*], clientData: ClientData): Integer
PRE:
  Product.allInstances().id->includesAll(product)
  and clientData.name<>'' and clientData.address<>''

POST:
  not product->forAll( p |
    stockRepresentation.getAvailableQuantity(p)>0
  ) implies
  clientDisplay^msg('product not available')
  and
  product->forAll( p |
    stockRepresentation.getAvailableQuantity(p)>0
  )
  and
  result = orderToPay.createOrder(product,clientData)
       ->asSequence()->first()
```

```
  and
  product->forAll( p |
    stockRepresentation^reduceQuantity(p)
  )
  and
  orderToPay->size()=orderToPay@pre->size()+1
```

The operation *pay* yields an error message if the external payment systems declines payment. Otherwise, the payment is executed.

```
pay(accountData: String, sum: Integer, orderID: Integer)
PRE:
  true
POST:
  not (paymentSystem.debitRequest(accountData,sum)='grant')
  implies
  clientDisplay.msg('Cannot debit amount')
POST:
  paymentSystem.debitRequest(accountData,sum)='grant'
  implies
  self^paid(orderID)
```

3.6 Life cycle

The last step of the requirements analysis phase consists of defining a life-cycle model for the machine, which describes the relations between the different subproblems.

We specify the possible activities from the viewpoints of the main actors in terms of the identified subproblems. In our case study, the client can browse the catalogue several times before buying some product, which we express by $C = Browse^*; Sell$. The managers update the catalogue, as expressed by $M = upCat$. Clients and managers can act in parallel: $System = C||M$.

We need the life-cycle model later, when we have to decide if a coordinator component should be introduced in the software architecture.

4 Architecture

We develop the software architecture for enterprise application systems in two steps. First, we set up an initial architecture that is then transformed into an implementable architecture.

4.1 Initial Architecture

In the initial architecture we collect information from the requirements analysis phase. On the one hand, we decide which domains of the problem diagrams are transformed into components of the software architecture. On the other hand, we transform interfaces of the problem diagrams into ports and interfaces of the software architecture.

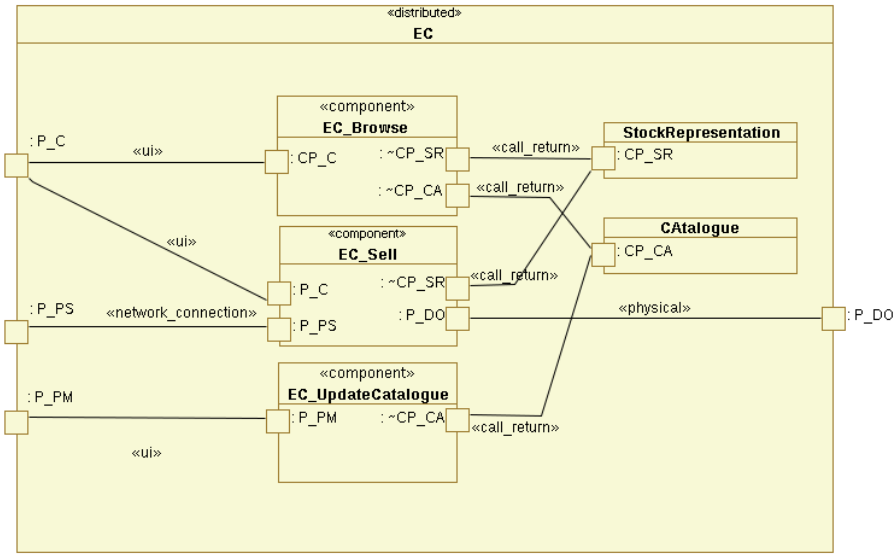

Fig. 11 Initial architecture of Electronic Commerce

First, each machine domain of a problem diagram becomes a component in the inital architecture, which has the same name as the machine domain in the context diagram. Second, business objects become components, because they are lexical and thus can exist only inside the machine. Business workers and external systems are outside the machine and thus do not become components. Interfaces of the domains that are mapped to components become interfaces to these components (represented by ports). If an interface belongs to a component corresponding to a machine domain, its interfaces are also interfaces of the overall software architecture.

