Using UML for Problem Frame Oriented Software Development

C. Choppy
LIPN, Université Paris XIII
99 Avenue Jean-Baptiste Clément,
F-93430 Villetaneuse, France

G. Reggio
DISI
Università di Genova
Genova, Via Dodecaneso 35, 16146, Italy

Abstract

We propose a software development approach that combines the use of the UML notation, the use of the structuring concepts provided by the problem frames, together with our methodological approach for well-founded methods. The problem frames are used to provide a first idea of the main elements of the problem under study. Then we provide ad hoc UML based development methods for some of the most relevant problem frames together with precise guidelines for the users. The general idea of our method is that, for each frame, several artifacts have to be produced, each one corresponding to a part of the frame. The description level may range from informal and sketchy, to formal and precise, while this approach is drawn from experience in formal specifications.

1 Introduction

While the early phases of software development are crucial to the success of a project, they remain sometimes quite uneasy to achieve well. Different techniques and notations are proposed to help, but it would be nice to be able to combine their advantages.

One of the difficult issues is to start the analysis of a complex problem. M. Jackson proposes "Problem Frames" [9] which are often used structures drawn from experience, that can be used by themselves or in combination to tackle with a first gross structuring of problems. Problem frames differ by their requirements, domain characteristics, involvements, and frame concern. For each problem frame, a diagram is settled, showing the involved domains, the requirements, the design, and their interfaces. Five basic problem frames are provided [9] together with some variants.

While the structuring concepts brought by problem frames seem highly valuable to help to start the development effort, showing clearly the items to consider and the general tasks to do, no development method is associated with them. In [7] we showed how, for each problem frame, we can devise a nicely tailored development method using as a modelling notation the formal algebraic specification languages Casl[4] and Casl-Ltl[11].

Now, since the UML notation [12] is widespread and also carries valuable concepts experienced in practice, our idea here is to propose for the various frames a development method using UML as a notation, which, to our knowledge, was never attempted before.

There are some drawbacks with the use of UML. While it provides a nice variety of constructs, they are redundant, and it may be difficult to choose which are appropriate. There are no means to fully insure the consistency between the different views used to build a model, and, moreover, the UML semantics is both informal and problematic. However, in [2, 3], it has been shown that UML may be used in a development method in a quite precise, structured and well-founded way, avoiding most of its typical problems (e.g., only a subset of UML with a semantics that may be formally given is used).

Here, taking advantage of the formal treatment made for the various problem frames in [7], we propose for each problem frame a development method using UML together with precise guidelines for the users, by tailoring the general one proposed in [2, 3]. These help to reduce the development time and prevent loosing time searching how to model the various aspects. This lead us also to discover how to properly handle, using UML, many kind of applications that do not easily fit inside the standard business case/use case approach, as proposed by many UML-based methods (e.g., RUP [10] and COMET [8]).

The general idea of our method is that, for each frame, several artifacts have to be produced, each one corresponding to a part of the frame. Following [2, 3] we group together those corresponding to the Domain Model, those to the Requirement Specification and to the Design Specification respectively, and then we show how to present them by UML models of a particular structure. As a consequence, each method may be complemented with a schematic model prepared using any of the supporting software tools, so as to provide further support of the developer effort.

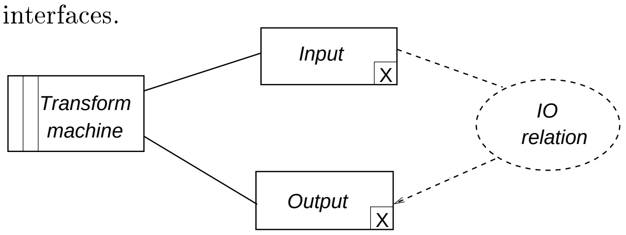
Moreover, the models produced following these methods may be made quite precise or more sketchy depending on the taste and aim of the developer. Indeed, the textual inscriptions on the models, such as the constraints and the method body, may be expressed in different ways, ranging from informal to formal. It may be convenient to start with an informal notation, such as the natural language, to quickly prepare an easily readable model, whenever this is considered as precise enough. Then, one may go to a more precise version, using OCL [13], or choose another formal language, e.g., close to algebraic specifications and their extensions as in [1].

In the following, we describe the development methods we propose for several problem frames, precisely the Transformation, Commanded Information, Required Behaviour, and we illustrate them on case studies.

2 Transformation Frame

2.1 The Frame

The problem frames distinguish the domain parts (drawn as rectangles), the requirements (as a dashed oval), and the design (as a double striped rectangle), which are connected through interfaces. Here, we do not show the events (or phenomena) that occur at the interfaces.

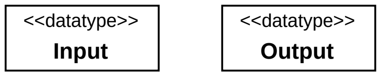


The transformation frame [9], schematically presented above, is described as follows. There are some data which must be transformed to give certain output data. The output data must be in a particular format, and must be derived from the input data according to certain rules. The problem is to build a machine that will produce the required outputs from the inputs. The X in the picture indicates that *Input* and *Output* are inert domains. The requirements are expressed through an input/output relation. The dashed line represents a requirements reference, and the arrow shows that it is a *constraining* reference.

2.2 The UML Models

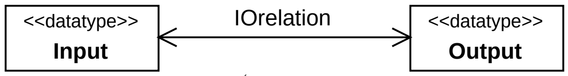
Domain Model

In this case the Domain Model contains the description of the *Input* and of the *Output* and is a UML model containing at least the following fragment; clearly any other UML constructs may be used to describe the two classes.



Requirement Specification

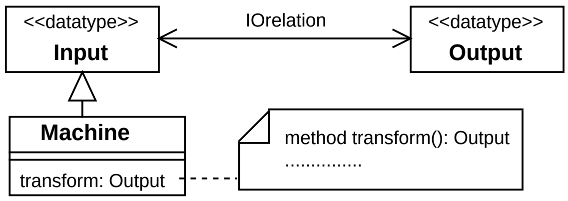
The requirements are specified by extending the Domain Model with the following association



and by qualifying it (fixing the multiplicities and adding invariant constraints to both the connected classes).

