A UML-Based Approach for Problem Frame Oriented Software Development

Christine Choppy [a] Gianna Reggio [b]

[a]*LIPN, Institut Galilée - Université Paris XIII, France*

[b]*DISI, Università di Genova, Italy*

Abstract

We propose a software development approach that combines the use of the structuring concepts provided by problem frames, the use of the UML notation, together with our methodological approach for well-founded methods. 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. Thus we show how problem frames may be used upstream of a development method to yield an improved and more efficient method equipped with the problem frames structuring concepts.

Key words: Problem Frames, UML based development method, Requirement Specification, Design Specification, “Rich” workpieces frame

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.

Given the complexity of the systems to be developed, concepts that help structuring (and abstraction) are essential. We focus here on those that provide pre-identified structures to be reused (and/or fitted, adapted) on problems under study. Depending on their granularity, these structures may be used at various stages of development. For instance, architectural styles [29, 5] help structuring a first stage of design, while design patterns [16] may help for a more detailed design (at a finer grain) or to structure some coding.

One of the difficult issues is to start the analysis of a complex problem. M. Jackson proposes “Problem Frames” [22] which are often used structures drawn from experience, that can be used by themselves or in combination to tackle with a first 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 [22] together with some variants.

Problem frames are also presented with the idea that, once the appropriate problem frame is identified, then the associated development method should be given “for free”. For instance, the Transformation frame was initially called JSP (Jackson Structured Programming [19]), while the Information Display frame was called JSD (Jackson System Design [20]). Thus, for large classes of software problems, specific development methods could be provided. This may be seen as a good compromise between a universal general method that may not be always fully helpful, and a large number of different methods fitted to small classes of problems and that are difficult to track.

Let us also note that, since problem frames are highly beneficial when starting to work at the requirements, they can be used upstream to accompany and enhance different development methods. What is needed then is a way to link problem frames with one’s favorite development method.

Since the structuring concepts brought by problem frames seem highly valuable to help start the development effort, showing clearly the items to consider and the general tasks to do, we propose development methods associated with them. In [13] 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[6] and CASL-LTL[28]. Another example is given in [18, 11] where architectures are associated with problem frames.

Now, since the UML notation [26] 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, 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 [3, 4], it has been shown that UML may be used in a development method in a quite precise, structured and well-founded way, so as to avoid most typical problems (e.g., only a subset of UML is used and its semantics may be formally expressed).

The multiview, use-case driven and UML-based software development process of [3, 4] requires to produce a Domain Model, followed by a Requirement Specification, and a Design Specification.

The Domain Model consists of aspects of the real world that are relevant for the system to be developed for providing a solution to the problem under consideration, and it has been inspired by M. Jackson's works [21]. We propose to model the various entities present in the domain by the Static View, a UML class diagram, where the classes may be active (thus with a dynamic behaviour), and we also allow to model the autonomous features of their behaviour. Then, the most relevant cooperation among these entities may be modelled in the Work Case View that consists of workflows of a special kind, named *work cases*.

In this approach the Requirement Specification artifacts consist of different views of the system, precisely
– the Context View describes the context of the system, that is which entities (*context entities*) of which kinds may interact with the system, and in which way they can achieve that. These entities are further classified into those taking advantage of the system (*service users*), and into those cooperating to accomplish the system functionalities (*service providers*).
– the Use Case View shows the main ways to use the system (*use cases*), and which actors take parts in them. These actors are just *roles* (*generic instances*) of some context entities depicted in the Context View.
– the Internal View describes abstractly the internal structure of the system, that is essentially its Abstract State. It will help precisely describe the behaviour of the use cases, by allowing to express how they read and update it.
– the Data View lists and states clearly all data appearing in the various views of the system to help guaranteeing the consistency of the concepts used in these views.

Similarly, the Design Specification consists of different views of the designed system, precisely:
– the Data View describes the datatypes used by the entities composing the system,
– the Static View introduces the classes typing the entities used to build the system, which are of the following four different kinds: *context*, external (with regard to the system) entities that interact with it; *boundary*, entities composing the system that take care of the interaction with some context entities; *executor*, entities composing the system that perform some core system activities; *store*, entities composing the system that contain persistent data.
– the Behaviour View describes the behaviour of a class introduced in the Static View.

Here, taking advantage of the formal treatment developed for the various problem frames in [13], 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 [3, 4] and summarized above. These help reduce the development time and prevent loosing time when searching how to model the various aspects. This lead us also to discover how to properly handle, using UML, many kinds 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 [27] and COMET [17]).

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 [3, 4] we group together those corresponding to the Domain Model, to the Requirement Specification and to the Design Specification respectively, and then we show how to present them using 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 written in a quite precise or more sketchy way 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 natural language, to quickly prepare an easily readable model, whenever this is considered as precise enough. Then, one may move towards a more precise version, using OCL [31], or another formal language, e.g., close to algebraic specifications and their extensions as in [2].

In the following, we describe the development method we propose for several problem frames, precisely the Transformation (very simple just to introduce our approach), the Commanded and Required Behaviour (covering a large class of relevant applications), the Commanded Information (showing how the underlying formal modelling of the frame [13] helps devise a systematic solution), and finally the (Rich) Workpieces. Each frame will be presented by intermixing it with the application to a case study.

Let us note that, in our opinion, these problem frames cover large classes of applications. Thus, we do not try to combine them, and we do not address the issue of multiframes or of hybrid frames. As shown, e.g., for the Workpieces problem frame, we would rather adapt/enrich them, so as to use the problem frame as it is, and to keep the concepts easy to work with.

We briefly report below the notation used in [22] to present a problem frame.

For each problem frame, a *frame diagram* is set up, which contains the different parts involved. Plain rectangles denote *application domains*. The characteristics of these domains play an important role in the application of a problem frame to a problem. Jackson distinguishes *causal domains* that may control some shared phenomena (e.g., events) at the interface with another domain, *biddable domains* (people), and *lexical, or inert domains* that are physical representation of data; denoted respectively by a C, B and X in the lower right corner.

A problem frame features also a *machine domain* denoted by a rectangle with a double vertical stripe, and a *requirement* denoted by a dashed oval. The machine is always a causal domain (so an explicit C is not needed).

The lines connecting the domain represent interfaces that consist of so-called "shared phenomena". *Causal phenomena* (e.g., events) are caused or controlled by some domain, and can cause in turn other phenomena. *Symbolic phenomena* (e.g., values) can be changed, but cannot change themselves or cause changes elsewhere.

2 The Transformation Frame

2.1 The Frame

The transformation frame [22], schematically presented in Fig. 1, 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. As indicated by the X in Fig. 1, the

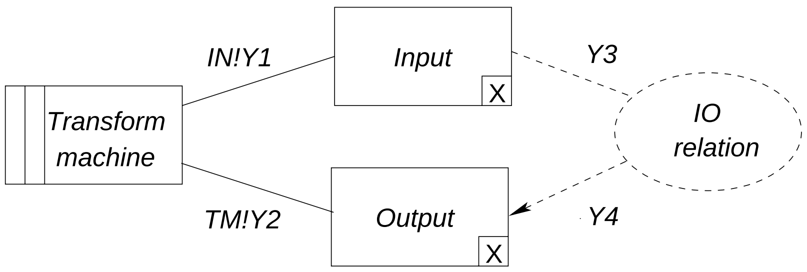


Fig. 1. The Transformation Frame

input and output domains are lexical. 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 (that is the outputs should comply to the *I/O relation*, while the inputs are given). The connections are labelled with *symbolic phenomena*, that is values. *Y1* and *Y3* (that may or may not be the same) are values of the input domain; *Y2* and *Y4* (that may or may not be the same) are values of the output domain produced by the *Transform Machine TM*.

This problem frame covers applications such as compilers, some XML tools, and format conversion applications.

2.2 Case study: Mailfiles Analysis

The goal is to analyse mailfiles, and build a report providing 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 [22]. The problem frame instance is in Fig. 2.

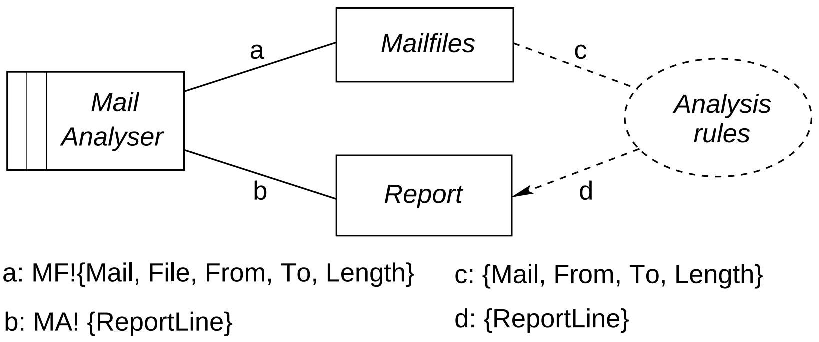


Fig. 2. Mailfiles Analysis Case Study

2.3 The UML Models (Domain Model, Requirement and Design Specification)

In the case of the transformation frame, the Domain Model and the Requirement Specification are given by a UML model containing at least the fragment shown in Fig. 3. The two classes correspond to the *Input* and *Output* of the frame and may be described using any UML construct. The requirements correspond to the association *I/O relation* that is qualified by fixing its multiplicities and adding invariant constraints to both connected classes.

The design is presented by extending the previous model with the class TransformMachine where the *transform* method is available, as shown in Fig. 4.

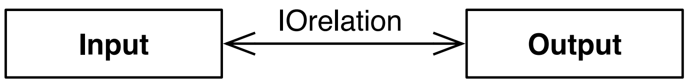


Fig. 3. Transformation Frame: Domain Model and Requirement Specification

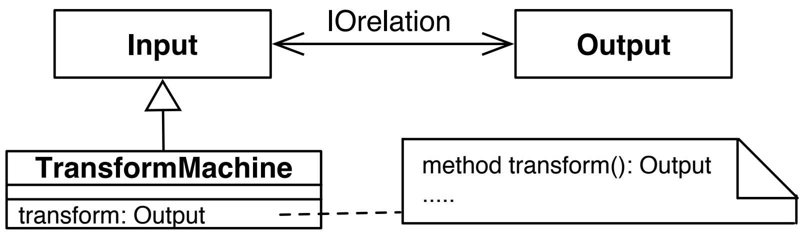
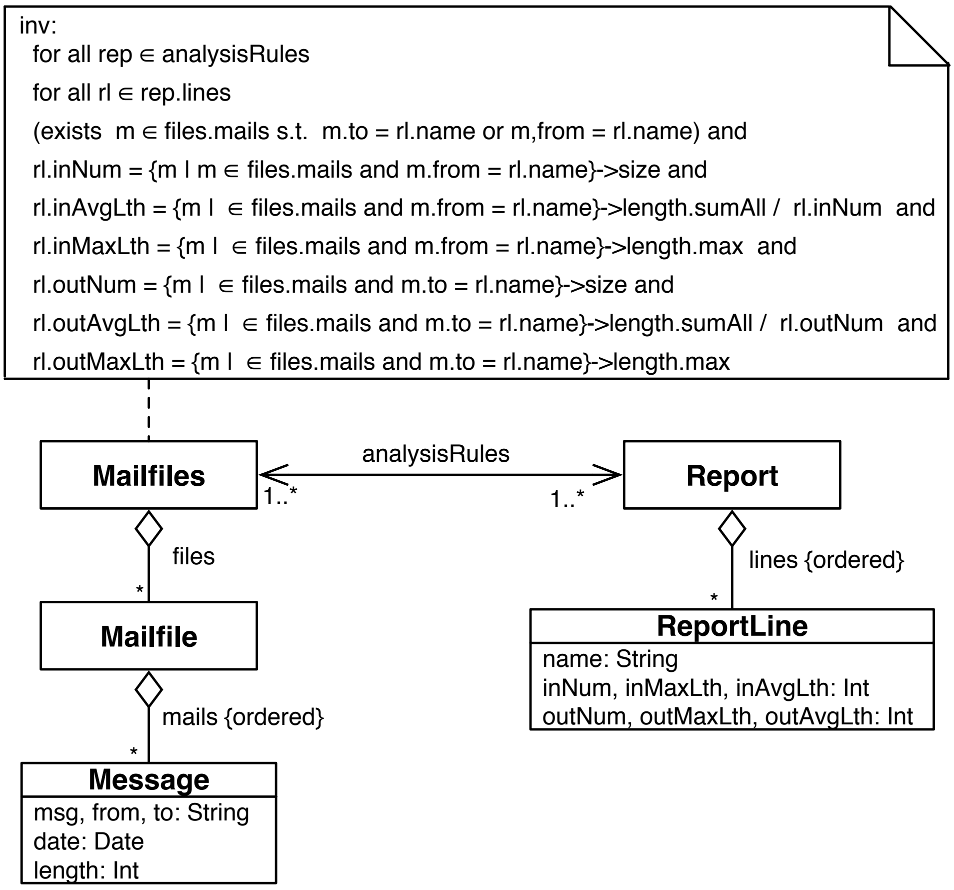


Fig. 4. Transformation Frame: Design Specification

2.3.1 The Mailfiles Analysis Case Study

The Domain Model and the Requirement Specification for the Mailfiles Analysis are given by the UML model in Fig. 5 where attributes are provided for the Mail and ReportLine classes, as well as invariants on the Mailfiles class.



```
inv:
  for all rep ∈ analysisRules
  for all rl ∈ rep.lines
  (exists  m ∈ files.mails s.t.  m.to = rl.name or m,from = rl.name) and
  rl.inNum = {m l m ∈ files.mails and m.from = rl.name}->size and
  rl.inAvgLth = {m l  ∈ files.mails and m.from = rl.name}->length.sumAll /  rl.inNum  and
  rl.inMaxLth = {m l  ∈ files.mails and m.from = rl.name}->length.max  and
  rl.outNum = {m l  ∈ files.mails and m.to = rl.name}->size and
  rl.outAvgLth = {m l  ∈ files.mails and m.to = rl.name}->length.sumAll /  rl.outNum  and
  rl.outMaxLth = {m l  ∈ files.mails and m.to = rl.name}->length.max
```

Fig. 5. Mailfiles Analysis: Domain Model and Requirement Specification

For the Design of the Mailfiles Analysis, we extend the class diagram produced in Fig. 5 introducing the description of the *Transform machine* that is the *Analyser* with the *analyse* method, as shown in Fig. A.1 of Appendix A.

3 The Commanded Behaviour Frame

3.1 The Frame

Jackson [22] describes this problem frame (sketched in Fig. 6) as follows. “There is some part of the physical world whose behaviour is to be controlled in accordance with commands issued by an operator. The problem is to build a machine that will accept the operator’s commands and impose the control accordingly.”

The C in the frame diagram indicates that the domain *Controlled domain* must be causal. The B stands for "biddable" and is used for domains that are people.

This problem frame covers applications such as embedded systems that provide user commands.

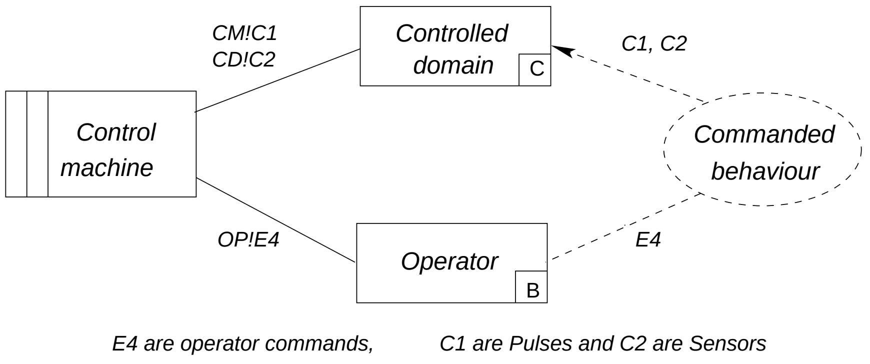


Fig. 6. The Commanded Behaviour Frame

3.2 *Case study: the Lift System*

A lift system consists of a *lift plant* (that is the cabin, the motor moving it and the doors at the various floors), some software automatically controlling the lift functioning (the *controller*), and the people using it (the *users*). The controller monitors the lift plant by means of *sensors*, which communicate the status of its various components (e.g., there is a sensor detecting the position of the cabin), and directs its behaviour by means of *orders* (e.g., it can order to open/close the doors). The lift

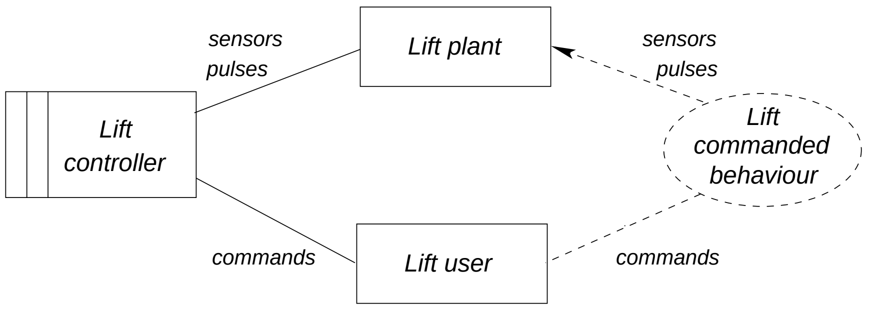


Fig. 7. Lift System Case Study

matches the Commanded Behaviour frame as shown in Fig. 7. The *Operator* is the lift user (further denoted by *User*), the *Control machine* is the lift controller (that is the software controlling the lift), and the *Controlled domain* is the *Lift plant* (that is the cabin, the motor, and the doors at the floors).

