An Analysis Pattern for Electronic Marketplaces

Anna Queralt and Ernest Teniente

Universitat Politècnica de Catalunya
C/ Jordi Girona Salgado 1-3
08034 Barcelona, Catalonia

[aqueralt | teniente]@lsi.upc.es

Abstract. An electronic marketplace supports the interaction among different users to exchange information about products to sell and buy. Significance of electronic marketplaces is given by the huge amount of web sites that are currently available providing services in almost any area one can think of. However, existing similarities among these sites are not precisely documented, which strongly restrains software reuse during the development of new electronic marketplaces.

To improve this situation, we propose an analysis pattern that describes both the structural and the behavioural properties of a generic electronic marketplace. Then, developing a new electronic marketplace will correspond just to provide a particular adaptation of our pattern. In this way, reuse of our pattern contributes to reducing the costs of the development of new electronic marketplaces.

Keywords: Analysis patterns, marketplaces, conceptual modelling, OO modelling, UML.

1 Introduction

Electronic commerce is devoted to share business information, maintain business relationships and conduct business transactions by means of telecommunications network [Cou99]. In this context, electronic marketplaces provide information about products to sell and buy and also support for the negotiations conducted to settle the price of the products being traded. In general, products are classified according to several categories that allow to distinguish the particular market being addressed. Thus, we have a real-estate marketplace when the products are flats, apartments or offices; an automobile marketplace when products refer to trucks, cars or motorbikes; etc. Moreover, negotiations are usually performed by means of different types of auctions that allow to achieve different business objectives.

Trading on the internet allows reaching a larger number of buyers and sellers than more traditional modes of communication. Moreover, electronic business transactions may be settled much cheaper since they do not require human intermediaries nor physical places where to perform the marketplace. This situation gives us an idea about the current importance of electronic marketplaces and the big effort that is being devoted to their development. Hundreds of electronic marketplaces are currently encountered on the internet (see www.internetauctionlist.com for a possible list). Among the most representative ones we may have www.ebay.com, www.onsale.com or www.amazon.com/auctions that allow to buy and sell items for a broad spectrum of categories according to several types of auctions.

These electronic marketplaces are only a few examples, randomly selected, but we could have chosen many others just by changing the country of reference or by considering only marketplaces for a particular category. The growing importance of electronic marketplaces is also substantiated in [Bak98, Fel00, MDD01].

In general, electronic marketplaces differ on the geographical area they address, on the kind of services they provide, on the type of items they handle, on the types of auctions considered, etc. However, all of them share several features regarding both the information they deal with and the functions they provide. Unfortunately, the existing similarities among different electronic

marketplaces have not been precisely documented yet. Therefore, we cannot take an explicit advantage of these similarities during the development of new electronic marketplaces.

To improve this situation, we propose in this paper an analysis pattern for electronic marketplaces. An *analysis pattern* [Fow97, FY00] is a conceptual schema resulting from the definition of a given generic information system for a certain application domain. By tailoring the pattern to a specific situation of the domain we can rapidly and correctly obtain the corresponding particular conceptual schema for that situation.

Accurate and precise conceptual modelling is an essential premise for a correct development of an information system. Reusable analysis patterns are able to facilitate this difficult and time-consuming task since they provide an additional value to increase reusability (and, thus, quality of the resulting system) during software development.

Our analysis pattern for electronic marketplaces is specified in UML and OCL and it has been drawn from an external (since we did not have access to their documentation nor code) in-depth study of some well-known electronic marketplaces: www.ebay.com, www.onsale.com, www.amazon.com (both its marketplace and auctions sites) and www.monster.com, a job search site. We have built our analysis pattern by considering the common functionalities among these systems, according to the methodology proposed in [SS02].

Our pattern covers the most important function regarding electronic marketplaces [Bak98]: matching buyers and sellers. This function is traditionally decomposed into three main activities: determining product offerings, search and price discovery; whose behaviour is also incorporated into our pattern. As far as price discovery is concerned, our pattern allows a fixed-price policy as well as all possible types of auctions and their variations (as defined in [KF98, HV99]). Hence, we show that our pattern is general enough to allow the definition of an analysis model for a new electronic marketplace just by tailoring it to the particular needs of users and owners of that marketplace.

Ontologies have been proposed as an important and natural means of representing real world knowledge that can be used during information systems development [SS02] as well as for modelling enterprises [FG97]. An ontology defines the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relationships between terms [LGSS99]. In this way, an ontology is a formal description of entities and their properties, constraints and behaviours [GF95]. For this reason, and due to the shallow differences that we find among ontologies and conceptual schemas, we may assert also that the analysis pattern we propose here can be regarded also as an ontology for the electronic marketplace domain.

Summarizing, we may conclude that the main contributions of this paper are threefold:

- First, we propose an analysis pattern for electronic marketplaces. This pattern is general enough and it describes both the structural and the behavioural properties that a generic marketplace should satisfy. Hence, it can be regarded also as an ontology for the electronic marketplace domain.
- Second, our pattern contributes to electronic marketplaces development in the same way than analysis patterns do for software development [GH02]. In particular, it reduces the development costs of new electronic marketplaces because of the reuse of existing solutions and it speeds up the definition of concrete conceptual schemas (while improving their quality) for particular application domains since we tailor existing sources that are known to be correct.
- Third, our work represents a step forward to the development of globally accepted (domain specific) analysis patterns (ontologies). This is an important field of open research as stated by several authors like [Neu03, SS02].

This paper is organized as follows. Next section reviews previous work on analysis patterns and how they can be defined using UML. Section 3 presents our analysis pattern for electronic marketplaces. Section 4 relates our pattern to previous work on electronic marketplace development. Finally, we point out our conclusions and comment on further work on Section 5.

2 Analysis Patterns and its Definition with UML

A pattern identifies a problem and provides the specification of a generic solution to that problem. The use of patterns in software development is becoming very important since they contribute to increase the reusability of software components and at the same time to reduce the number of errors of the software delivered. In general, patterns can be used at each stage of the software development process. Depending on the stage for which they are defined they can be classified as:

- *Analysis or conceptual modelling patterns,* which are defined at the analysis level of an information system.
- A*rchitectural patterns,* or set of classes that represent an architectural structure at the system level [BMR+96].
- *Design patterns*: design classes that describe a design construct [GHVJ95].
- *Language patterns,* idioms, that apply to one or a few languages.

It is largely agreed that a *design pattern* describes the structure of the solution to a problem that appears repeatedly during software design and the interaction between the different software components involved in the solution. In this sense, a design pattern is applicable to any software system, provided that the problem addressed by the pattern is encountered during the design of that system.

Unfortunately, we do not find such an agreement regarding the kind of patterns that can be defined at the analysis level of information systems development. In fact, the term analysis pattern was coined by Martin Fowler [Fow97] who proposed analysis patterns that define appropriate solutions to model specific constructs that may be found during the specification of different information systems. Patterns for object-oriented analysis had been already outlined in [Coa92].

More recently, E.B. Fernandez and X. Yuan [FY00] used the term analysis pattern with a different sense to define a whole (generic) conceptual schema for a single information system domain. Petia Wohed's patterns [Woh00] would be somehow in the middle of these two ideas.

According to these proposals, we think we should distinguish two different types of patterns regarding the definition of patterns at the analysis stage of the software development process: *conceptual modelling patterns* and *analysis patterns*.

A conceptual modelling pattern is aimed to represent a specific structure of knowledge (for instance a Part-Of relationship) that we may encounter in different domains. Then, they are domain-independent since they can be used in different domains. Fowler's patterns [Fow97] would follow this idea.

An analysis pattern is aimed to represent a generic, domain-dependent, knowledge required to develop an application for particular users. It provides a full conceptual model that is the whole specification of an information system to be developed for a given domain. In this sense, these patterns are domain-dependent since they can be used only in the domain for which they have been developed. [FY00] patterns as well as the pattern we propose in this paper fall into this type of pattern.

In addition to the previous considerations, we must note that an analysis model constitutes a permanent model of the reality in itself [Pre00, RBL+91] and, as such, it is independent of a particular implementation environment. For this reason, analysis patterns (as well as conceptual modelling patterns) must also be technology independent.

We use UML (with the support of OCL when required) to define our analysis pattern for electronic marketplaces. To our knowledge, previous work does not provide unfortunately a sufficient proposal to define analysis patterns in UML. Thus, for instance, [FY00, p.184] states that "a semantic analysis pattern describes a small set of coherent use cases that describe a basic generic application". However, their sequence diagrams specify object interaction and this can only be done if responsibilities are assigned to objects during analysis. However, taking this decision necessarily involves design and technological issues and, thus, it is not possible to define an analysis pattern that is technology independent. Similar limitations are found in [Fer98, Fow99].

As we have seen, an analysis pattern corresponds to a conceptual schema of a generic application domain. Therefore, the definition of an analysis pattern in UML should include all UML models required to define a conceptual schema. In this sense, we will mainly follow Larman's proposal [Lar98] who states that an UML analysis model must contain a Use Case Model, an Analysis Class

Diagram, a System Behaviour Model and an Analysis State Model. More recently, this proposal has been slightly adapted in [Lar02] to relate these models to the different stages of the Unified Process but its main features still remain the same.

One of the important features of Larman's proposal is to define the system behaviour as a 'black box' before proceeding to a logical design of how a software application will work. This decision has a clear impact during analysis on the sequence diagram definition and on the assignment of operations to classes. Hence, sequence diagrams are considered as *system sequence diagrams* that show the events that external actors generate; their order and the system response to those events. On the other hand, operations responding to those external events are not assigned to classes and they are recorded in an artificial type named *system*. In this way, responsibilities are not assigned to objects during analysis.

Moreover, we assume that the UML models of the analysis pattern are non-redundant [CST02]. Non-redundant models contribute to the desirable properties of an information system specification and, at the same time, facilitate software design and implementation. An UML specification is redundant when a certain aspect of the software system is defined in more than one of the UML models for analysis. Therefore, in our electronic marketplace analysis pattern each aspect will be specified into exactly one of the models that conform the pattern. More details about these ideas can be found in [Ten03].

3 Electronic Marketplace Analysis Pattern

We define in this section the different models and diagrams that conform our analysis pattern, i.e. the Use Case Model, the Analysis Class Diagram, the System Behaviour Model and the Analysis State Model.

3.1 The Use Case Model

The aim of this section is to provide an overview of the functionalities that must be offered by an e-marketplace. The use case model serves this purpose. Informally speaking, use cases are possible ways users may use a system to meet their goals. The use case model consists mainly of the definition of *use case diagrams* which identify the main ways a user may interact with the system.

Electronic marketplaces have three main functions [Bak98]: matching buyers and sellers, facilitating transactions and providing an institutional infrastructure which specifies the rules that govern the transactions of the electronic market. In order to perform these functions, e-marketplaces offer some basic services, such as a directory of its users, product catalogues, search tools to find products, on-line purchase orders and auctions, etc.

A generic e-marketplace supports the interaction between different users to exchange information about products that they want to sell or that they are looking for. A user may publish an announcement either to sell or to buy a product. If the user is looking for a certain kind of product, he specifies in the announcement the features requested. Otherwise, he provides as well the kind of negotiation he wants to use, which can be an advertisement to contact with interested users, a fixed price sale or an auction. The users interested in a published product may bid for it and, depending on the kind of announcement, the system determines the winner or winners.

An e-marketplace must also include a subsystem responsible for the management of the fees that users must pay for publishing announcements and selling products, and a method to give users an idea of the reliability of the other users of the marketplace.

To facilitate both the explanation and understanding of our analysis pattern, we group the functionalities provided by an e-marketplace into four different subsystems:

1. *Users, products and categories*: it shows how information related to users, products and categories is stored and managed.
2. *Publishing announcements*: this part of the pattern supports the setting up of an announcement so that users can start bidding.

3. *Bidding*: here we will see how users show their interest in the products offered in the marketplace and the way the system decides the users who can or must purchase the products, depending of the kind of announcement.
4. *Reputation and fees management*: it shows the structure that allows users to evaluate others' behavior in the marketplace and thus gives to the rest of participants an idea of the convenience of negotiating with him. In this section we will also know about the fees that must be paid by the users for announcing and selling products and how our pattern supports them.

According to this organization, the functionalities offered by an e-marketplace can be divided into a set of packages, shown in Fig. 1 together with the actors that interact with them.