Design

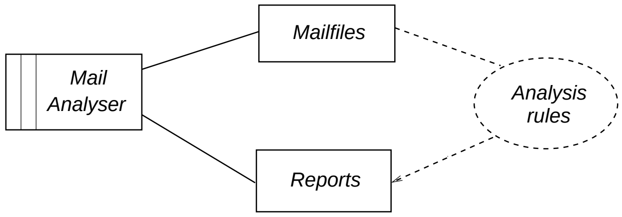
The design is presented by extending the requirement model with the class Machine where the *transform* method is available.



2.3 Case study: Mailfiles Analysis

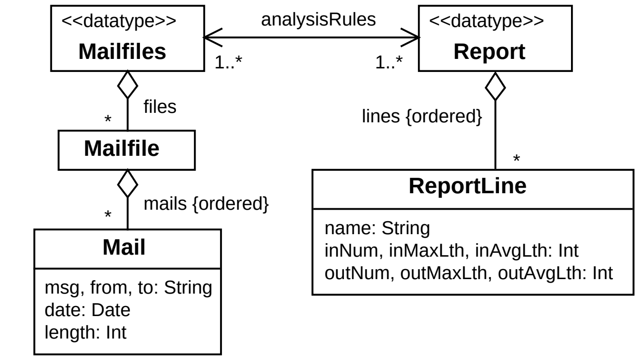
The goal is to analyse mailfiles, and build a report providing for instance, for each correspondent, the number of messages received, their maximum and average lengths, and the same information for the messages sent by the user of this facility [9].

The problem frame instance is as follows.



Domain Model and Requirement Specification

The Domain Model and the Requirement specification are given by the following UML model

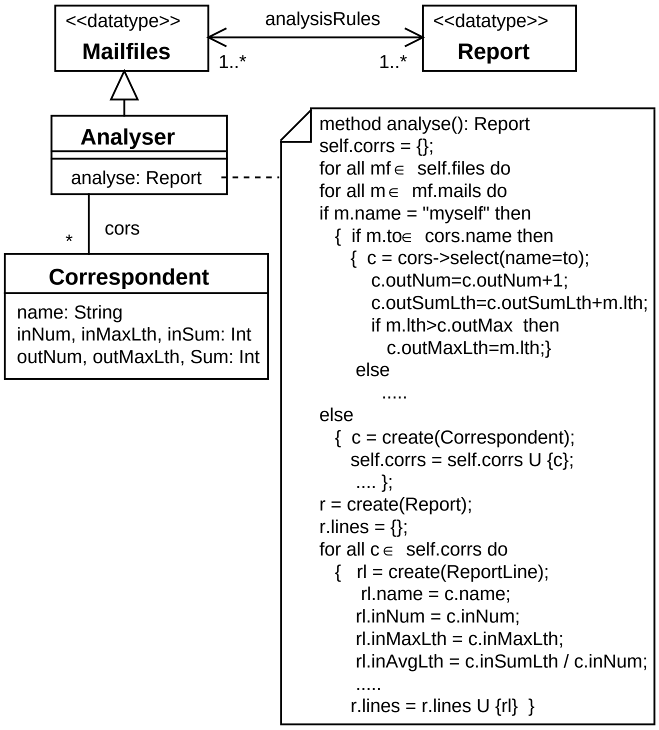


accompanied by these constraints

```
context mfs: MailFiles inv:
for all rep∈mfs.analysis
for all rl∈rep.lines
   exists m∈mf.mails s.t.
   (m.to = rl.name or m.from = rl.name) and
   rl.inNum =
      size({m | m∈mfs.files.mails and m.from = rl.name})
   and ...
```

Design Specification

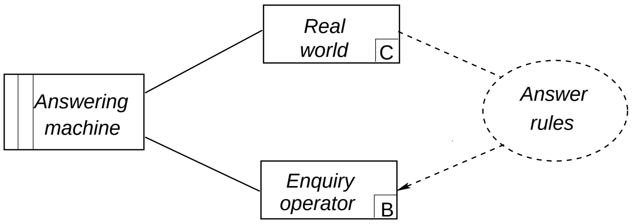
We extend the class diagram given before so as to introduce the description of the *Transform machine* that is the `Analyser` with the `analyse` method.



3 Commanded Information Frame

3.1 The Frame

The Commanded Behaviour frame is described as follows. There is some part of the physical world whose states and behavior are needed upon requests from an operator. The problem is to build a machine that will obtain this information from the world and present it in the required form.



This frame, schematically shown above, is a variant of the frame with the same name given in [9] because here the information is directly presented to the *Enquiry operator* and not by means of a *Display*. The B stands for "biddable" and is used for domains that are people.

3.2 The UML Models

Domain Model

The Domain Model in the case of the Comamnded Information Frame includes only the *Real world*, whereas the *Enquiry operator* will be introduced after in the Requirement Specification.

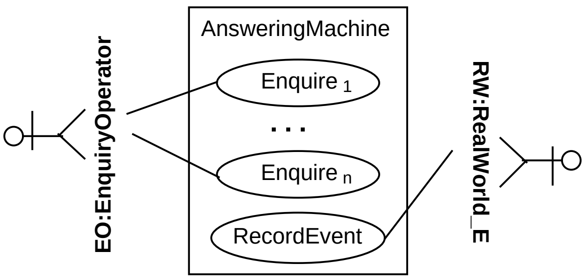
The *Real world* is described by a UML model containing (a class diagram with) an active class named RealWorld. This model may be very complex including a detailed description of the behaviour of RealWorld, or just a small conceptual model containing RealWorld and few other classes.

