Distributed Artificial Intelligence and Intelligent Agents (ID2209): Project assignment

Kim Hammar, Stockholm 16446

kimham@kth.se

I. Introduction

The work presented in this report is part of the final project in the course Distributed Artificial Intelligence and Intelligent Agents. The focus of the project is on Agent Oriented Software Engineering, where I apply various different methodologies and compare them. The methods are applied to the business case of the SmartMuseum framework as of which have been used during the course for numerous programming assignments.

If agents are to realise their potential as a software engineering paradigm, then it is necessary to develop software engineering techniques that are specifically tailored to them.[14]

II. Task 1 - Modeling with GAIA Methdology

In this section the result of modeling the SmartMuseum framework with the GAIA methodology [14] is presented.

The GAIA methdology is essentially a systematic procedure of transforming a set of articulated requirements for the system/organization to a design. For structural reasons the design is done in steps and is divided into various related models that use different levels of detail. The system in this context is a SmartMuseum Agent Framework, as of following the GAIA methodology [14] I will from here on frequently use the *organization* metaphor when referring to the system.

I. Analysis

I.1 Requirements Statement

I.1.1 Mission Statement

The SmartMuseum organization has the purpose of connecting different people and entities that are in some sense involved in consuming or providing services related to art. The goal of the organization is to improve the overall experience for everyone involved. The organization should make it easier for consumers to view and find interesting art, for art-curators to provide art and reach out to consumers, for tourguides to find interested consumers as well as building relevant tours and finally for artists to sell their work.

I.1.2 Organization Description

The activity of a consumer viewing an art-artifact involves atleast three, sometimes four, or five main divisions: *tour-guide division*, *art-curator division*, *artist-management division*, *user-service* *division* and *artist-division*. The activity is initiated by the consumer who contacts the user-service division and selects some type of art-service, the user-service divison support the consumer in requesting/retrieving the service from either the art-curator division or tour-guide-division. In parellel to managing consumer requests the tour-guide division browses art-artifacts that is curated by the art-curator division. Further more, the art-curator divison participates in auctions for obtaining art-artifacts from the artist-management division, in parallel to managing requests from consumers and tourguides. Finally, the artist-management division initiates auctions for art-artifacts on request from artists.

The activities described above can the be modelled as an organization in the following way. The organization consists of 7 roles. The ArtConsumer (AC) who consumes arts in different forms. The UserHandler (UH) which the consumer uses to purchase and browse services related to art. The TourGuide (TG) which builds and offers virtual tours. The ArtBuyer (AB) who buys art to include in its gallery/museum, the ArtQuoter (AQ) who quotes the price for arts and sells it to consumers. The ArtSeller (AS) who is hired by artists to sell their work to art buyers. And finally the Artist (A) who produces art.

I.2 Roles Model

The following assumption is necessary to avoid making decisions about implementation details when doing the analysis/design.

Assumption 1-$\mathcal{A}$. *Roles can find each other in some way in order to communicate*

Role Schema: ArtConsumer (AC)

Description:
Initiates activity of consuming art, either buying artifact or downloading a virtual tour

Protocols and activities:
DownloadVirtualTour, BuyArt, VisitArtifact, <u>ViewArtifact</u>

Permissions:

reads	supplied *availableServices*	//	*list of services*
	money	//	*money of the consumer*
	userProfile	//	*profile of the consumer*
	visitedArtifacts	//	*list of visited artifacts*
generates	*valuation*	//	*valuation of selected artifact*
	artifactTitle	//	*title of selected artifact*
	virtualTourTitle	//	*title of selected virtual-tour*
	moneyForArtifact	//	*money for selected artifact*
	supplied *virtualTour*	//	*downloaded virtual-tour*
	supplied *auctionResult*	//	*bought artifact or nil*

Responsibilities
Liveness:

$$\textsc{ArtConsumer} = (\text{GetService. ConsumeService})^{\omega}$$
$$\textsc{ConsumeService} = (\text{VisitArtifact} \mid \underline{\text{ViewArtifact}})$$
$$\textsc{GetService} = (\text{DownloadVirtualTour} \mid \text{BuyArt})$$

Safety:

- $moneyForArtifact \leq money$
- $artifactTitle \in availableServices.artifacts$
- $virtualTourTitle \in availableServices.artifacts$

Figure 1: Schema for role ArtConsumer

Role Schema: UserHandler (UH)

Description:

Receives request to buy art-services from consumers and manages the process of the consumer purchasing and obtaining the service.

Protocols and activities:

GetArtifact, GetVirtualTour, GetArtifactsList, GetVirtualTourList, <u>GenerateListOfArtServices</u>

Permissions:

generates	*availableServices*	// *list of services*
	strategy	// *strategy for dutch auction*
reads	supplied *virtualTours*	// *list of virtual tours*
	supplied *artifacts*	// *list of art-artifacts*
	supplied *moneyForArtifact*	// *consumer money to purchase artifact*
	supplied *valuation*	// *consumer valuation of artifact*
	supplied *artifactTitle*	// *title of artifact-purchase*
	supplied *virtualTourTitle*	// *title of virtual-tour selection*
	supplied *virtualTour*	// *virtual-tour downloaded by consumer*
	supplied *auctionResult*	// *artifact bought by consumer or nil*

Responsibilities

Liveness:

$\textsc{UserHandler} = (All)^{\omega}$

$\textsc{All} = (\text{PresentServices} \mid\mid \text{HandleConsumerRequest})^{\omega}$

$\textsc{PresentServices} = \text{GetServices}.\ \underline{\text{GenerateListOfArtServices}}$

$\textsc{GetServices} = \text{GetArtifactsList}.\ \text{GetVirtualToursList}$

$\textsc{HandleConsumerRequest} = \text{GetArtifact} \mid \text{GetVirtualTour}$

Safety:

- $availableServices = artifacts \cup virtualTours$
- $auctionResult \neq nil \implies auctionResult \in artifacts$
- $virtualTour \in virtualTours$

Figure 2: Schema for role UserHandler

Role Schema: TOURGUIDE (TG)

Description:

Responsible for constructing virtual tours of art-artifacts. Looks up available artifacts at curators and then builds different types of tours.
Sends tours to user-handlers.

Protocols and activities:

SendVirtualTours, SendVirtualTour, GetArtifactList, $\underline{\text{BuildVirtualTour}}$

Permissions:

generates	*virtualTour*	//	*virtual tour of art-artifacts*
	virtualTours	//	*list of virtual-tours*
reads	supplied *artifacts*	//	*list of artifacts*
	supplied *virtualTourTitle*	//	*specific virtual-tour title*

Responsibilities
Liveness:

$$\textsc{TourGuideBuilder} = (\text{ConstructTour} \mid\mid [\text{Send}])^{\omega}$$
$$\textsc{ConstructTour} = (\text{GetArtifactList}.\ \underline{\text{BuildVirtualTour}})^{\omega}$$
$$\textsc{Send} = \text{SendVirtualTours} \mid \text{SendVirtualTour}$$

Safety:

- $\forall virtualTour.artifact \quad virtualTour.artifact \in artifacts$

Figure 3: Schema for role TOURGUIDE

Role Schema: ARTBUYER (AB)

Description:
Buys art-artifacts from art-sellers.

Protocols and activities:
BuyArt, SendArtifacts, HandleVisit

Permissions:

generates	$artifacts$	// *list of purchased artifacts*
	$strategy$	// *strategy for dutch auction*
	$valuation$	// *valuation for artifact*
	$moneyForArtifact$	// *money for artifact*
reads	$money$	// *the buyer's money*
	$artifactTitle$	// *title for a specific artifact*
	supplied $artifactResult$	// *bought artifact or nil*

Responsibilities
Liveness:

$$\text{ARTBUYER} = ([\text{BuyArt}] \mid\mid [\text{SendArtifacts}] \mid\mid [\text{HandleVisit}]\ )^{\omega}$$

Safety:

- $moneyForArtifact \leq money$
- $artifactTitle \in artifacts$

Figure 4: Schema for role ARTBUYER

Role Schema: ArtQuoter (AQ)

Description:
Quotes art and resells it to consumers

Protocols and activities:
$\underline{\text{QuoteArt}}$, SellArt, GetArtifacts, SendArtifacts

Permissions:

reads	supplied *artifacts*	// *list of artifacts*
	supplied *artifact*	// *artifact for auction*
generates	*quote*	// *quote of artifact*
	rateOfReduction	// *rate of reduction for dutch auction*
	initialPrice	// *initial price for auction*
	reservePrice	// *reserved price for auction*
	price	// *price auction ended at*
	winner	// *winner of auction or nil*
	artifactResult	// *result of auction*
	bidders	// *bidders of auction*

Responsibilities
Liveness:

$$\text{ArtQuoter} = ((\text{GetArtifacts}.\ \underline{\text{QuoteArt}}.\ \text{SellArt})\ ||\ \text{SendArtifacts})^{\omega}$$

Safety:

- $winner \in bidders$
- $reservePrice \leq price \leq initialPrice$

Figure 5: Schema for role ArtQuoter

Role Schema: ArtSeller (AS)

Description:
Sells art to art-traders/curators.

Protocols and activities:
SellArt, GetArtifact

Permissions:

reads	supplied *artifact*	//	*artifact to be sold*
generates	*rateOfReduction*	//	*rate of reduction for dutch auction*
	initialPrice	//	*initial price for auction*
	reservePrice	//	*reserved price for auction*
	price	//	*price auction ended at*
	winner	//	*winner of auction or nil*
	artifactResult	//	*result of auction*
	bidders	//	*bidders of auction*

Responsibilities
Liveness:

$$\text{ArtSeller} = (\text{GetArtifact. SellArt})^{\omega}$$

Safety:

- $winner \in bidders$
- $reservePrice \leq price \leq initialPrice$

Figure 6: Schema for role ArtSeller

Role Schema: ARTIST (A)

Description:
Sells art to art-traders/curators.

Protocols and activities:
ProduceArt, SendArtifact

Permissions:
generates $artifact$ // *produced artifact*

Responsibilities
Liveness:
$$\text{ARTIST} = (\underline{\text{ProduceArt}}.\ \text{SendArtifact})^{\omega}$$
Safety:

- **true**

Figure 7: Schema for role ARTIST

I.3 Interaction Model

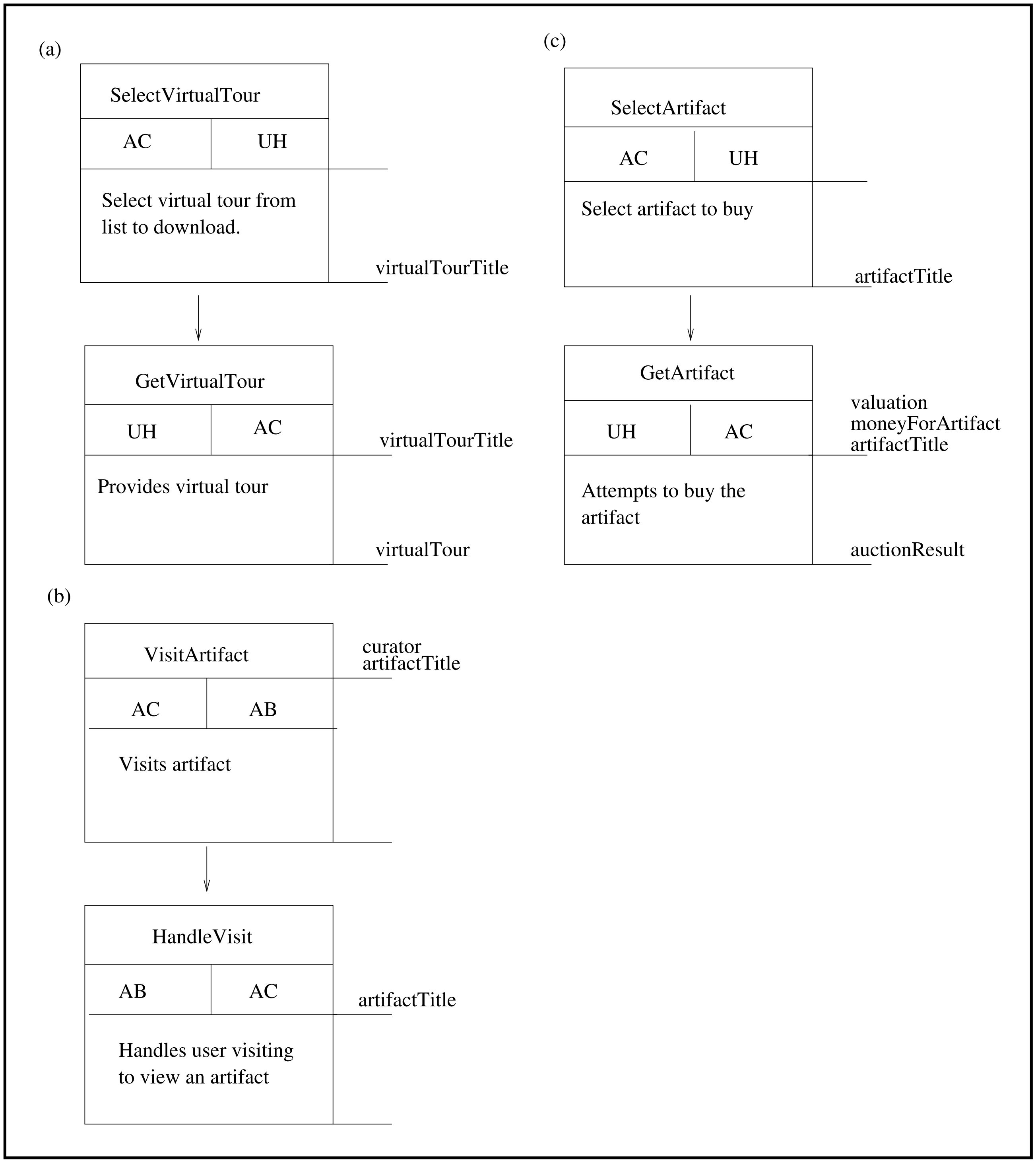


Figure 8: Definition of protocols associated with the ArtConsumer role: (a) DownloadVirtualTour, (b) VisitArtifact (c) BuyArt

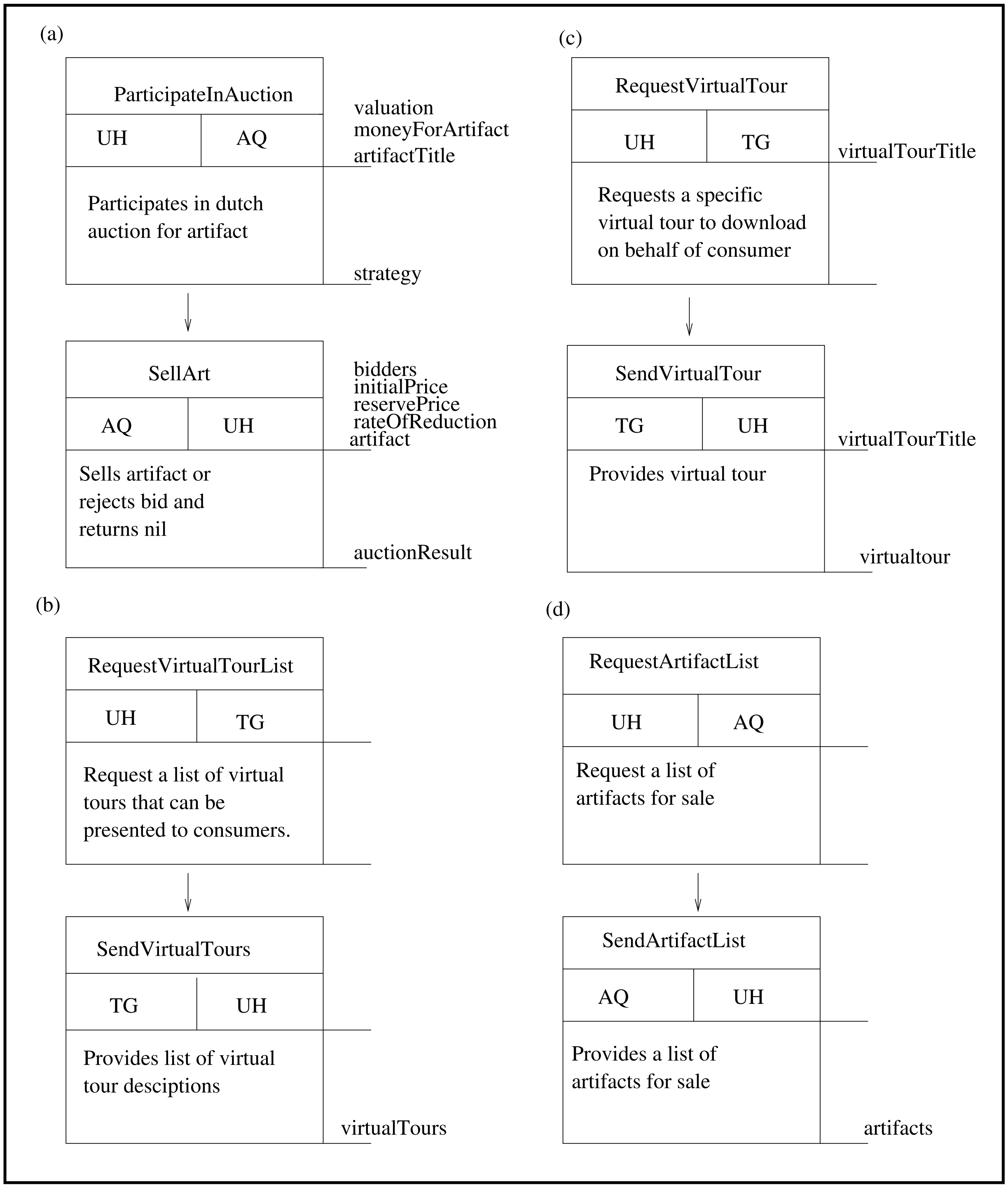


Figure 9: Definition of protocols associated with the UserHandler role: (a) GetArtifact, (b) GetVirtualTourList, (c) GetVirtualTour, (d) GetArtifactsList