3.3 The Domain Model

3.3.1 The UML model

The domain model in the case of the Commanded Behaviour Frame describes the *Controlled domain* and is expressed by a UML model that includes the fragment given in Figure 8 (note that a UML interface - keyword ≪interface≫ - is a named set of operations, and 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). The *Controlled domain* is equipped with some

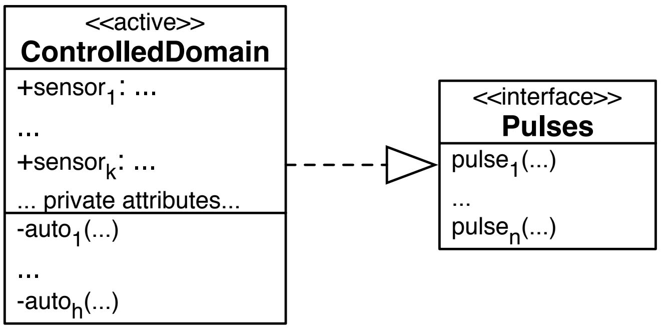


Fig. 8. Commanded Behaviour: Domain Model

sensors, which are modelled by the public attributes $\mathsf{sensor}_1, \ldots, \mathsf{sensor}_k$ (note that the fact that the sensors may break down is modelled by assuming that the corresponding attributes may contain some special values corresponding to failures), and is controlled by pulses, which are modelled by 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 self calls of the operations $\mathsf{auto}_1, \ldots, \mathsf{auto}_h$.

To describe the state of the *Controlled domain* other private attributes may be used. Moreover, some sensors may signal a value derived from other attributes. In this case the UML model should contain invariant constraints defining these sensors in terms of the other attributes.

The behaviour of the class *Controlled domain* is then modelled by a statechart (named *Controlled domain behaviour*)
– whose events are either time events or call events built by the operations pulse_1, $\ldots$, pulse_n, $\mathsf{auto}_1, \ldots, \mathsf{auto}_h$,
– and whose conditions and actions examine and update its attributes.

3.3.2 Lift Case Study: Domain Model

The *Controlled domain* in the Lift case study is the lift plant, and is modelled by the class LiftPlant presented in Fig. 9. There are sensors to detect the position of the

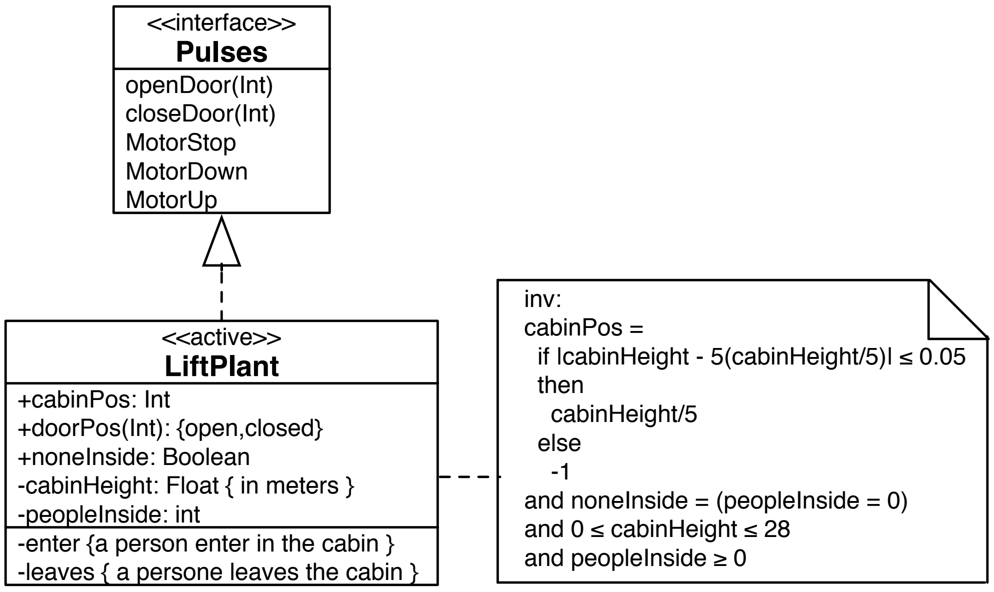


Fig. 9. Lift Case Study: Domain Model - Class Diagram

cabin (the floor number when it is at a floor, -1 when it is between two floors), the positions of the doors at the various floors (open or closed) and if there is someone inside the cabin. Two private attributes record the actual height of the cabin from the ground and how many persons are inside the cabin. The pulses may require to open or close the door at some floor, and to stop, move up or down the motor that moves the cabin.

The behaviour of the class LiftPlant is described by the statechart in Fig. 10. Note that this diagram shows that the doors can be opened/closed only when the cabin is at the corresponding floor, and that people may enter/leave the cabin only when the cabin is at some floor with open door; thus these security features will not be under the responsibility of the software controller. The fact, that following the problem frame approach, the developer has to explicitly consider and describe the existing parts of the real world interacting with the system to be developed is one of the most valuable aspect of this approach.

3.4 Requirement Specification

3.4.1 The UML Model

The requirement specification corresponds to the parts of the frame *Commanded behaviour* and *Operator*. In this case use cases are suitable to summarize the re-

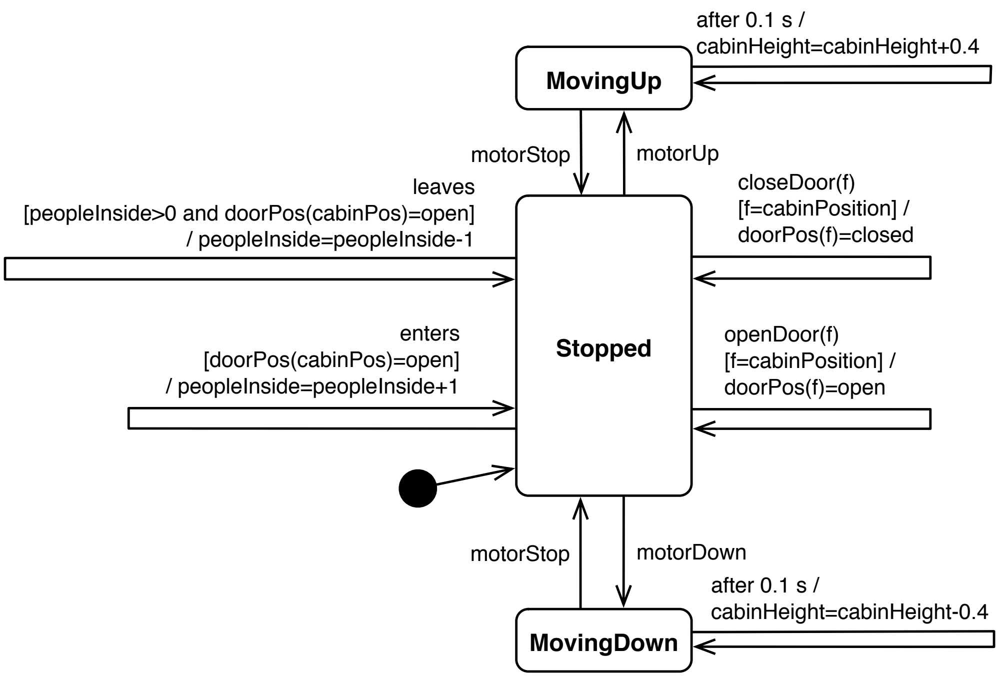


Fig. 10. Lift Case Study: Domain Model - Behaviour of LiftPlant

quirements, thus the Requirement Specification is a UML model with a use case diagram, a class diagram and various use case descriptions, one for each use case appearing in the diagram.

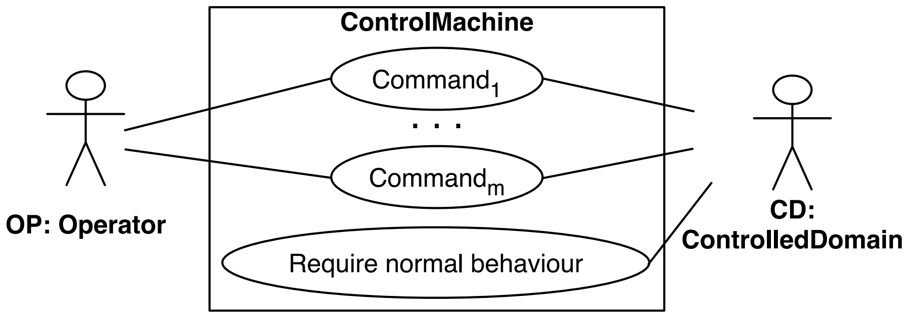


Fig. 11. Commanded Behaviour: Requirement Specification - Use case Diagram

Use Case Diagram The use case diagram for the Commanded Behaviour frame is displayed in Fig. 11. The *Control machine* is the application to develop, and the use cases Command$_1$, ..., Command$_m$ correspond to the possible commands sent by the *Operator* to the *Control machine*. The last use case, Require normal behaviour, models the fact that usually, in absence of commands from the *Operator*, the *Control machine* must ensure some behaviour. Obviously, if, in the problem under study, the *Control machine* acts only as a consequence of a command, this use case may be dropped. Recall that following [3, 4] we consider as actors all entities interacting with the application, and not only those using its services; thus also the Controlled domain is an actor (note that in [3, 4] a different icon - a parallelogram - is used for these “secondary” actors).

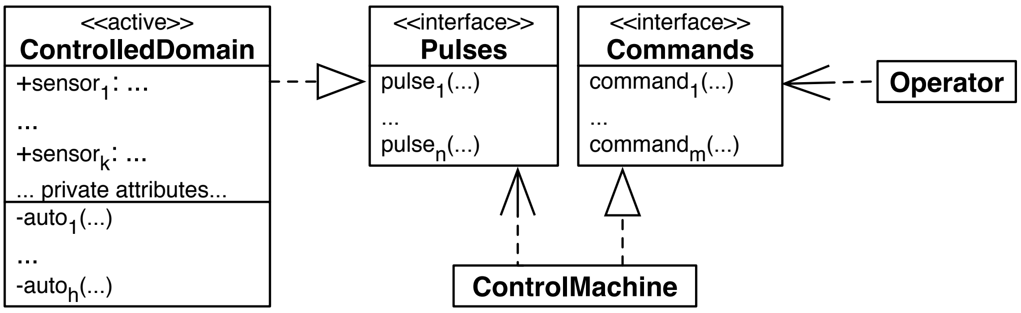


Fig. 12. Commanded Behaviour: Requirement Specification - Class Diagram

Class Diagram The class diagram, corresponding to the Context View of [3, 4], should contain the fragment reported in Fig. 12. This diagram introduces three classes modelling the three corresponding parts of the problem frame and shows which are the mutual interfaces. The ControlMachine interacts with the Controlled domain by sending it pulses (interface Pulses) and by accessing the "sensor" attributes, and with the Operator by receiving its commands (interface Commands); it is assumed that all commands are always correct and acceptable, and thus no error messages should be sent back from the *Control machine* to the *Operator* (note that variants of this frame may be developed, where commands may be refused in some situation and the *Operator* may make mistakes in sending the commands).

Use Case Description The *Commanded behaviour* may be described in terms of desired properties on the behaviour of the *Controlled domain*, whereas [22] mentions rules; 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 express 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., [10, 32]), but none 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 commanded behaviour will be modelled using properties.

On the other hand, the standard techniques to describe use cases focussing on showing the possible scenarios of interactions between the application to be developed and the actors (using natural language, or UML sequence/collaboration diagrams, or statecharts associated with the class modelling the application, as proposed by [3, 4]) are not very suitable in this case. Indeed, each use case has just a unique trivial scenario "The operator sends $\mathtt{command}_i$ to the *Control machine*, which in turns causes the *Controlled domain* to behave in the following way . . . ".

We think that, instead, a more appropriate way is to present the commanded behaviour by means of a statechart associated with the class Controlled domain (note that similarly [22] proposes on an example to use generic state machines) such that – the states are the same as in the statechart *Controlled domain behaviour* included

in the Domain Model (see Sect. 3.3),
– the events are either timed events or change events relative to 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 of external entities), they will be triggered by the change events.
Moreover, these activities consist in modifications of the internal state of the *Controlled domain*.

Whenever the commanded behaviour depends on the actual situation of the *Controlled domain* we need to give a statechart as described above for each state (of the statechart modelling the *Controlled domain*, see Sect. 3.3) in which the command has some effect.

The use case Require normal behaviour is described by statecharts as for the other use cases, modelling the behaviour that the *Control machine* must guarantee in absence of commands sent be the *Operator*, and by a set of invariant constraints that express the safety conditions that the *Control machine* must always guarantee.

3.4.2 Lift Case Study: Requirement Specification

Use case diagram The use case diagram part of the requirement specification for

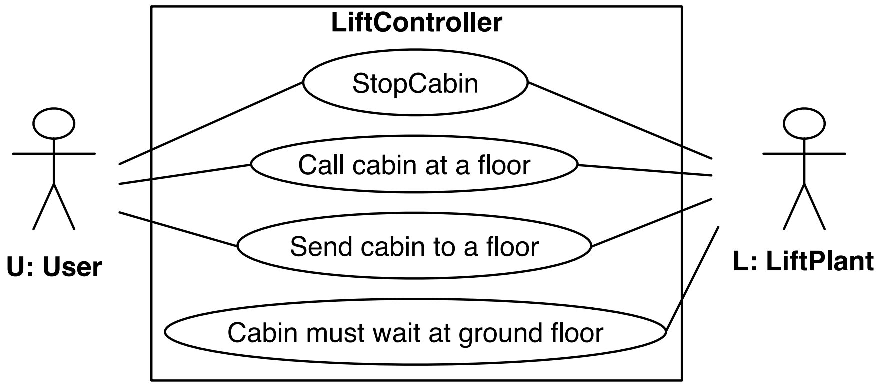


Fig. 13. Lift Case Study: Requirement Specification - Use Case Diagram

the lift case study is shown in Fig. 13. The lift user may request to stop the cabin (pressing some button inside the cabin), send the cabin to some floor pressing a button inside the cabin, and call the cabin to a floor by pressing a button at the door of this floor. When no one is using the lift, then cabin should go back to the ground floor.

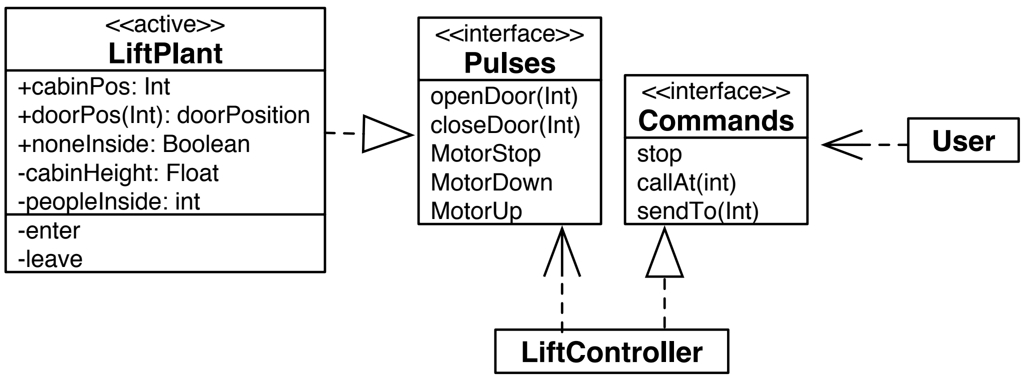
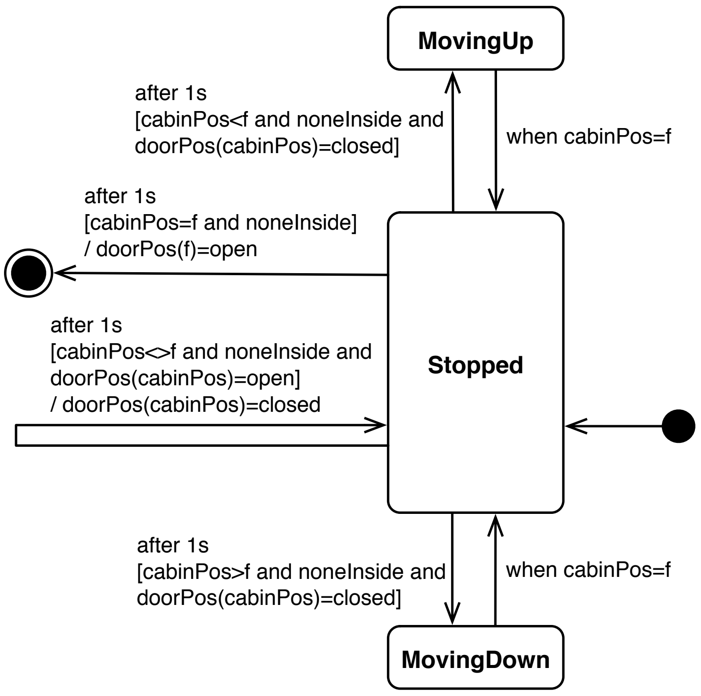


Fig. 14. Lift Case Study: Requirement Specification - Class Diagram

Class diagram The class diagram in Fig. 14 depicts the context of the lift controller (class LiftController) and how it may interact with the entities in this context (sending pulses to the *Lift plant*, reading its sensors, and receiving commands from the *User*).