Requirement Specification

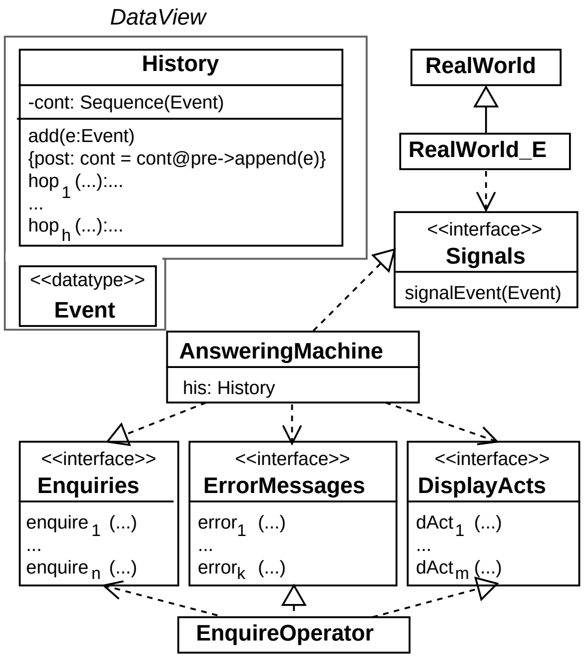
In [7] we provided a formal specification skeleton for the various parts of the Commanded Information Frame that introduces events (signalling relevant changes in the *Real world* state), and the history of the events that occurred, and here we follow the same basic ideas.

In this case use cases are suitable to present the requirements and we use them; thus the Requirement Specification, corresponding to the *Answer rules* and the *Enquiry operator* of the frame, is a UML model consisting of:

– **a use case diagram** of the following form, where the *Answering machine* is the system, and with a use case for each kind of enquiry, and one corresponding to record the events happening in the *Real world*. Recall that following [2, 3] we consider as actors all those entities interacting with the system either by taking advantage of its services (EO) or by helping with their services to fulfill its goals (RW).



– **a class diagram**, corresponding to the Data View, Context View and Internal View of [2, 3], containing the following fragment:[1]



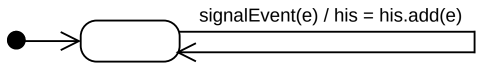
The class RealWorld_E models a specialization of RealWorld obtained by adding the capability of signalling to the *Answering machine* the relevant events when they happen (technically modelled by the fact that it uses the interface Signals). The events are modelled by the datatype Event. The class AnsweringMachine models the system to be developed (the *Answering machine* in the problem frame).

This diagram shows also which is the context of the AnsweringMachine and its interfaces towards it. We have that the AnsweringMachine interacts with the RealWorld_E by receiving its event signals, with the EnquiryOperator by receiving its enquiries (interface Enquiries), and by displaying the required information (interface DisplayActs) and possible errors made in the enquiries (interface ErrorMessages).

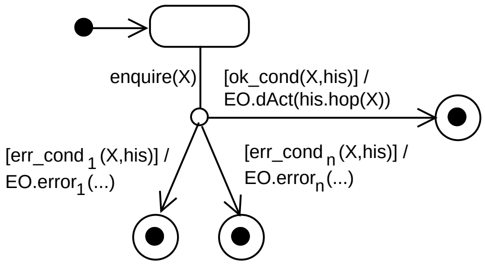
The abstract internal state of the AnsweringMachine just consists of the past history of the *Real world*. The class History abstractly models a history as the sequence of the events signalled by the *Real world*, and offers all the operations hop_i needed for answering to the enquiries. This abstract state will allow to precisely model the AnsweringMachine behaviour during each use case.

– **use case descriptions** (one for each use case in the use case diagram). Following [2, 3] we describe a use case by means of a statechart for the class modelling the system (here AnsweringMachine), which shows the complete behaviour of the system during that use case, including the messages exchanged with the actors and how they affect and are affected by its own "abstract" state.

The use case RecordEvent (in the history) is standard and its description is given by the following trivial statechart.



Whereas the statechart describing a generic use case Enquire has the following shape.

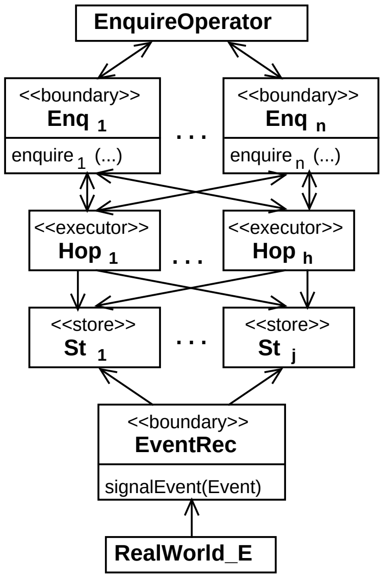


– **other diagrams and model elements**, which are used to describe all relevant information on the various classes appearing in the class diagram (such as EnquiryOperator, RealWorld_E, and Event). However, the operations of History must all be defined either by pre-postconditions or by associated methods (because they are the core of the description of each use case).

[1] A UML interface (keyword ≪interface≫) is a named set of operations, a dashed arrow from a class C to an interface I denotes that C will call the operations of I, whereas a dashed arrow with a solid head from I to C denotes that C will realizes I.

Design Specification

We propose a schematic design for the *Answering machine* following [3], which is described by a UML model consisting of:
– **a class diagram** (the StaticView in [3]) having the following form.



The classes EnquiryOperator and RealWorld_E model the external entities interacting with the *Answering machine*. The design methodology presented in [3] requires to organize the architecture of the system using three kinds of classes (each one represented by a stereotype):

≪boundary≫ (take care of the interaction of the system with the external entities),

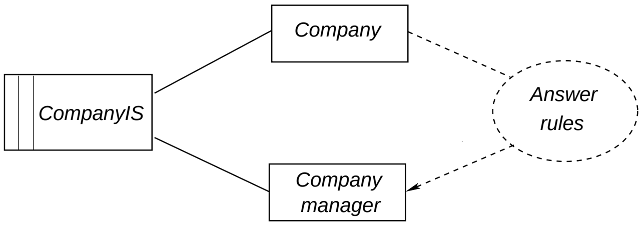
≪executor≫ (take care of the activities to be done by the system, by communicating among them and with the boundaries and the stores),

≪store≫ (take care of persistent data).
The associations, just model the fact that instances of a class send messages to instances of another class.