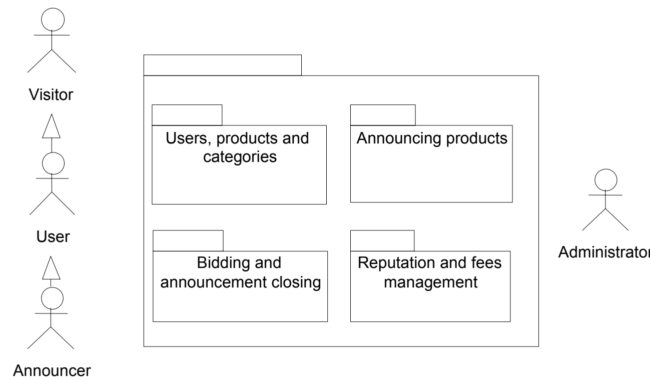

Fig. 1. – Use case packages

We may distinguish four kinds of actors in an e-marketplace:

- *Visitors*, which are the people not registered in the e-marketplace. They can only perform queries to the system.
- *Users,* which are the ones that have registered to buy products. They can also do the same things as visitors can.
- *Announcers*¸ who are the ones allowed to sell products as well as to do everything users can do.
- *Administrators*, who are responsible for maintenance tasks.

The package *Users, Products and Categories* includes the use cases regarding the management of these three entities. The use case diagram is shown in Fig. 2. We have chosen self-descriptive names for the use cases, so that further explanations will not be necessary in most cases.

When registering a product, the announcer will need to specify a set of features that will depend on the category to which the product belongs. Once this information is stored, the announcer must specify the format and the beginning and ending dates of the announcement he wants to publish about the product. This will be done in the use case *Publish announcement*, from the package *Announcing products.*

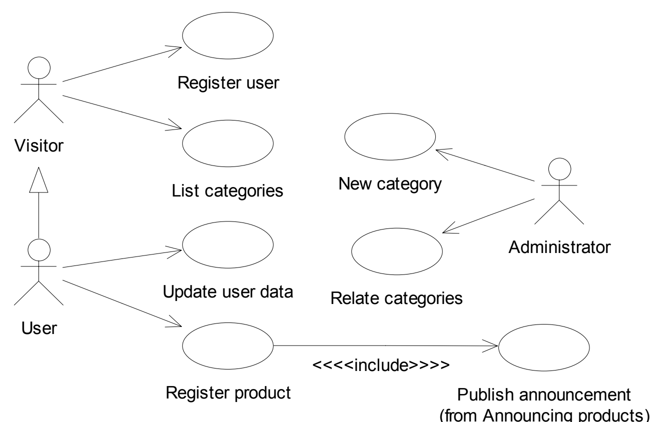

Fig. 2. – *Users, Products and Categories* use cases

The package *Announcing products* includes the use cases needed for publishing, modifying and deleting announcements of any kind. Concretely, the use cases that it includes are the ones shown in Fig. 3.

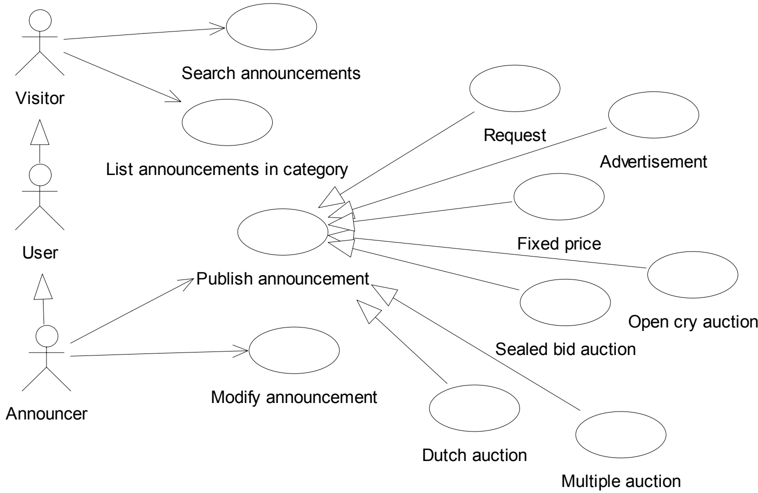

Fig. 3.- *Announcing products* use cases

We have used a generalization relationship between the use cases regarding the publication of each kind of announcement and the use case *Publish announcement*, in order to represent a common part of behaviour and semantics in all of them. Thus, in Fig. 2, the inclusion of *Publish announcement* in *Register product* implies that the behaviour of any of its specializations is included in the registration of a product in the e-marketplace. Apart from this, users can re-announce products that have not been bought or sold in their previous announcements. The specializations of *Publish announcement* also serve this purpose.

Bidding and announcement closing contains the use cases related to the actions that can be done when an announcement is open. They are shown in Fig. 4.

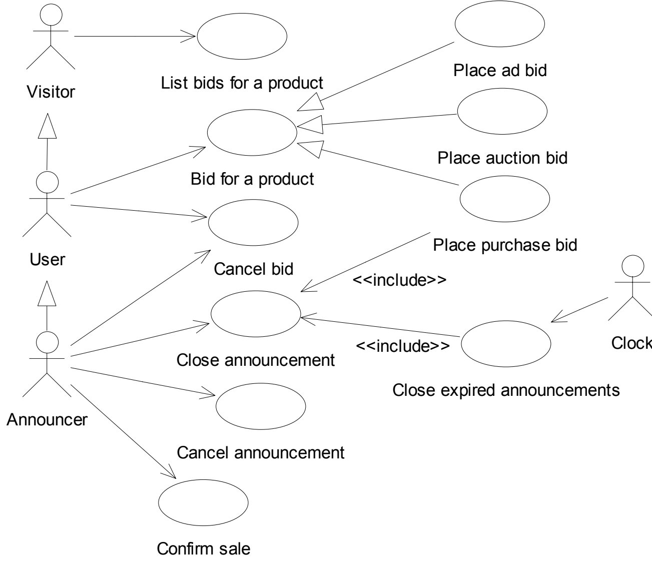

Fig. 4.- *Bidding and announcement closing* use cases

Finally, we have the use cases belonging to *Reputation and fees management,* shown in Fig. 5.

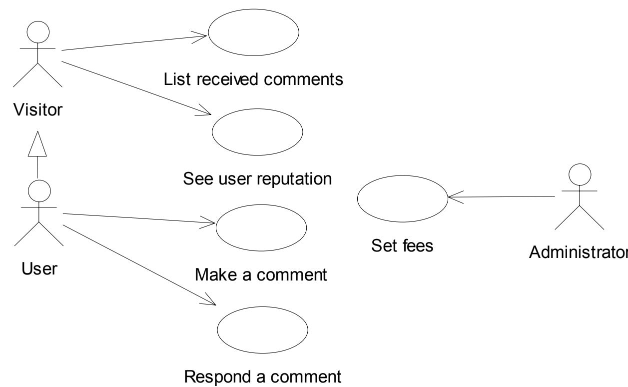

Fig. 5.- *Reputation and fees management* use cases

We hope the name of the use cases is clear enough to describe, at least intuitively, their intended functionality. We will provide a precise definition of the behaviour of some of these use cases in section 3.3, when presenting the system behaviour model.

3.2 The Analysis Class Diagram

The analysis class diagram describes structural properties of the objects that model concepts of the problem domain. It consists of a static structure diagram (a class diagram) in which no operations are defined.

As we said, we will define a non-redundant analysis pattern. Non-redundant analysis models are achieved by specifying as much aspects as possible in the class diagram instead of defining them in the other models. Therefore, the analysis class diagram is the most important part of the electronic marketplace pattern because it contains the definition of its main aspects. As we know, there are some conditions that the information base must satisfy but that can not be graphically specified in the analysis class diagram. We will specify these integrity constraints, as well as the derived information we consider necessary, by means of the OCL 2.0 [OMG03b], following the methods proposed in [Oli03a, Oli03b]. The basic idea of these methods is to express both integrity constraints and derivation rules as special types of operations of the context class.

We define the subset of the whole class diagram for each of the subsystems in which we have just divided the analysis pattern. We will start with the information related to the users of the marketplace, the products offered there and the categories in which they can be classified.

Users, products and categories

The users of an e-marketplace are usually identified by means of an email address and also have a password that allows them to log in, as well as some personal information (name, birth date, phone number, country, etc.). They may be either people or organizations. When a user registers a product in the marketplace, he becomes an announcer, and must provide a credit or debit card and checking account information.

Marketplace products are classified according to several categories. Each of these categories is usually divided into subcategories. For instance, the category real estate can be divided into residential, commercial, land, etc. A category is basic when it is not divided into subcategories. Products are classified into categories and, as such, they belong to all categories that are upon the one they are classified in. A category may also be related to several categories, for instance a category named *musical instruments* could be related to the category *musical magazines*.

Users may register products in several basic categories. Products registered in the marketplace are identified by means of a product identifier and the electronic marketplace records also the date when they are registered. Moreover, additional information of products may be specified depending on the category where they are classified.

The following analysis class diagram describes the information regarding users, categories and products in an electronic marketplace:

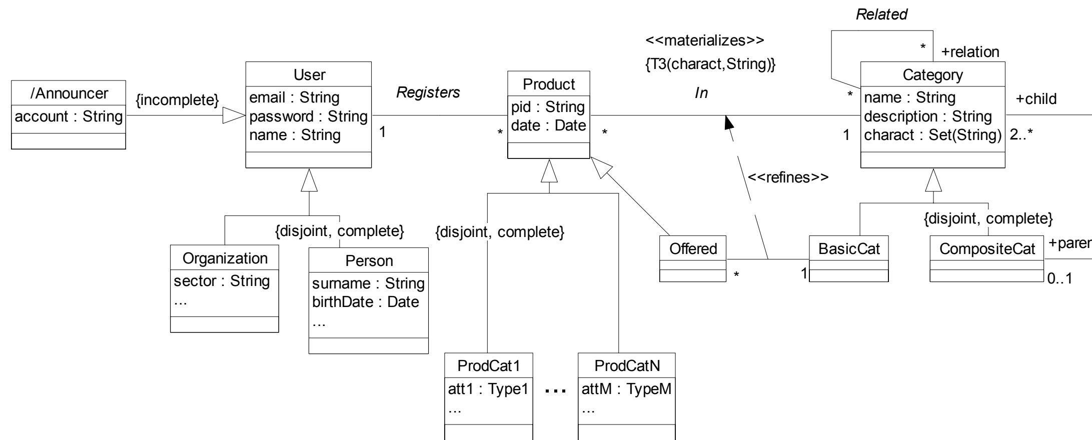

Fig. 6. – Users, products and categories in an e-marketplace

The classification of users into organizations and people corresponds to the well-known Party Pattern of [Fow97]. We have used the term User instead of Party since we believe it is more adequate in our context. A user is considered to be an Announcer when it has registered at least one product. Clearly, the class Announcer is derived since its instances can be computed from other elements of the schema.

The information about basic categories can be regarded as model as well as metamodel information since we will have a different subclass *ProdCatI* for each instance of *Category*. In the UML, this could be specified by means of a powertype [RJB99, p. 392] that is a metaclass (*Category* in our example) whose instances are subtypes of a given class (*Product*). However, as we want to allow the system administrator to add new categories, this solution is not powerful enough. We propose then the use of the materialization association [Cab03], which as well as maintaining the integrity of the model when adding, deleting or updating categories, allows a basic category to transfer information to the subtypes of Product. More precisely, each instance of *Category* will have an attribute that contains the set of attributes that the products belonging to it must have.

Products are related to the category they are associated in and to the user that has registered them. *Offered* products, i.e. products that a user wants to sell, can only be related to basic categories since they are specific products owned by the announcer. Requested products, i.e. products that a user is looking for, are related to any kind of category (basic or not).

Again, for the sake of simplicity, we assume that a requested product will have the same attributes as an offered product of the same category. Nevertheless, we could easily extend the hierarchy of products so that it admits different attributes for requested and offered products. Later, when we talk about publishing announcements, we will see how we make the distinction between offered and requested products.