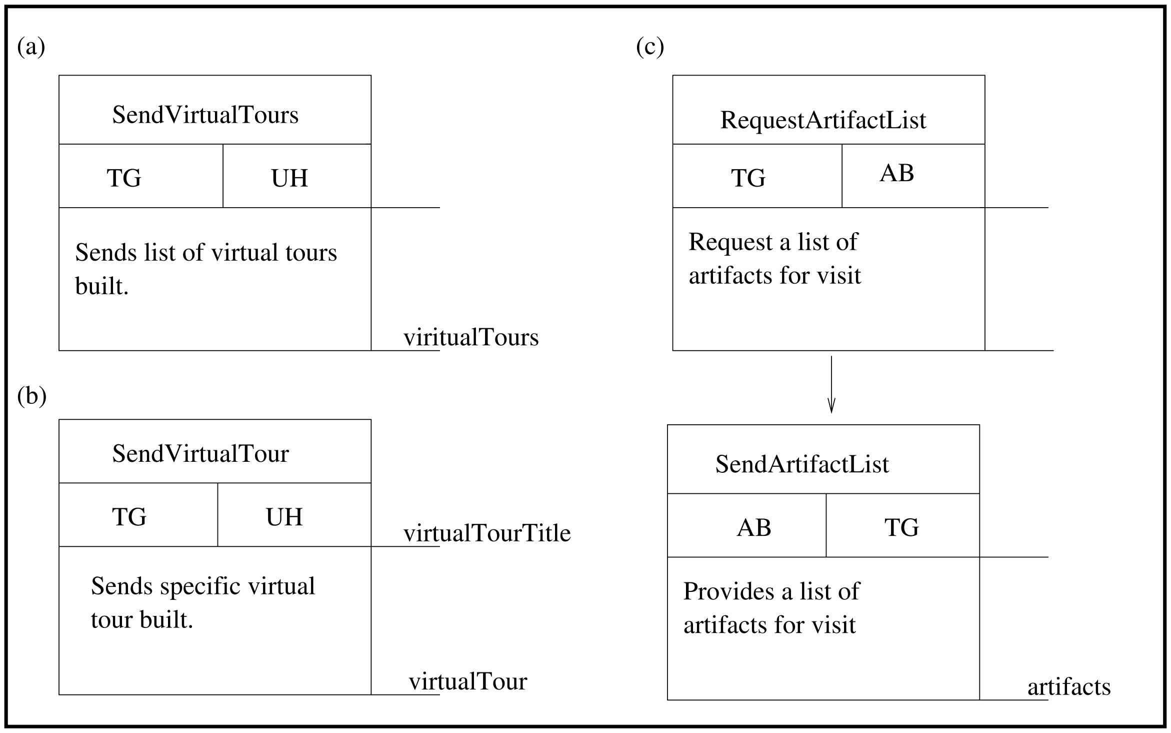

(a)
SendVirtualTours
TG
UH
Sends list of virtual tours built.
viritualTours
(b)
SendVirtualTour
TG
UH
virtualTourTitle
Sends specific virtual tour built.
virtualTour
(c)
RequestArtifactList
TG
AB
Request a list of artifacts for visit
SendArtifactList
AB
TG
Provides a list of artifacts for visit
artifacts

Figure 10: Definition of protocols associated with the TourGuide role: (a) SendVirtualTours, (b) SendVirtualTour, (c) GetArtifactList

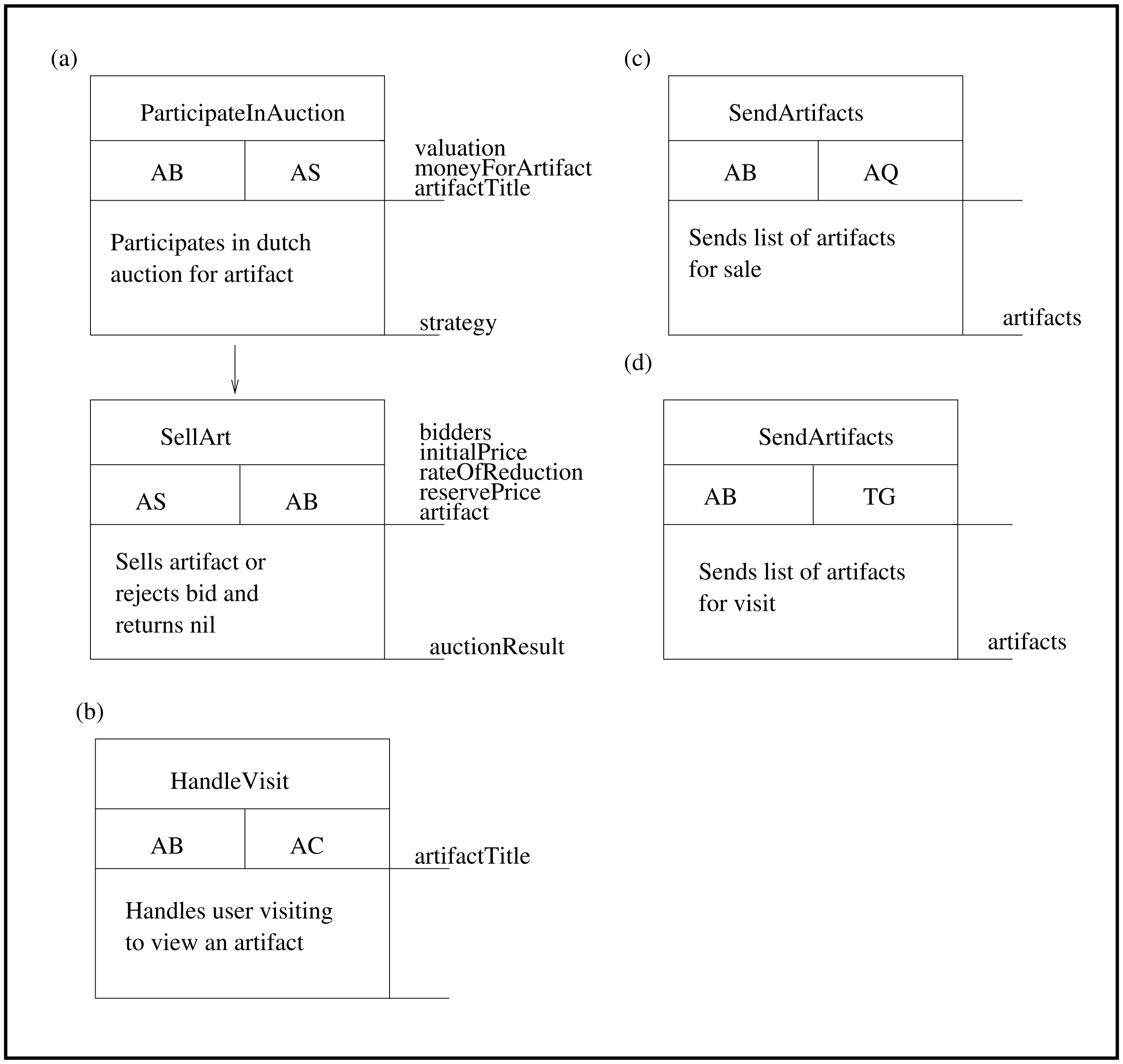


Figure 11: Definition of protocols associated with the ArtBuyer role: (a) BuyArt, (b) HandleVisit, (c) SendArtifacts (1), (d) SendArtifacts (2)

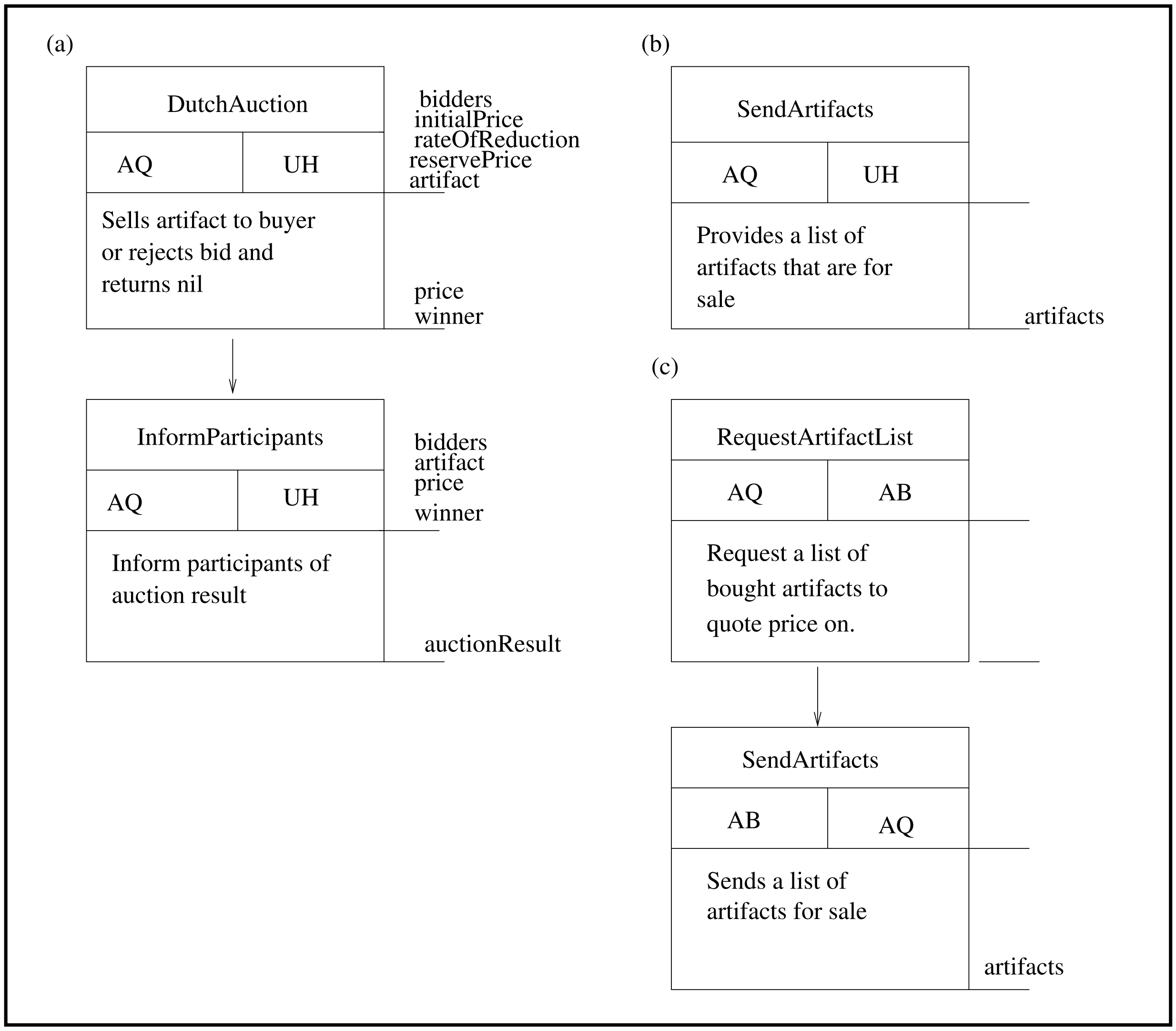


Figure 12: Definition of protocols associated with the ArtQuoter role: (a) SellArt, (b) SendArtifacts, (c) GetArtifacts

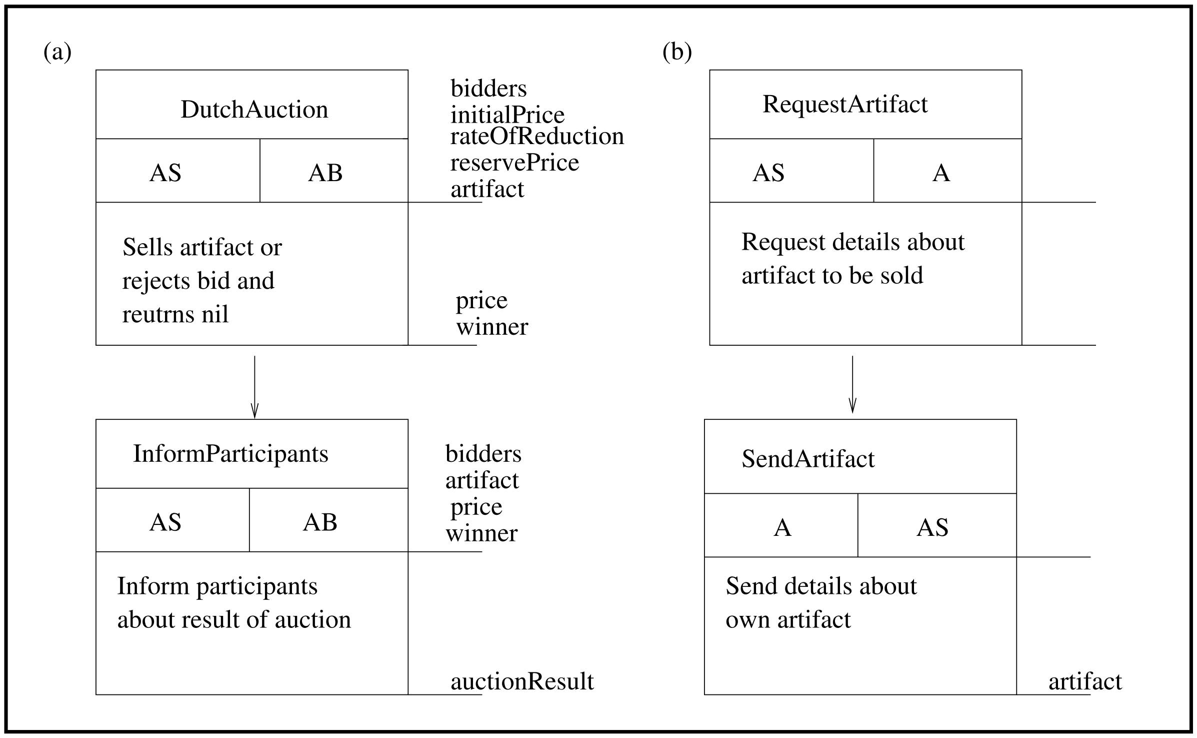


Figure 13: Definition of protocols associated with the ArtSeller role: (a) SellArt, (b) GetArtifact

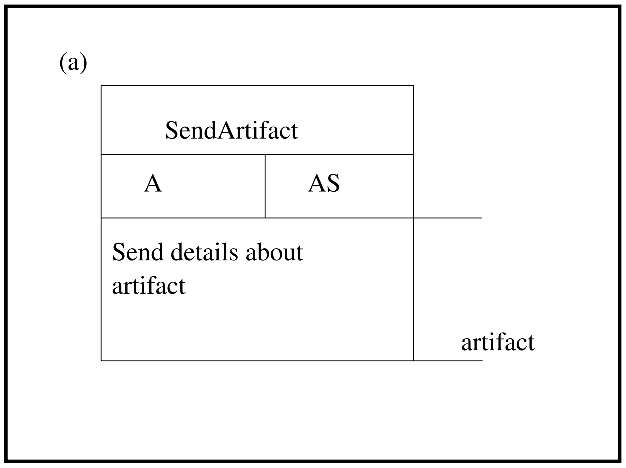


Figure 14: Definition of protocols associated with the Artist role: (a) SendArtifact

II. Design

II.1 Agent Model

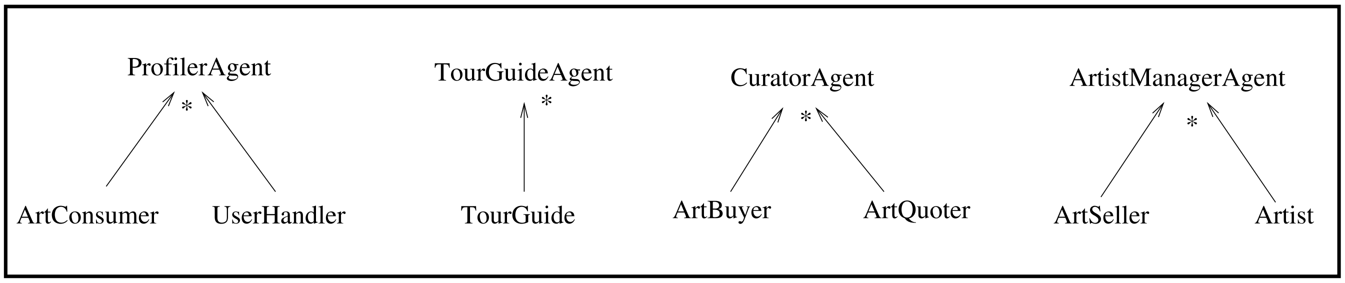


Figure 15: The agent model

II.2 Services Model

Table 1: Services model for agent ProfilerAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
obtain virtual-tour list		$virtualTours$	**true**	$virtualTours \neq nil$
obtain artifact list		$artifacts$	**true**	$artifacts \neq nil$
generate list of services	$virtualTours, artifacts$	$availableServices$	$\exists virtualTours, artifacts$	created list of available services
register as bidder for auction	$auctioneer, artifact$		auction exists	$self \in auctioneer.bidders \wedge strategy \neq nil$
receive CFP	currentPrice		is participating in the auction	**true**
place bid	currentPrice		$currentPrice \leq moneyForArtifact$	bid sent to auctioneer
receive bid result	$accept \vee reject$		have bidded	bid accepted or rejected
informed auction ended	$artifact \vee nil$		participated in auction	informed auction ended and received result
download virtual tour	$tourguide, virtualtour$		$\exists tourguide, virtualtour$	downloaded virtual tour
visitArtifact	$curator, artifactTitle$		$artifactTitle \in curator.gallery.titles$	$artifactTitle \in visitedArtifacts$

Table 2: Services model for agent TourGuideAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
obtain artifact list		$artifacts$	**true**	$artifacts \neq nil$
manage virtual-tour request	$virtualTourTitle$	$virtualTour \vee nil$	**true**	**true**
manage list of virtual-tours-request		$virtualTours$	**true**	**true**
build virtual tour	$artifacts$	$virtualTour$	$artifacts.size > 0$	$virtualTour \neq nil$

Table 3: Services model for agent ArtistManagerAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
get registered bidders		$bidders$	**true**	**true**
send inform-start-of-auction	$bidders$	$informMessage$	bidders are registered	bidders informed about start of auction
send CFP	$bidders$	CFP	bidders are registered and auction ongoing	bidders informed about current price and encouraged to bid
receive bid	bid	$bids$	bidder registered	$bid \in bids$
manage bids	$bids$	$bidResponses$	$bids > 0$	one bid was accepted and the bidder received the good, the rest was rejected and the bidders were informed
modify price	$reservePrice$, $rateOfReduction$, $currentPrice$	$newPrice$	no bids was received	$reservePrice \leq newPrice \leq currentPrice$
send inform-auction-closed	$bidders$, $auctionResult$	$informMessage$	bidders are registered	bidders informed about close of auction

Table 4: Services model for agent CuratorAgent

Service	Inputs	Outputs	Pre-condition	Post-condition
get registered bidders		$bidders$	**true**	**true**
register as bidder for auction	$auctioneer, artifact$		auction exists	$self \in auctioneer.bidders \wedge strategy \neq nil$
receive CFP	currentPrice		is participating in the auction	**true**
place bid	currentPrice		$currentPrice \leq moneyForArtifact$	bid sent to auctioneer
receive bid result	$accept \vee reject$		have bidded	bid accepted or rejected
informed auction ended	$artifact \vee nil$		participated in auction	curator were informed auction ended and received result
manage artifact-list request		$artifacts$	**true**	**true**
manage visit-artifact request	$artifactTitle$	$artifact$	**true**	provided artifact for visit only
quote art	$artifact$	$quote$	**true**	**true**
send inform-start-of-auction	$bidders$	$informMessage$	bidders are registered	bidders informed about start of auction
send CFP	$bidders$	CFP	bidders are registered and auction ongoing	bidders informed about current price and encouraged to bid
receive bid	bid	$bids$	bidder registered	$bid \in bids$
manage bids	$bids$	$bidResponses$	$bids > 0$	one bid was accepted and the bidder received the good, the rest was rejected and the bidders were informed
modify price	$reservePrice$, $rateOfReduction$, $currentPrice$	$newPrice$	no bids was received	$reservePrice \leq newPrice \leq currentPrice$
send inform-auction-closed	$bidders$, $auctionResult$	$informMessage$	bidders are registered	bidders informed about close of auction