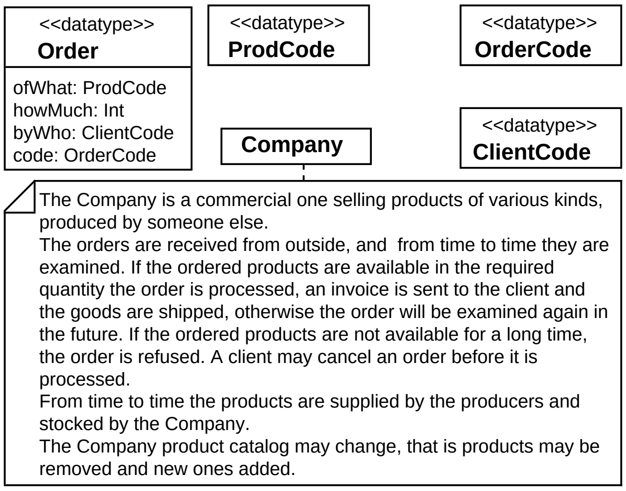
The suggested architecture for the *Answering machine* requires some stores for keeping in a convenient way the relevant information about the received events, a boundary taking care of receiving the events from RealWorld_E, a boundary for handling each kind of enquiries, and an executor for computing, using the information kept in the stores, each needed operation on the histories.
– **other diagrams and constraints** to describe the behaviour of the various classes in the above diagram (e.g., statecharts for the boundaries and executors, and methods or pre-postconditions for the operations of the store classes).

3.3 Case Study: Company Information System

The case study requires to develop an information system for a Company; this problem matches the Commanded Information Frame, as shown below. Indeed, the *Real world* is the *Company* itself, the *Enquiry operator* is a manager of the *Company* who needs to know some information on how the business is going to make decisions, the *Answer rules* describe how such information should be computed by looking at some relevant facts happening in the *Company*, and finally the *Answering machine* is the *Company Information System* itself (shortly *CompanyIS*).



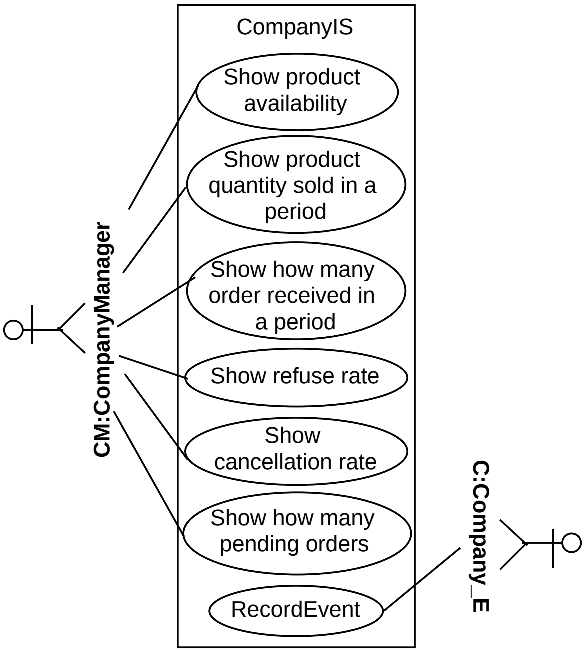
Domain Model



In this case, the behaviour of the class Company (The *Real world*) is simply described by natural text attached to the class using the UML note construct. That behaviour may have been described in a much more precise way by a statechart, but in this case we think that this is not relevant to the problem of building an information system, and so we took advantage from the flexibility of the UML opting for an informal description.

Requirement Specification

We first give the Use case diagram showing which are the pieces of information recovered by means of the *CompanyIS*.



The class diagram, see Fig. 1, is split in two parts, one presents the data structures and the other shows the context entities interacting with the *CompanyIS* and its internal abstract state.

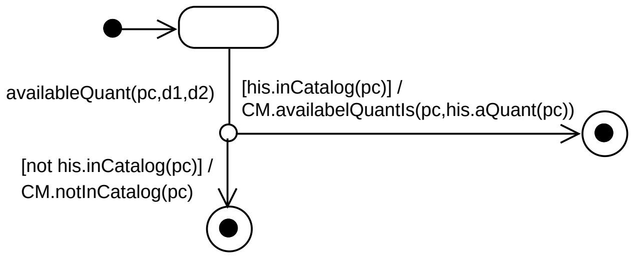
The events happening in the *Company* relevant to the Information System are those concerning the life of an order (received, processed, refused and cancelled), the stock refilling, and the addition/removing of products from the *Company* catalog (they are needed to discover wrong enquiries concerning products not in catalog). Notice that in Fig. 1 we use three different ways to present the constraints defining the History operations: some generic mathematical notation, pure OCL taking advantage of its iterate construct, and natural language. One can decide on the precision degree for the definition of these operations, but having to define them, in whatever way leads to make the requirements quite precise. For example, here it is clear that a product is considered sold when its order is processed, and not when its order is received.

The second part of the class diagram, which shows the context entities interacting with the *CompanyIS* and its internal abstract state, is again in Fig. 1. Notice how this diagram models the fact that the business of the *Company* should be adapted so as to signal to the *CompanyIS* all the events defined above whenever they happen. How to perform this task depends on the way the *Company* business is supported by sotware tools; for example, many of those events may be signalled by existing software systems.

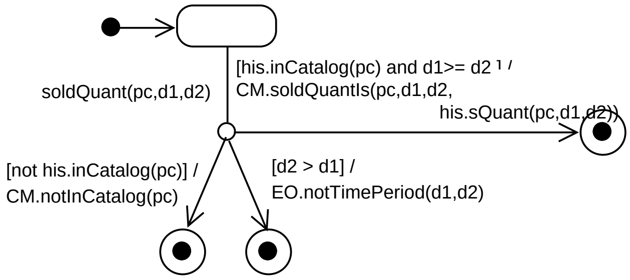
The non trivial use cases are described by statecharts presented in the following.