In order to express the conditions that the information base must satisfy but cannot be graphically specified, we include the following textual constraints, formally expressed in OCL:

- Users are identified by email

```
context User::UniqueEmail():Boolean
post: result = User.allInstances()->isUnique(email)
```

- Products are identified by pid

```
context Product::UniquePid():Boolean
post: result = Product.allInstances()->isUnique(pid)
```

- Categories are identified by name

```
context Category::UniqueName():Boolean
post: result = Category.allInstances()->isUnique(name)
```

- A category cannot be related to itself

```
context Category::NotSelfRelated():Boolean
post: result = self.relation->excludes(self)
```

- A category cannot be (recursively) a child of itself

```
context CompositeCat::isDescendant(c1:Category, c2:Category):Boolean
post:
result = c2.child-> includes(c|c=c1 or isDescendant(c1,c))

context CompositeCat::NotSelfDescendant():Boolean
post:
result = not isDescendant(self, self)
```

- A user can not be under 18

```
context Person::AdultUser():Boolean
post: result = today()-birthDate >= 18
```

We also need to include the derivation rules for each derived element:

- An announcer is a user who has registered at least one product

```
context Announcer::allInstances():Set(Announcer)
post: result = User.allInstances()->select(self.Product->notEmpty())
```

Announcing products

Once products are registered in the marketplace they can be announced to other users. In general, a product will be announced several times if it is not sold/bought the first time it is published in the marketplace. Users can increase the visibility of their announcement by choosing upgrades, for example using a bold font, highlighting the title of the announcement or making it appear in the home page of the e-marketplace.

An announcement may be dealt with in two different ways according to the kind of product being announced: either as an offer or as a request. In the first case, the user publishes the announcement in order to try to sell a product, whereas in the second he simply publishes the announcement with the purpose of finding sellers of a product with the characteristics specified in the announcement.

When a user wants to offer a product, he can either publish an advertisement or a sale announcement. In the first case, he simply waits for interested users to contact him and decides to which of them sell the product, while in the second the decision about selling the announced product is taken automatically by the e-marketplace.

There are two different kinds of automatic treatment of sales in an e-marketplace: fixed price sales of one or more items and auctions. A product can be offered at a fixed price and auctioned at the same time. There exist several types of auctions, each one having its own rules regarding the evaluation of bids and the auction closing. Here we present the distinctive characteristics of each of them [KF98a, KF98b]:

- *Open cry auctions*: each bid must be greater than the previous ones, which are available to the users of the marketplace at any time. The auction can either finish when the seller specifies, when new bids are not received for a certain period of time or whatever happens first. The user that wins the product auctioned is the one that has bidden higher.
- *Sealed bid auctions*: before the auction begins, the seller specifies one or more deadlines in which he will broadcast the highest bid received so far. Meanwhile, bids are kept secret for the interested users and when a deadline arrives, the seller decides whether he will go on with the auction or close it. The winner is the highest bidder in the last round.
- *Multiple item auctions*: several items are auctioned. Each interested user specifies the number of items desired and indicates the price he is willing to pay. The auction can finish either at the

time specified by the seller, when new bids are not received for a certain period of time, when there are no items remaining or whatever happens first. The winners will be the highest bidders such that the sum of the quantities they demand is at most the quantity of items offered. It may happen that the lowest winning bid gets a smaller number of items than the demanded one since only as many items as specified in the announcement are offered.

– *Dutch auctions*: one or more items are on sale. The seller specifies a high initial price, at which he expects no buyers, and reduces it while there are no bids for the item or items. In this case, a bid is an acceptance of the current price offered by the seller. When there are several items on sale, users specify in their bids the number of items they want to purchase at the price currently offered by the seller. The auction can finish at the time specified by the seller, when the price has decreased to a certain limit, when there are no remaining items or whatever happens first. There are several winners, which can pay the amount they bade or the lowest winning amount.

The seller can specify an initial and a reserve price for any kind of auctions. For all of them, except for Dutch auctions, the amount bidden must be greater than the initial price, and the product is sold to the highest bidder provided that the amount bidden is greater than the reserve price. In contrast, the meaning of the initial price in Dutch auctions is the best price for the seller, and the reserve price is the minimum amount at which he is willing to sell his product or products.

Additionally, there are some variations that can be applied to any kind of auction. One of them is the establishment of minimum bid increments in an auction, which are usually proportional to the current price of the product. A seller can also specify the price to be paid by the winner, which can be the amount he bade, the amount bidden by the second winner or, in auctions with more than one item, the lowest winning bid.

The diagram of Fig. 7 specifies the information regarding announcements in our pattern:

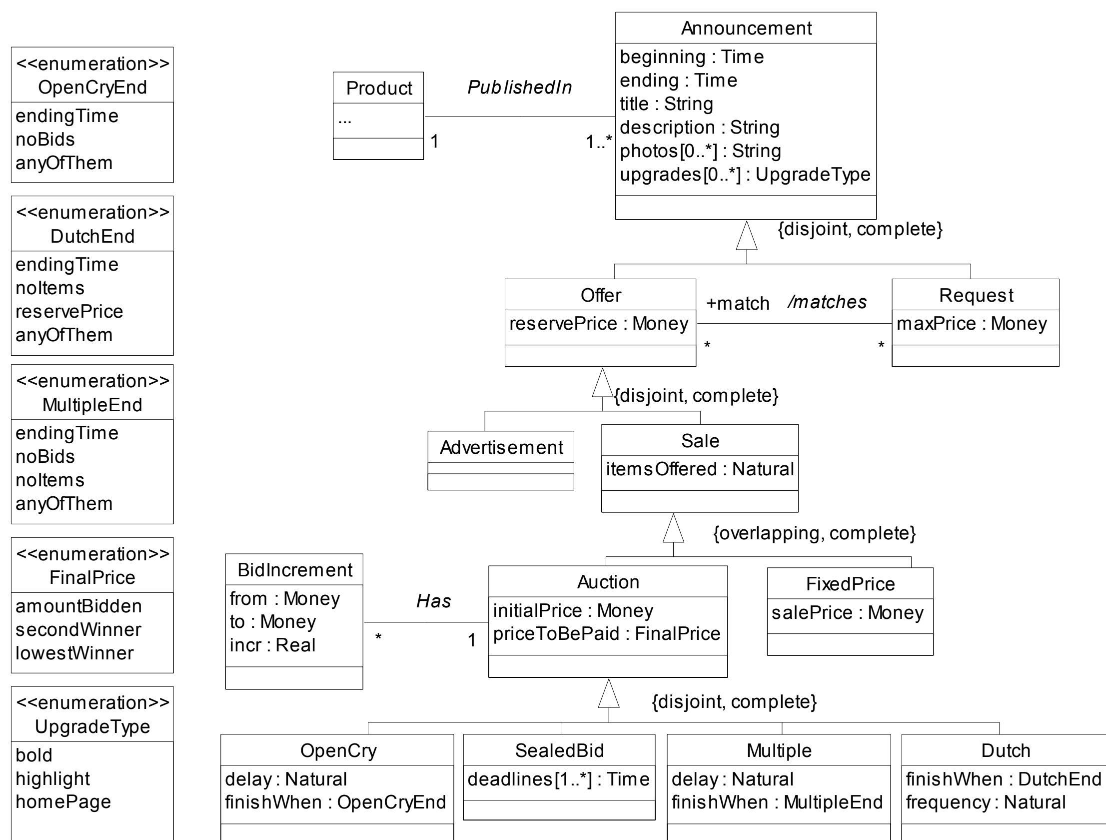

Fig. 7. – Ways of offering products in an e-marketplace

Our pattern allows announcing a product several times. When a user wants to publish an announcement, he must specify the period during which the announcement will be public in the

marketplace, a title and a description for the product. Optionally, he can add photographs or choose upgrades so that the announcement looks more attractive. Depending on the type of announcement, he will need to indicate different information.

When publishing an announcement requesting some product, a user can only specify the maximum price (*maxPrice*) he is willing to pay, as well as the desired characteristics of the product which, as we already know, are stored in the class *Product.*

On the other hand, if the user wants to offer a product in his announcement, he will optionally indicate a reserve price (*reservePrice*), which is the lowest price at which he is willing to sell his item or items. If the offer is a fixed price sale, indicating a sale price (*salePrice*) is compulsory.

The announcers of requested products will be automatically notified of the existence of matching offers in the e-marketplace, and will bid for them in case they are interested. This is why we need the derived association *matches*.

When an announcer wants to auction a product, he needs to give its initial price (*initialPrice*) and determine the price to be paid by the winner or winners of the auctions (*priceToBePaid*). He can also establish minimum bid increments for consecutive bids, which depend on the current price of the item and are usually expressed as a fixed quantity. Moreover, depending on the kind of auction, he must specify some other information so that the system can control the auction. More precisely, in an open cry auction (*OpenCry*) the information needed is the maximum delay between bids (*delay*) and the condition for finishing (*finishWhen*). In sealed bid auctions (*SealedBid*) the information needed is the set of instants in which the seller will publish the current winning bid (*deadlines*). In Dutch and multiple item auctions, the seller must indicate the number of items offered and the finishing condition. In multiple item auctions the maximum delay between bids must also be specified, and in Dutch auctions, the auctioneer must indicate the frequency (*frequency*) in which the price must be decreased. In this type of auctions, the bid increments specified will act as decrements, so they will be negative.

In order to complete this static part of the pattern we need some additional integrity constraints:

- The beginning date of an announcement is later than the date of the product

```
context Announcement::correctBeginning():Boolean
post: result = self.beginning >= self.Product.date
```

- The ending date of an announcement is later than its beginning date

```
context Announcement::correctEnding():Boolean
post: result = self.ending >= self.beginning
```

- If an auction is not Dutch, its initial price is lower than its reserve price. Otherwise, its reserve price is lower than its initial price

```
context Auction::correctPrices():Boolean
post: result =
if not self.oclIsTypeOf(Dutch) then self.initialPrice =< self.reservePrice)
else self.initialPrice >= self.reservePrice endif
```

- In Dutch Auctions, the bid increments must be negative

```
context Dutch::negativeIncrements():Boolean
post: result = self.BidIncrement->forall(b| b.incr < 0)
```

- A product can not have both offer and request announcements

```
context Product::notOfferAndRequest():Boolean
post: result = (self.Announcement->includes(a|a.oclIsTypeOf(Offer)) implies
not self.Announcement->includes(a|a.oclIsTypeOf(Request))) and
(self.Announcement-> includes(a|a.oclIsTypeOf(Request)) implies not
self.Announcement-> includes(a|a.oclIsTypeOf(Offer)))
```

- A fixed price sale must have a sale price

```
context FixedPrice::hasSalePrice():Boolean
post: result = self.salePrice->notEmpty()
```

- In a sale, at least one item must be offered

```
context Sale::atLeastOneItem():Boolean
post: result = self.offeredItems > 0
```

- In open cry auctions and sealed bid auctions, only one item can be offered

```
context Auction::onlyOneItem():Boolean
post: result = self.oclIsTypeOf(OpenCry) or self.oclIsTypeOf(SealedBid)
implies self.itemsOffered=1
```

- In a multiple item auction, more than one item must be offered

```
context Multiple::atLeastOneItem():Boolean
post: result = self.offeredItems > 1
```

- In a sealed bid auction, the last deadline must be the same as the ending of the announcement

```
context SealedBid::correctDeadlines():Boolean
post: result = self.deadlines->last() = self.ending
```

- If a product is auctioned and offered at a fixed price simultaneously, its sale price must be greater than its reserve price

```
context Sale::correctSaleAndReservePrice():Boolean
post: result = self.oclIsTypeOf(Auction) and self.oclIsTypeOf(FixedPrice)
implies self.reservePrice < self.oclAsType(FixedPrice).salePrice
```

- If the finishing condition for an open cry auction is not *endingTime*, the attribute *delay* must have a value

```
context OpenCry::hasDelayWhenNeeded():Boolean
post: result = not self.finishWhen=OpenCryEnd::endingTime implies self.delay-
> notEmpty()
```

- If the finishing condition for a multiple item auction is *noBids* or *allOfThem*, the attribute *delay* must have a value

```
context Multiple::hasDelayWhenNeeded():Boolean
post: result = self.finishWhen = MultipleEnd::noBids or self.finishWhen =
MultipleEnd::allOfThen implies self.delay->notEmpty()
```

- In open cry and sealed bid auctions, the price to be paid can only be the amount bidden or the amount bidden by the second winner. In multiple auctions, it can be the amount bidden or the amount bidden by the lowest winner.

```
context Auction::correctPriceToBePaid():Boolean
post: result = (self.oclIsTypeOf(OpenCry) or self.oclIsTypeOf(SealedBid)
implies self.priceToBePaid = FinalPrice::amountBidden or self.priceToBePaid =
FinalPrice::secondWinner) and (self.oclIsTypeOf(Multiple) implies
self.priceToBePaid = FinalPrice::amountBidden or self.priceToBePaid =
FinalPrice::lowestWinner)
```

- The specified bid increment intervals must be correct

```
context BidIncrement::correctInterval():Boolean
post: result = self.from <= self.to
```

- Two bid increments for the same auction can not overlap

```
context Auction::notOverlappingIncrements():Boolean
post: result = self.BidIncrement->forall(b1,b2|b1<>b2 and b1.to < b2.from or
b2.to < b1.from)
```

- The bid increments for an auction must not leave undefined intervals

```
context Auction::notUndefinedIntervals():Boolean
post: result =
let increments: Sequence(BidIncrement) = BidIncrement.allInstances()->
sortedBy(from)
in
Sequence{1..increments->size()-1}->forall(i:Integer | increments-> at(i).to +
0.01 = increments->at(i+1).from)
```

In this submodel there is one derived element. Its derivation rule is as follows:

- A request announcement matches an offer if all the information specified in the requested product is also in the offered product.

```
context Request::match():Set(Offer)
post: result = Offered.allInstances()->select(p | matching1(p,
self.Product)).Announcement-> select(a | a.beginning < now() and a.ending >
now()).oclAsType(Offer)
```

Bidding and announcement closing

Once we have seen the existing kinds of announcements, we will follow up describing how users bid for products and how the system decides their winners.