II.3 Acquaintance Model

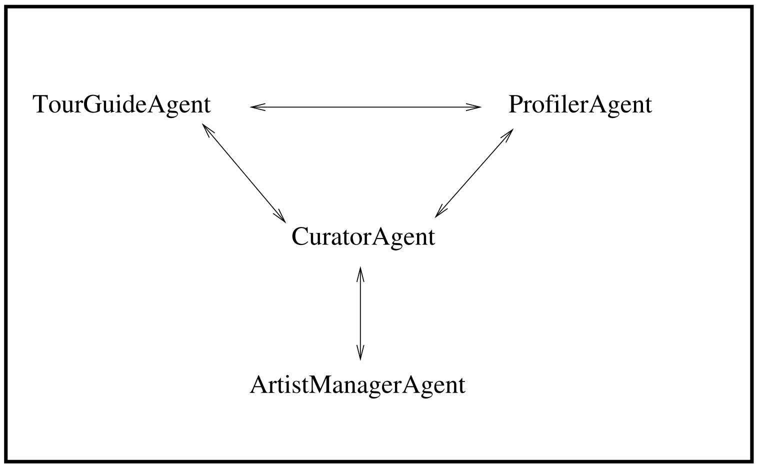


Figure 16: Acquaintance model

II.4 Mobility Model

Assumption 2-$\mathcal{A}$. *I've assumed the mobile architecture that I used for homework 3, i.e that only artistmanager agents and curator agents are mobile and can clone themself. Further more the cardinality of agents and places also follow from this assumption.*

Table 5: Place Types

Place Types	Description	Instances
Heritage Malta Container	Container where art-curators can reside and perform their services and where artistmanager agents can reside temporarily to perform auctions	1
Museo Galileo Container	Container where art-curators can reside and perform their services and where artistmanager agents can reside temporarily to perform auctions	1
ArtistManager Container	Container where artist-manager agents reside and where they come back to after performing auctions	*
ProfilerAgent Container	Container where profiler agents reside	*
TourGuideAgent Container	Container where tour-guide agents reside	*

Table 6: Agents and Places Specification

Agent Type	Mobile	Place Type	Constraints
ProfilerAgent	No	ProfilerAgent Container	
TourGuideAgent	No	TourGuideAgent Container	
CuratorAgent	Yes	Museo Galileo Container, Heritage Malta Container	
ArtistManagerAgent	Yes	ArtistManagerAgentContainer, Museo Galileo Container, Heritage Malta Container	

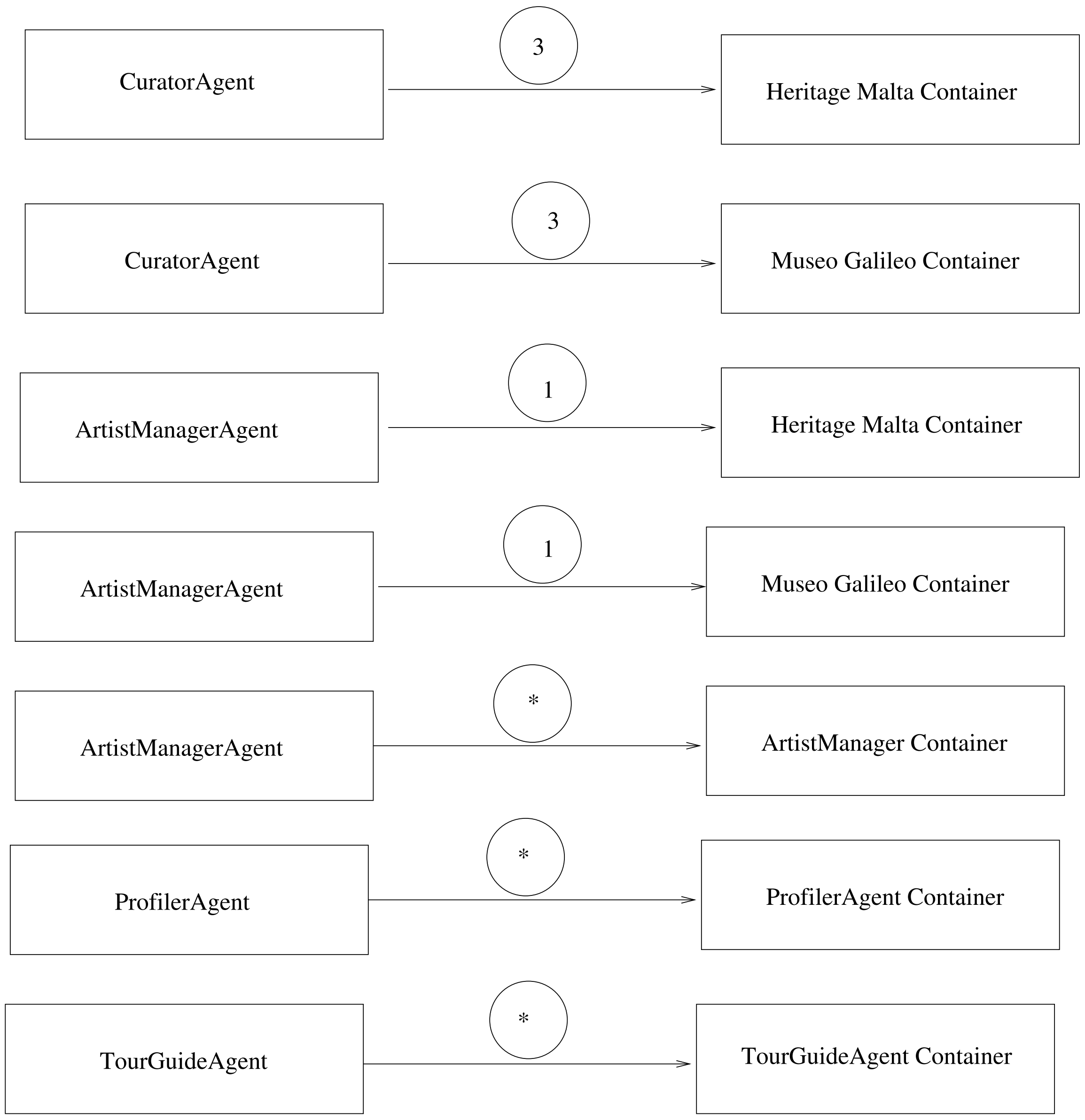


Figure 17: Cardinality of Agents and Places

Agent Type: CuratorAgent

Description: Can be cloned in current container to participate in auctions.
Origin: Heritage Malta Container or Museo Galileo Container.
Final Destination: Same as its origin container.
List of atomic movements:

1	Cloned in Heritage Malta Container
2	Cloned in Museo Galileo Container

Paths:

Cloned in the same container, no paths.

Figure 18: Travel schema for agentCuratorAgent

Agent Type: ArtistManagerAgent

Description: Can move between Museo Galileo Container, Heritage Malta Container and ArtistManager Container.
Origin: ArtistManager Container
Final Destination: ArtistManager Container
List of atomic movements:

1	Move from ArtistManager to Heritage Malta Container.
2	Move from Heritage Malta to ArtistManager Container.
3	Move from ArtistManager to Museo Galieo Container.
4	Move from Museo Galieo to ArtistManager Container.
5	Move from Museo Galieo to Heritage Malta Container.
6	Move from Heritage Malta to Museo Galieo Container.

Paths:

1	1.2
1	3.4
1	1.6
1	3.5
1	3.5.2
1	1.6.4

Figure 19: Travel schema for agentArtistManagerAgent

Agent Type:	ProfilerAgent
Description:	Static agent, not mobile.
Origin:	ProfilerAgentContainer
Final Destination:	ProfilerAgentContainer
List of atomic movements:	
	The agent is static and don't have any atomic movements.
Paths:	
	No paths

Figure 20: Travel schema for agentProfilerAgent

Agent Type:	TourGuideAgent
Description:	Static agent, not mobile.
Origin:	TourGuideAgentContainer
Final Destination:	TourGuideAgentContainer
List of atomic movements:	
	The agent is static and don't have any atomic movements.
Paths:	
	No paths

Figure 21: Travel schema for agentTourGuideAgent

III. Task 2 - Modeling with AgentUML

An alternative to the GAIA modeling approach is stick to UML, which is the dominant way of modeling in general software engineering and in particular object-oriented areas. However in the context of agent-oriented programing the UML standard have some obvious problems which are to be expected since UML was not designed for agent-oriented programming but rather object-oriented. Agent UML is an extension to UML with the purpose of making UML more usable for agent-based systems. In this section the result of modeling the SmartMuseum framework with the AgentUML method, and specifically with the approach used in [11] is presented.

I. The Overall Protocol

I.1 ArtistManager Auction package

High-level overview of the protocol where ArtistManagerAgents auctions art-artifacts to Curator-Agents using dutch auctions.

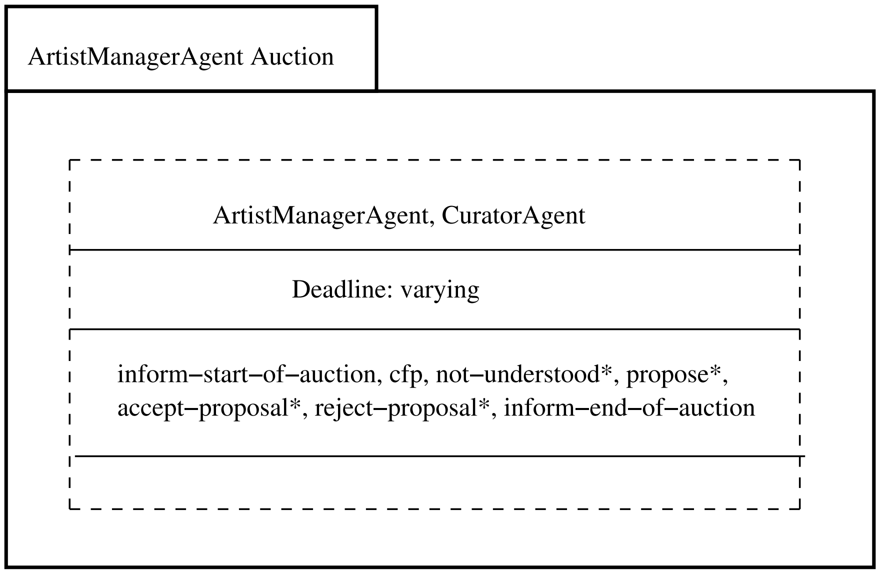

ArtistManagerAgent Auction
ArtistManagerAgent, CuratorAgent
Deadline: varying
inform–start–of–auction, cfp, not–understood*, propose*, accept–proposal*, reject–proposal*, inform–end–of–auction

Figure 22: ArtistManager Auction package

I.2 Curator Auction package

High-level overview of the protocol where CuratorAgents auctions art-artifacts to ProfilerAgents using dutch auctions.

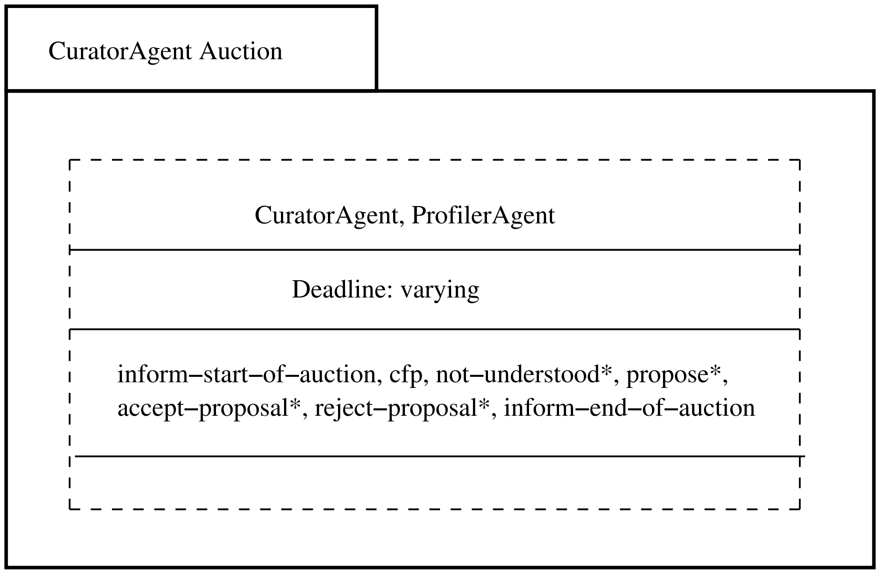

CuratorAgent Auction
CuratorAgent, ProfilerAgent
Deadline: varying
inform–start–of–auction, cfp, not–understood*, propose*, accept–proposal*, reject–proposal*, inform–end–of–auction

Figure 23: Curator Auction package

I.3 BuildVirtualTour package

High-level overview of the protocol where TourGuideAgents sonds the terrain of artifacts at different curators and then build virtual tours of different type.

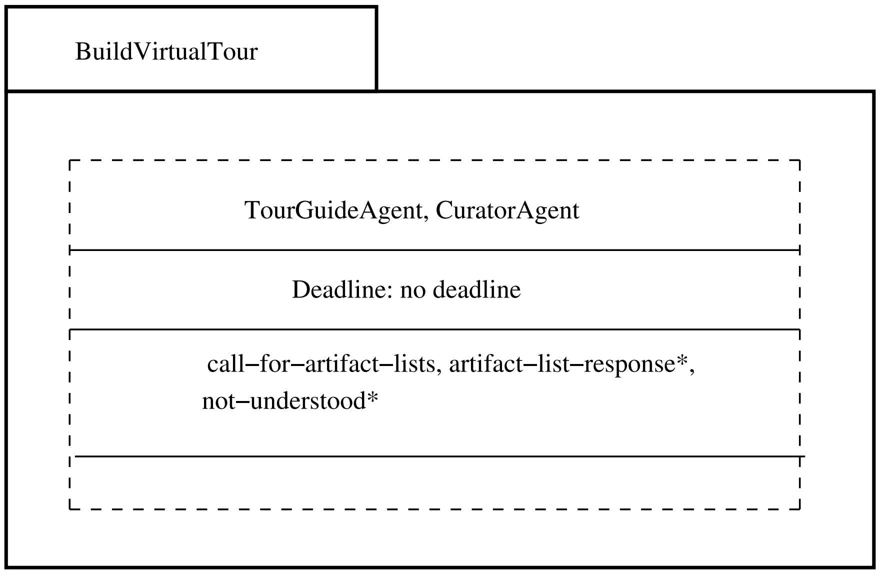

BuildVirtualTour
TourGuideAgent, CuratorAgent
Deadline: no deadline
call–for–artifact–lists, artifact–list–response*,
not–understood*

Figure 24: BuildVirtualTour package

I.4 FindVirtualTour package

High-level overview of the protocol where ProfilerAgents searches for virtual tours and rejects or selects the virtual tours.

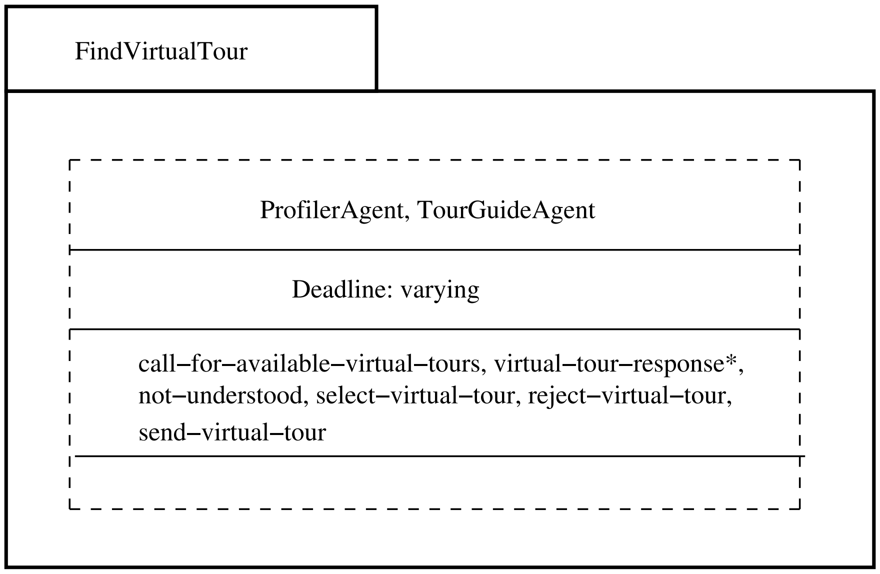

FindVirtualTour
ProfilerAgent, TourGuideAgent
Deadline: varying
call–for–available–virtual–tours, virtual–tour–response*,
not–understood, select–virtual–tour, reject–virtual–tour,
send–virtual–tour

Figure 25: FindVirtualTour package

I.5 VisitArtifact package

High-level overview of the protocol where ProfilerAgents visits artifacts from a virtual tour.

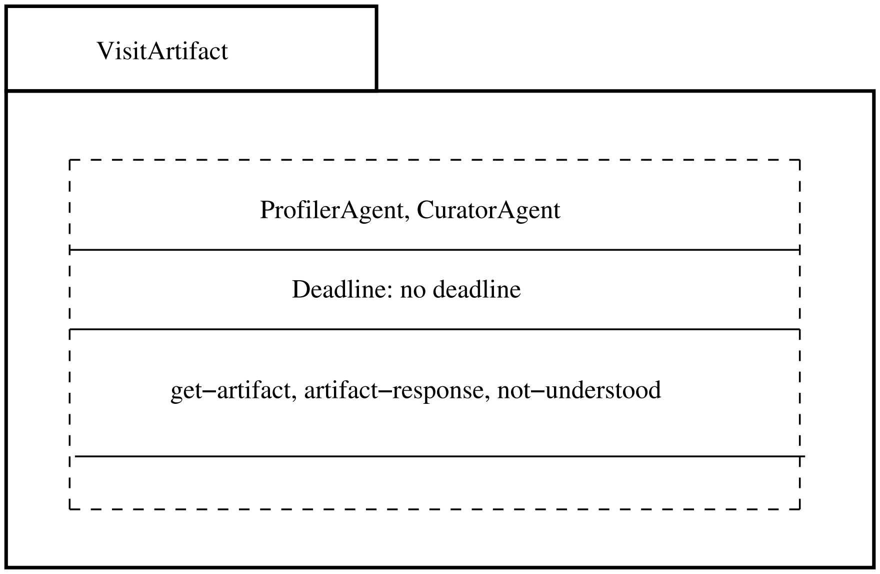

VisitArtifact
ProfilerAgent, CuratorAgent
Deadline: no deadline
get–artifact, artifact–response, not–understood

Figure 26: VisitArtifact package

II. Interactions Among Agents

II.1 ArtistManagerAgent Auction

Sequence diagram over the ArtistManagerAgent Auction protocol. When invoked the ArtistManagerAgent sends an $inform - start - of - auction$ message to n number of CuratorAgents, then it sends a call for proposal with the current price, $cfp - 1$ to n CuratorAgents. CuratorAgents can then either not respond at all or respond with either $not - understood$ or $propose$. The diamond and X indicates that one of the two choices, exlusive, need to be taken. The ArtistManagerAgent will then correspondingly take different action based on which response it receives. If it receives a $not - understood$ response it does nothing, if it receives a $propose$ response (bid) it will either reject or accept it. Finally if the ArtistManagerAgent did'nt receive any bids for a certain amount of time it can either send out another cfp or close the auction by sending a $inform - end - of - auction$ message in case the reservedPrice was reached.