Notice how using statecharts instead of sequence diagrams allows to clearly see many aspects of what is required on the system to build. For example (use case Show product quantity sold in a period), if a sold quantity enquiry contains two errors, the product is not in the catalog and the two dates do not make a time period, only one will be nodeterministically signalled.

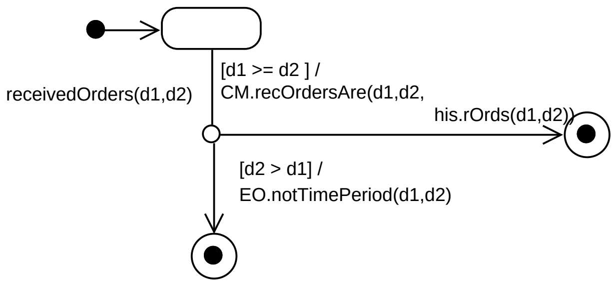
Show product availability



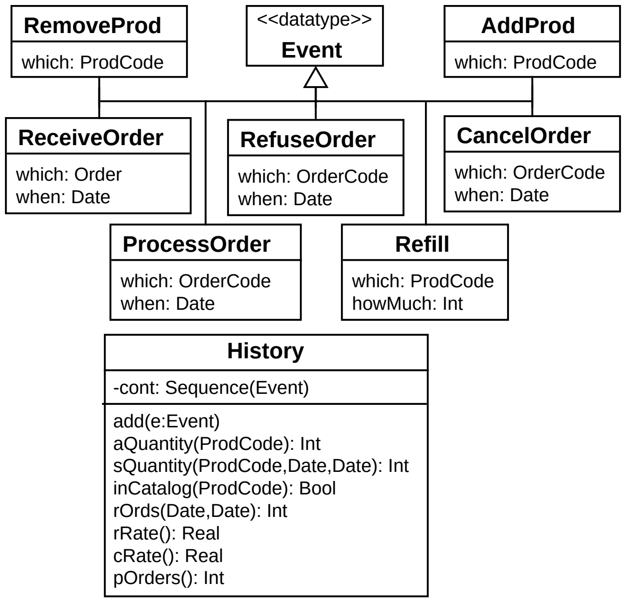
Show product quantity sold in a period



Show how many order received in a period



Show refuse rate



context inCatalog(pc:ProdCode): Bool post:
  result = exists i >= 1 s.t.
    self.cont[i] is the addition of pc and for all j > i self.cont[j] is not the removing of pc
context aQuantity(pc:ProdCode): Int post:
  result = cont -> iterate(e: Event; aQuant = 0 |
      if e.hasType(Refill) and pc = e.which then
        aQuant + e.howMuch
      else if e.hasType(ProcessOrder) and pc = e. which.ofWhat then
        aQuant - e.howMuch
    else aQuant)
context sQuantity(pc:ProdCode,d1,d2: Date): Int post:
  result = the sum of o.howMuch
    for all orders o of product pc processed in date d, with d2>=d>=d1
context Ords(d1,d2: Date): Int post:
  result = the number of the orders received in date d, with d2>=d>=d1
context rRate(): Real post:
  result = the number of all orders processed / the number of all orders refused
context cRate(): Real post:
  result = the number of all orders received / the number of all orders cancelled

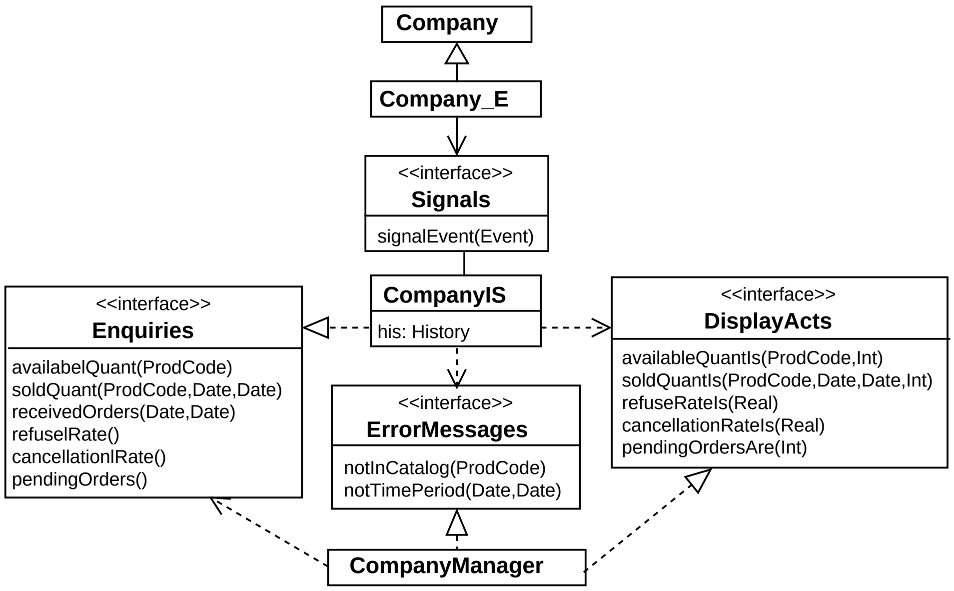
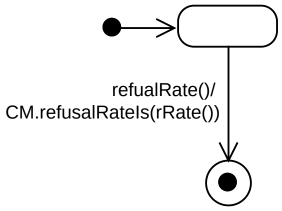
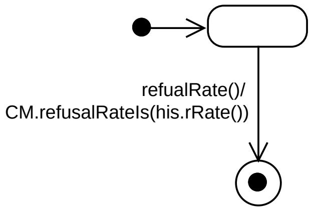


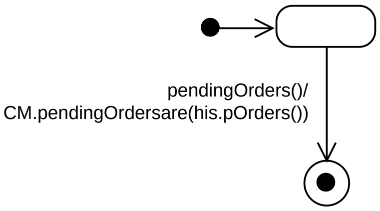
Figure 1: CompanyIS Requirement Specification: Class Diagram



Show cancellation rate



Show how many pending orders

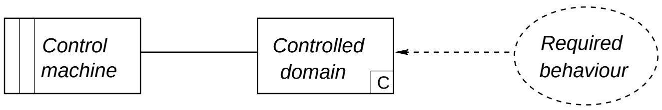


Design Specification

The class diagram of the UML model corresponding to the Design Specification of the *CompanyIS* is reported in Fig. 2. Notice that in this case, since the operation to be computed on the stores are quite simple the architecture of the design system has been simplified by dorpping trivial executors, thus in this case the boundaries directly calls operations of the stores. In Fig. 3 we show the statechart defining the behaviour of the class SoldQuant, thos concerning the other classes are omitted because they are quite similar.