Users will only bid for offered products, not for requested ones. In case there is someone having a product matching with a request but not published in the e-marketplace, we assume that he will contact the announcer of the requested product beyond the limits of the e-marketplace. In case there are matching products published, the announcer of the request will manually place his bid for the ones he is interested in.

When a user wants to purchase a product on sale, he makes a new bid indicating the quantity he is willing to pay and how many units of the product wants, in case there are several items offered. This can be done as many times as desired for a certain product, provided the announcement is open. If the product is offered at a fixed price, the user that has placed the bid will automatically be the winner of as many items as he has requested. On the other hand, if the product is auctioned, the bidder may win the items requested or not, depending on the bids made by other users and the rules of the corresponding auction type, as we have already explained. Optionally, in auctions the user can specify the maximum quantity he is willing to pay and the system places bids on his behalf up to the quantity specified. This is what in e-marketplaces is called *proxy bidding*, and allows users to pay less than the quantity they initially specified.

When a user is interested in an advertised product, he places his bid in the same way that he would do for a product on sale, but in this case the system will not decide automatically whether he wins the product or not. The announcer of the advertised product will contact the bidder in case he is interested in selling him his product.

If the product is offered at a fixed price, the announcement is closed when all the items are sold. If the product is auctioned, the announcement is closed when specified by the announcer (at a given time

[1] *Matching* is an auxiliary operation, not defined here, that determines if an offered product matches a request.

or when other conditions hold). Advertisements and request announcements are closed at the time specified by the announcer. Additionally, the announcer may close his announcement whenever he feels appropriate.

The diagram of Fig. 8 specifies the information about bids. We see that a user may place several bids for a product. In each bid (*Bid*) specifies the maximum amount he is willing to pay (*amount*) and the quantity of items requested (*itemsWanted*), which must not be greater than the quantity of items offered. For fixed price sales, the amount bidden will be the same as the sale price of the product.

If the bid is for a fixed price sale or for a Dutch auction (*PurchaseBid*), the user that has placed it will automatically win the items requested. On the contrary, if the bid placed is for an auctioned product (*AuctionBid*), the winners will be the highest bidders at the closing of the auction. Moreover, when a user places a bid for an auctioned product, he has the opportunity to use a proxy, which will place bids (*AutomaticBid*) for the minimum amount (*value*) that makes the user win, if possible, until the value reaches the *amount* specified in the *Bid.* The winners will be the users that have specified a greater *amount*, and the others will be outbid, with the possibility of placing another bid in order to continue participating in the auction.

Bids for advertised products (*AdBid)* are registered in order to inform the advertiser of the users interested in his product, so that he can contact them when he desires.

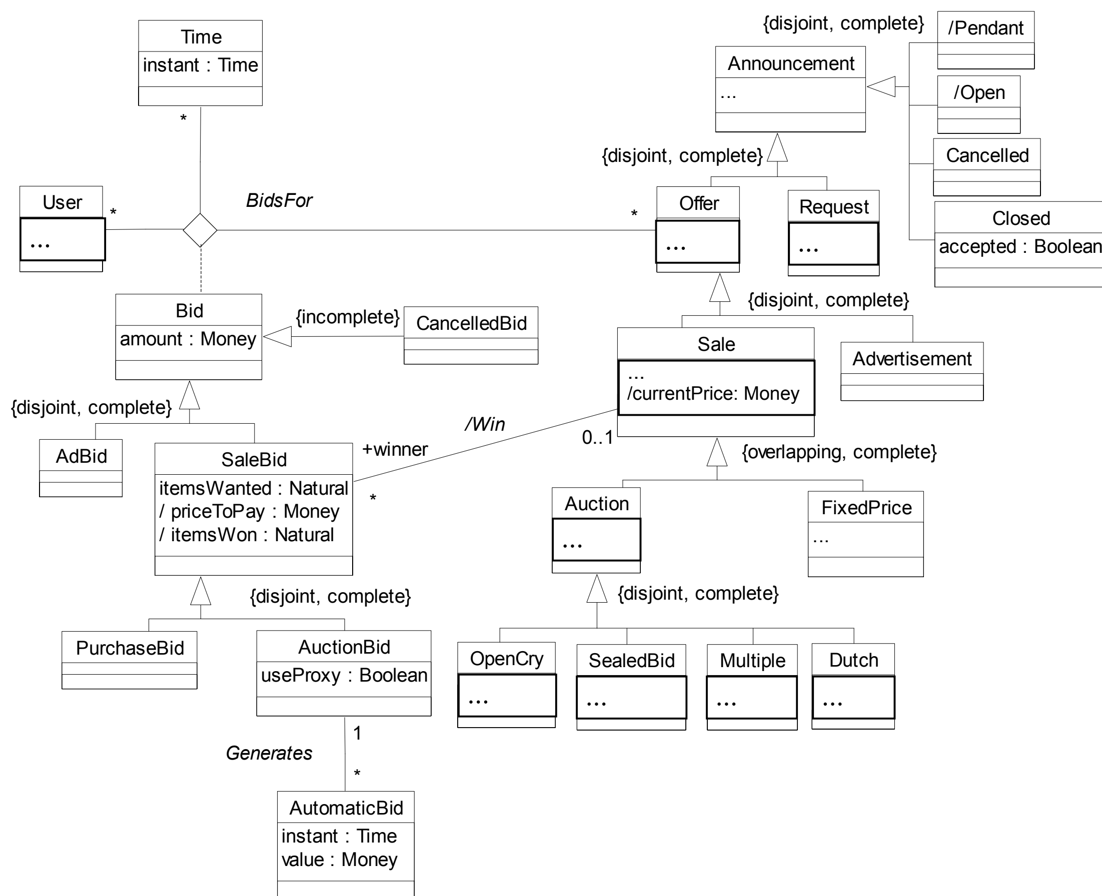

Fig. 8. – Bidding for products

Usually, an announcement will end at a specific time, but the announcer can close it whenever he wants. At this moment, he can either sell the item to the current winners or cancel the sale. Auctions can also end when some conditions hold, as we explained before. At this moment, the final winners of the auction will be the current winners and users will not be able to bid for that product anymore, unless the product is not sold and the announcer decides to publish another announcement of the same product.

As usual, we need some integrity constraints in order to complete the diagram:

- Two open announcements of the same product can not overlap

```
context Product::notOverlappingAnnouncements():Boolean
post: result = self.Announcement->select(a | not a.oclIsTypeOf(Cancelled))->
forall(a1,a2|a1<>a2 and a1.ending < a2.beginning or a2.ending < a1.beginning)
```

- Automatic bids can only be placed if the user has decided to use a proxy

```
context AuctionBid::proxyEnabled():Boolean
post: result = self.useProxy implies self.AutomaticBid->notEmpty() and
self.AutomaticBid->notEmpty() implies self.useProxy
```

- The number of items requested in a sale bid must be positive and not greater than the number of items offered

```
context SaleBid::validNumberOfItems():Boolean
post: result= self.itemsWanted>0 and self.itemsWanted<=self.Sale.itemsOffered
```

- The sum of the items wanted in all the purchase bids for the same product must not be greater than the number of items offered.

```
context Sale::enoughOfferedItems():Boolean
post: result = self.SaleBid->
select(b|b.oclIsTypeOf(PurchaseBid)).itemsWanted->sum() <= self.itemsOffered
```

- Creation IC[2]: A user can only bid for an open announcement

```
context Bid::openAnnouncement():Boolean
post: result = self.Offer.oclIsTypeOf(Open)
```

- A user can not bid for a product registered by himself

```
context Offer::announcerNotBidder():Boolean
post: result = self.Bid.User->excludes(self.Product.User)
```

- A user can only place a purchase bid if the product is offered at a fixed price or in a Dutch auction

```
context PurchaseBid::FixedPriceOrDutch():Boolean
post: result=self.Offer.oclIsTypeOf(FixedPrice)or
self.Offer.oclIsTypeOf(Dutch)
```

- Creation IC: A bid can not be cancelled if the corresponding sale announcement is not open

```
context CancelledBid::openAnnouncement():Boolean
post: result = self.Offer.oclIsTypeOf(Open)
```

- A purchase bid can not be cancelled

```
context CancelledBid::notPurchaseBid():Boolean
post: result = not self.oclIsTypeOf(PurchaseBid)
```

- A user can only place an auction bid if the product is auctioned

```
context AuctionBid::auctionSale():Boolean
post: result = self.Offer.oclIsTypeOf(Auction)
```

[2] According to [Oli03b], a Creation IC is a particular case of integrity constraint that is only evaluated when new instances of the entity where the constraint is defined are created.

- Creation IC: If the kind of sale is an open cry, sealed bid or multiple auction, the amount bidden must not be lower than the current price plus the corresponding increment. In fixed price sales and Dutch auctions, the amount bidden must be the current price.

```
context SaleBid::validAmount():Boolean
post:
let increment: Money = self.Offer.oclAsType(Auction).BidIncrement->
any(b|b.from <= self.Offer.oclAsType(Sale).currentPrice and b.to >=
self.Offer.oclAsType(Sale).currentPrice).incr
in
result =
if (self.Offer.oclIsTypeOf(OpenCry) or self.Offer.oclIsTypeOf(SealedBid) or
self.Offer.oclIsTypeOf(Multiple)) then
self.amount >= self.Offer.oclAsType(Sale).currentPrice + increment
else self.amount = self.Offer.oclAsType(Sale).currentPrice
```

- A closed announcement must be accepted if it is a fixed price sale or an auction in which the reserve price has been reached

```
context Closed::validAcceptance():Boolan
post: result = self.oclIsTypeOf(FixedPrice) or (self.oclIsTypeOf(Auction) and
self.oclAsType(Sale).currentPrice >= self.oclAsType(Offer).reservePrice)
implies self.accepted = true
```

We must also specify the derivation rules of our derived elements:

- Pendant announcements are the ones that have a future beginning date and have not been cancelled

```
context Pendant::allInstances():Set(Pendant)
post: result = Announcement.allInstances()->select(a | not
a.oclIsTypeOf(Cancelled) and a.beginning > now()
```

- Open announcements are the ones that are not pendant, closed or cancelled

```
context Open::allInstances():Set(Open)
post:
let notOpen: Set(Announcement) = Pendant.allInstances()->
union(Cancelled.allInstances())-> union(Cancelled.allInstances())
in
result = Announcement.allInstances()->reject(notOpen)
```

- The current price of a product on sale depends on the type of sale:
  - in fixed price sales the current price is always the sale price
  - in open cry and sealed bid auctions the current price is the highest automatically generated value of the winning bid , in case a proxy is used. Otherwise, the current price is the amount of the winning bid.
  - in multiple auctions the current price is the highest automatically generated value of the lowest winning bid, in case the user has chosen to use a proxy. Otherwise, the current price is the amount of the lowest winning bid.
  - in Dutch auctions the current price is given by the bid increments and the frequency chosen by the announcer

```
context Sale::increment(p: Money): Real
post: result = self.oclAsType(Auction).BidIncrement->select(b | b.from <= p
and b.to >= p).incr

context Sale::currentPrice():Money
post: result =
if self.oclIsTypeOf(FixedPrice) then
```

```
  self.oclAsType(FixedPrice).salePrice
else if self.oclIsTypeOf(OpenCry) or self.oclIsTypeOf(SealedBid) then
  if self.winner->any().oclAsType(AuctionBid).useProxy then
    self.winner-> any().oclAsType(AuctionBid).AutomaticBid-> sortedBy(value)->
    last().value
  else self.winner->any().amount endif
else if self.oclIsTypeOf(Multiple) then
  if self.winner->sortedBy(amount)->first().oclAsType(AuctionBid).useProxy
  then
    self.winner->sortedBy(amount)->
    first().oclAsType(AuctionBid).AutomaticBid-> sortedBy(value)->last().value
  else self.winner->sortedBy(amount)->first().amount endif
else -- Dutch auctions
let nDecrements: (now() -  beginning) / self.oclAsType(Dutch).frequency
in
Sequence{1..nDecrements}->iterate(i:Integer; acc:Money = initialPrice | acc +
increment(acc))
endif
```