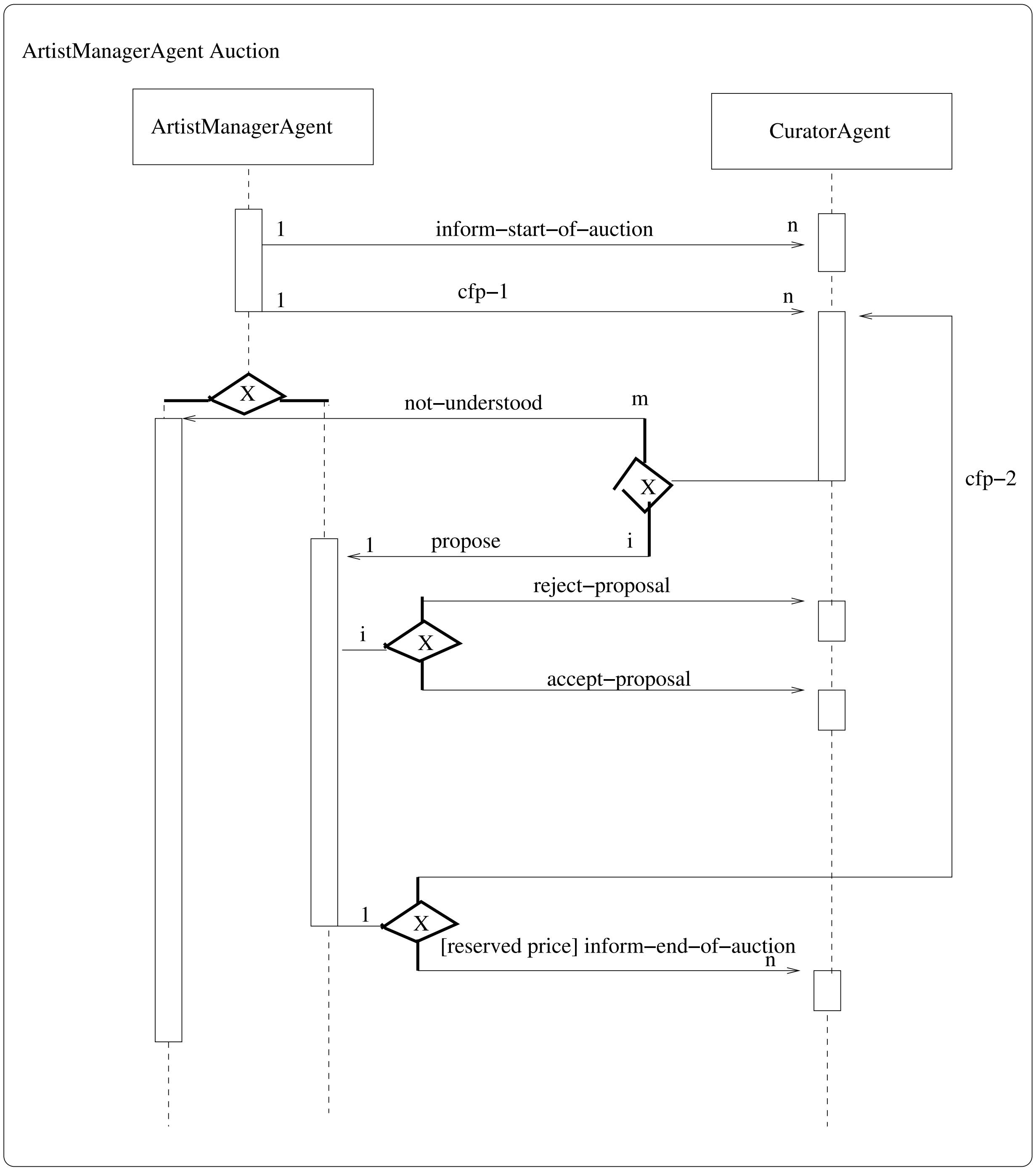


Figure 27: Sequence diagram over the interaction for ArtistManagerAgent Auction

II.2 CuratorAgent Auction

Sequence diagram over the CuratorAgent Auction protocol, this protocol follows the same dutch auction protocol as ArtistManagerAgentProtocol.

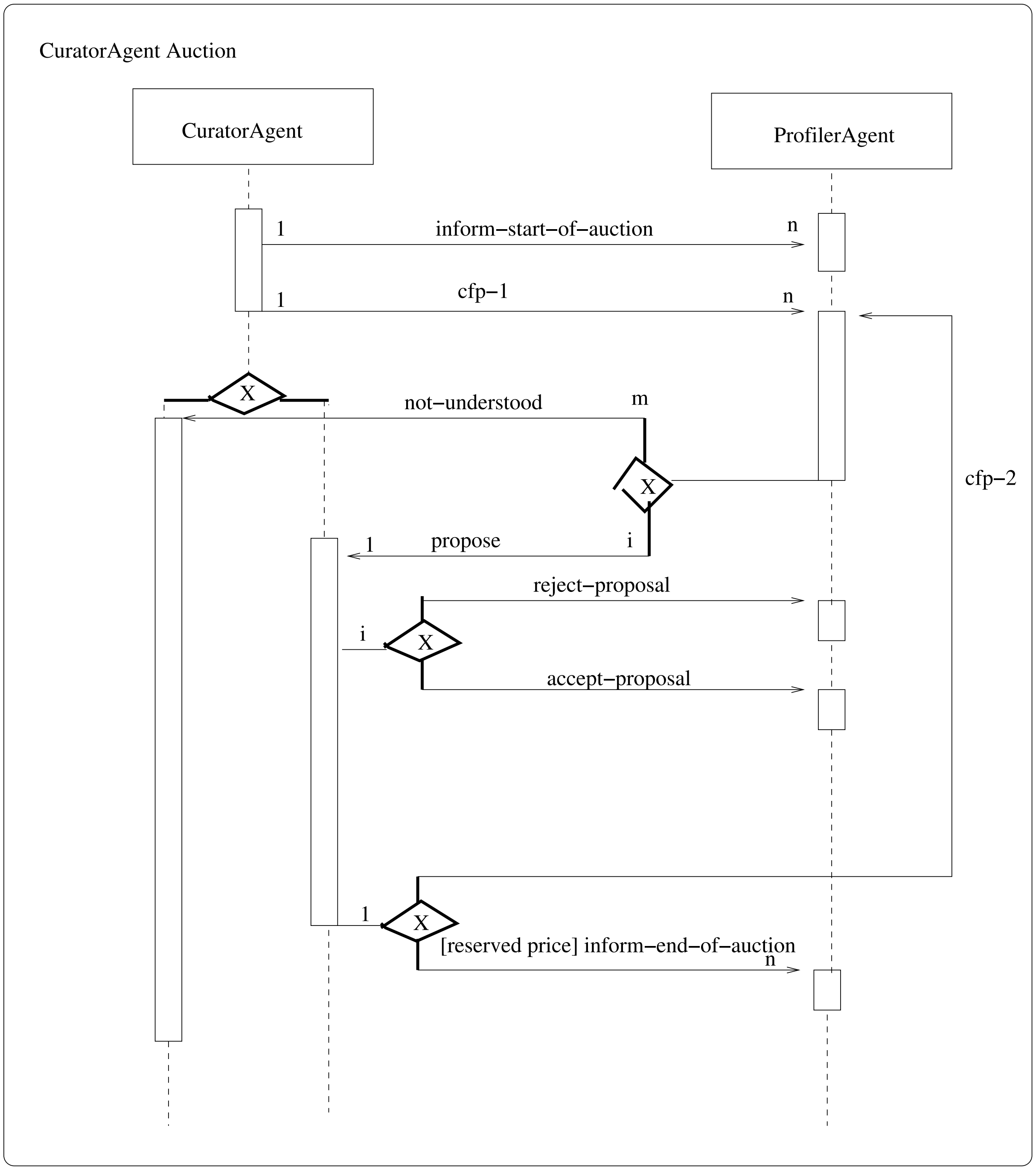


Figure 28: Sequence diagram over the interaction for CuratorAgent Auction

II.3 BuildVirtualTour

Sequence diagram over the BuildVirtualTour protocol. When invoked the TourGuideAgent sends a $call - for - artifact - lists$ to n CuratorAgents. CuratorAgents can then choose to not respond or respond with either $not - understood$ or $artifact - list - response$.

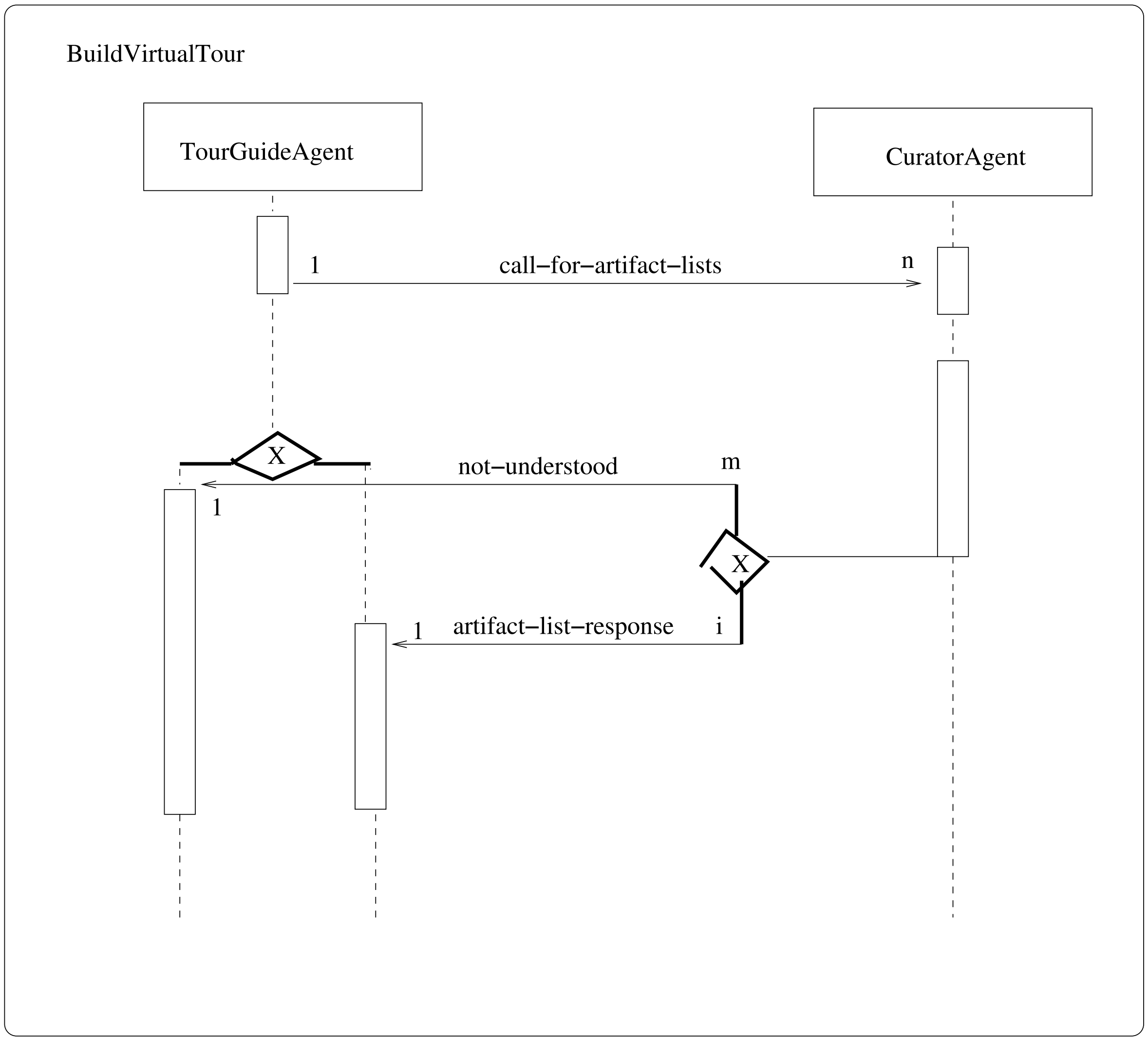


Figure 29: Sequence diagram over the interaction for BuildVirtualTour

II.4 FindVirtualTour

Sequence diagram over the FindVirtualTour protocol. When invoked the ProfilerAgent sends a $call - for - available - virtual - tours$ to n TourGuideAgents. TourGuideAgents can then choose to not respond or respond with either $not - understood$ or $virtual - tour - response$. If the ProfilerAgent receives a $not - understood$ message it does nothing, if it receives a $virtual - tour - response$ it either responds with $select - virtual - tour$ upon the tourguide responds with the full virtual-tour, or *reject − virtual − tour*.

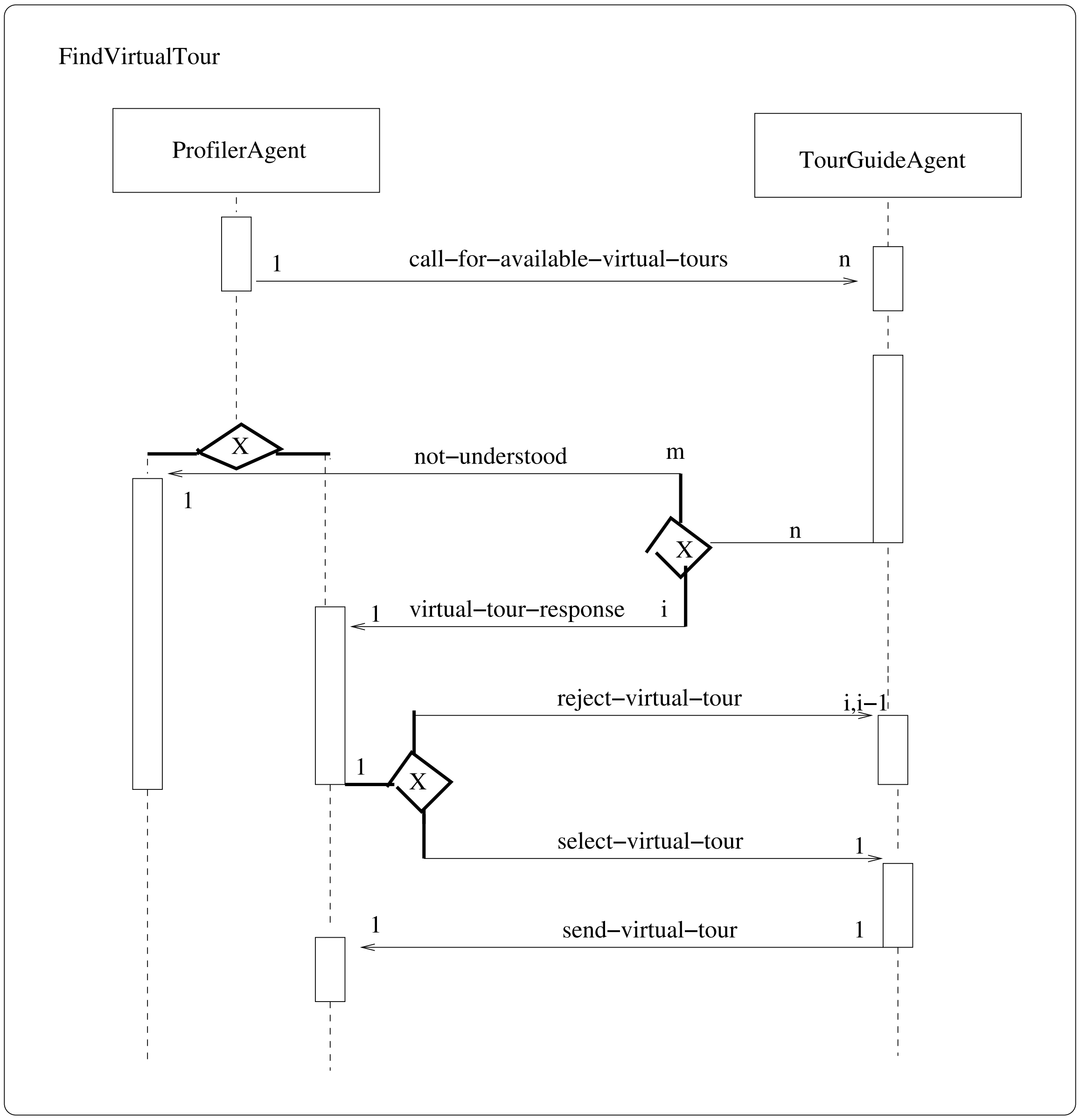


Figure 30: Sequence diagram over the interaction for FindVirtualTour

II.5 VisitArtifact

Sequence diagram over the VisitArtifact protocol. When invoked the ProfilerAgent sends a *get − artifact* message to 1 CuratorAgent. The CuratorAgent can then choose to not respond or respond with either *not − understood* or *artifact − response*.

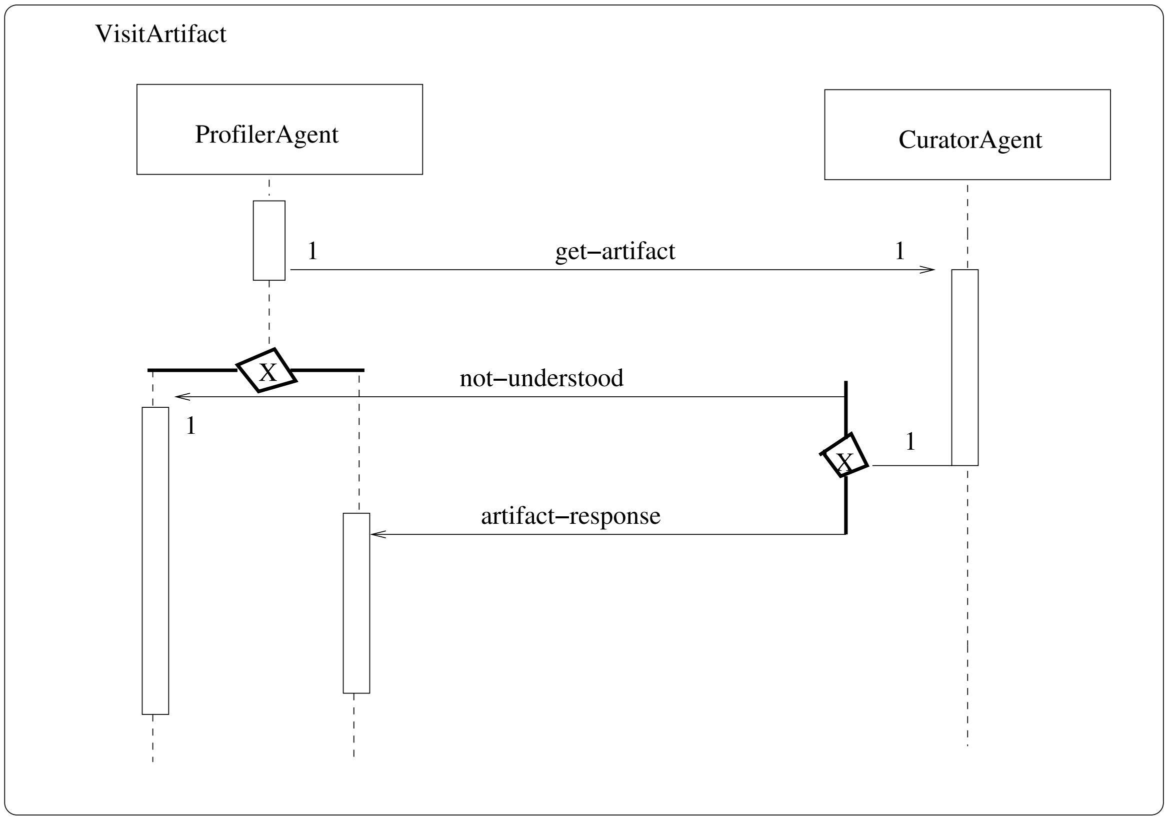


Figure 31: Sequence diagram over the interaction for VisitArtifact

III. Internal Agent Processing

III.1 ArtistManagerAgent

The ArtistManagerAgent contains internal processing for modifying prices in the dutch auction as well as selecting a winner when there is multiple bids. Many of the internal states depends on external events from the interaction protocol.