Use case descriptions We provide in Fig. 15 the description for the Call the cabin at floor f use case, and the other use cases are described in the appendix B.1. The command Call the cabin at floor f is ignored whenever the cabin is not stopped, or when it is already at the called floor with open doors. Otherwise, it requires that no one is inside the cabin and the door at the floor where is the cabin must be closed, then the cabin has to move to f, and after that the door at f will be opened.



MovingUp
after 1s
[cabinPos<f and noneInside and
doorPos(cabinPos)=closed]
when cabinPos=f
after 1s
[cabinPos=f and noneInside]
/ doorPos(f)=open
after 1s
[cabinPos<>f and noneInside and
doorPos(cabinPos)=open]
/ doorPos(cabinPos)=closed
Stopped
after 1s
[cabinPos>f and noneInside and
doorPos(cabinPos)=closed]
when cabinPos=f
MovingDown

Fig. 15. Use Case Description: Call the cabin at floor f

3.5 Design Specification

3.5.1 The UML Model

The design effort for a problem matching the Commanded Behaviour Frame consists in developing the *Control machine*. In general that task may be quite complex because many different possible design choices are possible. For example, the *Control machine* may
– be a simple synchronous process that, in any cycle, reads all the sensors and sends pulses depending on the values read beforehand and on the received command;
– be a simple process that waits to receive a command, then reads the sensors and sends the appropriate pulses, and goes back to waiting for commands;
– have a complex architecture, where some components receive the commands, others read the sensors and elaborate the read values, and others determine and send the pulses.
Moroever, the commands may be processed in a sequential way (one after the other) or in parallel, and policies may be defined to solve conflicts when several different commands are received and some of them have to wait to be processed.

Thus we will provide a family of patterns in the sense of [1] for the *Control machine* design, each one based on a set of coherent design choices, and accompanied by the schematic form of the UML model needed to present it. Here we present the pattern for a simple case.

Asynchronous Simple Control Machine The design of the *Control machine* proposed by this pattern assumes that it is a simple process receiving the commands and processing them one after the other. While it is processing a command it cannot receive another one, and when it is waiting for a command to take care of it should ensure the required normal behaviour. This pattern may be applied only when the description of the use case Required normal behaviour does not include safety constraints.

In this case the *Control machine* is simply modelled by an active class the behaviour of which is defined by a statechart. Thus, the design specification is a UML model containing the fragment in Fig. 16.

The operations $\mathsf{Command}_1, \ldots, \mathsf{Command}_m$ of the class ControlMachine are those of the Commands interface, and the attribute CD refers to the actual *Controlled domain*). The statechart describes the behaviour of ControlMachine, and the subcharts enclosed in any of the sequential hierarchical states (depicted by dots in that picture) are such that
– their events are only change events on the sensors of CD and timed events (while handling a command another one cannot be received),

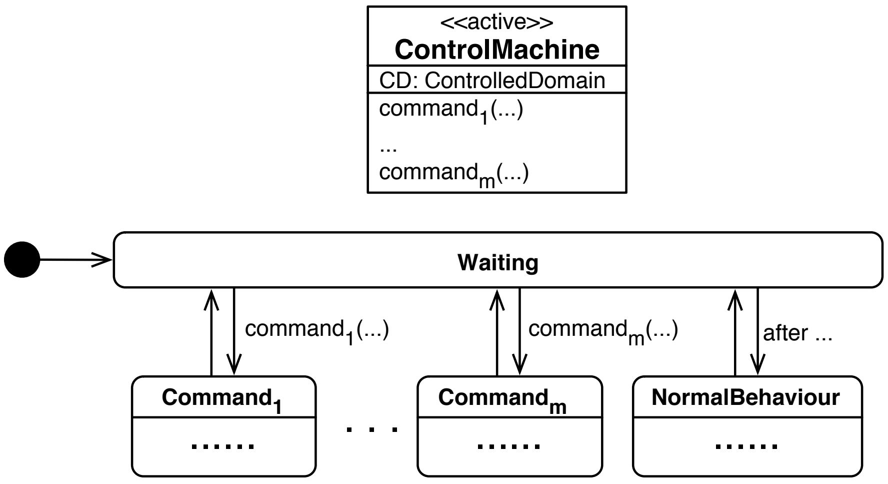


Fig. 16. Commanded Behaviour Frame: Design Specification

– their conditions are on the attributes of ControlMachine and on the sensors of CD,
– their actions may update the attributes of ControlMachine and call the operations of the Pulses interface,
– have a unique initial substate and at least one final substate (recall that a transition entering a structured state activates its initial substate, and that an unlabelled transition leaving a structured state will be fired whenever a final substate becomes active).

The *Control machine* modelled by the statechart in Fig. 16 waits for commands, when one is received it is processed sending pulses to the *Controlled domain* so as to obtain the commanded behaviour, and then goes back to wait; when no command is received for some given time, it sends pulses to the *Controlled domain* so as to obtain the behaviour required in the normal situation, and then goes back to wait.

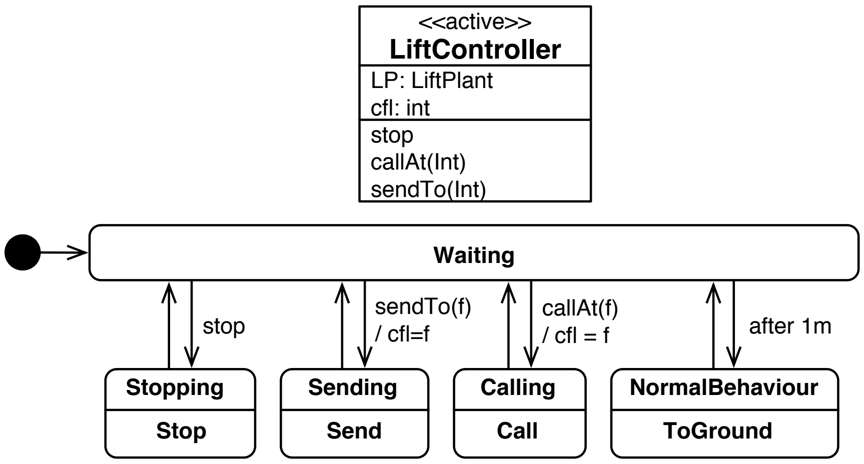


Fig. 17. Lift System Case Study: Design Specification

3.5.2 *Lift Case Study: Design Specification*

For the simple case of the lift we propose a design following the pattern "Asynchronous Simple Control", presented by a UML model in Fig. 17. The subchart Call is given in Fig. 18. The other subcharts are described in Appendix B.2.

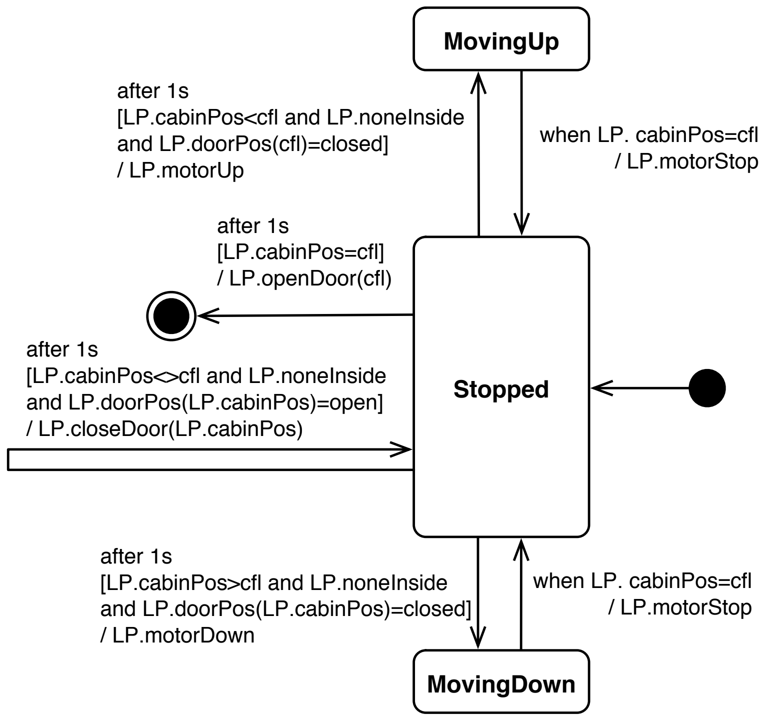

MovingUp
after 1s
[LP.cabinPos<cfl and LP.noneInside
and LP.doorPos(cfl)=closed]
/ LP.motorUp
when LP. cabinPos=cfl
/ LP.motorStop
after 1s
[LP.cabinPos=cfl]
/ LP.openDoor(cfl)
after 1s
[LP.cabinPos<>cfl and LP.noneInside
and LP.doorPos(LP.cabinPos)=open]
/ LP.closeDoor(LP.cabinPos)
Stopped
after 1s
[LP.cabinPos>cfl and LP.noneInside
and LP.doorPos(LP.cabinPos)=closed]
/ LP.motorDown
when LP. cabinPos=cfl
/ LP.motorStop
MovingDown

Fig. 18. Subchart Call

4 The Required Behaviour Frame

4.1 *The Frame*

The Required Behaviour frame, schematically shown in Fig. 19, 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." It also covers applications such as embedded systems,

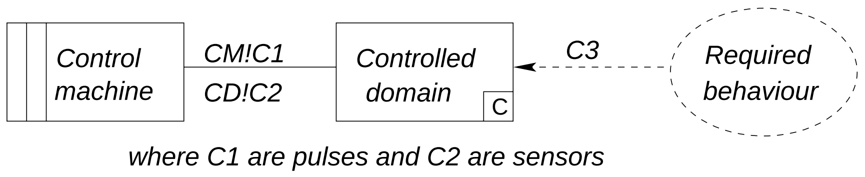


Fig. 19. The Required Behaviour Frame

but with no user commands.

This problem frame may be considered as a simplified version of the Commanded Behaviour since here there is no user. Thus, our approach for providing an associated UML description for this frame will use principles that are similar to those used for the commanded behaviour frame.

4.2 Case Study: Sluice Gate Control

A small sluice, with a rising and falling gate, is used in a simple irrigation system [22]. 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 moving clockwise CL, anticlockwise ACL, on and off pulses. There are sensors at the top and bottom of the gate travel; at the top the gate 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 in Fig. 20.

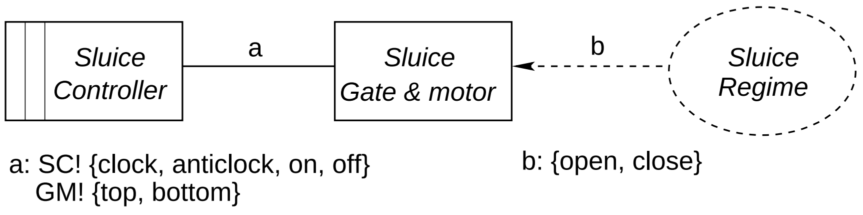


Fig. 20. Sluice Gate Control Case Study

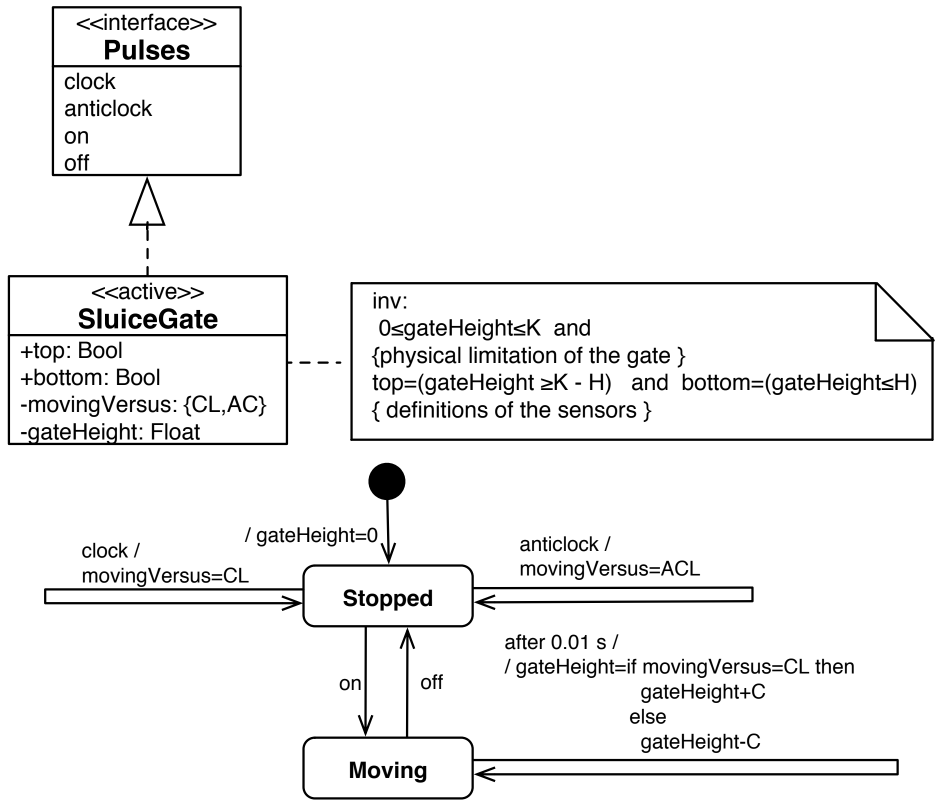
4.3 The Domain Model

4.3.1 The UML model

As for the Commanded Behaviour frame, the domain corresponds to the *Controlled domain* and is described by a UML model that includes the fragment shown in Figure 8 in Sect. 3.3.1.

The Sluice Gate Control: Domain Model

The Sluice Gate Control domain model is given in Fig. 21. Notice that in this case there are no autonomous acts. The statechart in that picture, 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 moving and stopped; whereas what is signalled by the sensors is described by the constraint in the same picture.



<<interface>>
Pulses
clock
anticlock
on
off
<<active>>
SluiceGate
+top: Bool
+bottom: Bool
-movingVersus: {CL,AC}
-gateHeight: Float
inv:
0≤gateHeight≤K and
{physical limitation of the gate }
top=(gateHeight ≥K - H) and bottom=(gateHeight≤H)
{ definitions of the sensors }
/ gateHeight=0
clock / movingVersus=CL
anticlock / movingVersus=ACL
Stopped
on
off
after 0.01 s / / gateHeight=if movingVersus=CL then gateHeight+C else gateHeight-C
Moving

Fig. 21. Sluice Gate Control: Domain Model

4.4 *Requirement Specification*

4.4.1 *The UML Model*

Use cases are not very suitable for the required behaviour. Indeed, there would just be a unique use case described by the sentence “The controlled domain must behave in the following way: . . . ”, and an implicit actor that interacts at most once with the system to start it, which would not be very interesting.

As in Sect. 3.4, we think that an appropriate way to present the required behaviour is by means of a statechart associated with the class *Controlled domain* 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 consist of modifications of the internal state of the *Controlled domain*.

4.4.2 Sluice Gate Control: 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 statechart in Fig. 22 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).

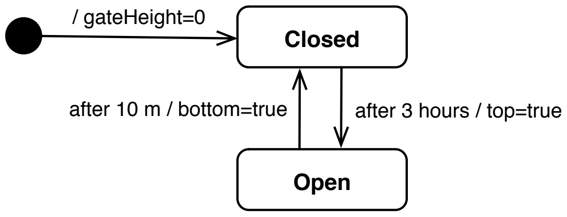


Fig. 22. Sluice Gate Control: Requirement Specification

4.5 Design Specification

4.5.1 The UML Model

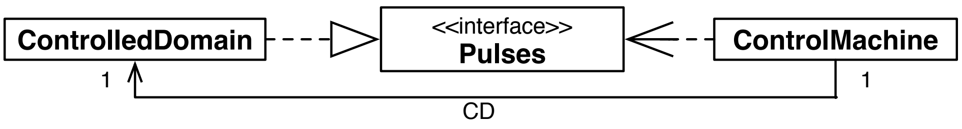


Fig. 23. Required Behaviour: Design Specification - Class Diagram

The design for the Required Behaviour Frame consists in modelling the *Control machine* by an active class, which is a process able to read the sensors of the *Controlled domain* and to send it pulses. The behaviour of the *Control machine* is modelled by a statechart. Thus, the design specification for this frame is a UML model containing the fragment in Fig. 23 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 concern its attributes and the sensors of the Controlled domain,
– its actions may update its attributes and call the operations of the Pulses interface.