- The price that the user who placed a bid must pay in case he wins the product is:
    - in a purchase bid, the price to pay is the current price of the item
    - in an auction bid, the price to pay depends on the value of the attribute *priceToBePaid,* in *Auction*

```
context SaleBid::priceToPay():Money
post: result =
if self.oclIsTypeOf(PurchaseBid) then self.Offer.oclAsType(Sale).currentPrice
else
  let price: FinalPrice = self.Offer.oclAsType(Auction).priceToBePaid
  in
  if price = FinalPrice::amountBidden then
    if self.oclAsType(AuctionBid).useProxy then
    self.oclAsType(AuctionBid).AutomaticBid->sortedBy(instant)->last().value
    else self.amount
    endif
  else if price = FinalPrice::secondWinner then
    let sortedBids: Sequence(SaleBid) = self.Offer.Bid->asSequence()->
    sortedBy(amount)
    in
    sortedBids->at(sortedBids->size()-1).amount
  else
    --lowestWinner: it is a Multiple auction, and the amount that must be paid
    --by the lowest winner is exactly the currentPrice
    self.Offer.oclAsType(Sale).currentPrice
  endif
endif
```

- The number of items won by a sale bid is:
    - the number of items demanded if it is a purchase bid
    - one if the bid is one of the winners and the announcement is not a multiple or Dutch auction, and zero if it is not winner
    - the number of items demanded if the announcement is a multiple auction and the bid is not the lowest winner
    - the number of remaining items if the announcement is a multiple auction and the bid is the lowest winner
    - zero otherwise

```
context SaleBid::itemsWon():Natural
post: result =
if self.oclIsTypeOf(PurchaseBid) then itemsWanted
else
  if not self.Offer.oclIsTypeOf(Multiple) then
    if self.Offer.oclAsType(Sale).winner->includes(self) then 1
    else 0
    endif
  else
    let isLowestWinner:Boolean = self.Offer.oclAsType(Sale).winner->
    sortedBy(amount)->first() = self
    in
    if isLowestWinner then
      self.Offer.oclAsType(Sale).itemsOffered –
      self.Offer.oclAsType(Sale).winner->reject(self).itemsWanted->sum()
    else
      if self.Offer.oclAsType(Sale).winner->includes(self) then itemsWanted
      else 0
      endif
    endif
  endif
endif
```

- The winning bids of a sale are:
  - in open cry and sealed bid auctions the winning bid is the one with the greater amount
  - in multiple auctions the winning bids are the ones with greater amount such that the sum of items they request is not greater than the number of items offered.
  - in Dutch auctions and fixed price sales the winning bids are all the bids placed for that sale

```
context Sale::winner():Set(Bid)
post: result =
let bids: Set(SaleBid) = self.Bid.oclAsType(SaleBid)
in
if self.oclIsTypeOf(OpenCry) or self.oclIsTypeOf(SealedBid) then
  bids->sortedBy(amount)->last()
else if self.oclIsTypeOf(Multiple) then
  if bids.itemsWanted->sum() <= self.itemsOffered then bids
  else
    let first:Integer = Sequence{1..bids->size()}->select(i: Integer | bids->
    subSequence(bids->at(i), bids->size()).itemsWanted->sum() >= itemsOffered
    and bids->subSequence(bids->at(i+1), bids-> size()).itemsWanted->sum() <
    itemsOffered)
    in bids->subSequence(first, bids->size())
  endif
else bids
endif
```

Reputation and fees management

Since the products offered in e-marketplaces usually belong to anonymous users, the rest of users can have problems in relying in the sellers. The same happens to sellers, who can never be sure if they will be paid by the winner or winners of the item. For these reasons, most e-marketplaces include a system to manage the reputation of its users.

Once an announcement is closed, both the announcer and the winner or winners can express their opinion about the others' behaviour regarding the transaction and evaluate their degree of satisfaction.

Each comment received can be responded by the alluded in order to apologise or defend himself in front of the accusation. These responses will not have an associated rating, so for a given transaction, a user will only be evaluated once by the same user.

The sum of all the ratings received by a user will give the others an idea of the convenience of negotiating with him.

An additional aspect that most marketplaces have in common is the fees charged to their announcers. There are three kinds of fees: the ones charged for publishing an announcement, the ones charged for the upgrades chosen and finally those charged to the announcer when his products are sold. The amount to pay for all these fees can depend on the category of the product, the kind of announcement and the price of the product.

We can see the representation of all this information in the diagram of Fig. 9:

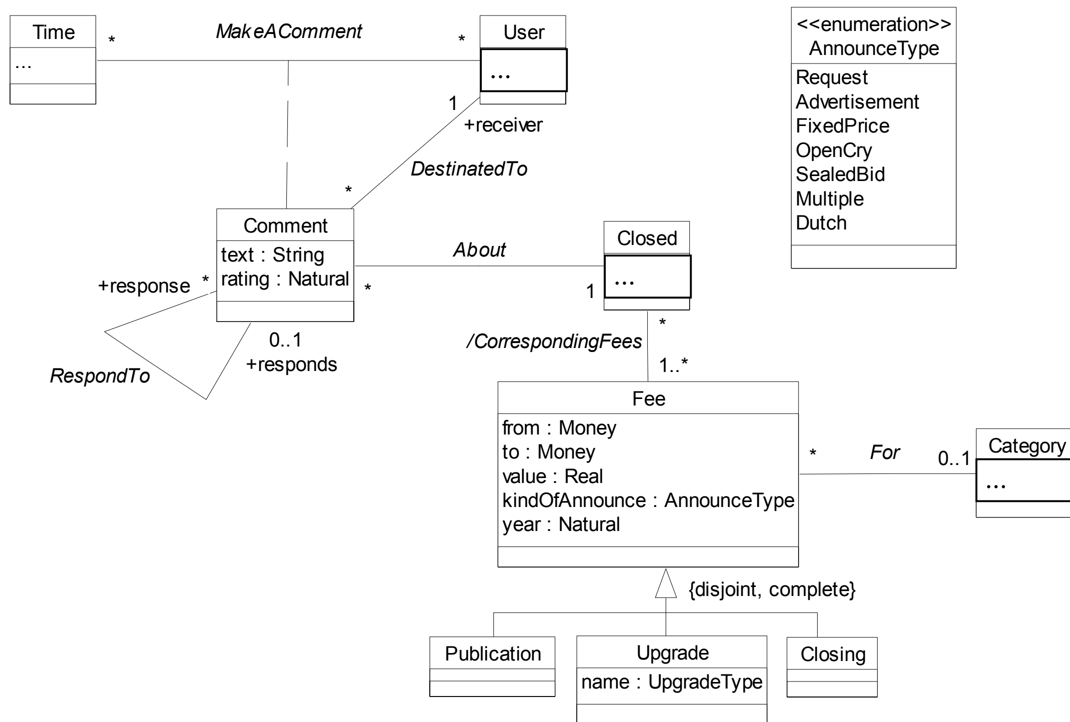

Fig. 9.- Comments left to users and fees applied to announcements

As can be seen in the diagram, a comment (*Comment*) can only be made regarding to a closed announcement. Each comment consists of the opinion of the emitter (*text*) and the rating (*rating*) he gives to the receiver, and can be responded by the alluded or emphasized by the emitter several times.

On the other hand, we have that several fees (*Fee*) can be applied to a closed announcement. As we already introduced, the amount of a fee can depend on the price of the product (*from, to*), the way in which the product is published (*kindOfAnnounce*), the category of the product and the current year (*year*). There are three kinds of fees: *Publication, Upgrade* and *Closing*, each of them corresponding to the explanation already given. The fees that must be applied to an announcement can be inferred from its characteristics.

We need some additional integrity constraints:

- Users can only make comments about sale announcements

```
context Comment::validAnnouncement():Boolean
post: result = self.Closed.oclIsTypeOf(Sale)
```

- A comment can only be emitted by the announcer or by the winners of the product to which it refers

```
context Comment::validEmitter():Boolean
post: result = self.User = self.Closed.Product.User or
self.Closed.oclAsType(Sale).winners-> include(self.User)
```

- If the author of a comment is the announcer, the receiver must be one of the winners

```
context Comment::emitterAnnouncerReceiverWinner():Boolean
post: result = (self.User = self.Closed.Product.User) implies
(self.Closed.oclAsType(Sale).winners-> include(self.User))
```

- If the author of a comment is one of the winners, the receiver must be the announcer

```
context Comment::emmitterWinnerReceiverAnnouncer():Boolean
post: result = (self.Closed.oclAsType(Sale).winners->includes(self.User))
implies (self.User = self.Closed.Product.User)
```

- Responses cannot have a rating

```
context Comment::responsesWithoutResponse():Boolean
post: result = self.response->notEmpty() implies self.responds->isEmpty()
```

- A user can only rate a comment regarding to a certain announcement once.

```
context Closed::oneRatingPerUser():Boolean
post: result= self.Comment->select(c| c.responds->isEmpty())-> isUnique(User)
```

- There can not exist two fees of the same kind referring to the same interval of prices, kind of announcement, category and year[3].

- The interval of prices to which a fee applies must be correct

```
context Fee::correctInterval():Boolean
post: result = self.from <= self.to
```

- All publication fees define a complete interval of prices for a given year, kind of announcement and category

```
context Publication::completeInterval():Boolean
post: result =
let fees: Sequence(Fee) = Fee.allInstances()-> select(f| f.year = self.year
and f.kindOfAnnounce = self.kindOfAnnounce and f.Category = self.Category)->
sortedBy(from)
in
Sequence{1..fees->size()-1}->forall(i:Integer | fees-> at(i).to + 0.01 =
fees-> at(i+1).from)
```

Upgrade and closing fees have analogous constraints.

The following operation corresponds to the derivation rule of the fees that must be applied to a closed announcement:

```
context Closed::isSubcategory(c1,c2:Category): Boolean
post: result = c1.Product.Category = c2 or isSubcategory(c1.parent, c2)

context Closed::Fee():Set(Fee)
post:
let publFee: Set(Publication) = Publication.allInstances()-> select(p |
p.from <= self.reservePrice and p.to >= self.reservePrice and
isSubcategory(self.Product.Category,p.Category) and p.year = year(today())
and p.kindOfAnnounce =
if self.oclIsTypeOf(Request) then AnnounceType::Request
else if self.oclIsTypeOf(Advertisement) then AnnounceType::Advertisement
```

[3] For the sake of simplicity, we omit the complex OCL expression required to specify this constraint.

```
else if self.oclIsTypeOf(FixedPrice) then AnnounceType::FixedPrice
else if self.oclIsTypeOf(OpenCry) then AnnounceType::OpenCry
else if self.oclIsTypeOf(SealedBid) then AnnounceType::SealedBid
else if self.oclIsTypeOf(Multiple) then AnnounceType::Multiple
else if self.oclIsTypeOf(Dutch) then AnnounceType::Dutch
endif endif endif endif endif endif endif)

let upgrFee: Set(Upgrade) = Upgrade.allInstances()-> select(u | u.from <=
self.reservePrice and u.to >= self.reservePrice and
isSubcategory(self.Product.Category,u.Category) and u.year = year(today())
and self.Upgrade->includes(u.name) and u.kindOfAnnounce =
if self.oclIsTypeOf(Request) then AnnounceType::Request
else fi self.oclIsTypeOf(Advertisement) then AnnounceType::Advertisement
else if self.oclIsTypeOf(FixedPrice) then AnnounceType::FixedPrice
else if self.oclIsTypeOf(OpenCry) then AnnounceType::OpenCry
else if self.oclIsTypeOf(SealedBid) then AnnounceType::SealedBid
else if self.oclIsTypeOf(Multiple) then AnnounceType::Multiple
else if self.oclIsTypeOf(Dutch) then AnnounceType::Dutch
endif endif endif endif endif endif endif)

let closeFee: Set(Closing) = Closing.allInstances()-> select(c | c.from <=
self.reservePrice and c.to >= self.reservePrice and
isSubcategory(self.Product.Category,c.Category) and c.year = year(today())
and c.kindOfAnnounce =
if self.oclIsTypeOf(Request) then AnnounceType::Request
else if self.oclIsTypeOf(Advertisement) then AnnounceType::Advertisement
else if self.oclIsTypeOf(FixedPrice) then AnnounceType::FixedPrice
else if self.oclIsTypeOf(OpenCry) then AnnounceType::OpenCry
else if self.oclIsTypeOf(SealedBid) then AnnounceType::SealedBid
else if self.oclIsTypeOf(Multiple) then AnnounceType::Multiple
else if self.oclIsTypeOf(Dutch) then AnnounceType::Dutch
endif endif endif endif endif endif endif)
in
result = publFee->union(upgrFee)->union(closeFee)
```

The analysis class diagram of Figs. 6 to 9 provides the full specification of the static structure of our electronic marketplace analysis pattern. This static structure (or an adaptation of it) is shared by all possible marketplaces, independently of the topic they address.