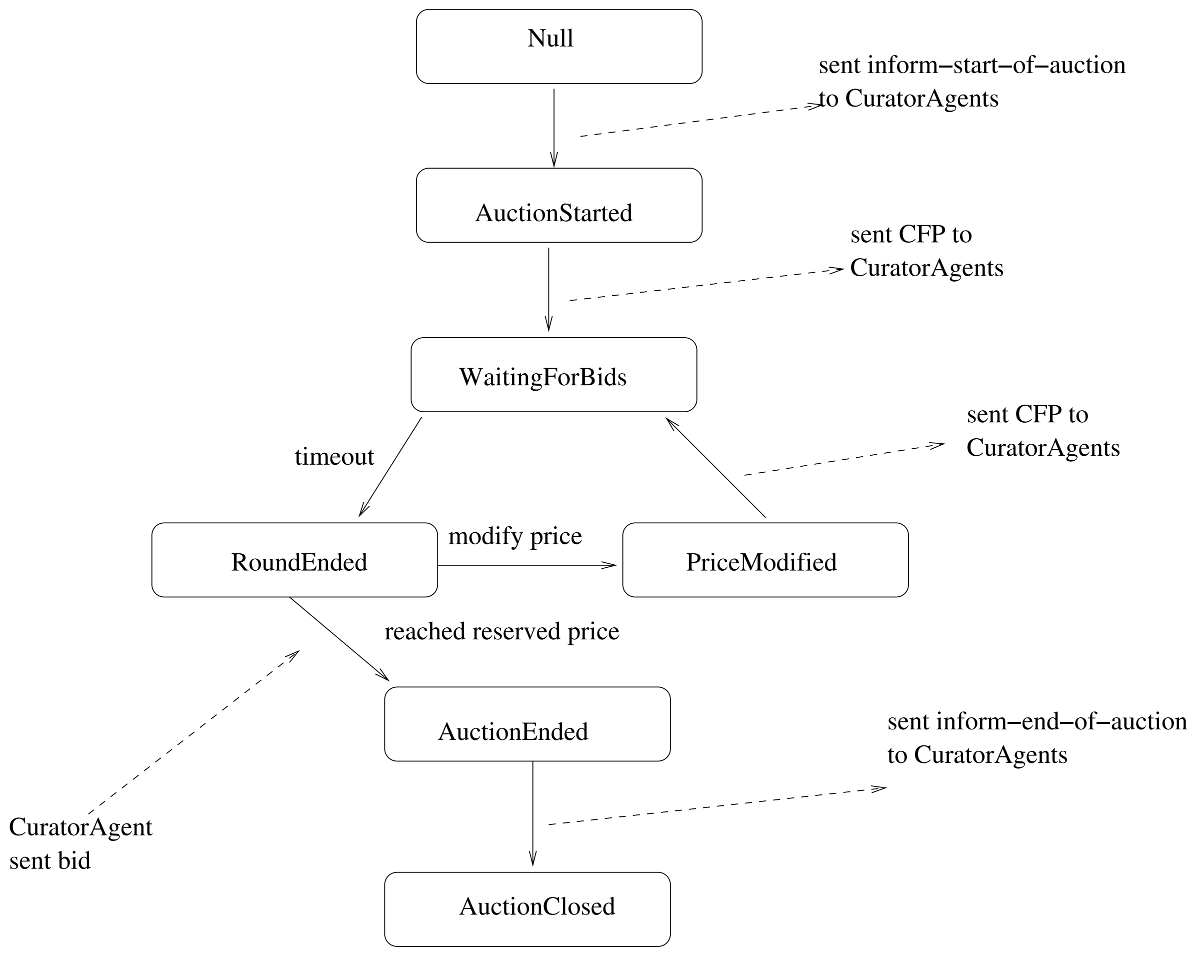


Figure 32: Statechart diagram for ArtistManagerAgent

III.2 TourGuideAgent

The TourGuideAgent contains internal processing for building virtual tours bases on artifacts. Many of the internal states depends on external events from the interaction protocol.

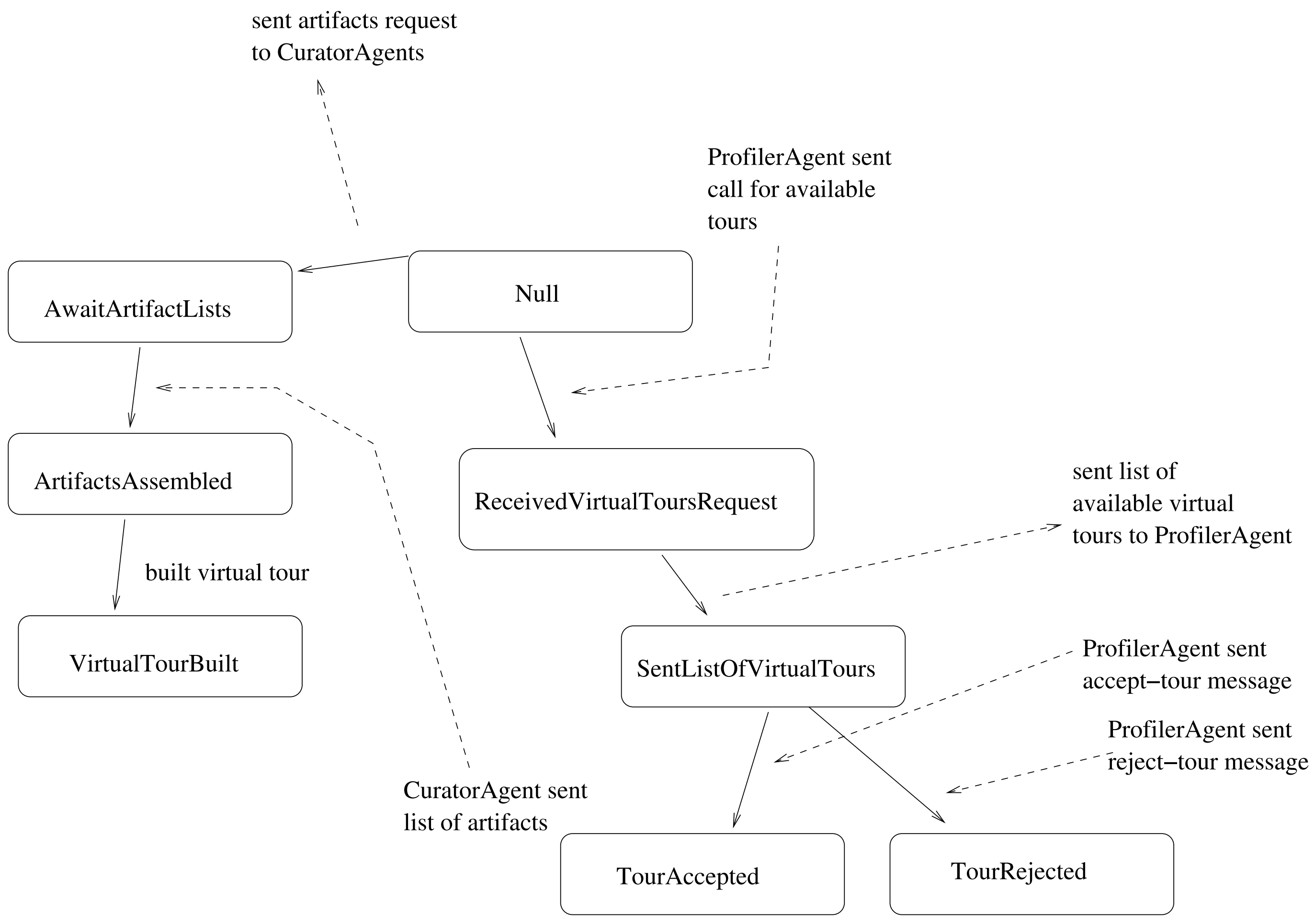

sent artifacts request to CuratorAgents
ProfilerAgent sent call for available tours
AwaitArtifactLists
Null
ArtifactsAssembled
ReceivedVirtualToursRequest
sent list of available virtual tours to ProfilerAgent
built virtual tour
VirtualTourBuilt
SentListOfVirtualTours
ProfilerAgent sent accept−tour message
ProfilerAgent sent reject−tour message
CuratorAgent sent list of artifacts
TourAccepted
TourRejected

Figure 33: Statechart diagram for TourGuideAgent

III.3 ProfilerAgent

The ProfilerAgent contains internal processing for choosing to participate in auctions, find virtual tours, as well as visiting artifacts. Many of the internal states depends on external events from the interaction protocol.

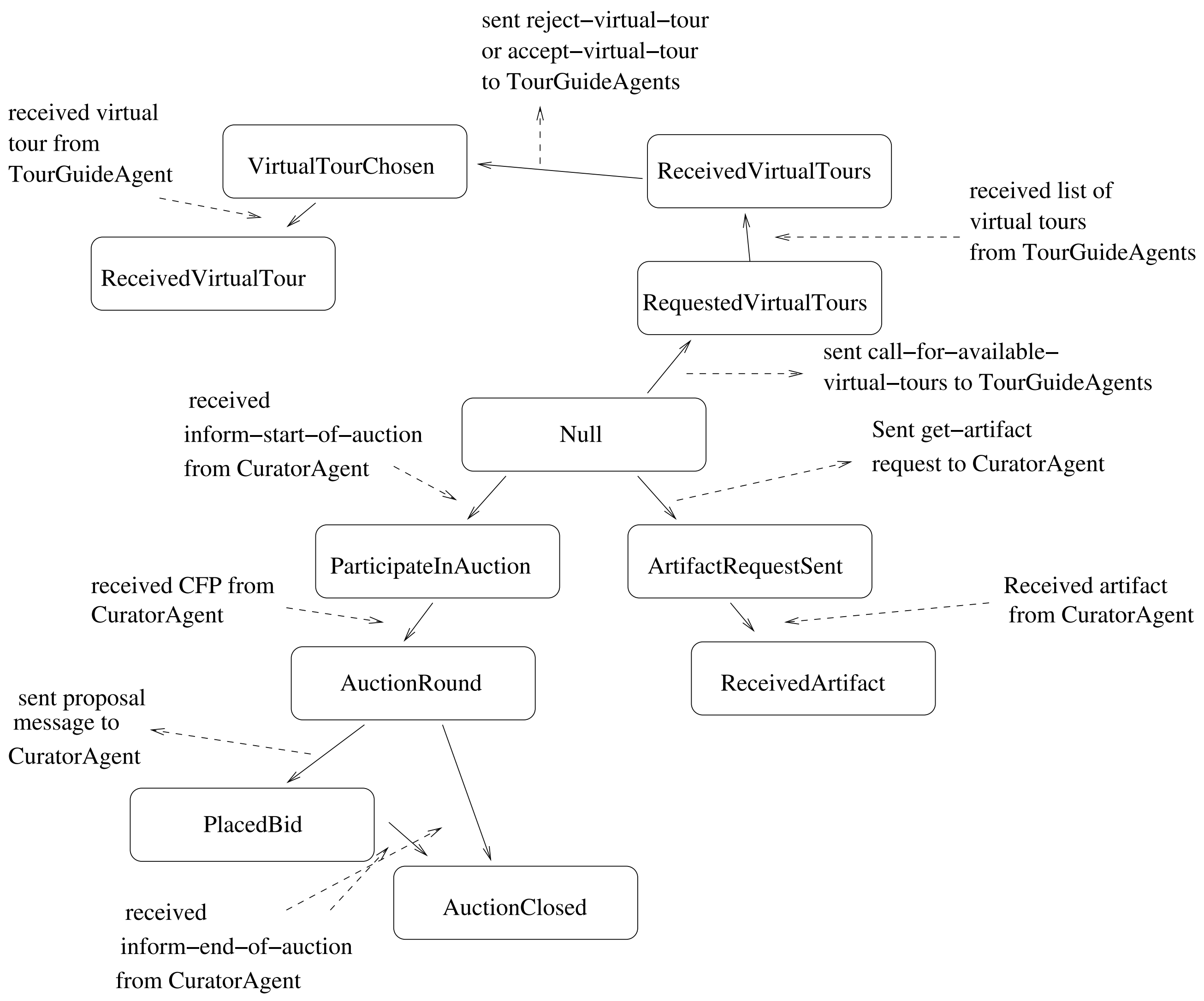
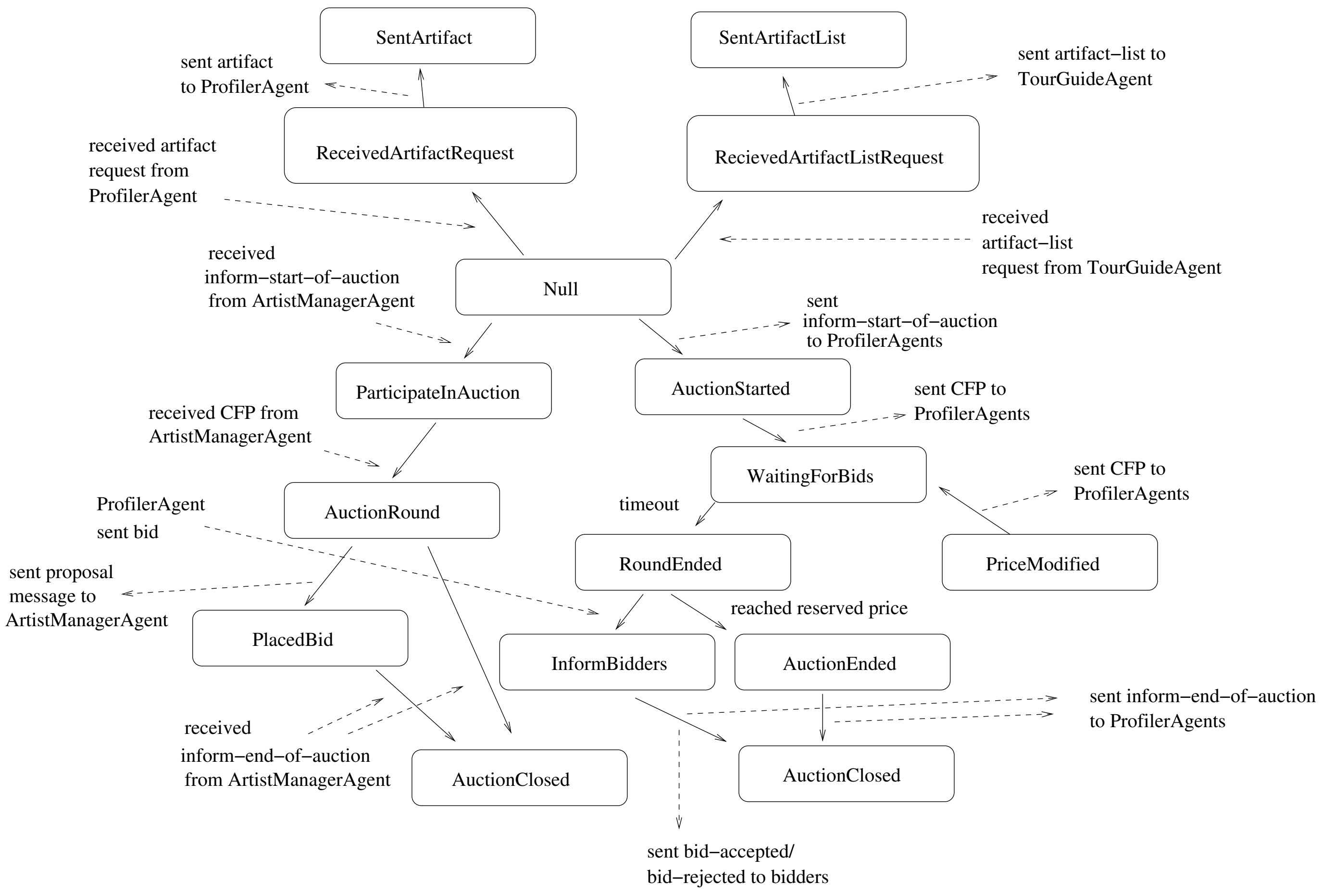


Figure 34: Statechart diagram for ProfilerAgent

III.4 CuratorAgent

The CuratorAgent contains internal processing for participating in auctions, modifying prices in the dutch auction as well as selecting a winner when there is multiple bids. Many of the internal states depends on external events from the interaction protocol.



SentArtifact
sent artifact to ProfilerAgent
ReceivedArtifactRequest
received artifact request from ProfilerAgent
SentArtifactList
sent artifact–list to TourGuideAgent
RecievedArtifactListRequest
received artifact–list request from TourGuideAgent
received inform–start–of–auction from ArtistManagerAgent
Null
sent inform–start–of–auction to ProfilerAgents
ParticipateInAuction
AuctionStarted
sent CFP to ProfilerAgents
received CFP from ArtistManagerAgent
WaitingForBids
sent CFP to ProfilerAgents
ProfilerAgent sent bid
AuctionRound
timeout
RoundEnded
PriceModified
sent proposal message to ArtistManagerAgent
PlacedBid
reached reserved price
InformBidders
AuctionEnded
sent inform–end–of–auction to ProfilerAgents
received inform–end–of–auction from ArtistManagerAgent
AuctionClosed
AuctionClosed
sent bid–accepted/ bid–rejected to bidders

Figure 35: Statechart diagram for CuratorAgent

IV. Task 3 - UML Class Diagram Revisited

In this section the results from modelling the SmartMuseum framework with class diagrams as described in [3] is presented.