Figure 11 shows the inital architecture for the electronic commerce system, which was set up according to the above rules. Note that we annotate the connections between the different components with appropriate stereotypes. The ports are either component ports (CP) or just ports (P), and the second part of their name refers to what they are connected to (e.g. P_PM is a port to Product Manager). We used the external ports (P) for the components if the components provide and required the same functionality as the external port. Moreover, we have to decide if

the software we are going to build will be distributed or not. We decide that the electronic commerce system will be a distributed system.

4.2 Implementable Architecture

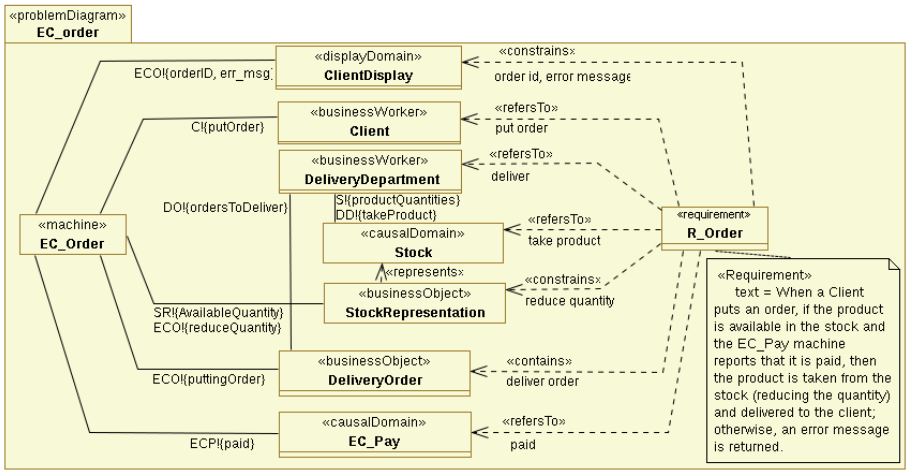

Fig. 12 Problem diagram for EC_order

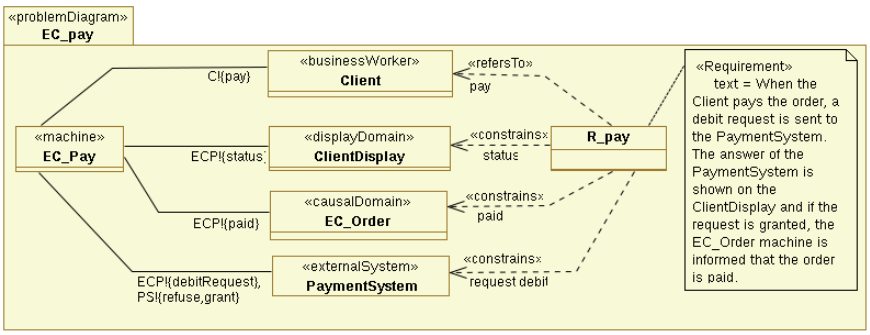

Fig. 13 Problem diagram for EC_pay

In the second step of the architectural design, we transform the initial architecture into an implementable architecture. If the initial architecture is annotated as distributed, it must be decided how the distribution will be achieved. Moreover, if a component is connected to several external ports, a facade component will be introduced. If from the life-cycle model it follows that some interactions have to take place before other interactions can be performed, a coordinator component is necessary.

In our example, we do not need a coordinator component, because placing an order (*Sell*) is possible without prior browsing. However, we decide that for security reasons the processing of orders and the processing of payments should be performed on different computers. As a consequence, we split the problem diagram *EC_sell* into two new problem diagrams as shown in Figs. 12 and 13.