3.3 The System Behaviour Model

The use case model of section 3.1 does not provide a precise definition of the way users interact with the information system. This is done by means of the *system behaviour model*. The system behaviour is defined as a 'black box' during analysis and it is illustrated by means of system sequence diagrams and the contracts of the operations appearing in these diagrams. A system sequence diagram shows, for a particular course of events within a use case, the external actors that interact directly with the system, the system as a 'black box', and system events that actors generate.

Due to space limitations, we only show here system sequence diagrams and operation contracts for some relevant use cases. We believe that the selected use cases illustrate the most important behavioural aspects of the system. The following sequence diagram corresponds to the typical course of events of the use case *Open cry auction,* which appears in Fig. 10. The sequence diagrams of the rest of use cases whose goal is publishing an announcement (*Request, Advertisement, Fixed price, Sealed bid auction, Multiple auction* and *Dutch auction*) are very similar to this one.

We assume that the product to be announced has already been registered and therefore we can use its object identifier. Similarly, we also consider that before any operation that requires user authentication, the user has logged in and the system has returned his object identifier.

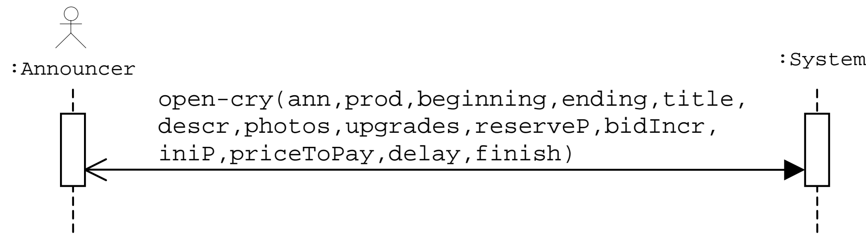

Fig. 10.- Sequence diagram for the use case *Open cry auction*

A system sequence diagram describes the basic interaction that an actor must perform to execute a given use case without going into particular details on how it will be actually performed in a given user interface. Therefore, it is not surprising that sequence diagrams remain so simple at the analysis level. Such system sequence diagrams may lead to more complex sequence diagrams during the design stage (after the external design of the interfaces of the electronic marketplace) when incorporating particular details of how users interact with a given interface.

The interaction required to execute *open-cry* is very simple. In fact, it is enough to specify the announcer, the product, the information required for all kinds of announcements (*beginning, ending, title, descr, photos* and *upgrades*) and the specific information for open cry auctions (*reserveP, bidIncr, iniP, priceToPay, delay* and *finish*)

A system operation is an operation that the system executes in response to a system event. Each operation is precisely specified by means of a contract that includes its signature; its semantics; its preconditions; i.e. the set of conditions that are guaranteed to be true when the operation is executed; and its postconditions, i.e. the set of conditions that hold after the operation execution. We will specify these conditions in OCL.

Note that, since our goal is to define a non-redundant analysis pattern, the operation contracts will remain quite simple since they must not include checking and dealing with the conditions that are already guaranteed by the constraints that are specified in the analysis class diagram.

The following contract defines the semantics of the operation *open-cry*:

Operation:
```
open-cry(ann: Announcer, prod: Product, beginning: Time, ending:
Time, title: String, descr: String, photos: Sequence(String),
upgrades: Sequence(UpgradeType), reserveP: Money, bidIncr:
Sequence(TupleType(from: Money, to: Money, incr: Real)), iniP: Money,
priceToPay: FinalPrice, delay: Natural, finish: OpenCryEnd)
```

Semantics: Create an announcement in open cry auction format for product *prod,* with the indicated characteristics

Preconditions:
```
prod.User = ann
```

Postconditions:
```
a.oclIsNew() and a.oclIsTypeOf(OpenCry) and a.Product=prod and
a.beginning=beginning and a.ending=ending and a.title=title and
a.description=descr and a.photos=photos and a.upgrades=upgrades and
a.reservePrice=reserve and a.initialPrice=iniP and
a.priceToBePaid=priceToPay and a.delay=delay and a.finishWhen=finish
and a.BidIncrement=bidIncr-> iterate(b;acc=Set{}| acc->including(bi|
bi.oclIsNew() and bi.oclIsTypeOf(BidIncrement) and bi.from=b.from and
bi.to=b.to and bi.incr=b.incr))
```

Bid for a product is another important use case of the e-marketplace, which is divided into three use cases: *Place auction bid, Place purchase bid* and *Place ad bid*. The semantics of *Place auction bid* is defined by means of the following system sequence diagram and operation contract:

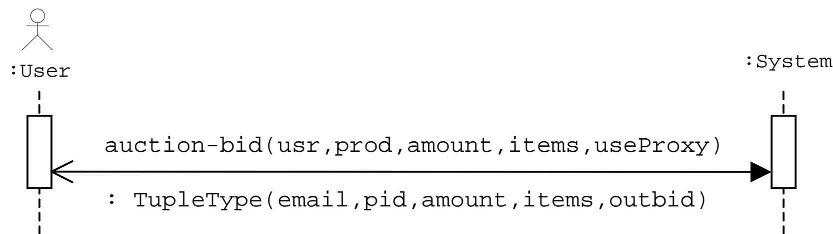

Fig. 11.- Sequence diagram for the use case *Place auction bid*

To place an auction bid, the system needs to know the user (*usr*) , the product he wants to purchase (*prod*), which is offered in an auction, the price he is willing to pay for it (*amount*), the number of items he demands (*items*) and whether he wants to use a proxy or not (*useProxy*). The system returns the e-mail of the bidder (*email*), and the product (*pid*) and the *amount* indicated, in order to send him a confirmation.

Note that in addition to placing the specified bid, this operation must generate automatic bids when necessary. In case *usr* has specified that he wants to use a proxy, the system must generate an automatic bid with the minimum *value* that makes *usr* a winner, in case the *amount* specified is the greatest placed so far. The value of this automatic bid will be the initial price if it is the first bid placed for the product. If *usr* is the first to reach the reserve price, then the value of the automatic bid will be the reserve price. Otherwise, the value will be the amount of the previous winning bid plus its corresponding bid increment.

In case it is not possible for *usr* to become the current winner, it means that there is another user that has placed a bid with a greater amount. Then, if this user specified when he placed his bid that he wanted to use a proxy, the operation *auction-bid* must generate an automatic bid for him. Its value will be the *amount* specified by *usr* plus the bid increment corresponding to this amount.

Once the corresponding automatic bids are generated, outbid users must be notified. If the new bid is the winner, an e-mail will be sent to the previous winner. On the contrary, *usr* will be notified that he has been outbid, and will need to place a new bid if he is still interested in the product and is willing to pay a greater amount for it. This kind of information facilities are very important in the context of e-marketplaces, since they allow users to be informed of the changes that occur to their bids.

Placing purchase bids is much simpler, since there are no proxies involved. However, the sale must be closed when there are no remaining items, that is, when the sum of the items demanded is equal to the quantity of items offered. The operation will only register a new *PurchaseBid* and will close the sale in case it is necessary, invoking the operation *close-ann* of the use case *Close announcement.*

As for ad bids, they are simply recorded and the announcement will be closed when its ending time arrives.

The operation *auction-bid* is defined in the following contract.

Operation: `auction-bid(usr: User, prod: Auction, amount: Money, items: Natural, useProxy: Boolean): TupleType(email: String, pid: String, amount: Money, items: Natural, outbid: String)`

Semantics: Place an auction bid for the announcement *prod*, with the specified *amount, items* and the option *useProxy*. The user *usr* will receive a confirmation e-mail and the user *outbid* will be notified that he must bid again if he wants to continue in the auction, because the new *amount* is greater than his.

Preconditions:

Postconditions:

```
b.oclIsNew() and b.oclIsTypeOf(AuctionBid) and b.User=usr and
b.Time=now() and b.Offer=prod and b.amount=amount and
b.itemsWanted=items and b.useProxy=useProxy and
```

```
result.email=usr.email and result.pid=prod.Product.pid and
result.amount=amount and result.items = items and
if useProxy then
--automatic bid generation
  ab.oclIsNew() and ab.oclIsTypeOf(AutomaticBid) and ab.AuctionBid=b
  and ab.instant=now() and
  if prod.currentPrice < prod.reservePrice and amount >=
  prod.reservePrice then
    ab.value = prod.reservePrice
  else
    if prod.winner->isEmpty() then ab.value = prod.initialPrice
    else
      if prod.winner->includes(b) then
        let newPrice:Money = prod.BidIncrement->select(bi|
        bi.from<=prod.currentPrice@pre and
        bi.to>=prod.currentPrice@pre).incr + prod.currentPrice@pre
        in
        if newPrice > b.amount then ab.value=b.amount
        else ab.value = newPrice
        endif
      else
        ab.value=b.amount and
        if prod.winner->first().oclAsType(AuctionBid).useProxy then
          let newPrice:Money = prod.BidIncrement->select(bi|
          bi.from<=b.amount and bi.to>=b.amount).incr + b.amount
          in
          ab2.oclIsNew() and ab2.oclIsTypeOf(AutomaticBid) and
          ab2.AuctionBid=prod.winner->first() and ab2.instant=now() and
          if newPrice > prod.winner->first().amount then
            ab2.value=prod.winner->last().amount
          else ab2.value=newPrice
          endif
        endif
      endif
    endif
  endif
endif
and result.outbid=
if prod.winner@pre = prod.winner then usr.email
else prod.winner@pre->first().User.email
endif
```

Another important use case in an e-marketplace is *Cancel bid*. By means of this use case, a user can retract a bid he has placed, and is notified of the cancellation. This use case can also be initiated by an announcer that wants to cancel a bid placed for his product. Its sequence diagram is shown in Fig. 12, followed by its corresponding operation contract.

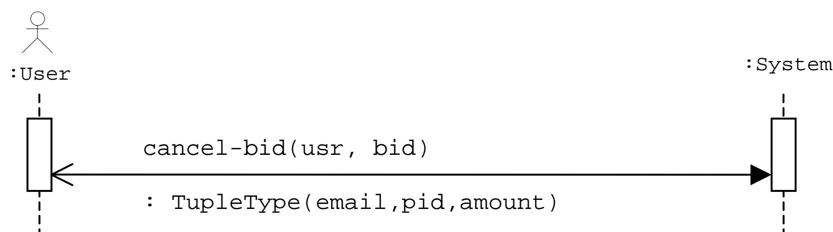

Fig. 12.- Sequence diagram for the use case *Cancel bid*

Operation: `cancel-bid(usr: User, bid: Bid): TupleType(email: String, pid: String, amount: Money)`

Semantics: Cancel the bid *bid* and notify by e-mail the user who placed it. The notification consists of the identifier of the product (*pid*) and the *amount* bidden.

Preconditions: `bid.User = usr or bid.Sale.Product.User=usr`

Postconditions: `bid.oclIsTypeOf(CancelledBid) and result.email = bid.User.email and result.pid = bid.Offer.Product.pid and result.amount = bid.amount`

The use case *Cancel announcement* includes *Cancel bid*, since when an announcement is cancelled all the bids placed for it are no longer valid, and the bidders must be notified. The following are the sequence diagram and operation contract that define the semantics of *Cancel announcement*.