The Sluice Gate Control:Design Specification

The simple *Control machine* designed in this case is presented by the UML model shown in Fig. 24, consisting of a class SluiceController and an associated statechart.

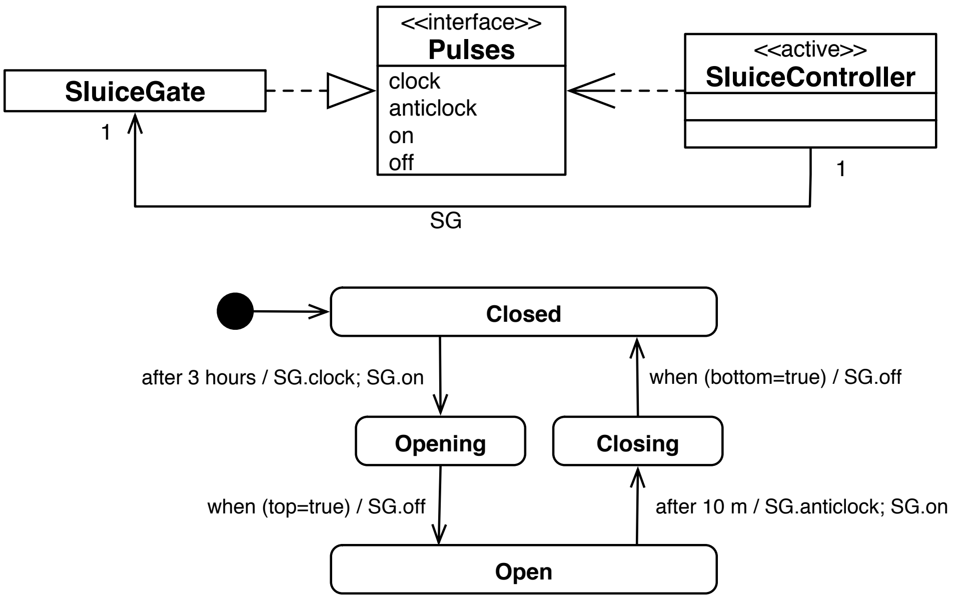
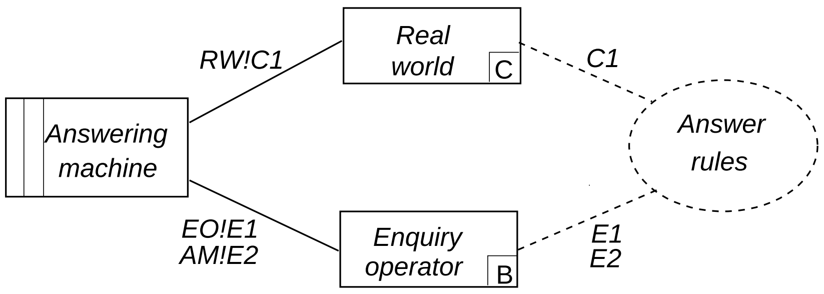


Fig. 24. Sluice Gate Control: Design Specification

5 The Commanded Information Frame

5.1 The Frame



C1 phenomena are referred to later as Events issued by the Real world
E1 are Enquiries from the Enquiry operator
E2 are Display Acts and Error Messages
displayed by the Answering Machine to the Enquiry Operator

Fig. 25. The Commanded Information Frame

This 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 in Fig. 25, is a variant of the Commanded Information Frame given in [22] 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.

5.2 *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 in Fig. 26. 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 in the *Company*, and finally the *Answering machine* is *CompanyIS* (that is the Company Information System itself).

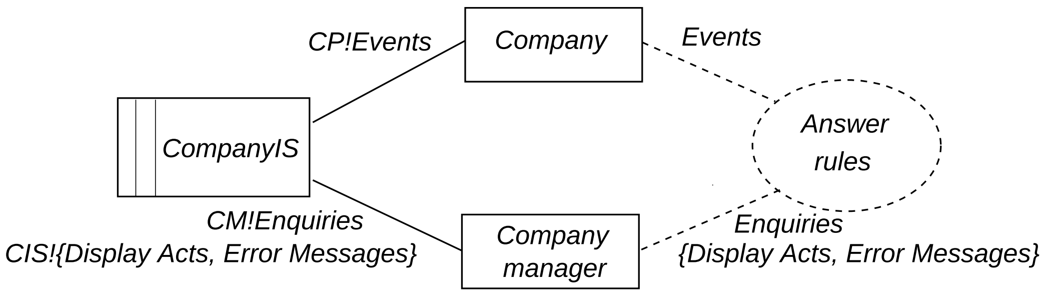


Fig. 26. Company Information System Case Study

5.3 *The Domain Model*

5.3.1 *The UML Model*

The Domain Model in this case 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.

5.3.2 *Company Information System: Domain Model*

In the case of the Company Information System the Domain Model is in Fig. 27. Notice, 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

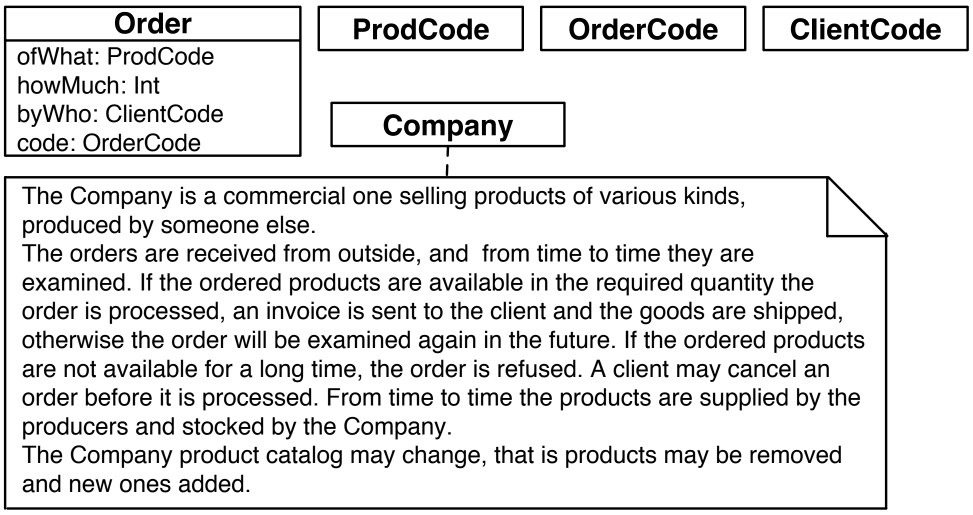

Order
ofWhat: ProdCode
howMuch: Int
byWho: ClientCode
code: OrderCode
ProdCode
OrderCode
ClientCode
Company
The Company is a commercial one selling products of various kinds, produced by someone else.
The orders are received from outside, and from time to time they are examined. If the ordered products are available in the required quantity the order is processed, an invoice is sent to the client and the goods are shipped, otherwise the order will be examined again in the future. If the ordered products are not available for a long time, the order is refused. A client may cancel an order before it is processed. From time to time the products are supplied by the producers and stocked by the Company.
The Company product catalog may change, that is products may be removed and new ones added.

Fig. 27. Company Information System: Domain Model

may be described in a more precise way by a statechart, but we think that this is not relevant to the problem of building an information system, and so we took advantage of the flexibility of the UML and opted for an informal description.

5.4 The Requirement Specification

5.4.1 The UML Model

In [13] we provided a formal specification skeleton for the various parts of the Commanded Information Frame that introduces events (that yield changes in the system state), and the history of the events that occurred, and here we follow the same basic ideas. Notice that here we assume that the interface between the *Real world* and the *Answering machine* is labelled by events generated by the *Real world* that convey information about it.

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 as in Fig. 28, where the *Answering machine* is the system, with a use case for each kind of enquiry, and one to record the events that take place in the *Real world*. Recall that following [3, 4] we consider as actors all entities that interact with the system either using its services (EO) or helping to provide services (RW).

a class diagram that corresponds to the Data View, Context View and Internal View of [3, 4], and contains the fragment shown in Fig. 29. The class RealWorld_E models a specialization of RealWorld with the capacity to signal to the *Answering machine* the relevant events when they happen (technically modelled

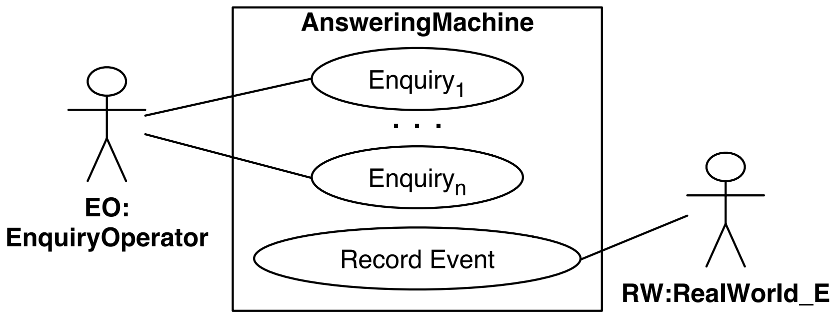


Fig. 28. Commanded Information: Requirement Specification - Use Case Diagram

by the fact that it uses the interface Signals). The events are modelled by the class Event. The class AnsweringMachine models the system to be developed (the *Answering machine* in the problem frame). The diagram in Fig. 29 shows

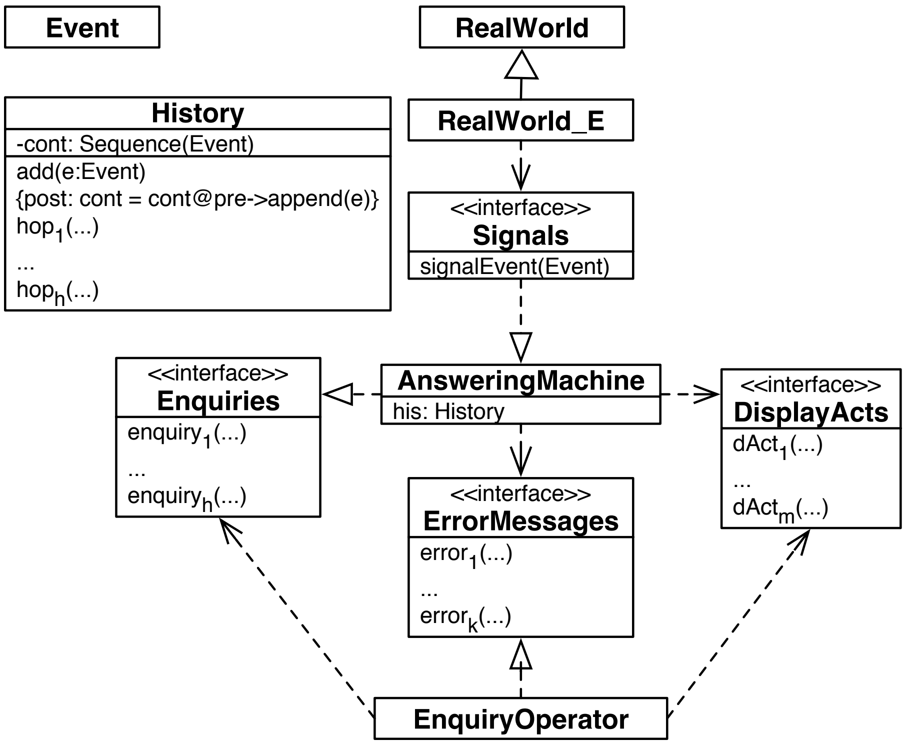


Fig. 29. Commanded Information: Requirement Specification - Class Diagram

also the context of the AnsweringMachine and its interfaces towards it. 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) as well as 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 the enquiries. This abstract state will allow to model precisely the AnsweringMachine behaviour for each use case.

other diagrams and model elements are used to describe all relevant information on the various classes in the class diagram (such as EnquiryOperator, RealWorld_E, and Event).

use case descriptions (one for each use case in the use case diagram). Following [3, 4] 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 for 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 trivial statechart reported in Fig. 30. Whereas the statechart describing a generic use case Enquiry is shown in the same picture.

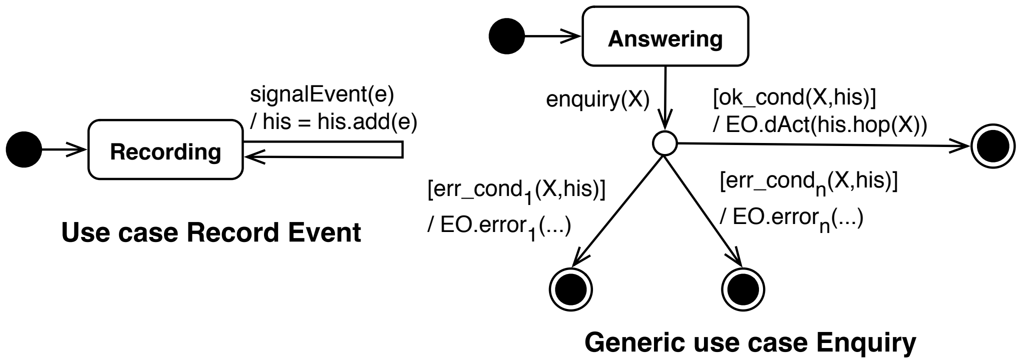
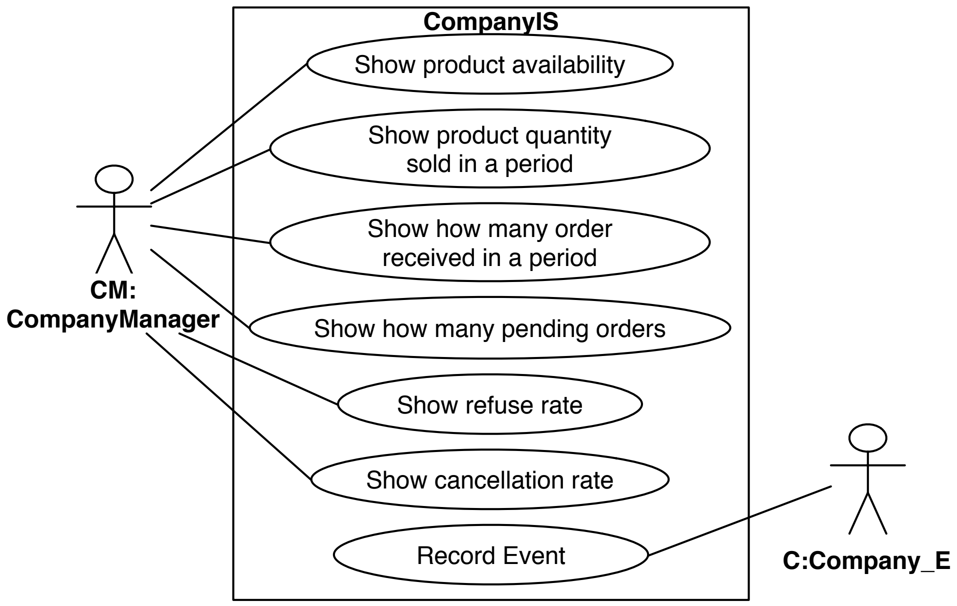


Fig. 30. Commanded Information: Requirement Specification - Use Case Descriptions

5.4.2 *Company Information System: Requirement Specification*

We first give, in Fig. 31, the use case diagram showing the pieces of information recovered through the *CompanyIS*. The class diagram (Fig. 32) is split in two parts, the upper part presents the data structures and the lower part 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* catalogue (they are needed to discover wrong enquiries concerning products not in the catalogue). Notice that we use three different ways to present the constraints defining the History operations: some generic mathematical notation (inCat), pure OCL taking advantage of its iterate construct (aQuant), and natural language (cRate). One can decide on the precision degree for the definition of these operations, but having to define them, in whatever way, leads to quite precise requirements. For example, it is clear that a product is considered as sold when its order is processed, and not when its order is received.

The second part of the class diagram shows the context entities interacting with the *CompanyIS* and its internal abstract state. Notice how this diagram models the fact that the business of the *Company* should be adapted so as to signal to the



CompanyIS
Show product availability
Show product quantity sold in a period
Show how many order received in a period
Show how many pending orders
Show refuse rate
Show cancellation rate
Record Event
CM: CompanyManager
C:Company_E

Fig. 31. Company Information System: Requirement Specification - Use case diagram

CompanyIS all the events defined above whenever they happen. How to perform this task depends on the way the *Company* business is supported by software tools; for example, many of those events may be signalled by existing software systems.

The Show product quantity sold in a period use case is described by the statechart in Fig. 33 (while the statecharts for the different use cases are provided in Appendix C.1). 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, if a sold quantity enquiry contains two errors (the product is not in the catalogue and the two dates do not make a time period) only one will be non-deterministically signalled.