I. ArtistManagerAgent

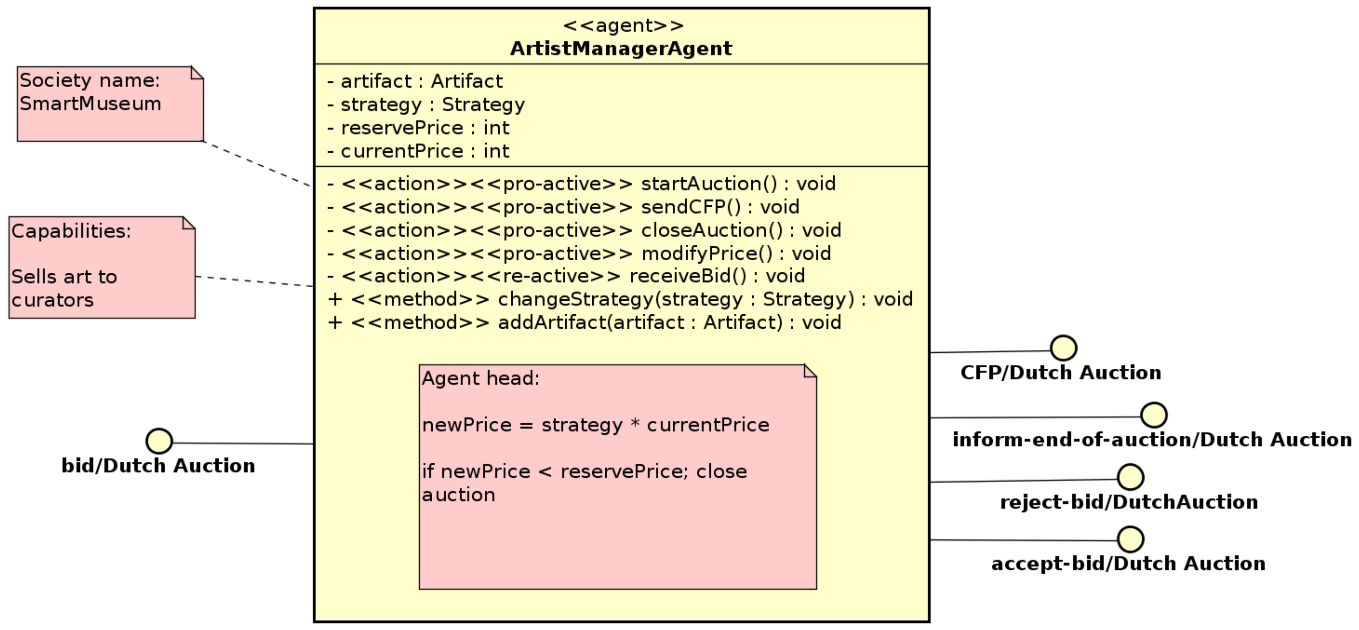

Society name:
SmartMuseum
Capabilities:
Sells art to curators
<<agent>>
ArtistManagerAgent
- artifact : Artifact
- strategy : Strategy
- reservePrice : int
- currentPrice : int
- <<action>><<pro-active>> startAuction() : void
- <<action>><<pro-active>> sendCFP() : void
- <<action>><<pro-active>> closeAuction() : void
- <<action>><<pro-active>> modifyPrice() : void
- <<action>><<re-active>> receiveBid() : void
+ <<method>> changeStrategy(strategy : Strategy) : void
+ <<method>> addArtifact(artifact : Artifact) : void
Agent head:
newPrice = strategy * currentPrice
if newPrice < reservePrice; close auction
bid/Dutch Auction
CFP/Dutch Auction
inform-end-of-auction/Dutch Auction
reject-bid/DutchAuction
accept-bid/Dutch Auction

Figure 36: Class diagram for ArtistManagerAgent

II. TourGuideAgent

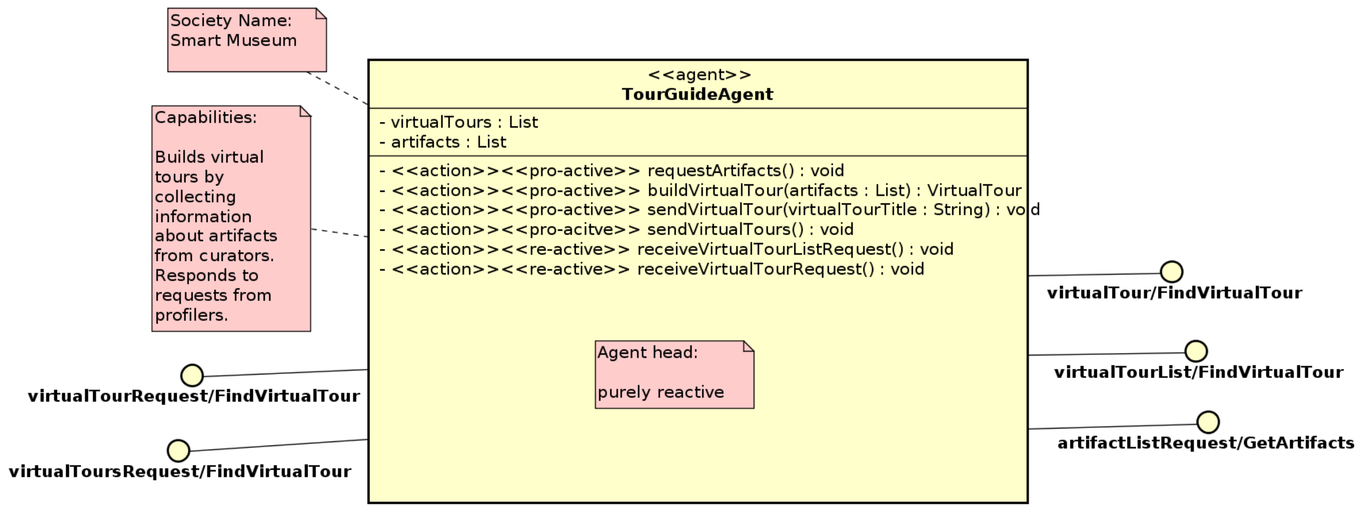

Society Name:
Smart Museum
Capabilities:
Builds virtual tours by collecting information about artifacts from curators. Responds to requests from profilers.
<<agent>>
TourGuideAgent
- virtualTours : List
- artifacts : List
- <<action>><<pro-active>> requestArtifacts() : void
- <<action>><<pro-active>> buildVirtualTour(artifacts : List) : VirtualTour
- <<action>><<pro-active>> sendVirtualTour(virtualTourTitle : String) : void
- <<action>><<pro-acitve>> sendVirtualTours() : void
- <<action>><<re-active>> receiveVirtualTourListRequest() : void
- <<action>><<re-active>> receiveVirtualTourRequest() : void
Agent head:
purely reactive
virtualTourRequest/FindVirtualTour
virtualToursRequest/FindVirtualTour
virtualTour/FindVirtualTour
virtualTourList/FindVirtualTour
artifactListRequest/GetArtifacts

Figure 37: Class diagram for TourGuideAgent

III. ProfilerAgent

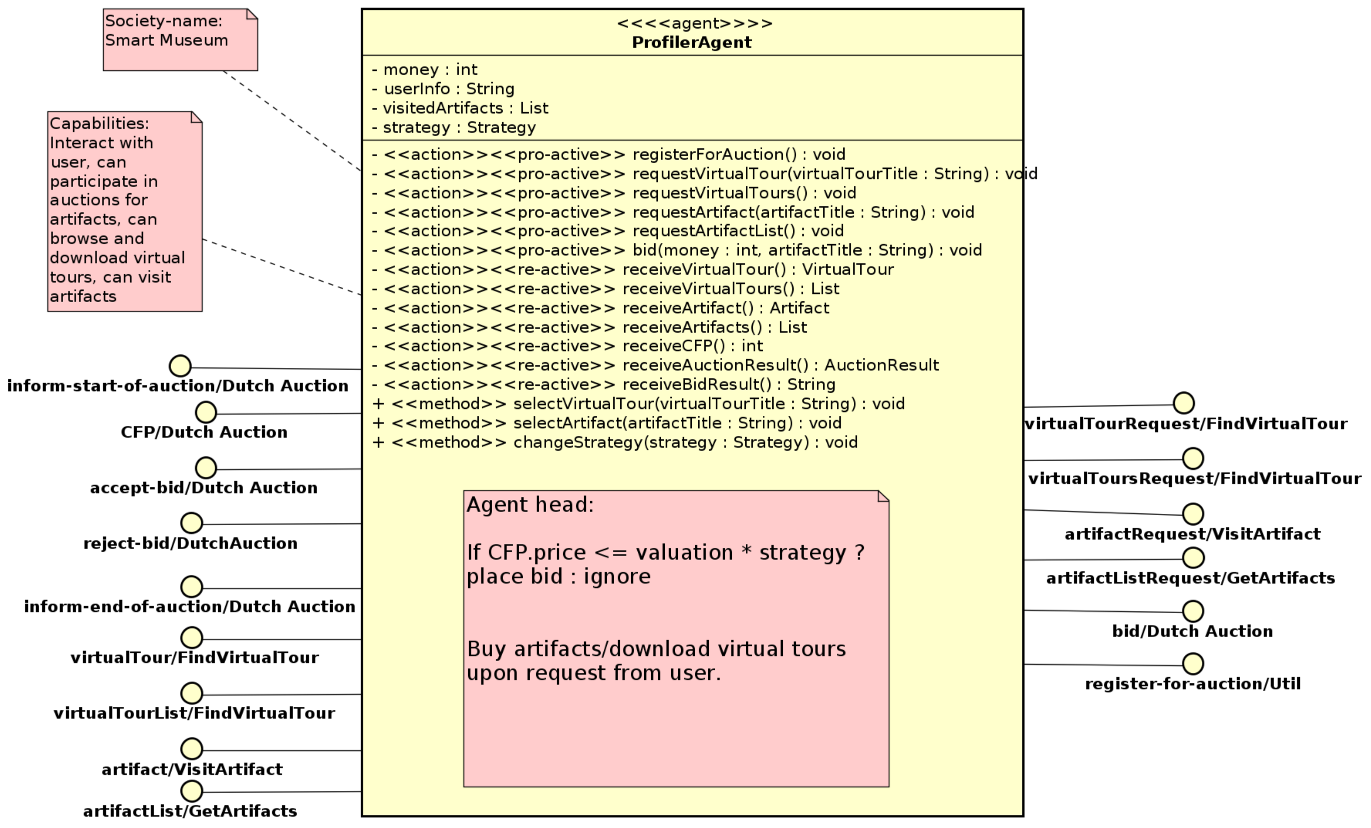

Society-name: Smart Museum
Capabilities: Interact with user, can participate in auctions for artifacts, can browse and download virtual tours, can visit artifacts
<<<<agent>>>>
ProfilerAgent
- money : int
- userInfo : String
- visitedArtifacts : List
- strategy : Strategy
- <<action>><<pro-active>> registerForAuction() : void
- <<action>><<pro-active>> requestVirtualTour(virtualTourTitle : String) : void
- <<action>><<pro-active>> requestVirtualTours() : void
- <<action>><<pro-active>> requestArtifact(artifactTitle : String) : void
- <<action>><<pro-active>> requestArtifactList() : void
- <<action>><<pro-active>> bid(money : int, artifactTitle : String) : void
- <<action>><<re-active>> receiveVirtualTour() : VirtualTour
- <<action>><<re-active>> receiveVirtualTours() : List
- <<action>><<re-active>> receiveArtifact() : Artifact
- <<action>><<re-active>> receiveArtifacts() : List
- <<action>><<re-active>> receiveCFP() : int
- <<action>><<re-active>> receiveAuctionResult() : AuctionResult
- <<action>><<re-active>> receiveBidResult() : String
+ <<method>> selectVirtualTour(virtualTourTitle : String) : void
+ <<method>> selectArtifact(artifactTitle : String) : void
+ <<method>> changeStrategy(strategy : Strategy) : void
Agent head:
If CFP.price <= valuation * strategy ? place bid : ignore
Buy artifacts/download virtual tours upon request from user.
inform-start-of-auction/Dutch Auction
CFP/Dutch Auction
accept-bid/Dutch Auction
reject-bid/DutchAuction
inform-end-of-auction/Dutch Auction
virtualTour/FindVirtualTour
virtualTourList/FindVirtualTour
artifact/VisitArtifact
artifactList/GetArtifacts
virtualTourRequest/FindVirtualTour
virtualToursRequest/FindVirtualTour
artifactRequest/VisitArtifact
artifactListRequest/GetArtifacts
bid/Dutch Auction
register-for-auction/Util

Figure 38: Class diagram for ProfilerAgent

IV. CuratorAgent

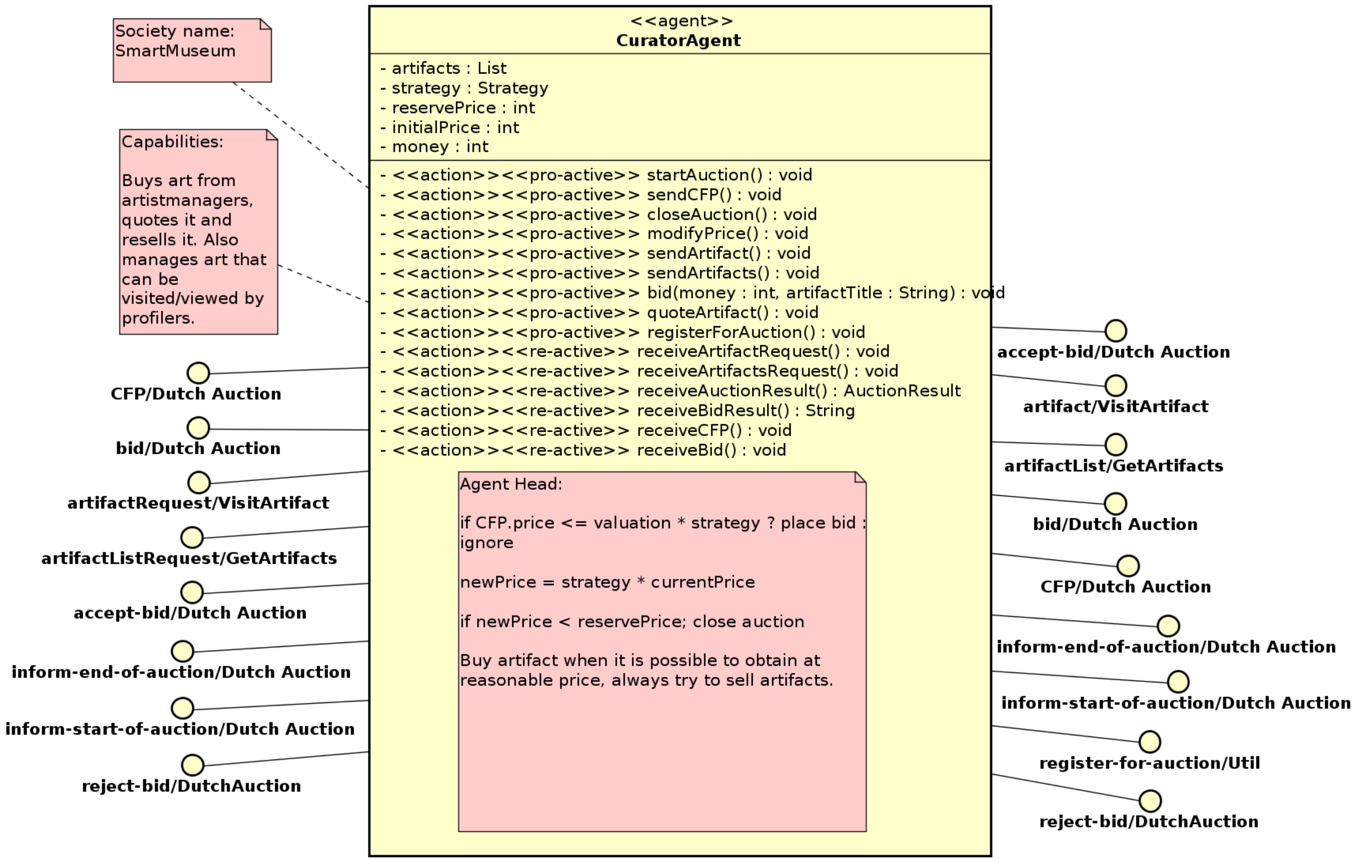

Society name:
SmartMuseum
Capabilities:
Buys art from artistmanagers, quotes it and resells it. Also manages art that can be visited/viewed by profilers.
<<agent>>
CuratorAgent
- artifacts : List
- strategy : Strategy
- reservePrice : int
- initialPrice : int
- money : int
- <<action>><<pro-active>> startAuction() : void
- <<action>><<pro-active>> sendCFP() : void
- <<action>><<pro-active>> closeAuction() : void
- <<action>><<pro-active>> modifyPrice() : void
- <<action>><<pro-active>> sendArtifact() : void
- <<action>><<pro-active>> sendArtifacts() : void
- <<action>><<pro-active>> bid(money : int, artifactTitle : String) : void
- <<action>><<pro-active>> quoteArtifact() : void
- <<action>><<pro-active>> registerForAuction() : void
- <<action>><<re-active>> receiveArtifactRequest() : void
- <<action>><<re-active>> receiveArtifactsRequest() : void
- <<action>><<re-active>> receiveAuctionResult() : AuctionResult
- <<action>><<re-active>> receiveBidResult() : String
- <<action>><<re-active>> receiveCFP() : void
- <<action>><<re-active>> receiveBid() : void
Agent Head:
if CFP.price <= valuation * strategy ? place bid : ignore
newPrice = strategy * currentPrice
if newPrice < reservePrice; close auction
Buy artifact when it is possible to obtain at reasonable price, always try to sell artifacts.
CFP/Dutch Auction
bid/Dutch Auction
artifactRequest/VisitArtifact
artifactListRequest/GetArtifacts
accept-bid/Dutch Auction
inform-end-of-auction/Dutch Auction
inform-start-of-auction/Dutch Auction
reject-bid/DutchAuction
accept-bid/Dutch Auction
artifact/VisitArtifact
artifactList/GetArtifacts
bid/Dutch Auction
CFP/Dutch Auction
inform-end-of-auction/Dutch Auction
inform-start-of-auction/Dutch Auction
register-for-auction/Util
reject-bid/DutchAuction

Figure 39: Class diagram for CuratorAgent

V. Task 4 - Role-based Modeling with RoMAS

In this section the result from role-based modeling with the RoMAS [12] method is presented.

RoMAS is a role-based modeling methods for agent systems, it introduces a slightly new concept of roles as compared to the role concept used in Task 1 for GAIA modeling. In particular RoMAS modeling assumes that agent and role bindings are dynamic.