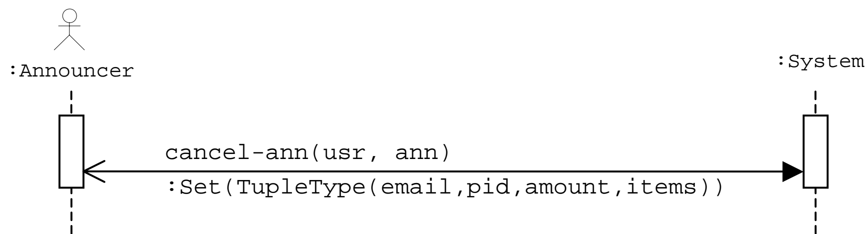

Fig. 13.- Sequence diagram for the use case *Cancel announcement*

Operation: `cancel-ann(usr: User, ann: Announcement): Set(TupleType(email: String, pid: String, amount: Money, items: Natural))`

Semantics: Cancel the announcement *ann*, published by *usr,* together with all the bids placed for it, in case it is a sale. The operation returns a set of all the notifications that must be sent by e-mail.

Preconditions: `ann.Product.User = usr`

Postconditions:

```
ann.oclIsTypeOf(Cancelled) and
if ann.oclIsTypeOf(Sale) then
  ann.oclAsType(Sale).Bid->iterate(b:Bid; result=Set{}| result->
  including(cancel-bid(usr,b)))
endif
```

Apart from cancelling their announcements, announcers can close them whenever they feel appropriate, meaning that the announcement finishes before it was scheduled but the announcer still wants to sell the product to the current winner or winners. When closing an announcement, its current winner or winners are notified that the sale has finished and they are the highest bidders. They receive an e-mail with the price they have to pay, the number of items they have won and the e-mail of the announcer. The announcer receives an e-mail with the e-mails of the winners, the final price of the sale and the number of items sold. All the non winning bidders are also notified of the ending of the announcement. The sequence diagram and operation contract corresponding to the use case *Close announcement* are the following.

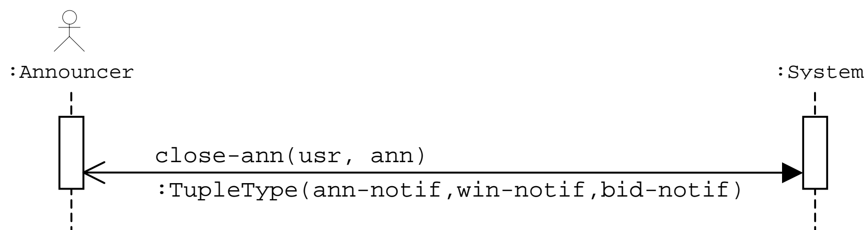

Fig. 14.- Sequence diagram for the use case *Close announcement*

Operation:

```
close-ann(usr: User, ann: Announcement):
TupleType(ann-notif: TupleType(email: String, pid: String,
finalPrice: Money, itemsSold: Natural, winners: Set(String)),
win-notif: Sequence(TupleType(email: String, pid: String, price:
Money, itemsWon: Natural, ann-email: String)),
bid-notif: TupleType(emails: Set(String), pid: String))
```

Semantics: Close the announcement *ann*, published by *usr,* and notify the cancelation to the announcer, the winners and the rest of bidders.

Preconditions:

```
ann.Product.User = usr
```

Postconditions:

```
ann.oclIsTypeOf(Closed) and
if ann.oclIsTypeOf(Offer) then
  if ann.oclIsTypeOf(Auction) and ann.oclAsType(Sale).Bid->notEmpty()
  and ann.oclAsType(Sale).currentPrice >=
  ann.oclAsType(Offer).reservePrice then
    ann.oclAsType(Closed).accepted
  else if ann.oclIsTypeOf(FixedPrice) and ann.oclAsType(Sale).Bid->
  notEmpty() then
    ann.oclAsType(Closed).accepted endif
  endif
endif
and result.ann-notif.email=ann.email and
result.ann-notif.pid=ann.Product.pid and
if ann.oclIsTypeOf(Sale) then
  result.ann-notif.finalPrice = ann.oclAsType(Sale).currentPrice and
  result.ann-notif.itemsSold = ann.oclAsType(Sale).Bid.itemsWanted->
  sum() and result.ann-notif.winner =
  ann.oclAsType(Sale).winner.User.email
  and Sequence{1..ann.winner->size()}->forall(i |
  result.win-notif->at(i).email = ann.winner->at(i).User.email and
  result.win-notif->at(i).pid = ann.Product.pid and
  result.win-notif->at(i).price = ann.winner->at(i).priceToPay and
  result.win-notif->at(i).itemsWon = ann.winner->at(i).itemsWon and
  result.win-notif->at(i).ann-email = usr.email)
endif
and if ann.oclIsTypeOf(Offer) then
result.bid-notif.emails = ann.Bid->reject(b|
b.oclIsTypeOf(CancelledBid)).User.email and
result.bid-notif.pid = ann.Product.pid endif
```

Another important sequence diagram, with its operation contract, is the one corresponding to the use case *Close expired announcements*. An announcement expires when its ending date arrives or when the conditions specified by the announcer hold, in case the announcement is an open cry, multiple or Dutch auction. Fig. 15 shows the sequence diagram corresponding to this use case, and the operation contract is specified below.

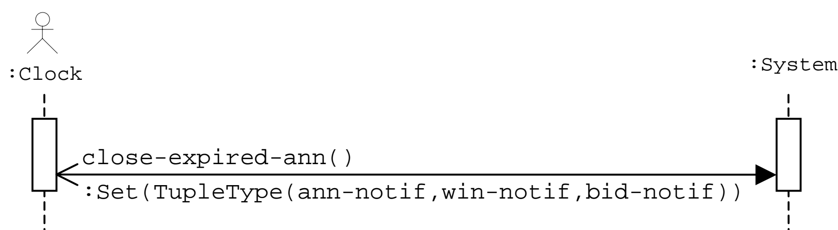

Fig. 15.- Sequence diagram for the use case *Close expired announcements*

.

Operation: `close-expired-ann():`

```
Set(TupleType(ann-notif: TupleType(email: String, pid: String,
finalPrice: Money, itemsSold: Natural, winners: Set(String)),
win-notif: Sequence(TupleType(email: String, pid: String, price:
Money, itemsWon: Natural, ann-email: String)),
bid-notif: TupleType(emails: Set(String), pid: String)))
```

Semantics: Close all the announcements that have arrived to their ending time or have reached the specific conditions specified by the announcer

Preconditions:

Postconditions:

```
let auctionsToClose: Set(Announcement) =
--open cry auctions
OpenCry.allInstances()->select(o| o.oclIsTypeOf(Open))->select(oc|
(oc.finishWhen=OpenCryEnd::endingTime and oc.ending <= now()) or
(oc.finishWhen=OpenCryEnd::noBids and oc.Bid->reject(b|
b.oclIsTypeOf(CancelledBid))->sortedBy(Time.instant)->
last().Time.instant + delay <= now()) or
(oc.finishWhen=OpenCryEnd::anyOfThem and (oc.ending <= now() or
oc.Bid-> reject(b| b.oclIsTypeOf(CancelledBid))->
sortedBy(Time.instant)->last().Time.instant + delay <= now())))->
--sealed bid auctions
union(SealedBid.allInstances()->select(s| s.oclIsTypeOf(Open))->
select(sb | sb.ending <= now()))->
--multiple auctions
union(Multiple.allInstances()->select(m| m.oclIsTypeOf(Open))->
select(m| (m.finishWhen=MultipleEnd::endingTime and m.ending<=now())
or (m.finishWhen=MultipleEnd::noBids and m.Bid->reject(b|
b.oclIsTypeOf(CancelledBid))->sortedBy(Time.instant)->
last().Time.instant + delay <= now()) or
(m.finishWhen=MultipleEnd::noItems and m.winner.itemsWon->sum() =
itemsOffered) or (m.finishWhen=MultipleEnd::anyOfThem and (m.ending
<= now() or m.Bid->reject(b| b.oclIsTypeOf(CancelledBid))->
sortedBy(Time.instant)-> last().Time.instant + delay <= now() or
m.winner.itemsWon->sum() = itemsOffered))))->
--Dutch auctions
union(Dutch.allInstances()->select(d| d.oclIsTypeOf(Open))->
select(d| (d.finishWhen=DutchEnd::endingTime and d.ending <= now())
or (d.finishWhen=DutchEnd::noItems and d.winner.itemsWon->sum() =
itemsOffered) or (d.finishWhen=DutchEnd::reservePrice and
d.currentPrice=reservePrice) or (d.finishWhen=DutchEnd::anyOfThem and
(d.ending <= now() or d.winner.itemsWon->sum() = itemsOffered or
d.currentPrice = reservePrice))))
let restOfAnnToClose: Set(Announcement) =
Announcement.allInstances()->reject(a | a.oclIsTypeOf(Auction) or not
a.oclIsTypeOf(Open))-> select(a | a.ending <= now())
in auctionsToClose->union(restOfAnnToClose)->iterate(a; result =
Set{} | result->including(close-ann(a.Product.User, a)))
```

Finally, we are going to specify the semantics of the use case *Confirm sale.* The announcer must explicitly confirm a sale when the product was offered in auction format and the highest bid for it, once the announcement is closed, doesn't reach the reserve price. In this case, the announcer has the possibility to accept or reject the sale at the current price. When the reserve price of an auction is reached or when the product is offered at a fixed price, the acceptance is automatic. Fig. 16 shows the sequence diagram corresponding to the use case *Confirm sale*, and the operation contract is shown below.

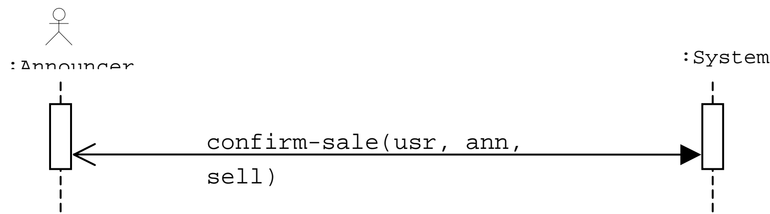

Fig. 16.- Sequence diagram for the use case *Confirm sale*

Operation: `confirm-sale(usr: Announcer, ann: Auction, sell: Boolean)`

Semantics: The announcement *ann* is accepted or rejected as specified by *sell*.

Preconditions: `ann.Product.User = usr and ann.oclIsTypeOf(Closed) and ann.currentPrice <= ann.reservePrice`

Postconditions: `ann.oclAsType(Closed).accepted = sell`

3.4 The Analysis State Model

The analysis state model consists of a set of statechart diagrams that show the events that originate changes on object states and specify the system response as a consequence of these events. The state model includes a statechart diagram for each object class with an important dynamic behaviour.

At the analysis level, statechart diagrams must be defined by means of a *protocol state machine* [OMG03a, 464]. The behaviour of each operation appearing in a protocol state machine is defined by an associated contract (as we have seen in the previous section), rather than through action expressions on transitions. This is why our statechart diagrams remain so simple and why we do not need to specify action expressions at the analysis level.

In the e-marketplace, only *bids* and *announcements* have an important dynamic behaviour that advises the definition of a state diagram for each of these classes.

As far as bids are concerned, a *Bid* can only be in two states: cancelled or not cancelled. The operation *cancel-bid* is responsible for this transition, as shown in Fig. 17.

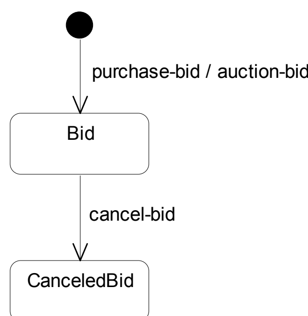

Fig. 17. – Statechart diagram for *Bid*

When an announcement is created, in can be *Pendant* or *Open*, depending on the beginning date specified by the announcer. From any of these states, it can be *Cancelled*, by means of the operation *cancel-ann*. The transition from *Pendant* to *Open* is performed automatically by means of the derivation rules that define the population of the derived subclasses *Pendant* and *Open* in the conceptual schema (see Section 3.2).

Once an announcement is *Open*, it can be closed manually by the announcer or automatically by the system. Both cases are included in the operation *close-ann*. This operation accepts automatically the sale for fixed price and auction announcements whose reserve price has been reached. When the reserve price of an auction has not been reached, the announcer can accept or reject the sale of the

product. For the rest of formats (request announcement and advertisement), no acceptance is done, since they do not require an economic transaction by means of the marketplace.

Figure 18 shows the statechart diagram that specifies this dynamic behaviour. The system operations that cause the transitions between states have already been defined in section 3.4.