5.5 *The Design Specification*

5.5.1 *The UML Model*

We propose a schematic design for the *Answering machine* following [4], which is described by a UML model consisting of:

a class diagram (the StaticView in [4]) as shown in Fig. 34. The classes Enquiry-Operator and RealWorld_E model the external entities interacting with the *Answering machine*. The design methodology presented in [4] requires to organize the architecture of the system using three kinds of classes (each one represented by a stereotype):
≪boundary≫ (to take care of the interaction of the system with the external entities),
≪store≫ (to take care of storing persistent data),

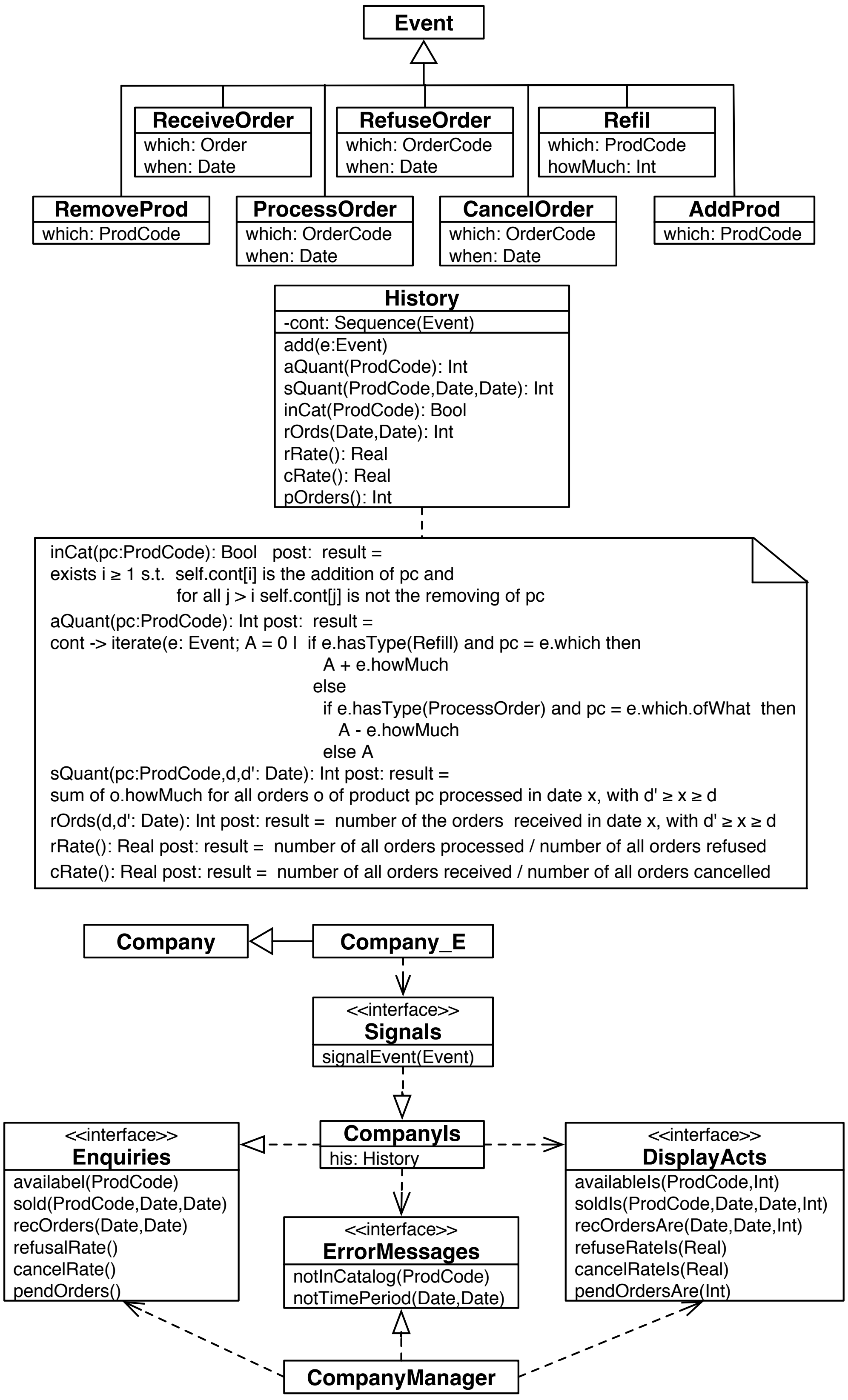
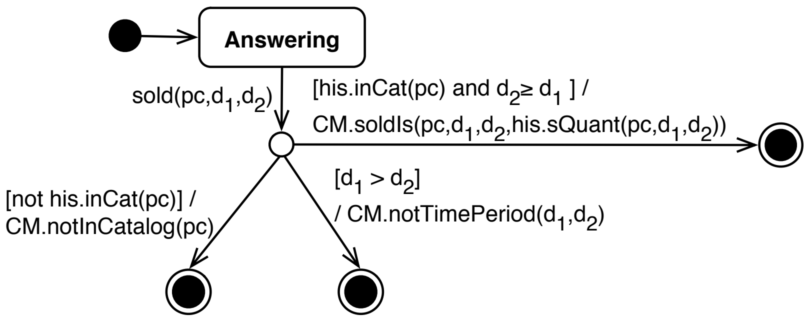

Event
ReceiveOrder
which: Order
when: Date
RefuseOrder
which: OrderCode
when: Date
Refil
which: ProdCode
howMuch: Int
RemoveProd
which: ProdCode
ProcessOrder
which: OrderCode
when: Date
CancelOrder
which: OrderCode
when: Date
AddProd
which: ProdCode
History
-cont: Sequence(Event)
add(e:Event)
aQuant(ProdCode): Int
sQuant(ProdCode,Date,Date): Int
inCat(ProdCode): Bool
rOrds(Date,Date): Int
rRate(): Real
cRate(): Real
pOrders(): Int
inCat(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
aQuant(pc:ProdCode): Int post: result =
cont -> iterate(e: Event; A = 0 | if e.hasType(Refill) and pc = e.which then
A + e.howMuch
else
if e.hasType(ProcessOrder) and pc = e.which.ofWhat then
A - e.howMuch
else A
sQuant(pc:ProdCode,d,d': Date): Int post: result =
sum of o.howMuch for all orders o of product pc processed in date x, with d' ≥ x ≥ d
rOrds(d,d': Date): Int post: result = number of the orders received in date x, with d' ≥ x ≥ d
rRate(): Real post: result = number of all orders processed / number of all orders refused
cRate(): Real post: result = number of all orders received / number of all orders cancelled
Company
Company_E
<<interface>>
Signals
signalEvent(Event)
<<interface>>
Enquiries
availabel(ProdCode)
sold(ProdCode,Date,Date)
recOrders(Date,Date)
refusalRate()
cancelRate()
pendOrders()
CompanyIs
his: History
<<interface>>
DisplayActs
availableIs(ProdCode,Int)
soldIs(ProdCode,Date,Date,Int)
recOrdersAre(Date,Date,Int)
refuseRateIs(Real)
cancelRateIs(Real)
pendOrdersAre(Int)
<<interface>>
ErrorMessages
notInCatalog(ProdCode)
notTimePeriod(Date,Date)
CompanyManager

Fig. 32. Commanded Information: Requirement Specification - Class Diagram


Answering
sold(pc,d1,d2)
[his.inCat(pc) and d2≥ d1] /
CM.soldIs(pc,d1,d2,his.sQuant(pc,d1,d2))
[not his.inCat(pc)] /
CM.notInCatalog(pc)
[d1 > d2]
/ CM.notTimePeriod(d1,d2)
Fig. 33. Use Case Show product quantity sold in a period

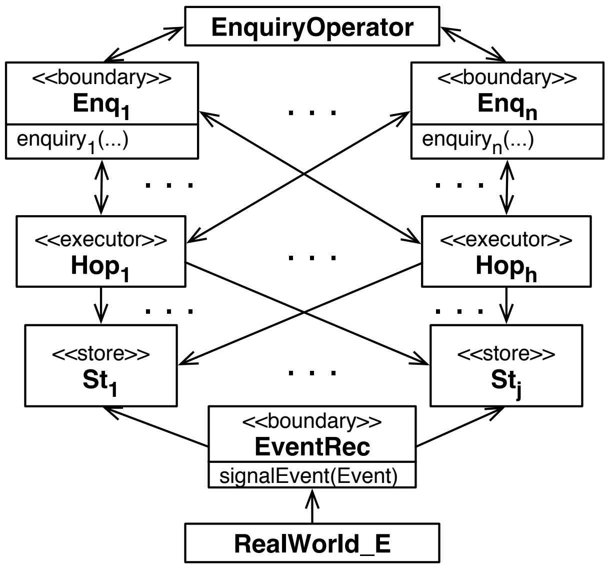


Fig. 34. Commanded Information: Design Specification - Static View

≪executor≫ (to take care of the activities to be done by the system, by communicating among them, with the boundaries and the stores).
The associations, all anonymous, 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 to conveniently keep the relevant information about the received events, a boundary to receive events from RealWorld_E, a boundary to handle each kind of enquiries, and an executor computation, using the information kept in the stores as well as 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).

5.5.2 Company Information System: Design Specification

The class diagram of the UML model corresponding to the Design Specification of the *CompanyIS* and the statechart defining the behaviour of the class SoldQuant are reported in Appendix C.2.

6 The Rich Workpieces Frame

6.1 The Frame

Rich Workpieces Frame, see Fig. 35, is described as follows. A tool is needed to allow a (single) user to create, delete, edit, browse and perform simple computations over a certain class of computer processable items (text, graphics, structured data, . . .), *the workpieces*. The problem is to build a tool that can act as this machine. Notice that in this case the structure and the meaning of the workpieces is not a responsability of the developer.

The Rich Workpieces Frame is schematically shown below. *OPERATIONS* are the operations that can be done over the workpieces to modify them, *MESSAGES* are sent back to communicate the corresponding result, and *LOOKS* are the actions that can be done over the workpieces to observe them. *SERVICES* are the services provided by the *Desktop tool* to the *User* concerning the manipulations of the workpieces.

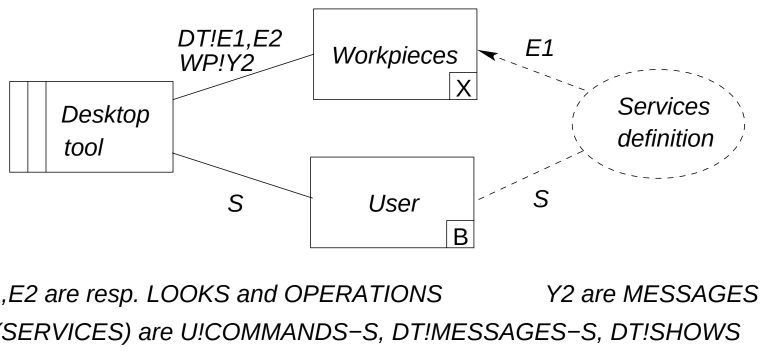

DT!E1,E2
WP!Y2
Workpieces
X
E1
Desktop tool
Services definition
S
User
B
S
E1,E2 are resp. LOOKS and OPERATIONS
Y2 are MESSAGES
S (SERVICES) are U!COMMANDS−S, DT!MESSAGES−S, DT!SHOWS

Fig. 35. The Rich Workpieces Frame

The services *SERVICES* are shared phenomena that cannot by classified using the kinds considered in the Jackson's book [22]: events, causal and lexical. Indeed, they correspond to complex interactions between two domains consisting of successive elementary phenomena, which we name *composing phenomena*. We name this new kind of phenomena *service* and use *S* to denote them. Since the initiators of the composing phenomena of a service may be different, shared phenomena of kind service do not have an initiator. The set of the composing phenomena of a service with their initiators give its *signature*.

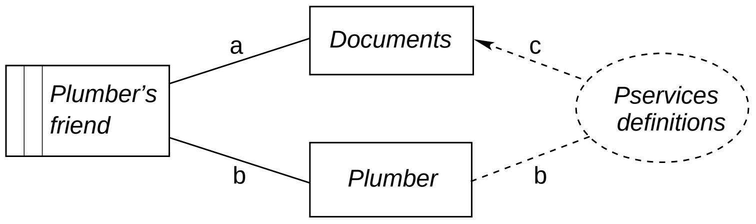
The signature of any service *S* belonging to *SERVICES* has the following form: U!*COMMANDS-S* (commands sent to the *Desktop tool* by the *User*), DT!*MESSAGES-S* (messages sent by the *Desktop tool* to the *User* about the results of the received commands, including error messages) and DT!*SHOWS-S* (communications from the *Desktop tool* to the *User* of information concerning the workpieces as result of received commands).

Many common development problems match this frame, among them we can recall text editors and word processors, tools supporting the use of visual modelling notations, such as the UML, tools to support the clerical work concerning handling simple documents, but only when the form of the documents and the way they are handled on the computer is already defined.

We named this frame "rich" since it considers a variant of the problem to handle computer-based workpieces that is quite more complex than the simple workpieces frame presented in the Jackson book [22]. This frame shows also that we follow an approach to handle complex problems sligthly different from the original one of Jackson [22]. Indeed, [22] presents a collection of elementary frames which have to be composed to get a schema to encompass the (real, usually complex) problem under examination. Our approach here is to propose to develop a collection of quite complex frames that directly match many real applications.

6.2 Case Study: Plumber's Friend

We have to develop a small application to support a plumber in handling the documents needed for his job, which are essentially the invoices to present to the clients (initially to approve a job and at the end to require the payment), and the records about the clients and the parts used for the various jobs (such as tubes, faucets, . . .). In this case, all those documents are stored in a relational database already existing. This problem matches the Rich Workpieces Frame, since it just concerns the handling of the documents and not their definition. Fig. 36 shows how the Plumber's Friend matches the Rich Workieces Frame. The workpieces in this case are the doc-



a: PF!{{SQLSelect}, {SQLInsert, SQLUpdate}} c: {SQLSelect}

b: PSERVICES = {AddClient, AddPart, CreateInvoice, ComputeBudget, ChangeClientAddress, ChangePartPrice, UpdateInvoice}

Fig. 36. Plumber's Friend Case Study

uments handled by the plumber, precisely the invoices, and the records containing the data about the clients and the parts. Moreover, since they are stored in a relational database, the way to look at them is by executing SQL select queries, and to use the SQL insert and update commands to modify them. No messages are sent back as result of performing these SQL commands, and so *MESSAGES* is empty, and not reported in the picture. The signature of the various services (that their composing commands, messages and shows) will be given when presenting their definitions.

6.3 The UML Models

Domain Model

The Domain Model for the Rich Workpieces case is just the *Workpieces*, and it will be described by a UML model containing a class diagram including at least the fragment shown in Fig. 37. The class Workpiece describes the considered work-

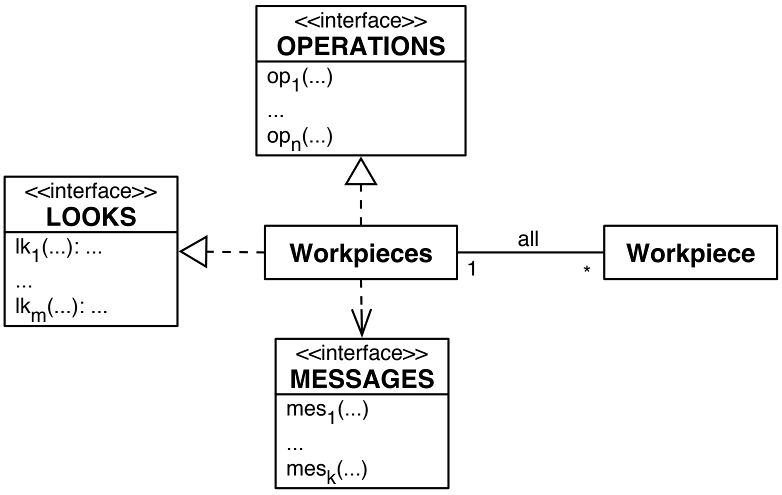


Fig. 37. Rich Workpieces Frame: Domain Model

pieces; if they are of different kinds, then it will have several specializations, one for each kind of workpieces. This class diagram may be complemented with other classes needed for the workpieces description. Moreover, the UML model may contain also other elements needed to define the operations of the various classes, e.g., constraints. The class Workpieces has been added to allow an easy way to define using UML, and thus in an object-oriented way, the elements of *OPERATIONS* and of *LOOKS*, as operations of that class; the association all allows to find all the instances of Workpiece representing existing workpieces. The three interfaces define the elements of the three corresponding parts of the frame.

Plumber's Friend: Domain Model

The Domain Model for the case of the Plumber's Friend describes the documents that he needs, precisely client and part records and invoices. An invoice consists of various lines, where a line may be just a comment, or the indication of parts used for the work or the indication of consumed labour hours.