4 Required Behaviour Frame

4.1 The Frame

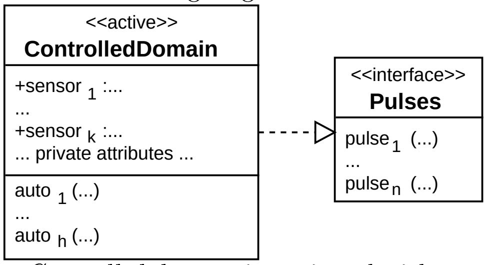


This frame, schematically shown above, is described as follows. "There is some part of the physical world whose behaviour is to be controlled so that it satisfies certain conditions. The problem is to build a machine that will impose that control."[9] The C in the frame diagram indicates that the domain *Controlled domain* must be causal. The machine is always a causal domain (so an explicit C is not needed).

4.2 The UML Models

Domain Model

The domain in this case corresponds to the *Controlled domain* and is described by a UML model that includes the following fragment.



The *Controlled domain* is equipped with some sensors, which are modelled by the public attributes sensor_1, ..., sensor_k)[2], and is controlled by sending it some pulses, which are modelled by the operations of the interface Pulses. However, a *Controlled domain* may change its state and the way it works even if it does not receive a pulse (for example, when a part breaks down or when some external entity acts over it). These "autonomous" activities are modelled by means of calls of the operations auto_1, ..., auto_h.

The behaviour of the class *Controlled domain* is then modelled by a statechart

– whose events are either time events or call events built by the operations pulse_1, ..., pulse_n, auto_1, ..., auto_h,

– and whose conditions and actions examine and update its attributes.

Moreover, for each sensor whose value is derived by other attributes, the model should contain an invariant constraint defining it.

Requirement Specification

In [7] the requirements part of a Required Behaviour Frame are some desired properties on the behaviour of the *Controlled domain*; but, to describe properties in UML is problematic. Indeed, the constraints (invariants for classes and pre-post for operations), written for example in OCL, can expressed a very limited form of properties on the behavior of a class. In the literature there are several proposals for extending OCL with temporal combinators (see, e.g., [6, 14]), but none

[2] The fact that they may break down is modelled by assuming that they may contain some special values corresponding to failures.

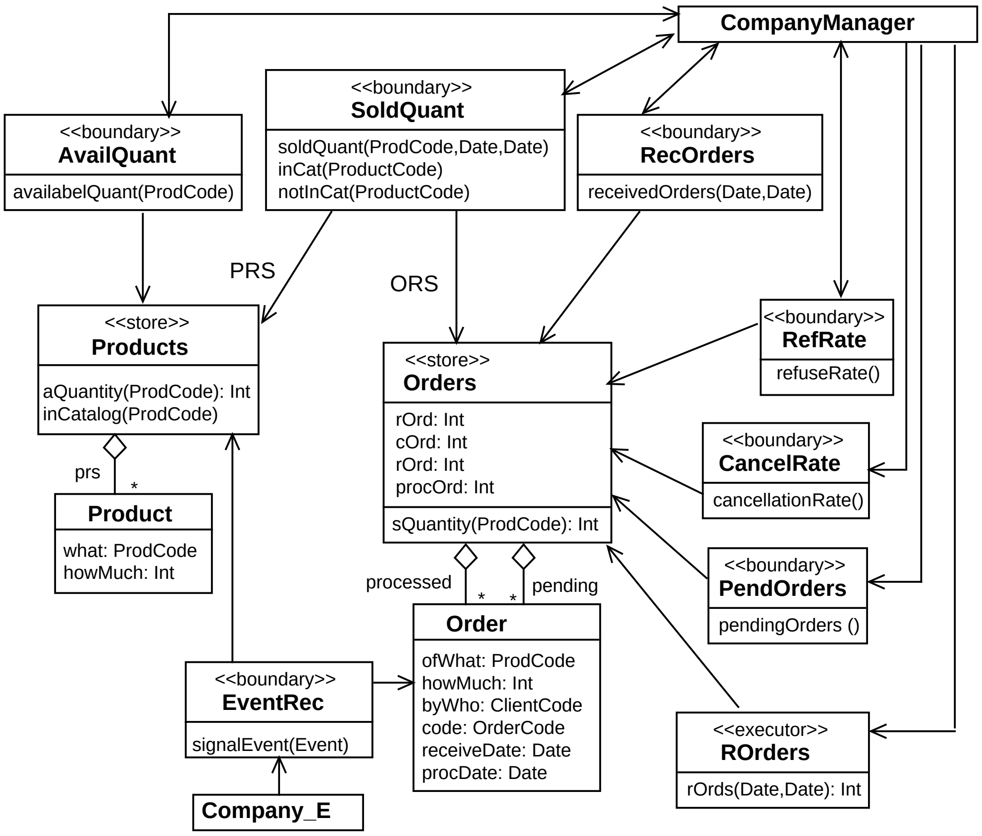


Figure 2: CompanyIS Design: Class Diagram

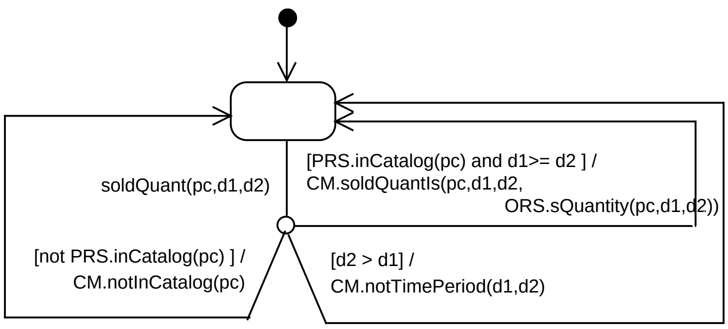


Figure 3: Behaviour of SoldQuant

of them has either reached the current practice or has been accepted as a standard. Thus, we do not think that it will be possible to propose a sensible UML-based method where the required behaviour will be modelled using properties.