I. Role-based Modeling of SmartMuseum Framework

I.1 Use cases

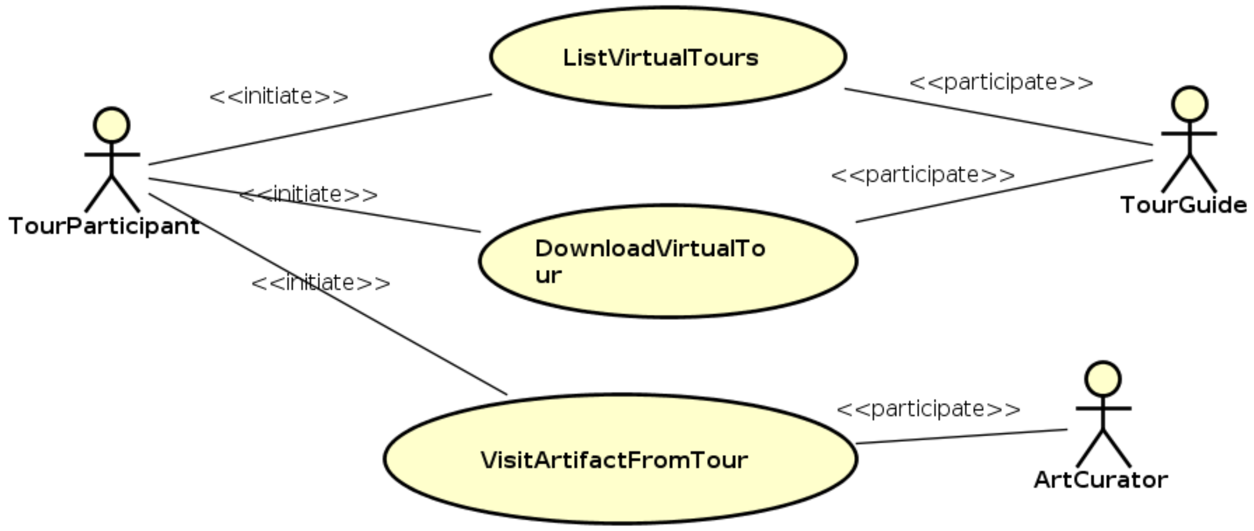


Figure 40: ConsumeVirtualTour usecase

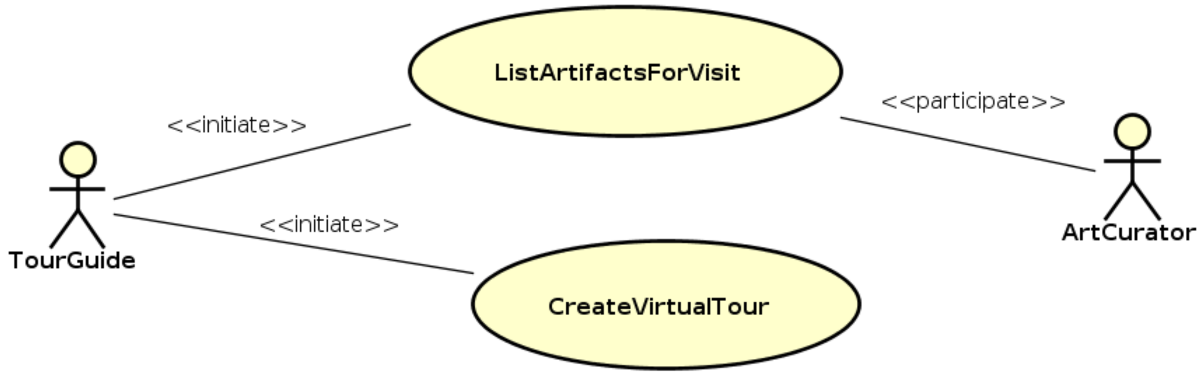


Figure 41: CreateVirtualTour usecase

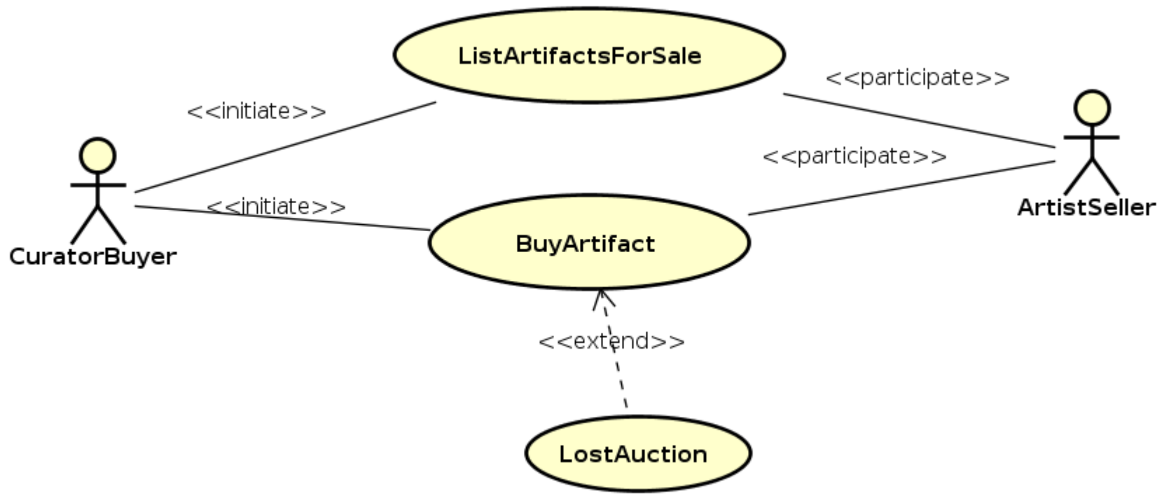


Figure 42: CuratorBuy usecase

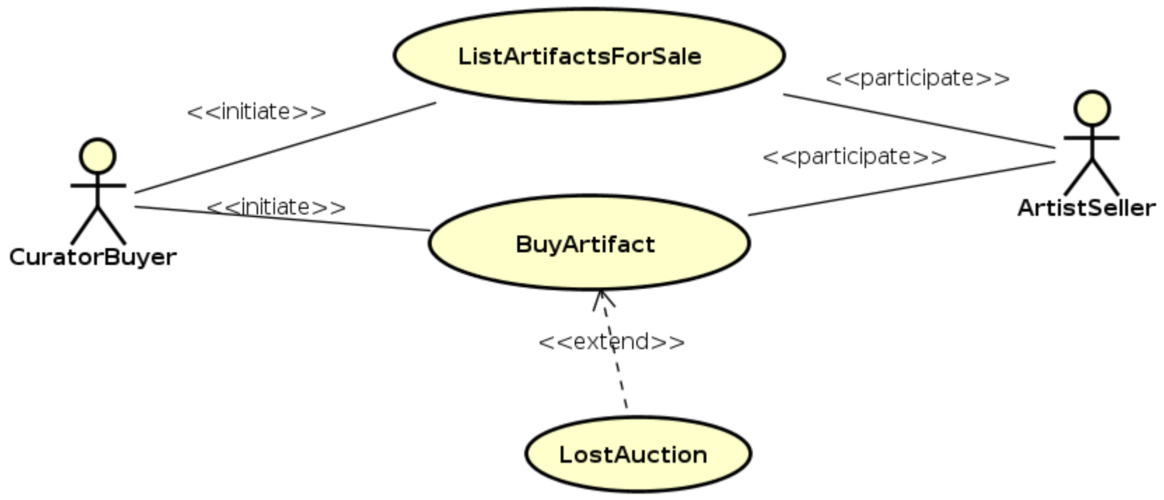


Figure 43: HobbyBuy usecase

I.2 Roles

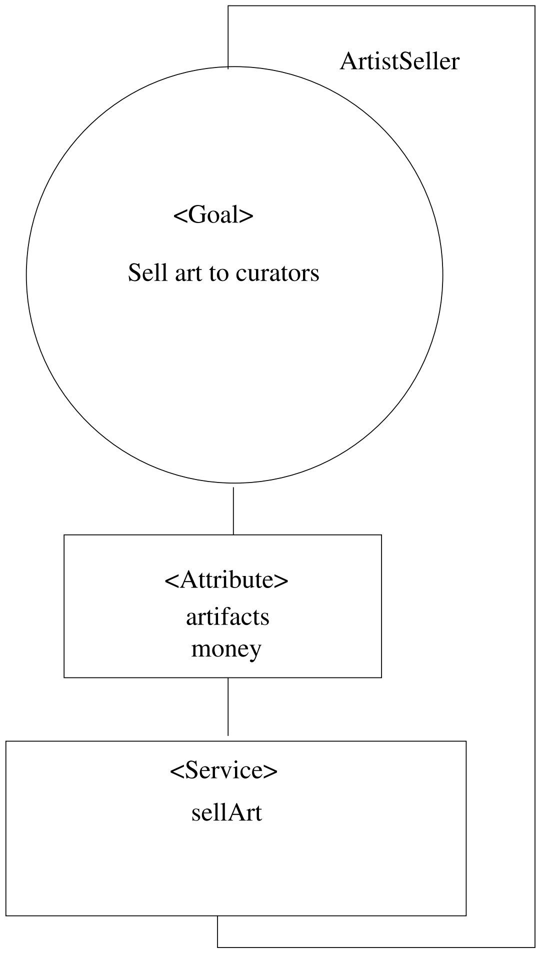


Figure 44: ArtistSeller role

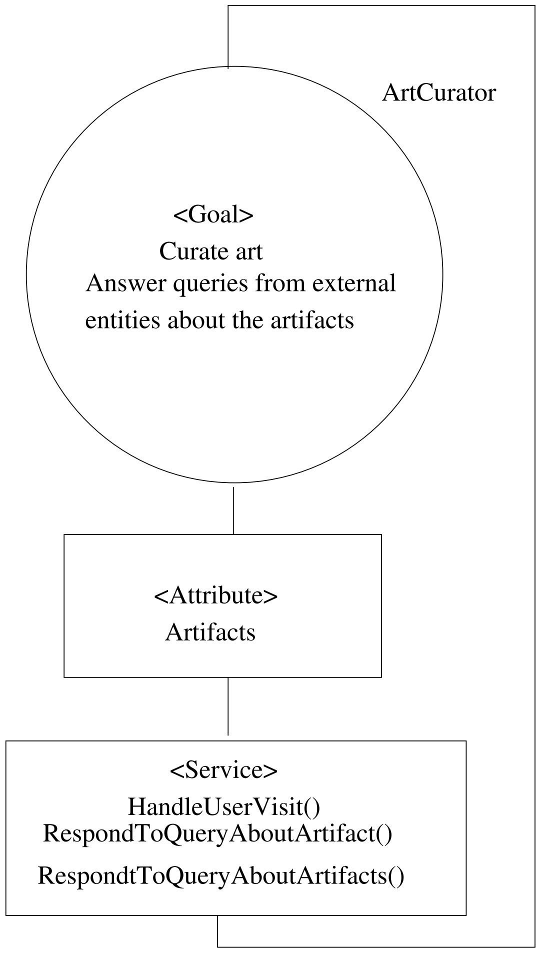

ArtCurator
<Goal>
Curate art
Answer queries from external entities about the artifacts
<Attribute>
Artifacts
<Service>
HandleUserVisit()
RespondToQueryAboutArtifact()
RespondtToQueryAboutArtifacts()