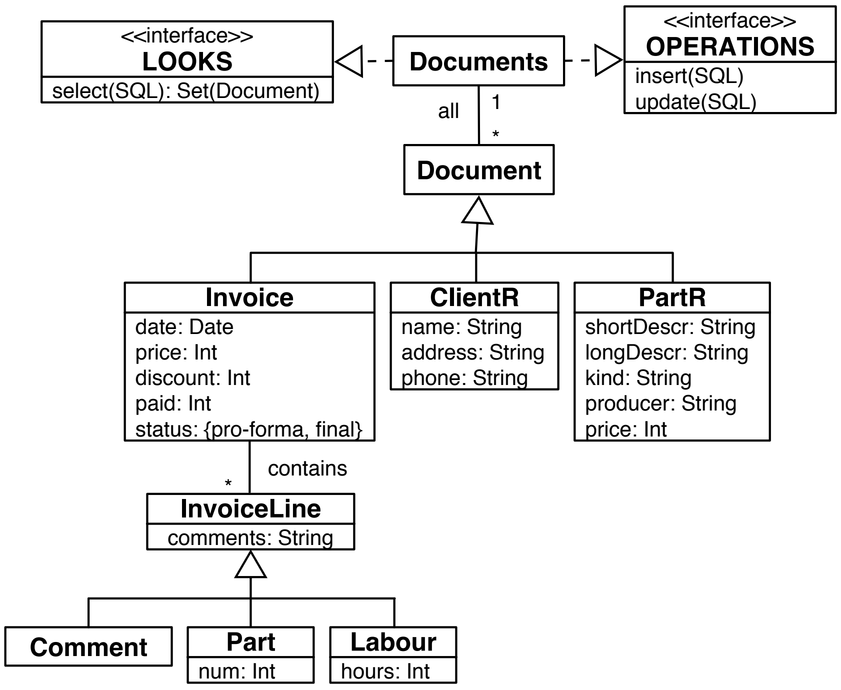


Fig. 38. Plumber's Friend: Domain Model

Requirement Specification

The *Services definitions* part of the Reach Workpieces Frame corresponds to the requirements, and so it is sensible to have use cases to specify them. Recall that to define the various services we need also to define their signature, that is the composing commands, shows and user messages. The Requirement Specification, corresponding to the *Services definitions* and to their signatures is then a UML model consisting of:

a use case diagram as in Fig. 39, where DesktopTool is the system, and with a use case for each service in *SERVICES*. Recall that following [3, 4] we consider as actors all entities interacting with the system either using its functionalities (the user U) or helping functionalities to fulfill the goals (the workpieces WPS).

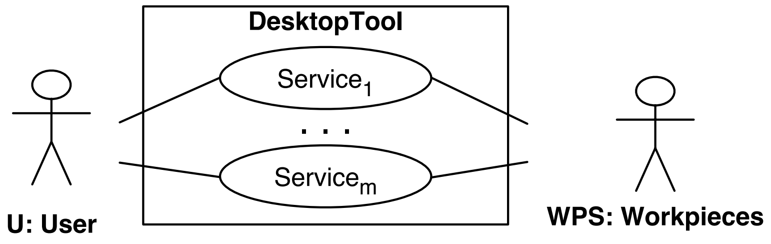


Fig. 39. Rich Workpieces Frame: Requirement Specification - Use Case Diagram

a class diagram corresponding to the Data View, Context View and Internal View of [3, 4], containing the fragment shown in Fig. 40. The class DesktopTool mod-

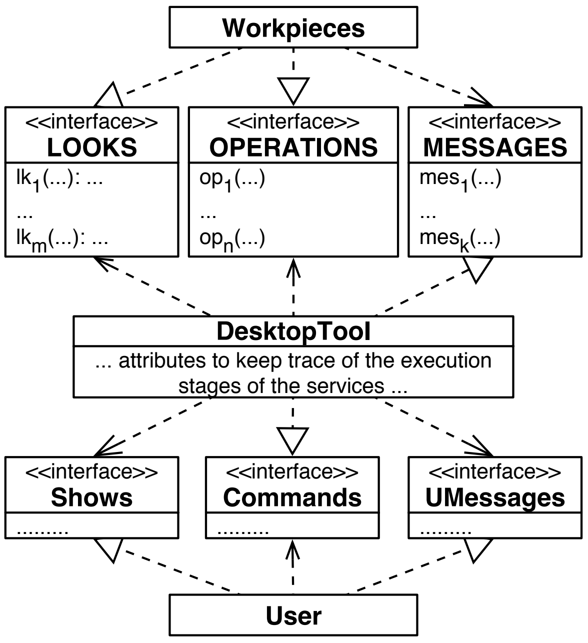


Fig. 40. Rich Workpieces Frame: Requirement Specification - Class Diagram

els the system to be developed (the *Desktop tool* in the problem frame).

This diagram shows also which is the context of the *Desktop tool* and its interfaces towards it. The *Desktop tool* interacts with the *Workpieces* by calling the operations acting on them and receiving the messages about the result of their execution, and by looking at their contents (through the "looks" commands), and interacts with the *User* by receiving commands (interface Commands) and by sending messages resulting from their execution (interface UMessages), and by showing information about the workpieces (interface Shows). The last three interfaces Shows, Commands and UMessages and are determined by the phenomena composing the various services in *SERVICES*.

use case descriptions (one for each use case in the use case diagram). Following [3, 4] we describe a use case by means of a statechart for the class modelling the system (here DesktopTool), which shows the complete behaviour of the system during that use case, including all interactions with the actors (in this case the user and the workpieces). The events of those state machines are event calls for operations in the realized interface Commands, whereas the action parts may contain calls of the operations of the (used) interfaces Looks (to examine the contents of the workpieces) and UMessages and Shows (to send messages and show information to the user), and statements corresponding to modifications of the attributes of the class DesktopTool. To abstractly model the effects on the workpieces required by the use case service we use invariant constraints for the class DesktopTool of the form *ST.isActive => cond(WP)*, where *ST* is a state of the statechart and *cond(WP)* a condition on the workpieces, which require that when state *ST* becomes active the condition *cond(WP)* must hold. To improve the readability a constraint of that form will be depicted as a note containing *cond(WP)* attached to the state *ST*. Recall that [3, 4] uses a special variation of the semantics of the statecharts: they represent fragments of possible lives of the instances of the context class and not the set of all their complete lives.

Notice that the above restrictions on the form of the state machine prevents to use in the definition of the services

– calls of the operations acting over the workpieces (interface Operations), and consequently receptions of the corresponding messages (interface Messages); in this way the requirements abstractly consider the effects of the services (and of their commands) on the workpieces without having to explicitly describe which operations are used and when they will be realized.

– time events, and this is quite reasonable, since time should not be relevant for this kind of applications.

– change events concerning the workpieces, thus modifications in the workpieces caused from other sources cannot be considered.

other diagrams and model elements are used to describe all relevant information on the classes appearing in the class diagram and to present additional views of the use cases, such as sequence diagrams.

Plumber's Friend: Requirement Specification

In Fig. 41 we show the use case diagram for the Plumber's friend with a use case for each service provided by the application. For lack of room here we detail in Fig. 42

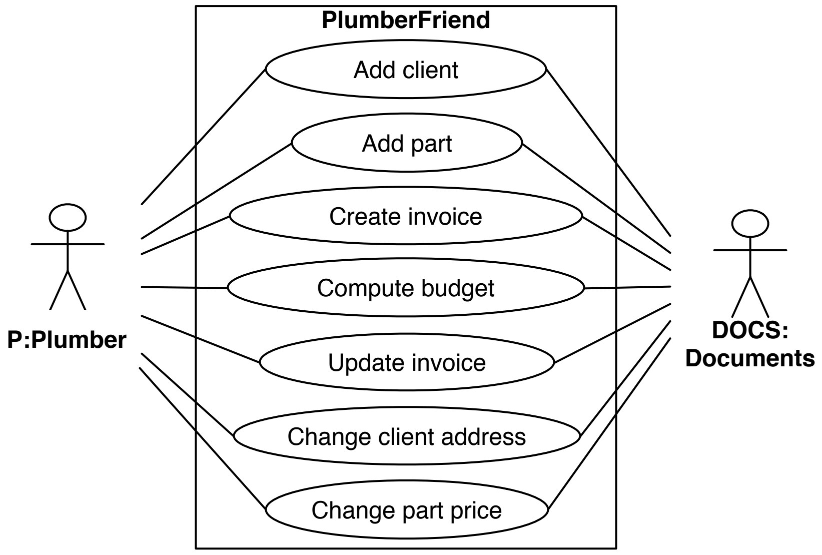


Fig. 41. Plumber's Friend: Requirement Specification - Use Case Diagram

the definition of only one use case (service), precisely Create invoice; thus in that picture we present the class diagram showing only the commands and the shows which are part of Create invoice. Similarly, only the attributes of PlumberFriend needed to define the use case/service Create invoice are shown. The state machine

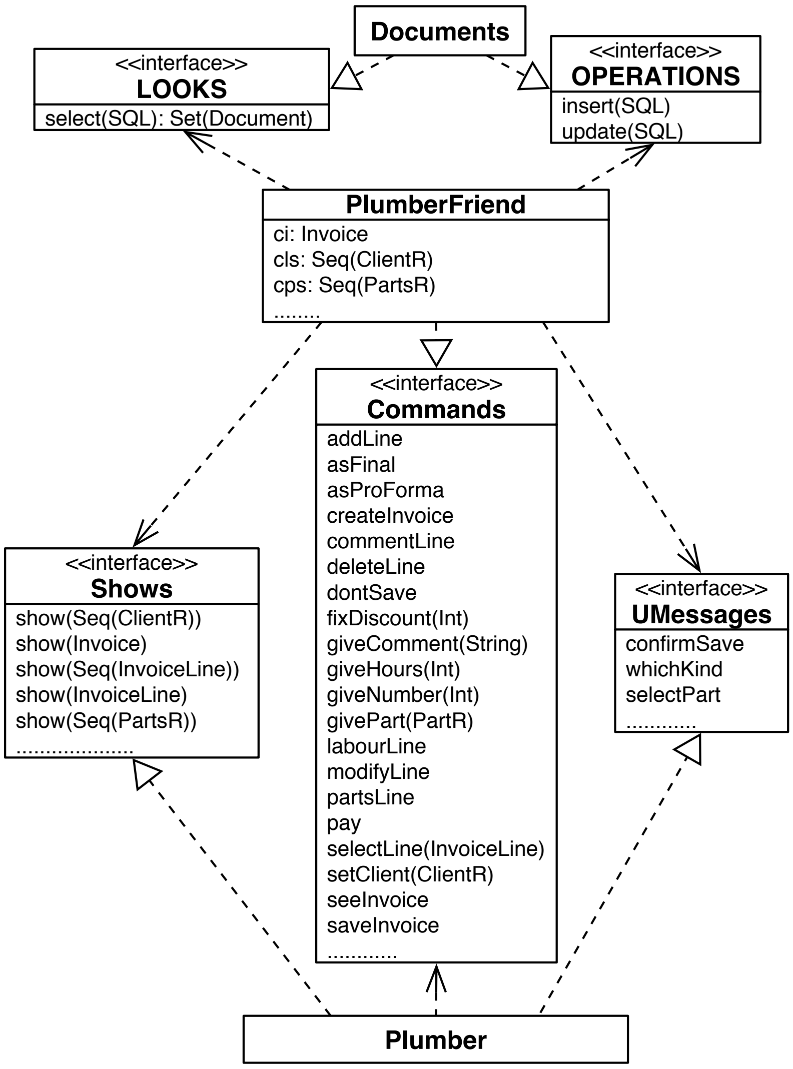


Fig. 42. Use Case (Service) Create invoice

describing Create invoice is in Fig. 43, where the subcharts (DeleteLine, ModifyLine and AddLine) are defined in Appendix D.

Design Specification

We propose a schematic design for the *Desktop tool* following [4], which is described by a UML model consisting of:

a class diagram (the StaticView in [4]) having the form shown in Fig. 44. The suggested architecture for the *Desktop tool* requires a boundary taking care of the access to the workpieces using the operations, messages and looks specified before, and for each service a boundary for handling the interaction with the user and an executor for performing the computations over the workpieces required

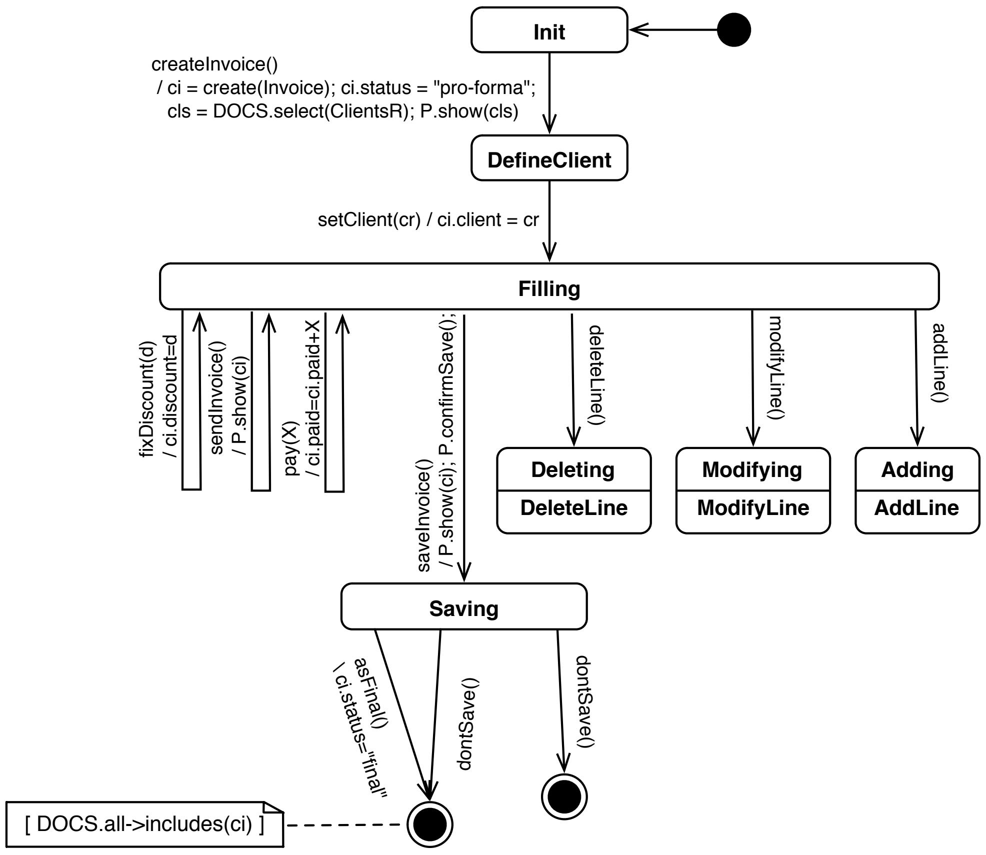


Fig. 43. Use Case Create invoice - statechart

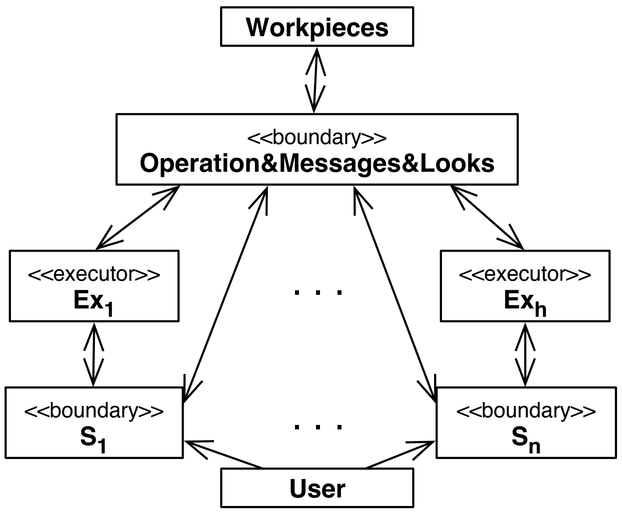


Fig. 44. Rich Workpieces Frame: Design Specification

by the service;

other diagrams and constraints to describe the behaviour of the various classes in the above diagram (e.g., statecharts for the boundaries).

Note that, since the precise understanding of the problem embedded in the frame makes clear that the storing and the organization of the workpieces is not a responsability of the developed application, the *Desktop tool*, the suggested design for the *Desktop tool* does not use classes of stereotype ≪store≫.

For lack of room, we do not present here a design for the Plumber's Friend case.

7 Conclusions

In this paper we have presented a software development approach for systems fitting problem frames 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 the problem frames provide a first overall structure for problems, our method shows, for each development phase, how to use appropriate UML constructs.

As mentioned in the introduction, we think problem frames are very good at providing a first requirement structure that is invaluable to start the analysis of a problem and understand its nature. The basic problem frames that are provided exhibit most common problem structures. However it may be necessary to adapt them to fit a problem. Even though, the process of trying to match a problem frame helps to characterize the problem under study, and this is usually not addressed in development methods that provide "general" solutions [17, 27]. Problem frames provide a means to reuse experience that is helpful to start a complex problem analysis with some structuring concepts in mind.