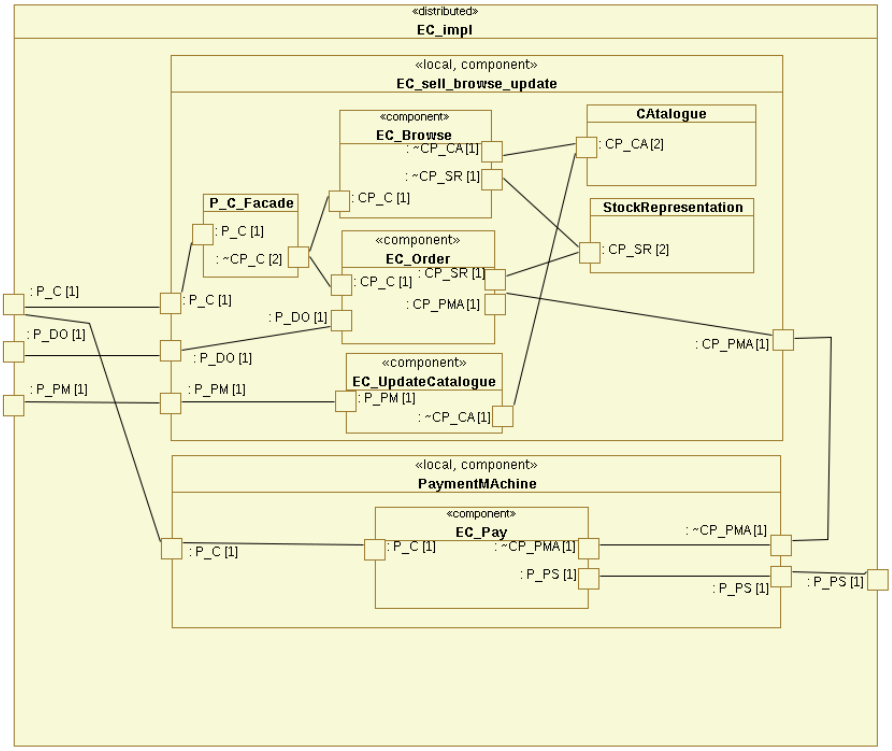

Fig. 14 Implementable architecture of Electronic Commerce

The machine *EC_Sell* is split into *EC_Order* and *EC_Pay*. The requirement *R_Sell* is split into *R_Order* and *R_Pay*. The interface *ECS!*{*msg*} is concretized by *ECO!*{*orderID, err_msg*} and *ECP!*{*status*}. The interface *C!*{*putOrderPay*} is concretized by *C!*{*putOrder*} and *C!*{*pay*}. For the other interfaces the controlling

domain needs to be *EC_Order* instead of *EC_Sell*. Therefore the interface names change, e.g., from *ECS!*{*reduceQuantity*} to *ECO!*{*reduceQuantity*}.

From the new problem diagrams, we derive the implementable architecture *EC_IMPL* shown in Fig. 14, which is a specialization of the EC machine. We have one component for the payment and one component for the other functionalities. The two components are connected via newly defined ports.

To split the business model (OCL), the operation *putOrderPay* of *EC_sell* is removed. The operation *putOrder* of *EC_sell* becomes a public operation of *EC_Order*. The operation *pay* of *EC_sell* becomes a public operation of *EC_Pay*. To allow the payment system (*EC_Pay*) informing the ordering system (*EC_Order*) a new operation *paid* is added to *EC_Order*. This operation triggers the product delivery with `orders^puttingOrder`.

```
paid(orderID: Integer)
PRE:
orderToPay.id->includes(orderID)
POST:
let otpay: OrderToPay =
   orderToPay->select(otp| otp.id=orderID)->asSequence()->first()
in
   orders^puttingOrder(otpay.product.id->asSet(),otpay.clientData)
```

The software architecture given in Fig. 14 can be implemented as it is defined. If wished for, it can further be transformed, for example into a layered architecture as described in (Choppy et al., 2011).

Note that our architecture is not optimized in any way. Other architectures could be developed, however, without support by our method. Our aim is to provide a systematic way to develop a working architecture. Moreover, we focus on functional requirements in this work. How quality requirements can be taken into account, too, is discussed in (Alebrahim, Hatebur, & Heisel, 2011).