Figure 45: ArtCurator role

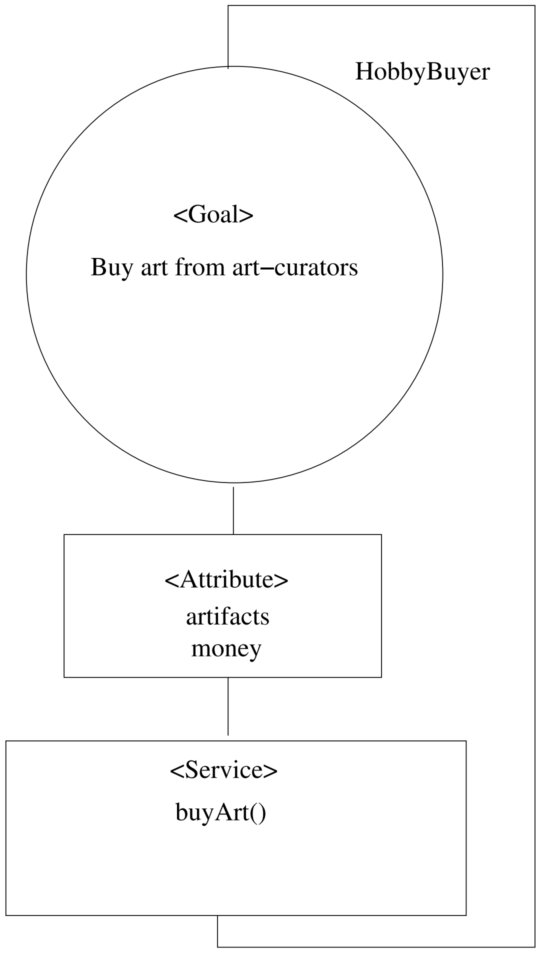


Figure 46: HobbyBuyer role

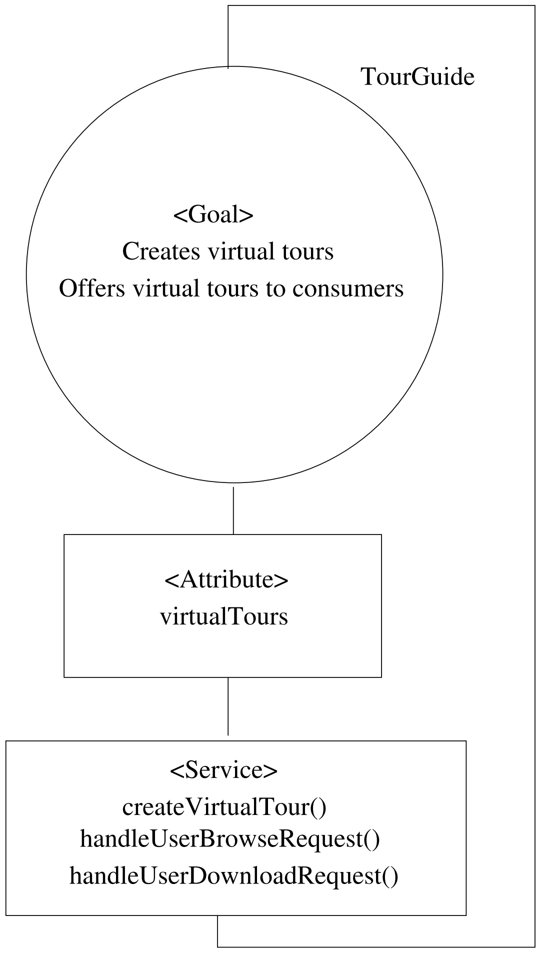


Figure 47: TourGuide role

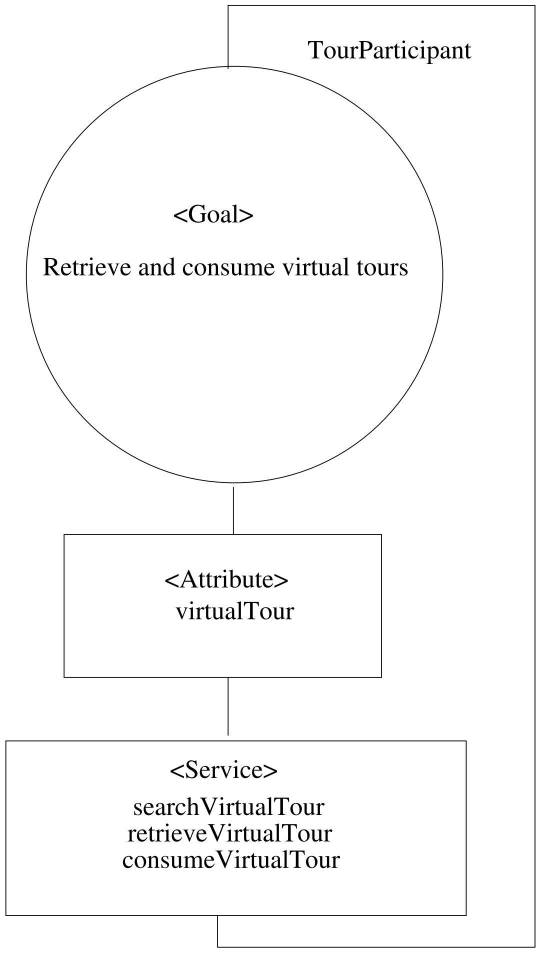


Figure 48: TourParticipant role

I.3 RoleOrganization

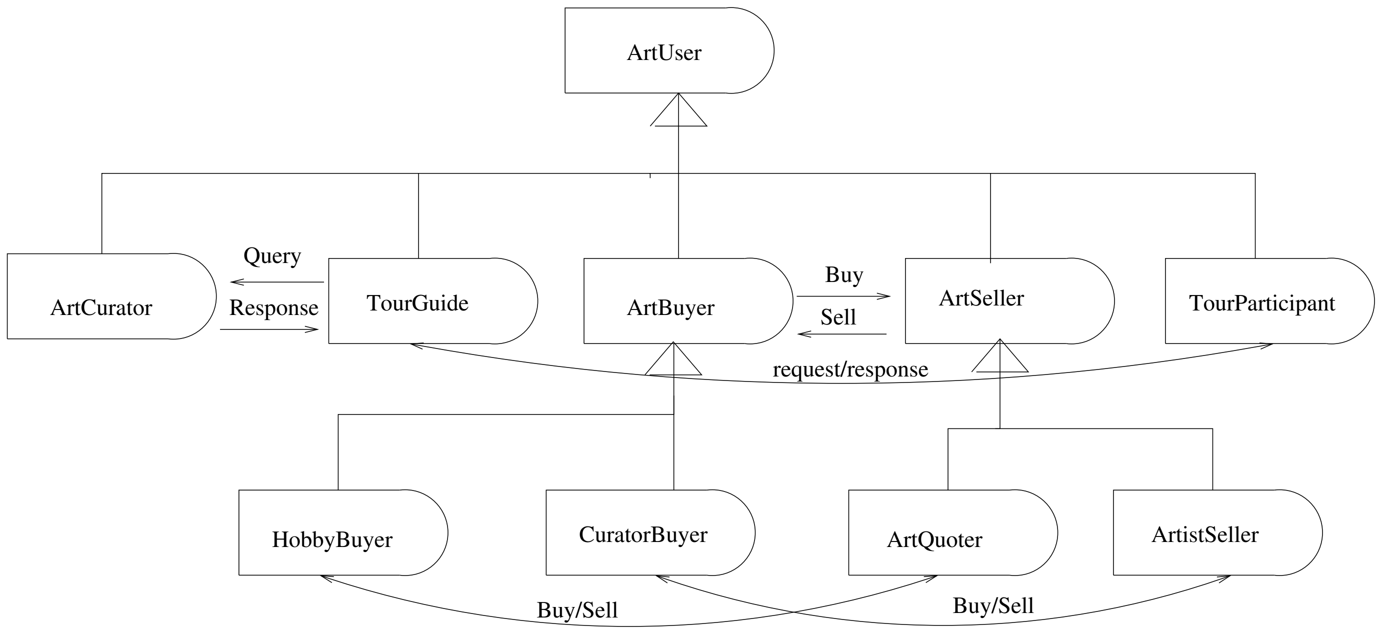


Figure 49: Roles Organization

I.4 Binding Agents to Roles

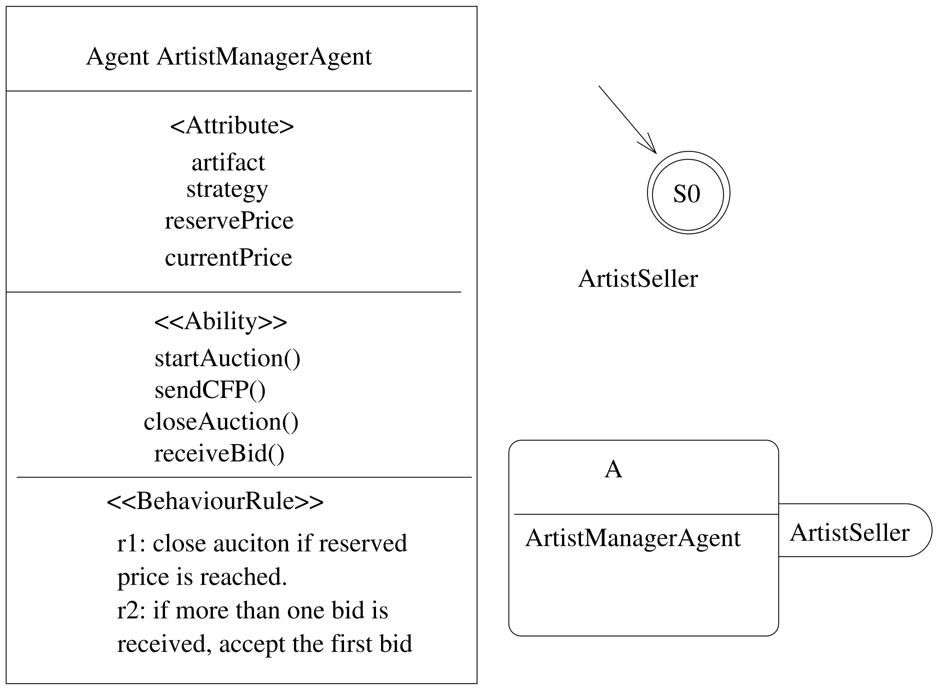


Figure 50: ArtistManagerAgent binding to roles

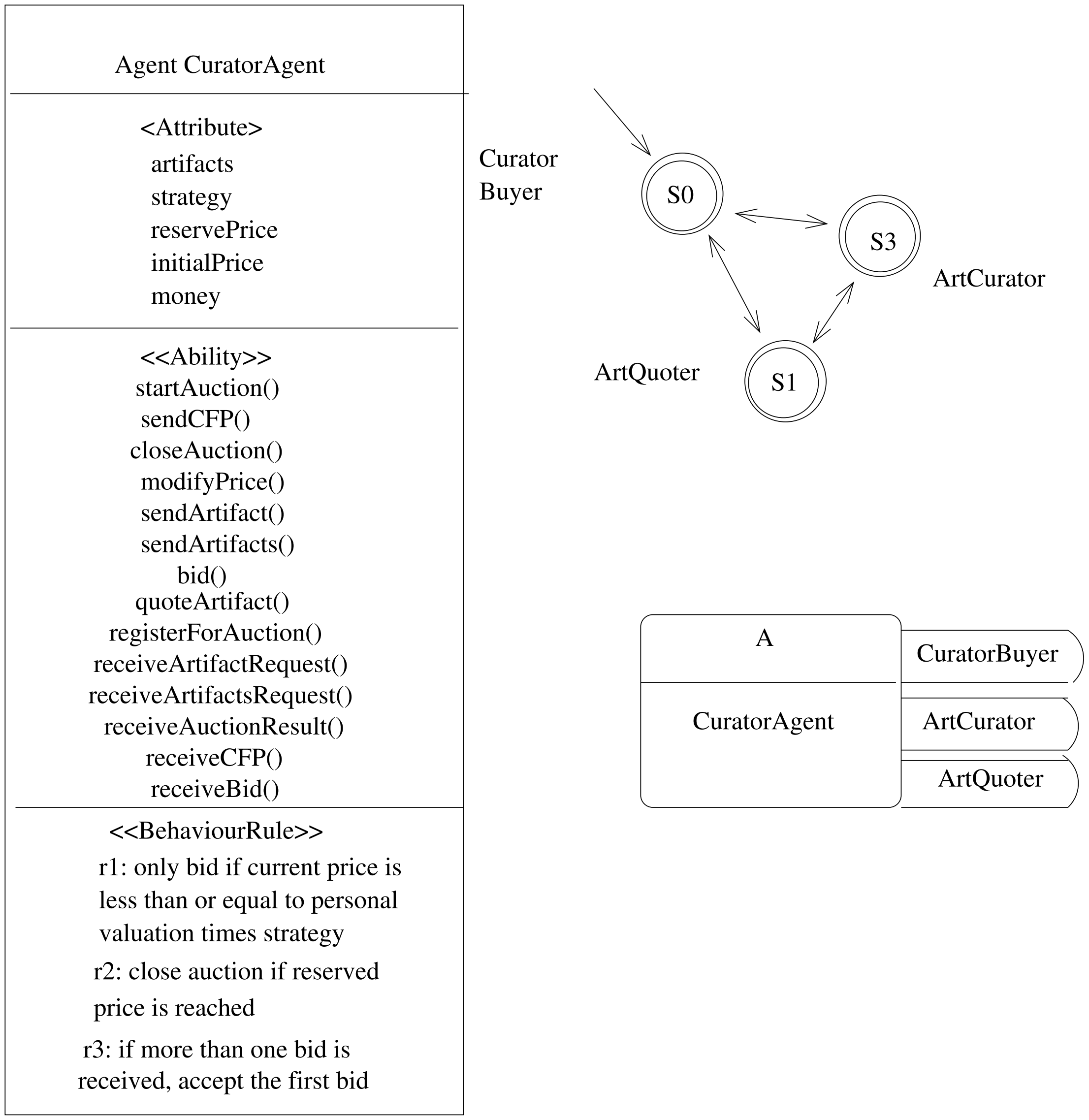


Figure 51: CuratorAgent binding to roles

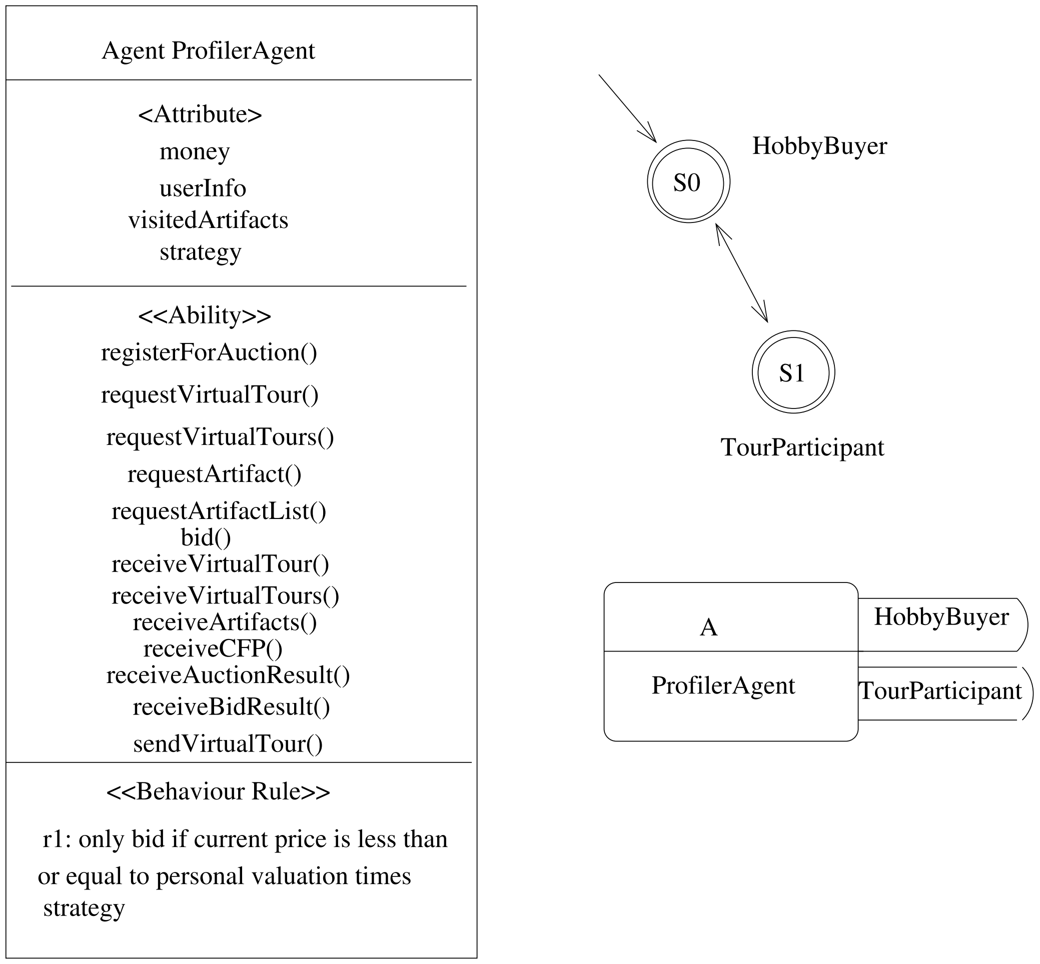


Figure 52: ProfilerAgent binding to roles

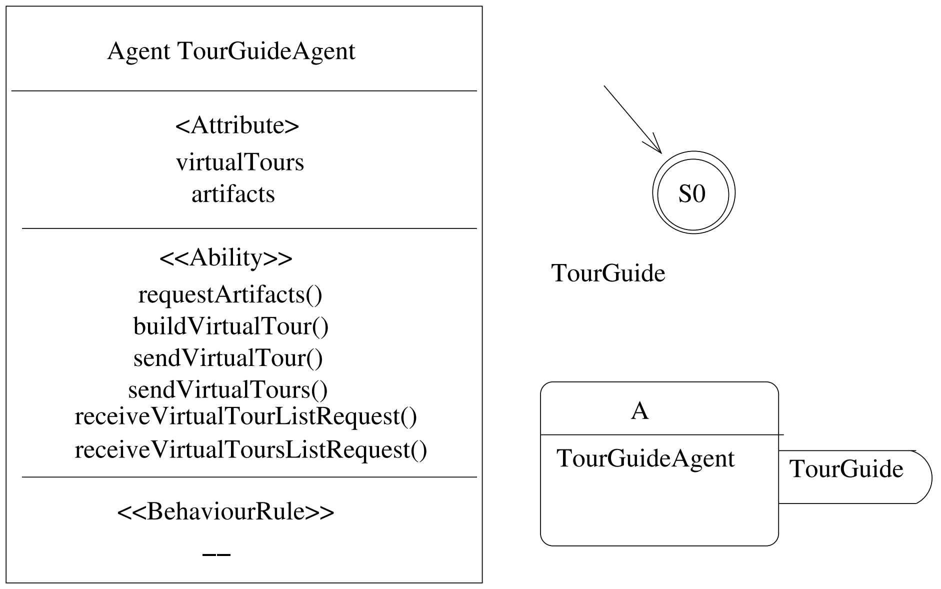


Figure 53: TourGuideAgent binding to roles

II. Comparison between RoMAS and GAIA

The resulting models are very similar, the same number of roles were identified in both cases. The models differ in that with RoMAS the notion that an agent is made out of roles is very explicit while in GAIA the roles gets "lost" when moving into the design phase. Another disparity between the two resulting models is that in the RoMAS model the agent can dynamically change roles over time, while in GAIA the model is assumed to be static. As stated in [14], the GAIA methdology is best suited for domains which inhabit the following characteristics:

> *The organisation structure of the system is static, in that inter-agent relationships do not change at run-time. The abilities of agents and the services they provide are static, in that they do not change at run-time.*[14]

In constrast, the RoMAS method do support dynamic binding of role to agent at runtime, and the model only declares the *initial* binding between agents and roles.

VI. Task 5 - Comparing JADE to other Agent Platforms

In this section it follows a high-level comparison of JAVA Agent DEvelopment Framework (JADE) [8], FIPA Open Source (FIPA-OS) [4], and JACK Intelligent Agents [2].

JADE is a framework for developing M.A.S in java where the developer is supposed to use the regular Java language but adopt the guidelines provided by the framework to construct agent systems. JACK is an agent platform developed on top of and integrated with Java, it works as an agent-oriented extension to the object-oriented Java, it is its own language although closely coupled with java. FIPA-OS is a set of components which constitute as the core of the FIPA specification, which means that the developer can utilize this and focus on solving the real problem instead of building agent infrastructure.

I. Services and Architectures

Platform	Architecture	Services
JADE	Container-oriented architecture. Distributed containers that are connected over the network through a message transport system provided by JADE. [6]	Agent Management System (AMS) can kill/create agents, provides a naming service for all agents on the platform. Directory Facilitator (DF) provides a Yellow Pages service which agents can use to find each other. Other notable parts of JADE's infrastructure are: ACL infrastructure, support for agent mobility, built-in support for FIPA-compliant protocols. [6]
FIPA-OS	Component-oriented architecture. When deploying an agent system the developer choose a set of components to use, some components are mandatory, some are optional. [5]	Core components: Agent Shell provides a shell for agent implementation, TM (Task Manager) support ability to split functionality of agents, CM (Conversation Manager) enables to track conversation state at the performative level, MTS (Message Transport Service) the general messaging service that enables agents to communicate. Beyond the mandatory components there are a bunch of optional components, e.g database factory, parser factory. FIPA-OS also provides Directory Facilitator and Agent Management System just as JADE. Other notable parts of FIPA-OS's infrastructure are: ACL infrastructure, support for agent mobility, built-in support for FIPA-compliant protocols. [5].
JACK	JACK Agent Kernel, a runtime engine that provides the infrastructure for developing agent systems. JACK uses a communication layer. The developer is not actively interacting with the kernel but instead uses constructs in the language declaring the name of agents, the address of other agents etc, which will allow the kernel to provide the underlying infrastructure, e.g communication between agents. [9]	JACK provides a default messaging service over UDP and has constructs for a naming-service in the language. One particular agent architecture have stronger support than others in JACK and that is the BDI architecture. JACK provides different services related to BDI such as BDI Models, ways of declaring plans, beliefs and external/internal events. TaskManager which allows agents to schedule tasks. JACK also provides services for team-oriented programming as a way of coordinating between agents. Other notable parts of JACK's infrastructure are: No support for agent mobility, built-in support for FIPA-compliant protocols. [10]

II. Implementation Comparison

- **JADE**:
  In JADE, service implementation can be done by developing agents that listens for certain type of messages, perform some action, and respond. The functionality for listening for a certain type of messages is provided by the JADE runtime and allows to have multiple services on the same host in a convenient way. Service registration and discovery is closley coupled with the AMS Service and the DF service. The AMS service ensures a global name space with unique names for adressing, the DF service is used to register services and to find other registered services. AMS and DF are agents on their own which means that the interaction with these services is done through messages passing. When registering a service one could define different properties like name, type, description etc.
- **FIPA-OS**:
  Service implementation in FIPA-OS can be done by developing agents that listens for specific type of incoming connections by developing Tasks, Tasks's can then be associated to different events which allows to have multiple tasks on the same host/agent. FIPA-OS uses the same type of services like JADE for service registration and discovery: AMS and DF [5].
- **JACK**:
  In JACK an agent service can be implemented by using the constructs in the language like `#handles` to declare which events this service/agent should react to, `#posts`, `#sends` for event/message sending, `#uses` for declaring plans. To set up a service in JACK which can be used by other agents, one can use a designated process/agent as a *name-server*, this name server can be designed in different ways, for example it could do lookups of names to port/address, or it could provide a service-registration service or similar. [9]

III. Notable Projects

- **JADE**:
  - *AMUSE (Agent-based Multi-User Social Environment)*: Software platform that facilitates the development of distributed social applications involving users that cooperate/compete to achieve common or private goals. Within this scope the primary focus of Amuse is on multi player on-line games [13].
- **FIPA-OS**:
  - *CRUMPET*: The overall aim of the CRUMPET project is to implement, validate, and trial tourism-related value-added services for nomadic users (across mobile and fixed networks). [7]
- **JACK**:
  - *Realistic Virtual Actors*: Simulation system in a military context, uses intelligent agents for the simulation [1].
  - *Human Behaviour Representation*: Application where realistic human behaviour is generated [1].

IV. Personal Judgement

JADE is the only platform I have had the chance to get practical experience working with throughout the assignments in this course. JACK and FIPA-OS are two other platforms with similar purposes as JADE, that I've only read about.

Something I've came to appreciate when using JADE to build agent systems is the simplicity in how much you can build just by using the few default services provided by JADE like DF, AMS and the message transport system. In my opinion JADE neatly provides the necessary infrastructure without getting in the way for the programmer who can focus on solving the specific problem at hand. Another pro of JADE in my opinion is the adoption of the FIPA specification, many protocols and message formats are supported out of the box. Something I feel is lacking in JADE is additional constructs in the framework for designing agents in the micro perspective. Agent design in JADE is done through composing general behaviours, there is no explicit constructs for using BDI architecture or similar.

Another deficiency with JADE in my opinion is the support for agent mobility, although JADE provide some tools that allows to implement mobile agents, the support is very poor if you compare to the standards of the rest of the platform. At the moment of writing this the documentation for agent mobility is also sparse, perhaps because this particular reason.

FIPA-OS is just as the name implies is also compliant with the FIPA specifications and is very similar to JADE. An advantage I've found with FIPA-OS compared to JADE is the ability to combine different components of the infrastructure archticture as you like, in this aspect FIPA-OS provide more eligibility than JADE. FIPA-OS uses the Task abstraction which seems to be analogous to the Behaviour abstraction in JADE.

JACK uses a different approach to FIPA-OS and JADE in that it has its own language. Something I feel missing in JACK is clear guidelines for designing agent systems in the macro perspective as both JADE and FIPA-OS provides with their default services, also JACK does not seem to have as good support for the FIPA specifications as JADE and FIPA-OS. JACK on the other hand gives more sophisticated structures for designing agents in the micro perspective and have very good support for BDI architectures in particular.

References

[1] AOS. Applications. `http://aosgrp.com/applications/`. [Online; accessed 13-Dec-2016].

[2] AOS. Jack. `http://aosgrp.com/products/jack/`, 2016. [Online; accessed 12-Dec-2016].

[3] Bernhard Bauer. *UML Class Diagrams Revisited in the Context of Agent-Based Systems*, pages 101–118. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[4] emorphia. Fipa-os. `http://fipa-os.sourceforge.net/index.htm`, 2003. [Online; accessed 12-Dec-2016].

[5] emorphia. Fipa-os developers guide. `http://fipa-os.sourceforge.net/docs/Developers_Guide.pdf`, 2016. [Online; accessed 12-Dec-2016].

[6] Tiziana Trucco Giovanni Rimassa Fabio Bellifemine, Giovanni Caire. Jade programmer's guide. `http://jade.tilab.com/doc/programmersguide.pdf`, 2016. [Online; accessed 13-Dec-2016].

[7] Foundation for Intelligent Physical Agents. External projects. `http://www.fipa.org/resources/projects.html#comma`. [Online; accessed 13-Dec-2016].

[8] Telecom Italia. Java agent development framework. http://jade.tilab.com/, 2016. [Online; accessed 11-Nov-2016].

[9] Agent Oriented Software Pty. Ltd. Jack intelligent agents agent manual. http://www.aosgrp.com/documentation/jack/Agent_Manual.pdf, 2005. [Online; accessed 13-Dec-2016].

[10] Michael Luck, Ronald Ashri, and Mark d'Inverno. *Agent-Based Software Development*. Artech House, 2004.

[11] James Odell, H. Van Dyke Parunak, and Bernhard Bauer. Representing agent interaction protocols in uml. In *First International Workshop, AOSE 2000 on Agent-oriented Software Engineering*, pages 121–140, Secaucus, NJ, USA, 2001. Springer-Verlag New York, Inc.

[12] QI YAN, LI-JUN SHAN, XIN-JUN MAO AND ZHI-CHANG QI. ROMAS: A ROLE-BASED MODELING METHOD FOR MULTI-AGENT SYSTEM*.

[13] Telecomin Italia SpA. Amuse. http://jade.tilab.com/amuseproject/. [Online; accessed 13-Dec-2016].

[14] Michael Wooldridge, Nicholas R. Jennings, and David Kinny. The gaia methodology for agent-oriented analysis and design. *Autonomous Agents and Multi-Agent Systems*, 3(3):285–312, September 2000.