On the other hand, the quite standard UML technique for requirements (i.e., use cases) is not very suitable for the required behaviour. Indeed, we should have just a unique use case described by the sentence "The controlled domain must behave in the following way: ...", an implicit actor that interacts at most once with the system to start it; and so a similar model will be not very interesting.

We think that an appropriate way to present the required behaviour is by means of a statechart[3] such that

– the events are either timed events or change events concerning its attributes,

– the conditions concern only its attributes,

– and the actions are updates of its attributes.

Indeed, the required behaviour of the *Controlled domain* is usually expressed in terms of *activities* to be done

– at certain time, they will be triggered by the timed events,

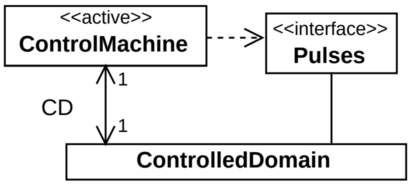
– or when something changes inside itself (e.g., due to failures or to acts by external entities), they will be triggered by the change events,

and that consists of modification of the internal state of the *Controlled domain*.

Obviously other operations for the class Controlled domain may be defined to help to express the conditions and the actions of this statechart in a simple and clear way.

Design Specification

The design for the Required Behaviour Frame consists in defining the *Control machine*, which is a process able to read the sensors of the *Controlled domain* and to send to it pulses. The *Control machine* is modelled by an active class whose behaviour is defined by a statechart. Thus, the design specification for this frame is a UML model containing



[3] Similarly [9] proposes on an example to use generic state machines.

and a statechart associated with the class ControlMachine such that

– its events are only change events on the sensors of the Controlled domain and timed events,

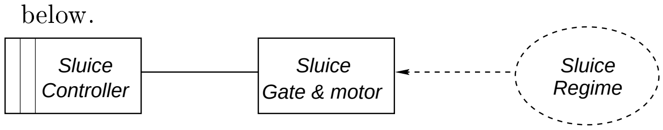
– its conditions are on its attributes and on the sensors of the Controlled domain,

– its actions may update its attributes and call the operations of the Pulses interface.

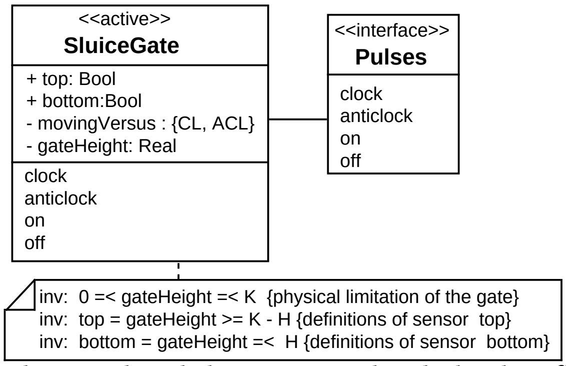
Obviously, also in this case we can define other operations for the class ControlMachine to help express the conditions and the actions of this statechart in a simple and clear way.

4.3 Case Study: Sluice Gate Control

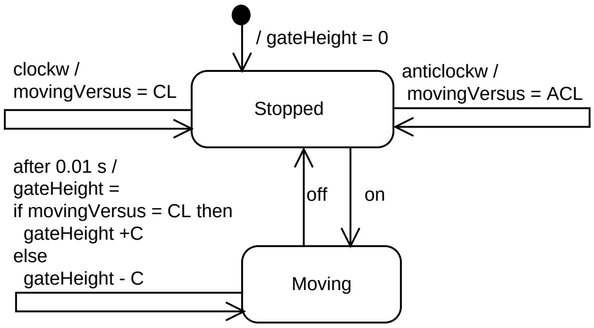
A small sluice, with a rising and falling gate, is used in a simple irrigation system. A computer system is needed to control the sluice gate: the requirement is that the gate should be held in fully open position for ten minutes in every three hours and otherwise kept in the fully closed position. The gate is opened and closed by rotating vertical screws. The screws are driven by a small motor, which can be controlled by clockwise CL, anticlockwise ACL, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top it is fully open, at the bottom it is fully shut. The connection to the computer consists of four pulse lines for motor control and two status lines for the gate sensors. The sluice gate control is an instance of the Required Behaviour Frame as shown by the schema below.



Domain Model

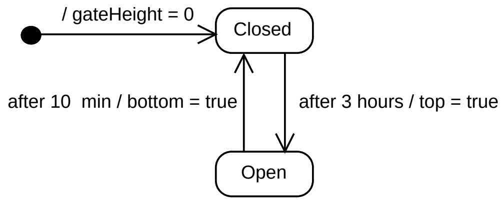


The statechart below, associated with the class *SluiceGate*, shows which is the effect of the pulses on the Sluice Gate motor and how it evolves when it is working and stopped; whereas what is signalled by the sensors is described by the constraints in the above class diagram.



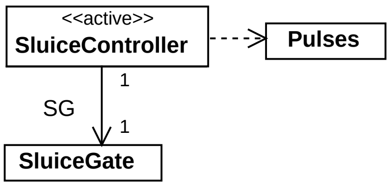
Requirement Specification

In this case the required behaviour is as follows: the sluice gate should be closed for three hours and then open for ten minutes and so on for ever. This behaviour is modelled by the following statechart associated with the class *SluiceGate*. Notice that we assume that the time needed to open and close the gate is negligible as regards those when the gate is open and closed (less than one second w.r.t. minutes and hours).