5 Related work

Hall, Rapanotti et al. (Hall, Jackson, Laney, Nuseibeh, & Rapanotti, 2002; Rapanotti, Hall, Jackson, & Nuseibeh, 2004) introduce architectural concepts into problem frames (introducing "AFrames") so as to benefit from existing architectures. In (Hall et al., 2002), the applicability of problem frames is extended to include domains with existing architectural support, and to allow both for an annotated machine domain, and for annotations to discharge the frame concern. In (Rapanotti et al., 2004), "AFrames" are presented corresponding to the architectural styles Pipe-and-Filter and Model-View-Controller (MVC), and applied to transformation and control problems. In contrast, we keep requirements and architectural documents separate.

Barroca et al. (Barroca, Fiadeiro, Jackson, Laney, & Nuseibeh, 2004) extend the problem frame approach with *coordination* concepts. This leads to a description of *coordination interfaces* in terms of *services* and *events* (referred to respectively here

as actuators and sensors) together with required properties, and the use of *coordination rules* to describe the machine behavior. Our method is more concerned with structure than with behavior.

Lavazza and Del Bianco (Lavazza & Bianco, 2006) also represent problem diagrams in a UML notation. They use component diagrams (and not stereotyped class diagrams) to represent domains. Jacksons interfaces are directly transformed into used/required classes (and not observe and control stereotypes that are translated in the architectural phase). In a later paper, Del Bianco and Lavazza (Lavazza & Bianco, 2008) suggest enhance problem frames with scenarios and timing, which we do not consider in this paper.

Hofmeister et al. (Hofmeister, Nord, & Soni, 1999) describe software architectures in four views (conceptual, module, execution, and code) with UML and stereotypes. Five industrial architecture design methods are compared in (Hofmeister et al., 2007), and a general approach is extracted where the design activities are the architecture analysis, synthesis (i.e. the core of the design) and evaluation. We may consider that, although our approach is quite different, it complies with these design activities.

Bleistein et al. (Bleistein, Cox, & Verner, 2006) describe how to come from strategic level business requirements to machine level requirements. They also use problem diagrams, but combine them with VMOST (Sondhi, 1999). VMOST is a technique for deconstructing business strategies into core components. The presented approach complements our procedure for deriving the specifications of the components in the implementable architecture.

Attribute Driven Design (ADD) (Wojcik et al., 2006) is a method to design a conceptual architecture. It focuses on the high-level design of an architecture, and hence does not support detailed design.

Charfi et al. (Charfi, Gamatié, Honoré, Dekeyser, & Abid, 2008) use a modelling framework, *Gaspard2*, to design high-performance embedded systems-on-chip. They use model transformations to move from one level of abstraction to the next. To validate that their transformations were performed correctly, they use the OCL language to specify the properties that must be checked in order to be considered as correct with respect to Gaspard2. We have been inspired by this approach. However, we do not focus on high-performance embedded systems-on-chip. Instead, we target general software development challenges.

Choppy and Heisel give heuristics for the transition from problem frames to architectural styles. In (Choppy & Heisel, 2003), they give criteria for choosing between architectural styles that could be associated with a given problem frame. In (Choppy & Heisel, 2004), a proposal for the development of information systems is given using *update* and *query* problem frames. A component-based architecture reflecting the repository architectural style is used for the design and integration of the different system parts.

In (Choppy, Hatebur, & Heisel, 2005), Choppy, Heisel and Hatebur propose architectural patterns for each basic problem frame proposed by Jackson (Jackson, 2001). In a follow-up paper (Choppy, Hatebur, & Heisel, 2006), the authors show how to merge the different sub-architectures obtained according to the patterns pre-

sented in (Choppy et al., 2005), based on the relationship between the subproblems. Hatebur and Heisel (Hatebur & Heisel, 2009) show how interface descriptions for layered architectures can be derived from problem descriptions.

In a more recent paper (Choppy et al., 2011), Choppy, Heisel and Hatebur describe how to derive software architectures from problem diagrams in a general setting. This paper emphasizes the integrity conditions that have to hold for the different models that are set up in the process.