In this paper, we showed how to accompany problem frames with a UML description in a methodological way. We chose UML because it is widespread, and also because we think using problem frames may improve significantly available methods for UML. We also hope that providing problem frame based methodology for UML will positively promote the use of problem frames within the UML community. To our knowledge, our approach, proposing a development method for the combined use of problem frames and UML, and aiming at covering a wide range of applications represented by the various (more or less "basic") problem frames, is original.

Lavazza and Del Bianco [23] do not aim at a development method per se, but they provide a description of commanded and required behaviour problem frames in UML-RT focussing on active objects or "capsules" communicating through ports (defined by protocols), and they provide a real time version of OCL called OTL.

Another direction is to associate descriptions in other languages than UML e.g., formal specification languages, as we did in [13] to provide CASL and CASL-LTL specifications for the JSP problem frame and the IS frame. D. Bjorner in [7] provides an expression using the Z specification language, and Nelson et al. in [25] provide a description in the Alloy language.

In [14], taking advantage of our experience in [13] we developed the Commanded Behaviour problem frame associated UML method, while in [15] we gave a short glimpse of our treatment for the Transformation, Commanded Information, and Required Behaviour. In this paper, we can provide a full and more complete view of what we propose for these problem frames, and we introduce a “rich” workpieces frame, that in our opinion covers an interesting range of applications, and we provide an associated UML development.

We think that, with respect to the use of standard UML-based methods, our method should be 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.

Let us note that our precise form of the various statecharts used for domain, requirement and design may be at the basis of methods to help guarantee correctness and other good properties we plan to investigate.

We did not formally address yet the issue of validation for this method. However, we used it and consider that the resulting description obtained with our combined use of problem frames and UML is significantly better than when we used a general method alone. We also think it helps students produce more adequate models.

The issue of whether problem frames might be used in conjunction with other development methods (KAOS [30], Agile methods [9], . . .) so as to add its requirement value may be subject to future work. While we developed this method to produce UML descriptions, we think its principles are reusable for other target languages, and also may be easily adapted to other (basic) frames such as Information Display.

Aiming at covering a wide range of applications, we would like to explore our approach on web-based applications (considering, e.g., frames as given in [8]).

Up to now our approach has been to look for problem frames general enough to cover a valuable class of applications. However, Jackson [22] always mentioned that decomposition and recomposition are needed on complex problems.

An alternative way to handle this is proposed in [12], where decomposition is based on the different use cases. The level of decomposition of use cases should be so as to be able to match a problem frame. Then, along ideas also developed in [11], an architectural style is associated with the problem frame, so as to move on to the design phase. Recomposition is then achieved through a component based approach.

However, we think the decomposition/recomposition with problem frames raises interesting issues, as shown in [24] for web applications, and we would like to explore how our approach could provide some support for this.

Acknowledgements The authors thank the anonymous referees and the editors for their helpful and constructive comments.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language.* Oxford University Press, 1977.

[2] 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.

[3] 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.

[4] 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.

[5] L. Bass, P. Clements, and R. Kazman. *Software Architecture in Practice.* Addison-Wesley, 1998.

[6] M. Bidoit and P. D. Mosses. CASL *User Manual*, number 2900 in Lecture Notes in Computer Science (IFIP Series). Springer Verlag, Berlin, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.

[7] 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. Hinchey and L. ShaoYing, editors, *Proc. Intl.Conf. on Formal Engineering Methods, Hiroshima, Japan, 12-14 Nov.1997*, pages 263–270. IEEE Press, 1997.

[8] S. J. Bleistein, K. Cox, and J. Verner. Problem Frames Approach for e-Business Systems. In K.Cox, J. Hall, and L. Rapanotti, editors, *Proc. 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF)*. IEEE Press, 2004.

[9] B. Boehm and R. Turner. *Balancing Agility and Discipline: A Guide for the Perplexed.* Addison-Wesley, 2003.

[10] J. Bradfield, J. K. Filipe, and P. Stevens. Enriching OCL Using Observational Mu-Calculus. In R.-D. Kutsche and H. Weber, editors, *Proc. FASE 2002*,

number 2306 in Lecture Notes in Computer Science, pages 203–217. Springer Verlag, Berlin, 2002.

[11] C. Choppy and M. Heisel. Use of patterns in formal development: Systematic transition from problems to architectural designs. In M. Wirsing, R. Hennicker, and D. Pattinson, editors, *Recent Trends in Algebraic Development Techniques, 16th WADT, Selected Papers*, number 2755 in Lecture Notes in Computer Science, pages 205–220. Springer Verlag, Berlin, 2002.

[12] C. Choppy and M. Heisel. Une approche à base de "patrons" pour la spécification et le développement de systèmes d'information. In *AFADL 2004, Approches Formelles dans l'Assistance au Développement de Logiciels*, 2004.

[13] 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`.

[14] C. Choppy and G. Reggio. A UML-Based Method for the Commanded Behaviour Frame. In K. Cox, J. Hall, and L. Rapanotti, editors, *Proc. of the 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF 2004)*, pages 27–34. IEEE Press, 2004.

[15] C. Choppy and G. Reggio. Using UML for Problem Frame Oriented Software Development. In W. Dosch and N. Debnath, editors, *Proc of the ISCA 13th Int. Conf. on Intelligent and Adaptative Systems and Software Engineering (IASSE-2004)*, pages 239–244. The International Society for Computers and Their Applications (ISCA), 2004.

[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley, Reading, 1995.

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

[18] J. G. Hall, M. Jackson, R. C. Laney, B. Nuseibeh, and L. Rapanotti. Relating Software Requirements and Architectures using Problem Frames. In *Proceedings of IEEE International Requirements Engineering Conference (RE'02)*, Essen, Germany, 9-13 September 2002.

[19] M. Jackson. *Principles of Program Design*. Academic Press, 1975.

[20] M. Jackson. *System Development*. Prentice Hall, 1986.

[21] M. Jackson. *Software Requirements & Specifications: a Lexicon of Practice, Principles and Prejudices*. Addison-Wesley, 1995.

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

[23] L. Lavazza and V. D. Bianco. A UML-Based Approach for Representing Problem Frames. In K.Cox, J. Hall, and L. Rapanotti, editors, *Proc. 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF)*. IEEE Press, 2004.

[24] Z. Li, J. G. Hall, and L. Rapanotti. Reasoning about decomposing and re-

composing Problem Frames developments: a case study. In K.Cox, J. Hall, and L. Rapanotti, editors, *Proc. 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF)*. IEEE Press, 2004.

[25] M. Nelson, T. Nelson, P. Alencar, and D. Cowan. Exploring problem-frame concerns using formal analysis. In K.Cox, J. Hall, and L. Rapanotti, editors, *Proc. 1st International Workshop on Advances and Applications of Problem Frames (IWAAPF)*. IEEE Press, 2004.

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

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

[28] 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`.

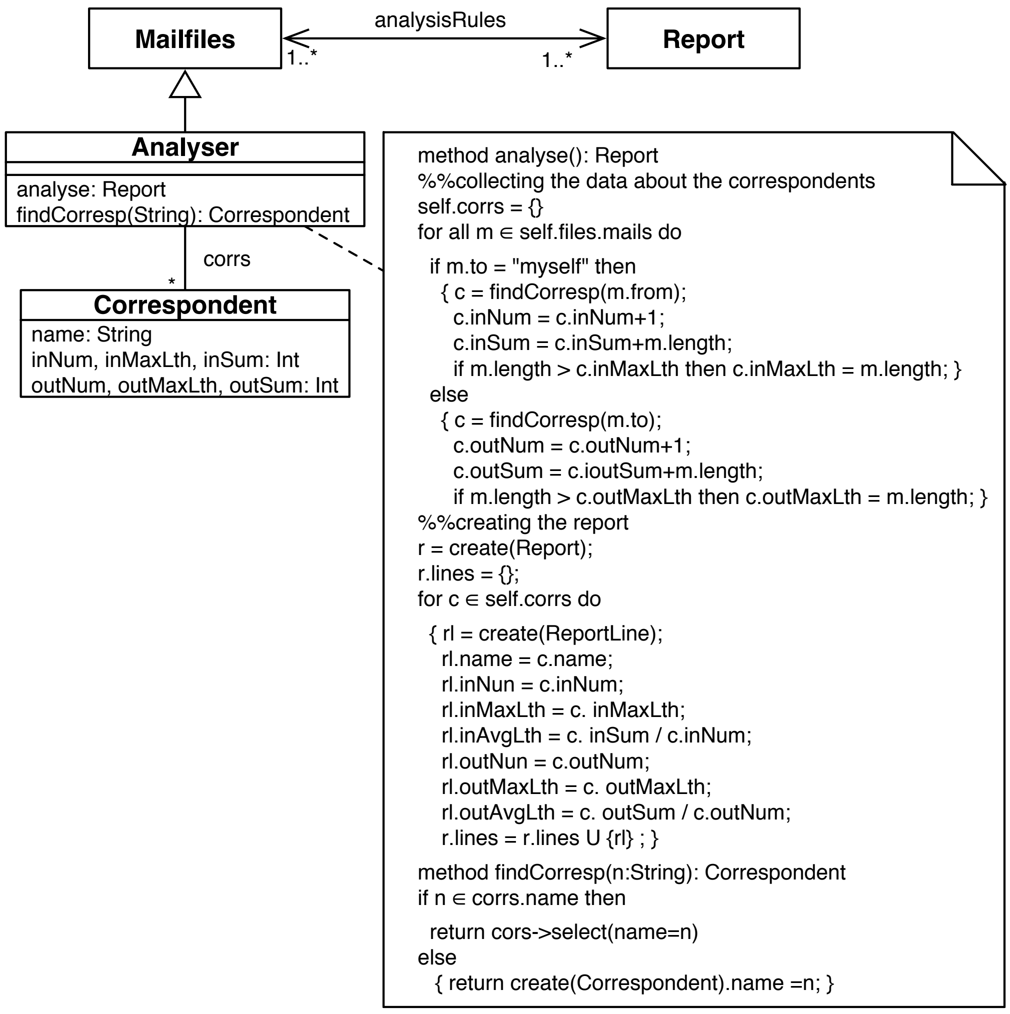
[29] M. Shaw and D. Garlan. *Software Architecture. Perspectives on an Emerging Discipline*. Prentice-Hall, 1996.

[30] A. van Lamsweerde. Goal-Oriented Requirements Engineering: A Roundtrip from Research to Practice. In *Proceedings of RE'904, 12th IEEE Joint International Requirements Engineering Conference, Kyoto, Sept. 2004, 4-8 (Invited Keynote Paper)*, 2004.

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

[32] 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.

A The Mailfiles Analysis Case Study: Design Specification


Mailfiles
analysisRules
Report
1..*
1..*
Analyser
analyse: Report
findCorresp(String): Correspondent
corrs
*
Correspondent
name: String
inNum, inMaxLth, inSum: Int
outNum, outMaxLth, outSum: Int

```
method analyse(): Report
%%collecting the data about the correspondents
self.corrs = {}
for all m ∈ self.files.mails do
  if m.to = "myself" then
   { c = findCorresp(m.from);
     c.inNum = c.inNum+1;
     c.inSum = c.inSum+m.length;
     if m.length > c.inMaxLth then c.inMaxLth = m.length; }
  else
   { c = findCorresp(m.to);
     c.outNum = c.outNum+1;
     c.outSum = c.ioutSum+m.length;
     if m.length > c.outMaxLth then c.outMaxLth = m.length; }
%%creating the report
r = create(Report);
r.lines = {};
for c ∈ self.corrs do
  { rl = create(ReportLine);
    rl.name = c.name;
    rl.inNun = c.inNum;
    rl.inMaxLth = c. inMaxLth;
    rl.inAvgLth = c. inSum / c.inNum;
    rl.outNun = c.outNum;
    rl.outMaxLth = c. outMaxLth;
    rl.outAvgLth = c. outSum / c.outNum;
    r.lines = r.lines U {rl} ; }
method findCorresp(n:String): Correspondent
if n ∈ corrs.name then
  return cors->select(name=n)
else
  { return create(Correspondent).name =n; }
```

Fig. A.1. Mailfiles Analysis: Design Specification

B The Lift Case Study (Commanded Behaviour Frame)

B.1 Use Case descriptions

Stop the cabin (Fig. B.1) The stop command requires that
– if the cabin is moving, it must reach the nearest floor and then the corresponding door is opened (recall cabinPosition = -1 when the cabin is between two floors),
– if, instead, it is already stopped at a floor, then the door at that floor must be opened.

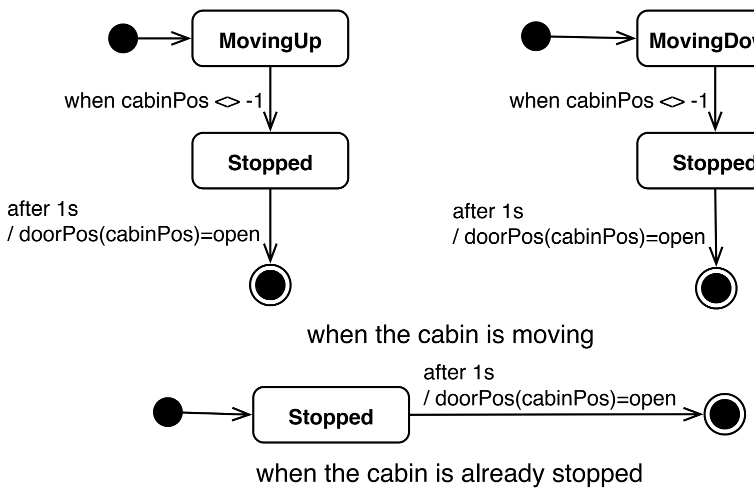


Fig. B.1. Stop the cabin Use Case Description

Send the cabin to floor f (Fig. B.2) This command is ignored whenever the cabin is not stopped or when it is already stopped at f with open door. Otherwise, it requires that the door at the floor where is the cabin must be closed, then the cabin has to move to f, and after that the door at f will be opened. There is a little subtle difference with the previous use case (the former requires that no one is inside the cabin before it can start to move), but the proposed method allows to precisely describe both of them.

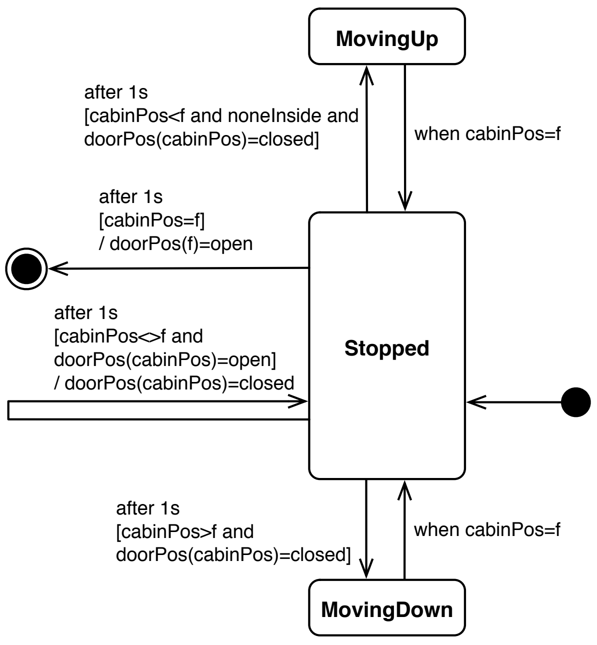


Fig. B.2. Send the cabin to floor f Use Case Description

The cabin must wait at ground floor (Fig. B.3) When no one is using the lift for 1 minute, the cabin must be sent to the ground floor and the door at that floor opened.

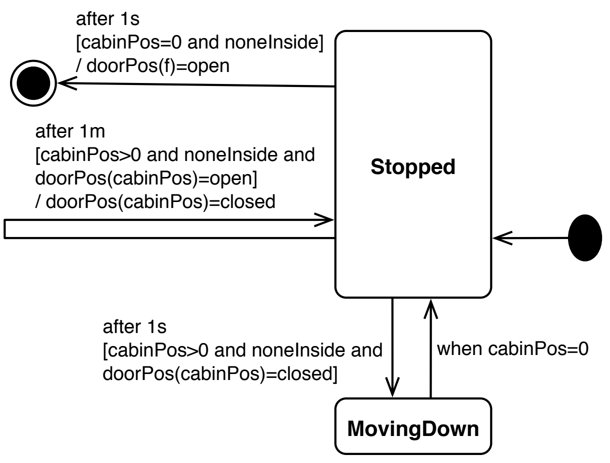

after 1s
[cabinPos=0 and noneInside]
/ doorPos(f)=open
after 1m
[cabinPos>0 and noneInside and
doorPos(cabinPos)=open]
/ doorPos(cabinPos)=closed
Stopped
after 1s
[cabinPos>0 and noneInside and
doorPos(cabinPos)=closed]
when cabinPos=0
MovingDown

Fig. B.3. The cabin must wait at ground floor Use Case Description

B.2 Design Specification: Subcharts

In Fig. B.4, B.5 and B.6 we report the subscharts of the statechart of Fig. 17.