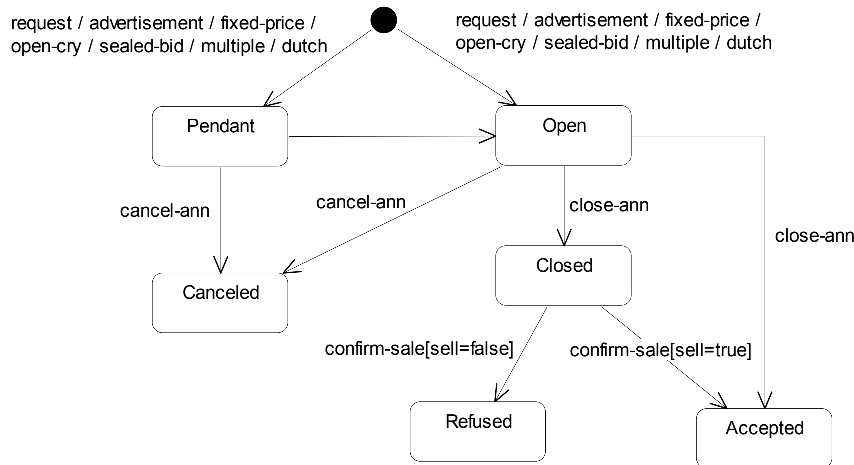

Fig. 18. – Statechart diagram for *Announcement*

4 Related Work

Previous proposals related to electronic marketplace development have been mainly concerned with the design of the information system rather than with its specification. For this reason, these proposals have addressed mainly architectural and technological issues related to electronic marketplace development. Due to the technology-independent nature of our analysis pattern and to the different field of concern, we think it would be too difficult (and perhaps even unfair) to try to establish an exhaustive comparison with them. Nevertheless, it is worthy to summarize the main achievements in this area.

One of the first papers to address electronic marketplace development is [KF98] that reasons about the design of an information system for auctioning. In this sense, it outlines a process flow model that describes the different actions that should be taken into account to implement different types of auctions. Moreover, the paper addresses technological issues such as security requirements, repositories and servers that may participate in the system behaviour, usability, navigation of the web site, etc.

From a similar perspective, [Dog98] describes an architecture for an open marketplace exploiting the workflow technology and data exchange and metadata representation over the web. The main concern of this proposal relies also on technology issues thus addressing distribution infrastructure, architecture of the marketplace, communication of agents in the system, workflow implementation, etc.

More recently, [TM00] propose an agent-based intermediary architecture intended as a generalized platform for the specification of goods and services. Again, the paper is mainly devoted to describing algorithms and technologies used as well as the architecture of the system proposed, rather than to specifying the conceptual aspects of the system.

All technological issues addressed by the three proposals we have just reviewed are important ones and should be taken into account during the implementation of any conceptual schema drawn from our electronic marketplace analysis pattern.

As far as e-marketplace analysis is concerned, the work closer to ours is [RBM01] which defines a pattern language for online auctions management that gathers the functional aspects involved in an online auction system rather than design or implementation issues. This language allows the definition of the static (structural) properties of the system (in a similar way than we do in our analysis class diagram) and it incorporates several types of auctions and the possibility to bid for them.

We may state several limitations of this language when comparing it to the analysis pattern we propose in this paper. First, it deals only with online auction systems while our pattern covers a broader spectrum since we deal with electronic marketplaces in general. Second, it addresses only the structural part of the system while we address also the behavioural one. Third, it assumes an external (manual) treatment of auctions (including the decision about the winner) while we assume that the system alone (automatically) knows the state of the auction according to the bids that have been performed. We believe that ours is a more realistic option since the auction rules are fixed a priori and, hence, can be directly and autonomously incorporated in the system behaviour. Finally, they do not consider the definition of integrity constraints nor derived information associated to their class diagrams, while we do. This is an important limitation since constraints and derivation rules capture business rules of the application domain [SS02] and, thus, they can be directly incorporated in the analysis pattern definition.

Another interesting proposal is that of [CLL99] which proposes to build Java models by means of archetypes. An archetype is a form or template for some class category that specify attributes, links, methods, plug-in points and interactions that are typical for classes in that category. It distinguishes four different kinds of archetypes: *moment-interval*, *role*, *description* and *party, place or thing*, each of them being modelled by means of a different colour. Using these archetypes, they describe several ready-to-use Java models dealing with making or buying, selling, relating and coordinating and supporting.

We see at least two important differences between this work and the one we propose here. First, their models are not specific to a particular application domain and may be found during the implementation of different information systems. For this reason, they do not address specifically e-marketplace development nor the desirable treatment of auctions and announcements which has been the main concern of our work. Second, they address models from a more technological perspective than ours. Their interest is to describe models to be executable in Java while we are mainly concerned on the conceptual specification of the concepts to be described.

5 Conclusions and Further Work

We have proposed an analysis pattern for electronic marketplaces. Our pattern has been drawn from an external study of some well-known electronic marketplaces (www.ebay.com , www.onsale.com, www.amazon.com and www.monster.com) and it covers both the structural and the behavioural properties of the main functionalities provided by an e-marketplace: determining product offerings, searching for products and price discovery. Hence, our pattern is general enough to allow the definition of an analysis model for a new electronic marketplace just by tailoring it to the particular needs of users and owners of that marketplace.

Accurate and precise conceptual modelling is an essential premise for a correct development of an electronic marketplace. Our analysis pattern is able to facilitate this difficult and time-consuming task since it provides an additional value to increase reusability (and, thus, quality of the resulting system) during software development.

Finally, and due to the shallow differences we find among ontologies and conceptual schemas, we may assert also that the analysis pattern we have proposed in this paper can also be regarded as an ontology for the electronic marketplace domain. In this sense, our work also represents a step forward to the development of globally accepted (domain specific) analysis patterns (ontologies) which is an important field of open research as stated for instance in [Neu03, SS02].

As further work, we would like to suggest design patterns and reliable solutions that fit our analysis pattern and solve common problems we find during the design of electronic marketplaces.

Acknowledgements

We would like to thank Dolors Costal, Cristina Gómez, Òscar Hernández, Antoni Olivé and Maria-Ribera Sancho for many useful comments to previous drafts of this paper.

This work has been partially supported by the Ministerio de Ciencia y Tecnología and the FEDER funds, under the project TIC2002-00744.

6 References

[Bak98] Y. Bakos. "The Emerging Role of the Electronic Marketplaces on the Internet", Communications of the ACM, Vol. 41, No. 8, pp. 35-42, 1998.

[BMR+96] F. Buschmann; R Meunier; H. Rohnert et al. "Pattern-Oriented Software Architecture. A System of Patterns", John Wiley & Sons, 1996.

[Cab03] J. Cabot. "La relación de materialización en UML", VIII Jornadas de Ingeniería del Software y Bases de Datos (JISBD'03), 2003.

[CLL99] P. Coad; E.Lefebvre; J. de Luca. "Java Modeling in Color", Prentice-Hall, 1999.

[Coa92] P. Coad. "Object-Oriented Patterns", Communications of the ACM, Vol. 35, No. 9, pp. 152-159, 1992.

[Cou99] A. Coulson. "Electronic Commerce: the Ever-Evolving Online Marketplaces", IEEE Communications Magazine, Vol. 37, No. 9, pp. 58-60, 1999.

[CST02] D. Costal; M.R. Sancho; E. Teniente. "Understanding Redundancy in UML Analysis Models", Proc. of the 14th Int. Conf on Advanced Information Systems Engineering (CAiSE'02), LNCS 2348, Springer, pp. 659-674, 2002.

[Dog98] A. Dogac et al. "A Workflow Based Electronic Marketplace on the Web", SIGMOD Record, Vol. 27, No. 4, pp. 25-31, 1998.

[Fel00] S. Feldman. "Electronic Marketplaces", IEEE Internet Computing, pp. 93-95, July 2000.

[Fer98] E.B. Fernandez. "Building Systems Using Analysis Patterns", Proc. of the 3rd Int. Software Architecture Workshop (ISAW3), ACM, 1998, pp. 37-40.

[FG97] M.S. Fox; M. Grüninger. "On Ontologies and Enterprise Modelling". Int. Conf. on Enterprise Integration Modelling Technology (ICEIMT'97), Springer, 1997.

[Fow97] M. Fowler. "Analysis Patterns–Reusable Object Models", Addison-Wesley, 1997.

[Fow99] M. Fowler. "Analysis Patterns", http://www.martinfowler.com/apsupp, 1999.

[FY00] E.B. Fernandez; X. Yuan. "Semantic Analysis Patterns", Proc. of the 19th Int. Conf on Conceptual Modelling (ER'00), LNCS 1920, Springer, 2000, pp. 183-195.

[GF95] M. Grüninger; M.S. Fox. "Methodology for the Design and Evaluation of Ontologies", Proc. of the Workshop on Basic Ontological Issues in Knowledge Sharing held in conjunction with IJCAI-95, 1995.

[GH02] A. Geyer-Schulz; M. Hahsler. "Software Reuse with Analysis Patterns". Proc. of the 8th American Conference on Information Systems (AMCIS'02), 2002.

[GHJV95] E. Gamma; R. Helm; R. Johnson; J. Vlissides. "Design Patterns – Elements of Reusable Object-Oriented Software", Addison-Wesley, 1995.

[HV99] M.N. Huhns; J.M. Vidal. "Online Auctions". IEEE Internet Computing, pp. 103-105, May-June 1999.

[KF98a] M. Kumar; S.I. Feldman. "Internet Auctions", USENIX Workshop on Electronic Commerce, 1998.

[KF98b] M. Kumar; S.I. Feldman. "Business Negotiations on the Internet"

[Lar98] C. Larman. "Applying UML and Patterns", Prentice Hall, 1998.

[Lar02] C. Larman. "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process", 2nd Ed., Prentice Hall, 2002.

[LGSS99] M.F. López; A. Gómez-Pérez; J.P. Sierra; A.P. Sierra. "Building a Chemical Ontology using Methontology and the Ontology Design Environment", IEEE Intelligent Systems and their Applications, 14(1), pp. 37-45, 1999.

[MDD01] Z. Maamar; E. Dorion; C. Daigle. "Toward Virtual Marketplaces for E-Commerce Support", Communications of the ACM, Vol. 44, No. 12, pp. 35-38, 2001.

[Neu03] E. Neuhold. "Semantic Web Applications Modelling". 22nd Int. Conf on Conceptual Modelling (ER'03), LNCS 2813, Springer, 2003, Keynote Speech.

[Oli03a] A. Olivé. "Integrity Constraints Definition in Object-Oriented Conceptual Modeling Languages". 22nd Int. Conf on Conceptual Modelling (ER'03), LNCS 2813, Springer, pp. 349-362, 2003.

[Oli03b] A. Olivé. "Derivation Rules in Object-Oriented Conceptual Modeling Languages". Proc. of the 15th Int. Conf on Advanced Information Systems Engineering (CAISE'03), LNCS 2681, Springer, pp. 404-420, 2003.

[OMG03a] OMG. "UML 2.0 Superstructure Specification", August 2003.

[OMG03b] OMG. "Response to the UML 2.0 OCL RfP (ad/2000-09-03)", January 2003.

[Pre00] R. Pressman. "Software Engineering: A Practitioner's Approach", Fifth Edition. McGraw-Hill, 2000.

[RBL+91] J. Rumbaugh; M. Blaha; W. Lorensen et al. "Object Oriented Modelling and Design", Prentice-Hall, 1991.

[RBM01] R. Ré; R.T.V. Braga; P.C.Masiero. "A Pattern Language for Online Auctions Management", Proc of the 8th Conference on Pattern Languages of Programs (PLOP'01).

[RJB99] J. Rumbaugh; I. Jacobson and G. Booch. "The Unified Modelling Language Reference Manual", Addison-Wesley, 1999.

[SS02] V. Sugumaran; V.C. Storey. "Ontologies for Conceptual Modelling: their Creation, Use and Management". Data and Knowledge Engineering, Vol. 42, pp. 251-271, 2002.

[Ten03] E. Teniente. "Analysis Pattern Definition in the UML", Proc. of the IRMA conference. Idea Group Pub., pp. 774-777, 2003.

[TM00] G. Tewari; P. Maes. "Design and Implementation of an Agent-Based Intermediary Infrastructure for Electronic Markets", EC'00, pp. 86-94, 2000.

[Woh00] P. Wohed. "Conceptual Patterns for Reuse in Information Systems Analysis", Proc. of the 12th Int. Conf on Advanced Information Systems Engineering (CAiSE'00), LNCS 1789, Springer, pp. 157-175, 2000.