6 Conclusion and Perspectives

In this chapter, we have presented a method that supports the development of enterprise application systems. It covers the phases requirements analysis and architectural design. Requirements analysis is based on a specialization of the problem frame approach to enterprise applications, in particular, a specific enterprise application frame. The business processes to be automated are explicitly represented using domain knowledge diagrams. The result of the requirements analysis phase are a set of problem diagrams, each covering a relevant aspect of the software developement problem. These diagrams form the basis of the architectural design phase, which makes use of the information collected in the analyis phase. The architectural design takes into account the specifics of enterprise application systems, in particular the business objects that have to be stored appropriately, often using databases.

Different diagrams expressed in UML are set up during requirements analysis as well as architectural design.

In summary, the contributions of our work are the following:

- We provide a systematic development method for enterprise systems, based on well-structured requirements documents.
- The specifics of enterprise systems are taken into account by specific domain types and a specialized enterprise problem frame.
- From the problem description, most interfaces and the data specification can be derived in a systematic way.
- The different artifacts developed with our method are linked; thus, traceability and change propagation are supported.
- The method is tool-supported, which relieves developers of tedious modeling and validation tasks.

Future work on this trend includes to adapt and extend the validation conditions to take into account the enterprise problem frame and our approach that includes domain knowledge diagrams and diagrams to describe the behaviour (here activity diagram). We also intend to semi-automatically generate the architectures for enterprise systems.

References

Alebrahim, A., Hatebur, D., & Heisel, M. (2011). A method to derive software architectures from quality requirements. In T. D. Thu & K. Leung (Eds.), *Proceedings of the 18th asia-pacific software engineering conference (APSEC)* (pp. 322–330). IEEE Computer Society.

Barroca, L., Fiadeiro, J. L., Jackson, M., Laney, R. C., & Nuseibeh, B. (2004). Problem frames: A case for coordination. In *Coordination models and languages, proc. 6th international conference coordination* (p. 5-19).

Bertrand, P., Darimont, R., Delor, E., Massonet, P., & Lamsweerde, A. van. (1998). GRAIL/KAOS: an environment for goal drivent requirements engineering. In *Icse'98 - 20th international conference on software engineering.*

Bleistein, S. J., Cox, K., & Verner, J. (2006, March). Validating strategic alignment of organizational it requirements using goal modeling and problem diagrams. *J. Syst. Softw.*, *79*(3), 362–378. Available from http://dx.doi.org/10.1016/j.jss.2005.04.033

Bresciani, P., Perini, A., Giorgini, P., Giunchiglia, F., & Mylopoulos, J. (2004). Tropos: An agent-oriented software development methodology. *Autonomous Agents and Multi-Agent Systems*, *8*(3), 203–236.

Charfi, A., Gamatié, A., Honoré, A., Dekeyser, J.-L., & Abid, M. (2008). Validation de modèles dans un cadre d'IDM dédié à la conception de systèmes sur puce. In *4èmes journées sur l'ingénierie dirigée par les modèles (idm 08).*

Choppy, C., Hatebur, D., & Heisel, M. (2005). Architectural patterns for problem frames. *IEE Proceedings – Software, Special Issue on Relating Software Requirements and Architectures*, *152*(4), 198–208.

Choppy, C., Hatebur, D., & Heisel, M. (2006). Component composition through architectural patterns for problem frames. In *Proc. xiii asia pacific software engineering conference (apsec)* (pp. 27–34). IEEE.

Choppy, C., Hatebur, D., & Heisel, M. (2011). Systematic architectural design based on problem patterns. In P. Avgeriou, J. Grundy, J. Hall, P. Lago, & I. Mistrik (Eds.), *Relating software requirements and architectures* (pp. 133–159). Springer.

Choppy, C., & Heisel, M. (2003). Use of patterns in formal development: Systematic transition from problems to architectural designs. In *Recent Trends in Algebraic Development Techniques, 16th WADT, Selected Papers* (pp. 205–220). Springer Verlag.

Choppy, C., & Heisel, M. (2004). Une approache à base de "patrons" pour la spécification et le développement de systèmes d'information. In *Proceedings approches formelles dans l'assistance au développement de logiciels - afadl'2004* (pp. 61–76).