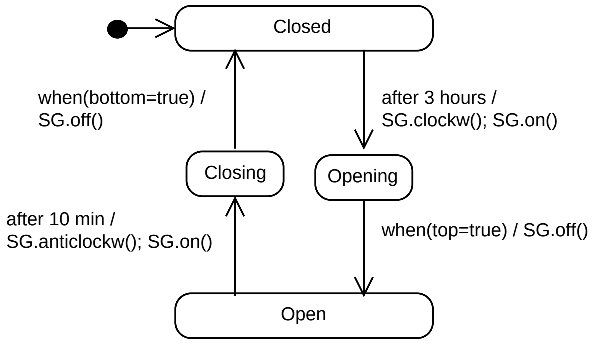


Design Specification

The simple *Control machine* designed in this case is presented by the following UML simple model consisting of



and of the following statechart associated with SluiceController.



5 Conclusion and future work

In this paper we have presented a software development approach that combines the use of the UML notation, the use of the structuring concepts provided by the problem frames, together with our methodological approach for well-founded methods.

While problem frames provide a first overall structure for problems, Jackson's idea was that, once a convenient problem frame is identified for a given problem, then the appropriate development method could be available. Thus we propose here development methods associated with three "basic" problem frames that cover quite frequent and relevant cases, with different characteristics. Our methods show, for each development phase, how to use appropriate UML constructs.

There are, to our knowledge, no similar approaches in the literature, proposing a method for the combined use of problem frames and UML.

However, some work was done to associate formal specifications with problem frames, as we did in [7], and D. Bjorner in [5]. Here, we took advantage of our experience in [7] to propose a well-founded way to use UML for the various frames.

We think that, with respect to the use of standard UML-based methods, our method is more efficient since the user does not need to loose time to devise the better way to use UML, deciding among the various constructs to use and in which way, thus (s)he may concentrate on the relevant aspects of the development instead of the questions related to the use of UML.

For each frame we offer schematic UML models for the three main phases of the development (Domain modelling, Requirement specification and Design specification) from which may also be derived schematic projects using some of the UML software support tools, such as Poseidon, Rational Rose and Together, further helping the developer effort.

We plan to provide similar methods for the other basic problem frames, such as Commanded Behaviour, Required Information, Workpieces, and we see no particular problems to do that. Whereas, for handling other cases, instead of following the Jackson idea of handling any problem by combining the various basic frames, we plan to devise new frames to cover other relevant development problems, such as Web Applications and three tiered Enterprise Applications.

References

[1] E. Astesiano, M. Cerioli, and G. Reggio. Plugging Data Constructs into Paradigm-Specific Languages: towards an Application to UML (Invited Lecture). In T. Rus, editor, *Proc. AMAST 2000*, number 1816 in Lecture Notes in Computer Science, pages 273–292. Springer-Verlag, Berlin, 2000. `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll00a.ps`.

[2] E. Astesiano and G. Reggio. Towards a Well-Founded UML-based Development Method. In A. Cerone and P. Lindsay, editors, *Proc. of SEFM '03*, pages 102–117. IEEE Computer Society, Los Alamitos, CA, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ AstesianoReggio03g.pdf` and `ftp://ftp.disi.unige.it/person/ReggioG/ AstesianoReggio03g.ps`.

[3] E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications. In M. Wirsing, A. Knapp, and S. Balsamo, editors, *Radical Innovations of Software and Systems Engineering in the Future, Proc. 9th Monterey Software Engineering Workshop, Venice, Italy, Sep. 2002.*, number 2941 in Lecture Notes in Computer Science. Springer Verlag, Berlin, 2004. Available at `ftp://ftp.disi.unige.it/person/ReggioG/AstesianoEtAll03f.pdf`.

[4] Michel Bidoit and Peter D. Mosses. CASL *User Manual.* LNCS 2900 (IFIP Series). Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[5] D. Bjorner, S. Kousoube, R. Noussi, and G. Satchok. Michael Jackson's Problem Frames: Towards Methodological Principles of Selecting and Applying Formal Software Development Techniques and Tools. In M.G. Hinchey and Liu ShaoYing, editors, *Proc. Intl.Conf. on Formal Engineering Methods, Hiroshima, Japan, 12-14 Nov.1997*, pages 263–270. IEEE CS Press, 1997.

[6] J.C. Bradfield, J. Kuster Filipe, and P. Stevens. Enriching OCL Using Observational Mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE 2002*, Lecture Notes in Computer Science. Springer Verlag, Berlin, 2002.

[7] C. Choppy and G. Reggio. Using CASL to Specify the Requirements and the Design: A Problem Specific Approach. In D. Bert and C. Choppy, editors, *Recent Trends in Algebraic Development Techniques, Selected Papers of the 14th International Workshop WADT'99*, number 1827 in Lecture Notes in Computer Science, pages 106–125. Springer Verlag, Berlin, 2000. A complete version is available at `ftp://ftp.disi.unige.it/person/ReggioG/ChoppyReggio99a.ps`.

[8] H. Gomaa. *Designing Concurrent, Distributed and Real-Time Applications with UML.* Addison-Wesley, 2000.

[9] M. Jackson. *Problem Frames: Analyzing and Structuring Software Development Problems.* Addison-Wesley, 2001.

[10] Rational. Rational Unified Process© for System Engineering SE 1.0. Technical Report Tp 165, 8/01, 2001.

[11] G. Reggio, E. Astesiano, and C. Choppy. CASL-LTL : A CASL Extension for Dynamic Reactive Systems Version 1.0– Summary. Technical Report DISI-TR-03-36, DISI – Università di Genova, Italy, 2003. Available at `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll03b.ps` and `ftp://ftp.disi.unige.it/person/ReggioG/ReggioEtAll03b.pdf`.

[12] UML Revision Task Force. *OMG UML Specification 1.3*, 2000. Available at `http://www.omg.org/docs/formal/00-03-01.pdf`.

[13] J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1999.

[14] P. Ziemann and M. Gogolla. OCL Extended with Temporal Logic. In M. Broy and A. V. Zamulin, editors, *Proc. PSI 2003*, number 2890 in Lecture Notes in Computer Science, pages 351–357. Springer Verlag, Berlin, 2003.