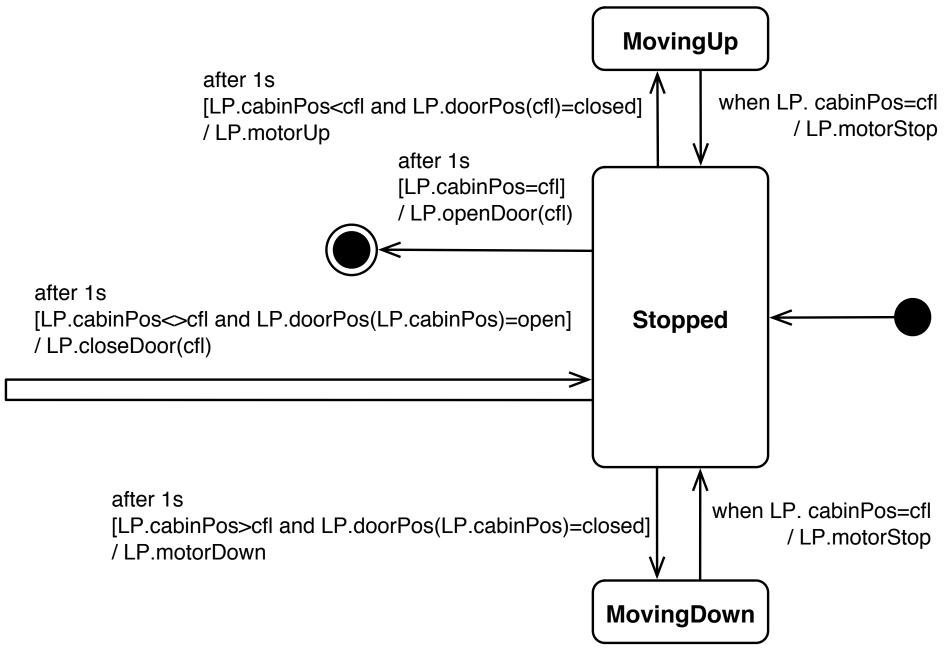


Fig. B.4. Subchart Send

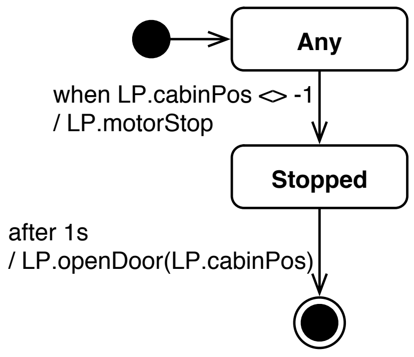


Fig. B.5. Subchart Stop

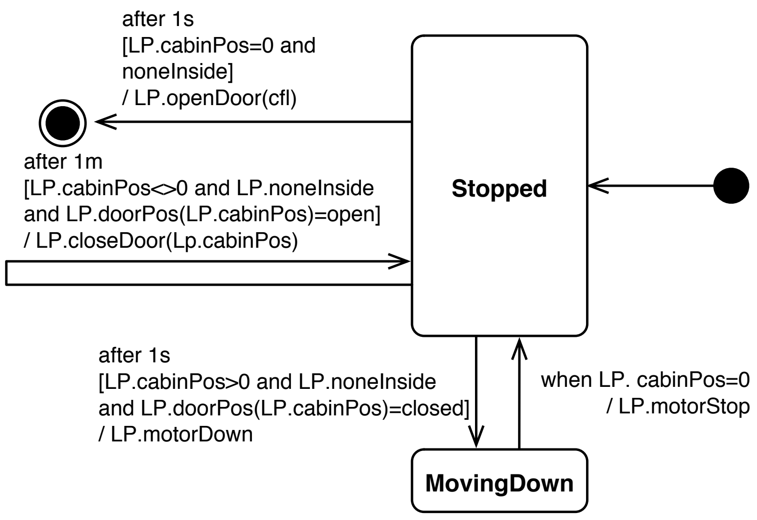

after 1s
[LP.cabinPos=0 and
noneInside]
/ LP.openDoor(cfl)
after 1m
[LP.cabinPos<>0 and LP.noneInside
and LP.doorPos(LP.cabinPos)=open]
/ LP.closeDoor(Lp.cabinPos)
Stopped
after 1s
[LP.cabinPos>0 and LP.noneInside
and LP.doorPos(LP.cabinPos)=closed]
/ LP.motorDown
when LP. cabinPos=0
/ LP.motorStop
MovingDown

Fig. B.6. Subchart ToGround

C The Company Information System Case Study

C.1 The Company Information System Requirement Specification

In Section 5.4.2, the use case diagram and the class diagrams (Fig. 32) are provided together with the statechart for the use case Show product quantity sold in a period. Fig. C.1 shows a complete picture for the different use case statecharts.

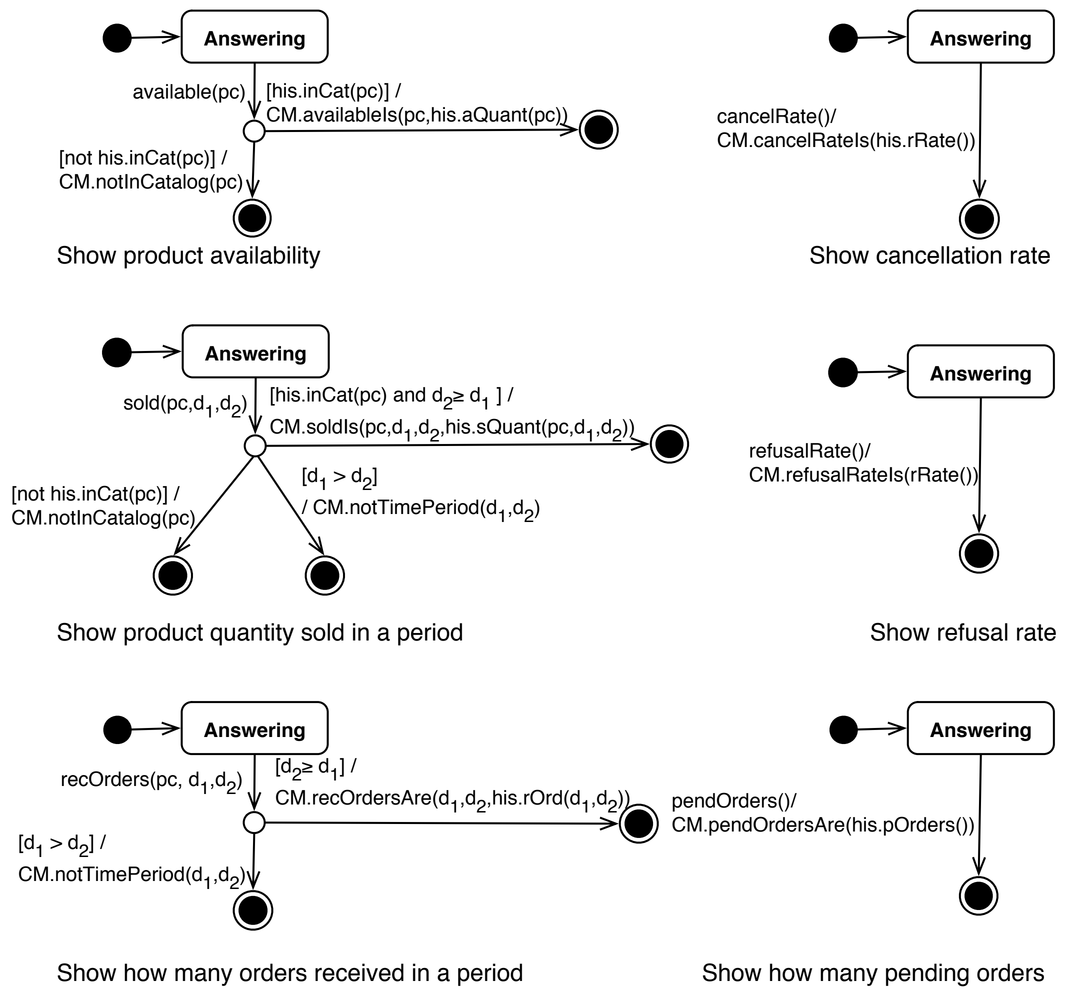

Answering
available(pc)
[his.inCat(pc)] /
CM.availableIs(pc,his.aQuant(pc))
[not his.inCat(pc)] /
CM.notInCatalog(pc)
Show product availability
Answering
cancelRate()/
CM.cancelRateIs(his.rRate())
Show cancellation rate
Answering
sold(pc,d1,d2)
[his.inCat(pc) and d2≥ d1] /
CM.soldIs(pc,d1,d2,his.sQuant(pc,d1,d2))
[not his.inCat(pc)] /
CM.notInCatalog(pc)
[d1 > d2]
/ CM.notTimePeriod(d1,d2)
Show product quantity sold in a period
Answering
refusalRate()/
CM.refusalRateIs(rRate())
Show refusal rate
Answering
recOrders(pc, d1,d2)
[d2≥ d1] /
CM.recOrdersAre(d1,d2,his.rOrd(d1,d2))
[d1 > d2] /
CM.notTimePeriod(d1,d2)
Show how many orders received in a period
Answering
pendOrders()/
CM.pendOrdersAre(his.pOrders())
Show how many pending orders

Fig. C.1. Company Information System: Requirement Specification - Use Case Descriptions

The class diagram of the UML model corresponding to the Design Specification of the *CompanyIS* is reported in Fig. C.2. Notice that in this case, since the operation

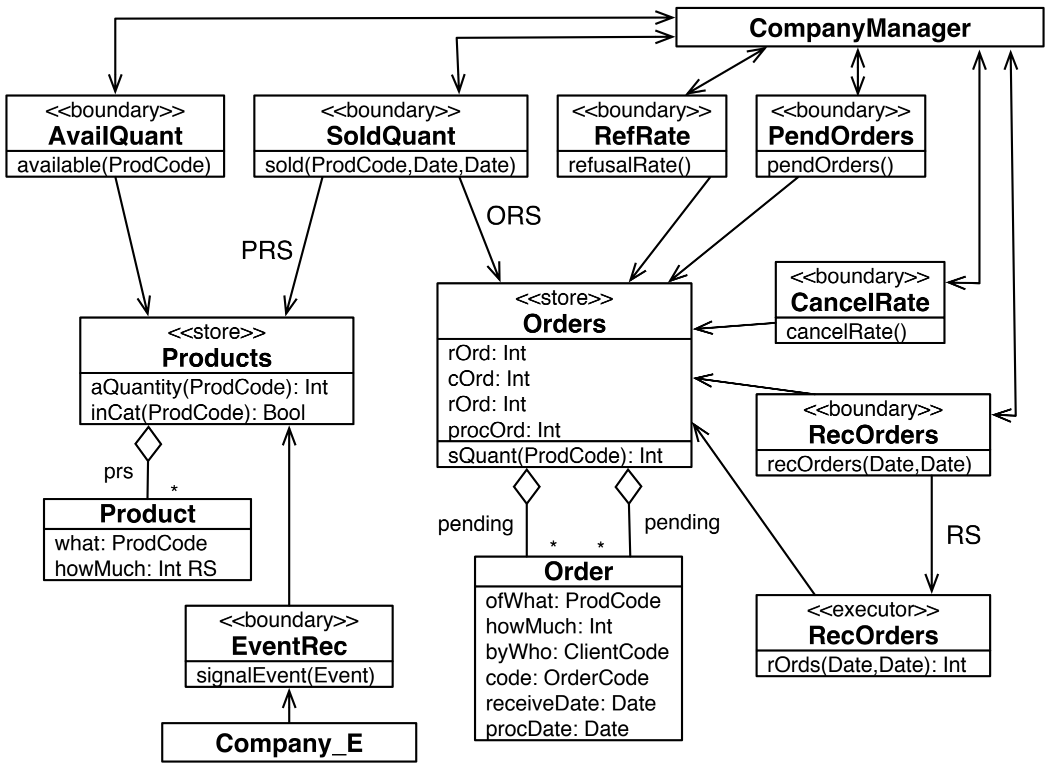


Fig. C.2. Company Information System: Design Specification - Class Diagram

to be computed on the stores are quite simple, the architecture of the design system is simplified by dropping trivial executors, thus in this case boundaries directly call stores operations. In Fig. C.3 we show the statechart defining the behaviour of the class SoldQuant. The other classes behaviours are omitted because they are similar.

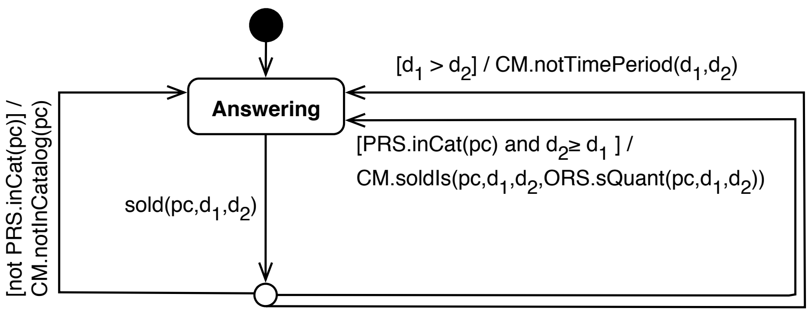

Answering
[d1 > d2] / CM.notTimePeriod(d1,d2)
[PRS.inCat(pc) and d2≥ d1] /
CM.soldIs(pc,d1,d2,ORS.sQuant(pc,d1,d2))
sold(pc,d1,d2)
[not PRS.inCat(pc)] /
CM.notInCatalog(pc)

Fig. C.3. Behaviour of SoldQuant

D The Plumber’s Friend case study: Requirement Specification

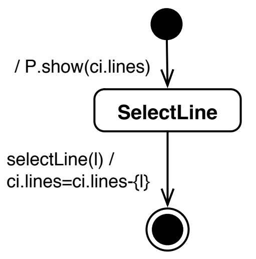


Fig. D.1. Subchart DeleteLine

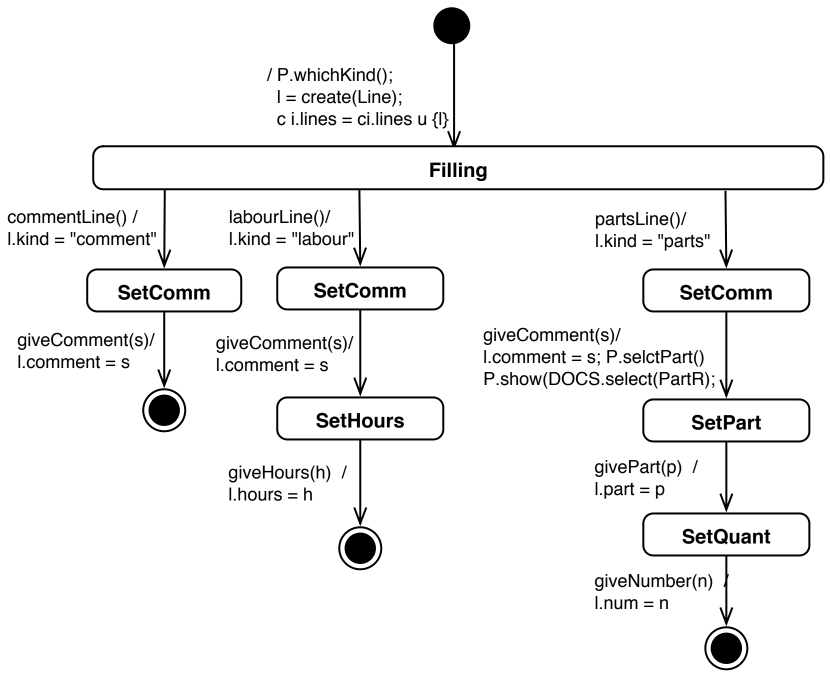


Fig. D.2. Subchart AddLine

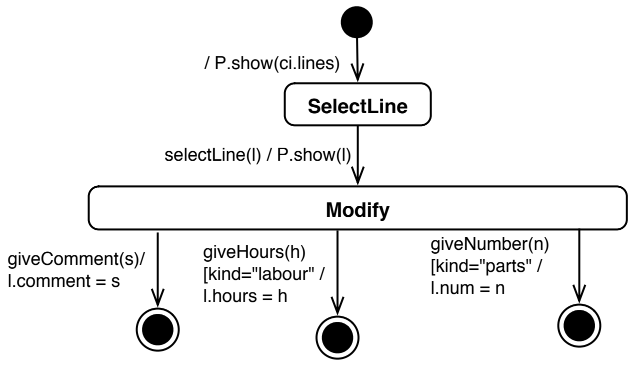


Fig. D.3. Subchart ModifyLine