Choppy, C., & Reggio, G. (2005). A UML-Based Approach for Problem Frame Oriented Software Development. *Journal of Information and Software Technology*, *47*, 929-954.

Choppy, C., & Reggio, G. (2006). Requirements capture and specification for enterprise applications: a UML based attempt. In J. Han & M. Staples (Eds.),

Proc of the australian software engineering conference (aswec 2006), ieee (p. 19-28).

Hall, J. G., Jackson, M., Laney, R. C., Nuseibeh, B., & Rapanotti, L. (2002, 9-13 September). Relating software requirements and architectures using problem frames. In *Proceedings of ieee international requirements engineering conference (re'02).* Essen, Germany.

Hatebur, D., & Heisel, M. (2009). Deriving software architectures from problem descriptions. In *Software engineering 2009 - workshopband* (pp. 383–302). GI.

Hatebur, D., & Heisel, M. (2010). Making pattern- and model-based software development more rigorous. In J. S. Dong & H. Zhu (Eds.), *Proceedings of international conference on formal engineering methods (ICFEM)* (Vol. LNCS 6447, pp. 253–269). Springer.

Heisel, M., & Hatebur, D. (2005). A model-based development process for embedded systems. In T. Klein, B. Rumpe, & B. Schätz (Eds.), *Proc. workshop on model-based development of embedded systems.* Technical University of Braunschweig. (Available at http://www.sse.cs.tu-bs.de/publications/MBEES-Tagungsband.pdf)

Hofmeister, C., Kruchten, P., Nord, R. L., Obbink, H., Ran, A., & America, P. (2007). A general model of software architecture design derived from five industrial approaches. *Journal of Systems and Software*, *80*(1), 106–126.

Hofmeister, C., Nord, R. L., & Soni, D. (1999). Describing software architecture with UML. In *Proceedings of the first working ifip conference on software architecture* (pp. 145–160). Kluwer Academic Publishers.

Jackson, M. (2001). *Problem frames. analyzing and structuring software development problems*. Addison-Wesley.

Jackson, M., & Zave, P. (1995). Deriving specifications from requirements: an example. In *Proc. 17th int. conf. on software engineering* (pp. 15–24). ACM Press.

Lavazza, L., & Bianco, V. D. (2006). Combining Problem Frames and UML in the Description of Software Requirements. *Fundamental Approaches to Software Engineering*.

Lavazza, L., & Bianco, V. D. (2008, February). Enhancing Problem Frames with Scenarios and Histories in UML-based software development. *Expert Systems - The Journal of Knowledge Engineering*, *25*(1).

Nelson, M., Cowan, D., & Alencar, P. (2001). Geographic problem frames. In *Fifth ieee international symposium on requirements engineering* (pp. 306–307).

Rapanotti, L., Hall, J. G., Jackson, M., & Nuseibeh, B. (2004, 6-10 September). Architecture driven problem decomposition. In *Proceedings of 12th ieee international requirements engineering conference (re'04).* Kyoto, Japan.

Sondhi, R. (1999). *Total strategy*. Airworthy Publications International Ltd.

"UML Revision Task Force". (2009, February). Omg unified modeling language: Superstructure [Computer software manual]. (available at http://www.omg.org/docs/formal/09-02-02.pdf)

UML Revision Task Force. (2010, February). Object Constraint Language Specification [Computer software manual]. Retrieved 10-08-2011, from http://www.omg.org/spec/OCL/

Warmer, J., & Kleppe, A. (2003). *The object constraint language 2.0: Getting your models ready for MDA* (2nd ed.). Pearson Education.

Wojcik, R., Bachmann, F., Bass, L., Clements, P., Merson, P., Nord, R., et al. (2006). *Attribute-Driven Design (ADD)* (Version 2.0). Software Engineering Institute. Available from ftp://ftp.sei.cmu.edu/pub/documents/06.reports/pdf/06tr023.pdf

Yu, E. (1997). Towards modelling and reasoning support for early-phase requirements engineering. In *Proceedings of the 3rd ieee intern. symposium on re* (pp. 226 – 235).